



Introduction to SQL

ECE 356
University of Waterloo
Dr. Paul A.S. Ward

Acknowledgment: slides derived from Dr. Wojciech Golab
based on materials provided by
Silberschatz, Korth, and Sudarshan, copyright 2010
(source: www.db-book.com)



SQL Basics

- DML syntax and semantics
 - Selection, projection, Cartesian product and joins, renaming
 - set operations
 - aggregation and grouping
 - ordering results
 - nested subqueries
- DDL syntax and semantics
 - SQL data types
 - creating tables, constraints, views
 - insert, update, and delete rows
 - auto-increment attributes
 - stored procedures, cursors, and triggers



Brief History of SQL

- SEQUEL (Structured English Query Language) developed for *System R* project at IBM San Jose Research Lab in early 1970s. Later shortened to SQL (Structured Query Language).
- Standardized by ANSI (since 1986) and ISO (since 1987):
 - SQL-86 (first cut)
 - SQL-89 (added integrity constraints)
 - **SQL-92** (DATE and VARCHAR types, NATURAL JOIN)
 - SQL:1999 (regular expressions, recursive queries, triggers)
 - SQL:2003 (XML support, auto-generated values)
 - SQL:2008 (revisions to cursors and triggers)
 - SQL:2011 (support for temporal databases)
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.



SQL DDL, DML, DCL, TCL

- The SQL **Data Definition Language (DDL)** provides the ability to specify schema includes domains and integrity constraints.
 - e.g., creating tables and declaring foreign keys
- The SQL **Data Manipulation Language (DML)** provides the ability to query data, as well as insert/delete/update tuples.
 - e.g., SELECT statement
- The SQL **Data Control Language (DCL)** provides the ability to grant access privileges to users and to revoke such privileges.
- The SQL **Transaction Control Language (TCL)** provides the ability to control the execution of transactions.
- We will cover **SQL DML** and **DDL**.



Schema for Running Example

- *instructor* (*ID*, *name*, *dept_name*, *salary*)
- *teaches* (*ID*, *course_id*, *sec_id*, *semester*, *year*)
- *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_no*, *time_slot_id*)
- *course* (*course_id*, *title*, *dept_name*, *credits*)

instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
32456	G. J. Kelly	Finance	87000

teaches

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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



Basic Query Structure

- A simple SQL query has the form:

select A_1, A_2, \dots, A_n

from t_1, t_2, \dots, t_m

where P

Not all attributes need be selected

Tables to be joined

A condition over the table join

- A_i represents an attribute
- t_i represents a table
- P is a predicate over the joined tables

- The result of an SQL query is a multi-set relation.



What is t_1, t_2, \dots, t_m ?

- The most basic form of joining two tables t_1 and t_2 is selecting all rows from t_2 and appending them to each row of t_1 .
- You may think of this as the following:
 for each row r_1 in t_1
 for each row r_2 in t_2
 output r_1, r_2
- There are better ways of joining tables than this, so do not think of this as ideal.



The select Clause

- The **select** clause lists the requested attributes in the result of a query.
- Example: find the names of all instructors:

```
select name  
from instructor
```

- In general relation names are case-sensitive but attribute names are not.
 - (Some exceptions, like MySQL running on Windows.)



The select Clause (Cont.)

- The keyword **distinct** eliminates duplicates. The keyword **all** keeps duplicates. In MySQL, the default is **all**.
- Example: find the names of all departments with at least one instructor, and remove duplicates.

```
select distinct dept_name  
from instructor
```

- Example: find the names of all departments with at least one instructor, and keep duplicates.

```
select all dept_name  
from instructor
```



The select Clause (Cont.)

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

select * **from** *instructor*

- The **select** clause can contain arithmetic expressions involving the operators +, −, *, 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 (and renamed “salary/12”).



The where Clause

- The **where** clause specifies conditions that the result must satisfy.
- Example: find all instructors in Physics with *salary* > 80000.
select *name*
from *instructor*
where *dept_name* = 'Physics' **and** *salary* > 80000
- Comparison results can be combined using the logical connectives **and**, **or**, and **not**.
- Comparisons can be applied to results of arithmetic expressions.



String Pattern Matching

- SQL includes a string matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters (wildcards):
 - percent (%): matches any substring
 - underscore (_): matches any one character
 - Example: Find the names of all instructors whose name includes the substring “ar”.

```
select name
from instructor
where name like '%ar%'
```
- Example: Match the string “100 %”.

```
like '100 \%' escape '\'
```
- Note: Patterns are case-sensitive.



The from Clause

The **from** clause lists the relations involved in the query.

- Example: find the Cartesian product *instructor* x *teaches*:

```
select *  
from instructor, teaches
```

- This query generates every possible instructor-teaches pair,
- with all attributes from both relations.

Warning: the result set of a Cartesian product can be very large!



Cartesian Product: *instructor X teaches*

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	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...
...



Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common attribute.
- Example: **select * from *instructor* natural join *teaches***

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010



Natural Join (Cont.)

- **Danger in natural join:** beware of unrelated attributes with same name that get equated incorrectly.
- Example: list the names of instructors along with the titles of courses that they teach.
 - Incorrect solution (makes `course.dept_name = instructor.dept_name`):
select *name, title*
from *instructor* **natural join** *teaches* **natural join** *course*
 - Correct solution:
select *name, title*
from *instructor* **natural join** *teaches, course*
where *teaches.course_id = course.course_id*
 - Another correct solution:
select *name, title*
from (*instructor* **natural join** *teaches*)
inner join *course* **using** (*course_id*)



More Joins

- **Join operations** take two relations and return another relation as a result.
- **Join condition:** defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type:** defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<i>Join types</i>	<i>Join Conditions</i>
inner join left outer join right outer join full outer join	natural on <predicate> using (A_1, A_1, \dots, A_n)



More Joins (Cont.)

■ Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

■ Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that
prereq information is missing for CS-315 and
course information is missing for CS-437



Inner and Outer Joins

select * from course inner join prereq on
course.course_id = prereq.course_id

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>	<i>course_id</i>
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

Note: some databases allow you to omit the keyword “**inner**”.

select * from course left outer join prereq on
course.course_id = prereq.course_id

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>	<i>course_id</i>
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	<i>null</i>	<i>null</i>



Outer Joins

select * from course right outer join prereq using (course_id);

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prere_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	<i>null</i>	<i>null</i>	<i>null</i>	CS-101

select * from course full outer join prereq using (course_id)

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>
CS-347	<i>null</i>	<i>null</i>	<i>null</i>	CS-101

■ Note: full outer join is not supported in MySQL 5.0.



Theta Join

- A Theta join is obtained using an **inner join** with the **on** clause.
- Example: find the course ID, semester, year and title of each course offered in the Physics department.

```
select section.course_id, semester, year, title  
from section inner join course  
on (section.course_id = course.course_id and  
      dept_name = 'Physics')
```



The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:
old-name as new-name
- Example:
select *ID, name, salary/12 as monthly_salary*
from *instructor*
- Find the names of all instructors who have a higher salary than some instructor in 'Physics'.

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

Keyword **as** is optional and may be omitted
instructor as T \equiv *instructor T*

- (Note: keyword **as** must be omitted in Oracle)



Set Operations

- Find courses offered 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 offered in Fall 2009 and in 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 offered 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)
```

Note: MySQL does not support **intersect** or **except**. Remedy?



Set Operations

- Set operations **union**, **intersect**, and **except** automatically eliminate duplicates.
- To retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.
- Suppose a tuple occurs m times in r and n times in s , then, it occurs:
 - $m + n$ times in r **union all** s
 - $\min(m, n)$ times in r **intersect all** s
 - $\max(0, m - n)$ times in r **except all** s



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

Use the keyword **distinct** to eliminate duplicates.

- Example: ... **count(distinct ID)** ...



Aggregate Functions (Cont.)

- Example: find the average salary of instructors in the Physics department.

```
select avg(salary)
from instructor
where dept_name= 'Physics'
```

- Example: find the total number of instructors who teach a course in the Spring 2010 semester.

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

- Example: find the number of tuples in the *course* relation.

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

- Note: avoid spaces between the aggregate function and (



Aggregation with Grouping

- Find the average salary of instructors in each department

```
select dept_name, avg(salary)
from instructor
group by dept_name
```

- Note: departments with no instructor will not appear in result

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

<i>dept_name</i>	<i>avg_salary</i>
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 the **group by** list:
- The following has undefined results:

```
select dept_name, name, avg(salary)
from instructor
group by dept_name
```

Returns a group for each distinct *dept_name* in relation *instructor*.

Note: MySQL 5.5 does accept the above syntax. For each department group it will return one instructor name from that group.

Note: MySQL 5.7 does **not** accept the above syntax.



Aggregation with “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.



Ordering the Display of Tuples

Example: list in alphabetical order the names of all instructors.

```
select distinct name  
from instructor  
order by name
```

- Specify **desc** for descending order or **asc** for ascending order, for each attribute. Ascending order is the default.
 - example: **order by name desc**
- Possible to sort on multiple attributes.
- example:

```
select distinct dept_name, name  
from instructor  
order by dept_name asc, name desc
```



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.
- Common uses of subqueries including performing tests for set membership (e.g., a value is in the set or not) and set cardinality (e.g., set is empty or not), as well as performing set operations (e.g., union, intersect, difference).
- The **in** construct is used for testing set membership. It returns **true** if the value on the left is an element of the relation generated by a subquery on the right.

select ... from ... where value in (subquery)

select ... from ... where value not in (subquery)



Testing for Set Membership

- Example: 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)
```

- Example: 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)
```




Testing for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
 - **exists** $r \Leftrightarrow r \neq \emptyset$
 - **not exists** $r \Leftrightarrow r = \emptyset$
- Example: 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)
```

Correlated subquery: uses values from outer query in **where** clause.



Test for Empty Relations (Cont.)

- Example: find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name  
from student as S  
where not exists ( (select course_id  
                    from course as C  
                    where C.dept_name = 'Biology' and  
                    not exists  
                    (select T.course_id from takes as T  
                    where C.course_id = T.course_id  
                    and S.ID = T.ID) ) )
```



Scalar Subquery

- **Scalar subquery** is one that yields a single value.

- Example:

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

Example:

```
select name  
from instructor  
where salary * 10 >  
      (select budget from department  
       where department.dept_name = instructor.dept_name)
```

- Note: An error occurs if the subquery returns more than one result tuple.



Modifying Relations – Insertion

- Example: Add a new tuple to *course*.
insert into *course*
values ('ECE-656', 'Databases', 'ECE', 0.5)
- Example: Add a new tuple to *course* with *credits* set to *null*.
insert into *course*
values ('ECE-656', 'Databases', 'ECE', *null*)



Modifying Relations – Updates

- Example: Give a 3% salary increase to all instructors whose salary is below \$80,000.

```
update instructor  
  set salary = salary * 1.03  
  where salary < 80000
```



Modifying Relations – Deletion

- Example: Delete all instructors.

delete from *instructor*

Example: Delete all instructors from the Math department.

delete from *instructor*

where *dept_name* = 'Math'

- Example: Delete all instructors whose departments are in the EIT building.

delete from *instructor*

where *dept_name* in (**select** *dept_name*
from *department*
where *building* = 'EIT')



Data Definition Language

The SQL **data-definition language (DDL)** is used to specify of information about relations, including:

- the schema for each relation
- the domain of values associated with each attribute
- integrity constraints
- the set of indexes to be maintained for each relation
- the physical storage structure of each relation
- (e.g., choice of InnoDB vs. MyISAM storage engine in MySQL)



Domain Types in SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*.
- **varchar(*n*)**. Variable length character string, with user-specified maximum length *n*.
- **int**. Integer (a machine-dependent finite subset of the integers).
- **smallint**. Small integer (a machine-dependent subset of int).
- **numeric(*p*,*d*)**. Fixed point number, with user-specified precision of *p* significant digits, with *d* digits to the right of decimal point.
- **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.
- **date, time**. Calendar date (YYYY-MM-DD format), and time of day (hh:mm:ss format).



Create Table Construct

- An SQL relation is defined using the **create table** command:

```
create table  $r$  ( $A_1$   $D_1$ ,  
    ...,  
     $A_n$   $D_n$ ,  
    (integrity-constraint1),  
    ...,  
    (integrity-constraintk)  
);
```

- [Note: r is a relation, A_i is an attribute, and D_i is the domain of A_i]
- Example:

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



Views

- In some cases, it is not desirable for all users to see all the relations stored in a database instance.
- Consider a person who needs to know an instructor's name and department, but not the salary. This person should see a relation described in SQL by the following query:

```
select ID, name, dept_name  
from instructor
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not part of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.



View Definition

- A view is defined using the **create view** statement, which follows the form

create view *view_name* **as** < query expression >

where <query expression> is any legal SQL query.

- Once a view is defined, the view name can be used to refer to the virtual relation.
- A view need not be defined over a single table. It can be defined over the result set of a join or group by query.
- Note: Views are dynamic. In other words, changing the data in the relations referenced by a view causes analogous changes in the virtual relation corresponding to the view.



Example Views

A view of instructors without their salary

```
create view faculty as  
select ID, name, dept_name  
from instructor
```

Find all instructors in the Biology department

```
select name  
from faculty  
where dept_name = 'Biology'
```

Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as  
select dept_name, sum(salary)  
from instructor  
group by dept_name
```

```
select * from departments_total_salary
```



Drop and Alter Table Constructs

- **drop table** *student*
 - Deletes the table and its contents. Assumes table exists.
- **drop table if exists** *student*
 - Deletes the table and its contents if table exists.
- **delete from** *student*
 - Deletes all tuples from table, but retains table.
- **alter table**
 - **alter table** *r* **add** *A D*
 - ▶ Adds attribute with name *A* and domain *D* to relation *r*. All existing tuples in the relation are assigned *null* as the value for the new attribute.
 - **alter table** *r* **drop** *A*
 - ▶ Drops attribute *A* from relation *r*. Not supported by many databases.



Declaring Integrity Constraints

- **not null**: disallows null values
- **primary key** (A_1, \dots, A_n): ensures uniqueness
- **unique** (A_1, \dots, A_n): ensures uniqueness (think superkey)
- **foreign key** (A_m, \dots, A_n) **references** r (A'_m, \dots, A'_n):
- defines a foreign key in the child (referencing) table that points to a referenced key in a parent (referenced) table r
- **default** V : makes V the default value for an attribute
- Example:

```
create table instructor (  
    ID           char(5),  
    name        varchar(20)    not null,  
    dept_name   varchar(20),  
    salary     numeric(8,2)   default 0,  
    primary key (ID),  
    foreign key (dept_name) references department (dept_name)  
);
```



Primary Keys and Superkeys

Example of **primary key** and **unique** constraints in SQL:

```
create table customer (  
    customer_id          int,  
    social_insurance_num  numeric(9,0),  
    first_name           varchar(20),  
    last_name            varchar(20),  
    primary key (customer_id),  
    unique (social_insurance_num)  
);
```

- Note 1: **primary key** and **unique** both identify superkeys.
- You can use **primary key** at most once per table but you can use **unique** more than once.
- Note 2: A primary key attribute cannot be null. The **primary key** constraint on an attribute implies the **not null** constraint.
- Note 3: For a **unique** attribute, the number of tuples that may have a *null* value is system-dependent (zero, one, or many).



Foreign Keys

Example of a **foreign key** (a.k.a. **referential integrity**) constraint:

```
create table instructor (  
    ID   char(5)      primary key,  
    name  varchar(20) not null,  
    dept_name varchar(20),  
    salary numeric(8,2),  
    foreign key (dept_name) references department(dept_name)  
);
```

```
create table department (  
    dept_name varchar(20) primary key,  
    building varchar(20) not null,  
    budget int  
);
```

- Note: In MySQL the referenced key (in this case *dept_name*) must be a superkey of the referenced table, or a prefix of a multi-attribute primary key of the referenced table.



Referential Actions

- **Referential actions** define the behavior of the DB in cases when an update or deletion on the parent (*i.e.*, referenced) table SQL statement affects a value referenced by a child (*i.e.*, referencing) table.
- SQL 92 defines four actions:
 - **cascade**: automatically update or delete foreign key in matching rows of child table.
 - **set null**: set foreign key columns to *null* in matching rows of child table. (Assumes foreign key is nullable.)
 - **set default**: set foreign key columns to the default value in matching rows of child table.
 - **no action**: reject operation and generate error.
 - (Also known as **restrict** in MySQL.)



Referential Actions (Cont.)

Example: cascade on update, set null on delete

```
create table instructor (  
    ID          char(5) primary key,  
    name        varchar(20) not null,  
    dept_name varchar(20),  
    salary     numeric(8,2),  
    foreign key (dept_name) references department(dept_name)  
    on update cascade on delete set null  
);
```



Foreign Key Caveats in MySQL 5.0

- MySQL will enforce foreign key constraints only in some cases:
 - The storage engine for the parent and child tables must support foreign keys (e.g., InnoDB).
 - Corresponding attributes in parent and child table must have “similar internal data types”.
 - (e.g., integers of the same size and sign, or strings of possibly different lengths.)
 - Foreign key and referenced key must be indexed.
- **Note: we will discuss storage engines and indexing later on in the course.**
- For further information consult the MySQL reference manual:
- <http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html>



Check Constraints

In SQL 92, constraints can be expressed using predicates and declared using the **check** clause.

- (Not supported in MySQL 5.7.)

Note: “The CHECK clause is parsed but ignored by all storage engines.”

- Example: ensure that salaries observe institutional limits.

```
create table instructor (  
    ID          char(5)          primary key,  
    name        varchar(20)      not null,  
    dept_name   varchar(20),  
    salary      numeric(8,2),  
    foreign key (dept_name) references department(dept_name),  
    check (salary > 50000 and salary < 150000 )  
);
```



Auto-increment Attributes

- Auto-increment attributes can be used to automatically generate primary key values.
- Example:

```
create table instructor_auto (  
    ID          int          auto increment primary key,  
    name        varchar(20),  
    dept_name   varchar(20),  
    salary     numeric(8,2)  
)
```

```
insert into instructor_auto (name, dept_name, salary)  
values ('Watson', 'Biology', 90210)
```

```
select * from instructor_auto
```

- **Note:** *ID* = 1 should be assigned automatically to the first tuple.



Stored Procedures

- A **stored procedure** is a subroutine that consolidates and centralizes logic that would otherwise be implemented in database applications. Using stored procedures reduces redundant SQL code in applications.
- Example:

```
DELIMITER @@  
CREATE PROCEDURE ProcTopSalary  
(IN dept_name VARCHAR(20))  
BEGIN  
    SELECT max(salary) FROM instructor  
    WHERE instructor.dept_name = dept_name;  
END @@  
DELIMITER;  
CALL ProcTopSalary('Physics');
```

Stored procedures are generally long-lived, similarly to tables. A stored procedure can be deleted using the drop keyword:

```
DROP PROCEDURE ProcTopSalary;
```



Cursors

- A database **cursor** is a control structure that enables traversal of records in a database. Cursors are similar to iterators in Java, and are used inside stored procedures.
- Example: assign every instructor in Biology department to teach 'BIO-101', one instructor per year, consecutive years starting at 2016.
-

```
DELIMITER @@  
CREATE PROCEDURE CursorDemo()  
BEGIN  
    DECLARE year INT DEFAULT 2016;  
    DECLARE my_id VARCHAR(5);  
    DECLARE my_dept_name VARCHAR(20);  
    DECLARE done INT DEFAULT 0;  
    DECLARE cur CURSOR FOR  
        SELECT ID, dept_name from instructor;  
    DECLARE CONTINUE HANDLER FOR  
        NOT FOUND SET done = 1;
```

< continued on next slide >



Cursors (Cont.)

```
OPEN cur;  
my_loop: LOOP  
    FETCH cur INTO my_id, my_dept_name;  
    IF done = 1 THEN  
        LEAVE my_loop;  
    END IF;  
    IF my_dept_name = 'Biology' THEN  
        INSERT INTO teaches  
            VALUES (my_id, 'BIO-101', 1, 'Summer', year);  
    END IF;  
    SET year = year + 1;  
END LOOP my_loop;  
CLOSE cur;  
END @@  
DELIMITER;  
  
CALL CursorDemo();
```




Triggers

- A **trigger** is procedural code that is automatically executed in response to certain events on a particular table or view in a database.
- MySQL supports three types of trigger events: insert, update, and delete. A trigger can be executed either before or after an event.
- Example of using triggers for validation:

```
DELIMITER @@
```

```
CREATE TRIGGER SalaryTrigger
```

```
BEFORE UPDATE ON instructor FOR EACH ROW
```

```
BEGIN
```

```
    IF NEW.salary > OLD.salary * 1.10 THEN
```

```
        SIGNAL SQLSTATE '45000'
```

```
        SET MESSAGE_TEXT = 'increase higher than 10%';
```

```
    END IF;
```

```
END; @@
```

```
DELIMITER;
```

```
UPDATE instructor SET salary = salary * 3.14
```

```
WHERE salary < 80000;
```