Inserting data into a table

The basic insertion of data into a table is done with the INSERT instruction

```
INSERT INTO tablename
(fieldname1, fieldname2, fieldname3, ... )
VALUES (value1, value2, value3, ... )
```

The **values list must match in number and order with the fieldnames list**. In the values list, 'strings' must be quoted and numbers should not be. If the row contains an auto_increment field, leave it out of both lists. If there is a default value for a column which you want to use, leave it out of both lists.

File upload is accomplished by using LOAD DATA LOCAL INFILE where the data is **tab delimited**:

```
LOAD DATA LOCAL INFILE 'filename'
INTO TABLE tablename
(fieldname1, fieldname2, fieldname3, ... )
```

The **fieldnames list must match the order of the data in the file**, not in the order of the fields in the table. If the data is **comma delimited**, indicate this with FIELDS TERMINATED BY ','

```
LOAD DATA LOCAL INFILE 'filename'
INTO TABLE tablename
FIELDS TERMINATED BY ','
(fieldname1, fieldname2, fieldname3, ... )
```

Header lines can be ignored with IGNORE <number> LINES.

```
LOAD DATA LOCAL INFILE 'filename'
INTO TABLE tablename
IGNORE 2 LINES
(fieldname1, fieldname2, fieldname3, ... )
```

Default values can be set for unnamed fields.

```
LOAD DATA LOCAL INFILE 'filename'
INTO TABLE tablename
(fieldname1, fieldname3, ... )
set fieldname2 = 'value'
```

Other options for specifying the input data format are available. See the MySQL documentation.

Retrieving data from a single table

The basic form of SQL query (Structured Query Language) for a single table is:

SELECT field name, field name, ...
FROM table name
WHERE condition [AND|OR condition etc.]

The individual query parts are referred to as clauses:

- the **SELECT** clause lists the fields in the output,
- the **FROM** clause lists the table where the data is stored
- the **WHERE** clause restricts the output to records that meet the stated conditions.

The simplest query returns the whole table:

SELECT *
FROM Professor

Here, * means "all fields." To restrict the fields, use field names:

SELECT pid, lname, fname
FROM Professor

To restrict records, impose a condition

SELECT pid, lname, fname
FROM Professor
WHERE rank = 'Associate'

There are a number of different ways to select strings (like 'Associate') including using equals (=), LIKE, and REGEXP.

String comparison:

- '=' strings must match exactly, except trailing white space is ignored
  usage: expression = pattern
  not case sensitive

- 'LIKE' strings must match exactly, including trailing white space,
  can use wildcards in pattern:
  '%' for zero or more letters
  '_' for one letter
  usage: expression LIKE pattern
  not case sensitive

```
SELECT pid, lname, fname
FROM Professor
WHERE rank like 'Assoc%'
```

- REGEXP regular expression matching (used for advanced substring matching)
  can use regular expression wildcards
    '^' match beginning of a string only
    '$' match end of a string only
    '.' (period) match any character
    .* match zero or more characters (any)
  usage: expression REGEXP pattern

```
SELECT pid, lname, fname
FROM Professor
WHERE lname REGEXP 's'
```

String formatting:

Output can be formatted in various ways:

- concat('a','b') creates an output string from the concatenation of the two input strings
- format('number',length) creates a decimal string with commas separating the thousands, and length values to the right of the decimal point

```
SELECT concat( '$', format(budget,2))
FROM Project
WHERE budget >= 500000
```

- strings can be output as part of the select clause

```
SELECT 'Prof.', lname
FROM Professor
ORDER BY lname ASC
```

Retrieving data from multiple tables

Cross Product

Queries that combine fields from two tables execute a **cross product**:

Professor X Project

This cross product returns a result table which combines **every row** from **Professor** with **every row** from **Project**. Each row of the result contains **one column for each field** from both tables.
For example:

If
Professor has 4 rows and 5 columns
Project has 6 rows and 4 columns
Then
Professor X Project has 24 rows and 9 columns

We can think of a cross product as a nested loop in a program:

For each tuple, **p**, in **Professor** do
For each tuple, **q**, in **Project** do
Add **<p, q>** to the result

In a **SELECT** statement, a cross product is specified by the following notation in the **FROM** clause:

```
SELECT …
FROM Professor, Project
WHERE …
```

Cross products are uncommon because they do not use links between data in the tables.

Joins

Consider the following cross product SELECT statement:

```
SELECT …
FROM Professor p, Project q
WHERE p.pid = q.pid AND q.sponsor = 'NIH'
```

Here, **p** and **q** are abbreviations that allow you to identify which table contains a named field. There are two types of Boolean conditions expressed in the WHERE clause:

- The condition **p.pid = q.pid** is a **join condition.** It involves comparison of related fields, one from each table and *restricts the output of the cross product* to only those rows which meet the condition.

- The condition **q.sponsor = 'NIH'** involves the contents of only a single field and is not involved in linking data in the tables. It limits the rows considered from the Project table

A **join** is an operation on tables which share a foreign key. It returns only those result rows that meet particular Boolean relationships on the foreign key. It is more efficient than a cross product.

JOIN notation provides two alternate ways to write the join condition.

USING:

```
SELECT …
FROM Professor p JOIN Project q USING(pid)
WHERE q.sponsor = 'NIH'
```

used only when the Boolean condition is **equality** and the **field names are identical**, or

ON:

```
SELECT …
FROM Professor p JOIN Project q ON p.pid = q.pid
WHERE q.sponsor = 'NIH'
```

used for all Boolean operators and any combination of field names.

Types of Joins and their output

- **Condition Join**
  Used for any Boolean condition
  Employs the "ON" notation
  Result contains the same **columns** as a cross product, that is, all columns from both tables.

```
SELECT *
FROM Professor p JOIN Project q ON p.pid > q.pid  (doesn't make
                                                   sense, but is a
                                                   Boolean
                                                   condition)
WHERE q.sponsor = 'NIH'
```

Number of columns is 9, same as cross product.

- **Equijoin**

  Has only equality conditions between identical field names
  Employs the "USING" notation
  Result **eliminates one duplicate column** for each field name in the USING list

```
SELECT *
FROM Professor p JOIN Project q USING(pid)
WHERE q.sponsor = 'NIH'
```

  Number of columns is 8 with only one column named pid

- **Natural Join**

  Tests for equality on ALL identical field names
  Has NO qualifying phrase like "ON" or "USING"
  and doesn't name the fields
  Employs the keyword "NATURAL"
  Result **eliminates one duplicate column** for each pair of identical field names
  If no pair of identical field names exists, the result is a cross product

```
SELECT *
FROM Professor p NATURAL JOIN Project q
WHERE q.sponsor = 'NIH'
```

  Number of columns is 8 with only one column named pid

Writing Select Statements for Single and Multiple Tables

Use the following database schema for multiple table queries:

Prof (pid, lname, fname, rank, research)
Dept (did, dname, address, *chair*) (Note *chair* refers to Prof.pid)
Appt (*pid, did*, percentage)
Proj (prid, sponsor, budget, *pid*) (Note *pid* means PI on Proj)
CoPI (*prid, pid*)

Table names are in bold. Primary keys are underlined. *Foreign keys* are shown in italics and match the same field name in another table unless specified.

Professor Table Instance

pid	Lname	Fname	Rank	Research
101	Benson	Gary	Associate	Repetitive DNA patterns
102	Kasif	Simon	Full	Functional annotation
103	Segre	Daniel	Associate	Metabolic flux analysis
104	Korolev	Kirill	Assistant	Evolutionary dynamics

Department Table Instance

did	Dname	Address	Chair (pid professor)
1	Biology	5 Cummington	87
2	Bioinformatics	24 Cummington	38
3	Biomedical Engineering	44 Cummington	231
4	Computer Science	111 Cummington	57
5	Physics	590 Comm. Ave	24

Appointment Table Instance

pid (professor)	did (department)	percentage
101	2	50
101	4	50
102	3	100
103	1	50
103	3	50
104	2	25
104	5	75

Project Table Instance

prid	pid	sponsor	budget
801	101	NSF	250000
802	103	NIH	750000
803	102	NIH	35000
804	104	Wellcome Trust	150000
805	101	IGERT	1000000

CoPI Table Instance

prid (project)	pid (professor)
801	103
802	104
803	101
801	104

Queries

Single Table

1. List last names of all professors in alphabetic order (lname). Again, but with no duplicates.

   Table involved:
   Query:

2. List names of all professors who work on DNA (fname, lname).

   Table involved:
   Query:

Multiple Table

3. For each project sponsored by the NIH list the PI (prid, pid, lname, fname).

   Tables involved:
   Query:

4. For each professor in the computer science and biology departments, list the professor name, department name, and appointment percentage (lname, fname, dname, percentage).

Tables involved:
Query:

5. List the project budgets over $100,000 for each department broken down as follows: department name, professor id, and budget (dname, pid, dept budget). A grant budget goes to a department if the PI is appointed in the department. The department share of the budget is calculated as budget* percentage/100.

Tables involved:
Query:

6. List all professors who are CoPIs on a project (lname, fname, prid, sponsor).

Tables involved:
Query:

7. List all professors who are PIs on at least two grants (lname, fname).

Tables involved:
Query:

8. What indexes should be available for these queries?