# Databases

DR. YIHSIANG LIOW (FEBRUARY 2, 2020)

## Contents

File: intro.tex

# 1 Introduction

A relational database management system RDBMS is a software for managing a relational database.

A relational database is made up of tables which is made up of rows of data. You can think of it like a spreadsheet except that RDBMS is a lot more powerful and more complicated.

Briefly, in this set of notes I will talk about

1. RDBMS and SQL
2. DB design
3. Programmatic access to a RDMBS
4. Web programming
5. ORM
6. RDBMS implementation

(Maybe there will be other topics if I have time to write them up.)

It's better to learn RDBMS by trying out the software. In order not to get bogged down with details, we'll first go over some examples, using only the simplest commands, i.e., the smallest and most useful subset of the SQL language.

After you have a general overview of SQL language, then I'll talk about details on SQL. You also need to know how to access the RDBMS with a program written such as in Python, C/C++, Java, PHP, etc. I'll also talk about DB design issues as well as how this related to object-oriented software design.

When you're done using the RDBMS, including how to include it in a complete object-oriented software construction, you will need to know the mathematics, design, and construction of the RDBMS (data structures and algorithms behind the DB.) This will give you a better understand of how to optimize your database.

I will be using MySQL (version 5), a very popular RDBMS. (There are many others: Oracle, MS SQL, DB2, etc.)

You should download a copy of the MySQL reference manual and keep in on your laptop at all times. It's free.

File: getting-started.tex

# 2 Getting Started

I assume that you have a Fedora machine.

**STEP 1.** To install MySQL do this (in your bash shell after logging in as root):

```
yum install -y mysql mysql-server
```

**STEP 2.** The next thing to do is to enable and run MySQL service (it will run in the background whenever the system boots/reboots.)

FOR FEDORA 18: Instead of the above, do the following:

```
/sbin/chkconfig mysqld on
service mysqld start
```

FOR FEDORA 22: Instead of the above, do the following

```
dnf install -y mariadb mariadb-server
systemctl enable mariadb.service
systemctl start mariadb
```

FOR FEDORA 25: Instead of the above, do the following

```
dnf install -y mariadb mariadb-server --allowerasing
systemctl enable mariadb.service
systemctl start mariadb
```

(The second command above does not work, try this `systemctl enable mariadb`.)

FOR MACOS: First check if if you have homebrew installed:

```
brew list
```

If get an error, run the following command to install homebrew.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

(the above is one command on one line.) Now, run the following command to install MySQL

```
brew install mysql
```

Now, run the following command to start the MySQL server

```
brew services start mysql
```

**STEP 3.**

Fedora 18: The next thing to do is to change the MySQL root user password:

```
mysqladmin -u root password 'root'
```

The new password is within quotes. (If the above does not work, try `/usr/bin/mysqladmin` instead of `mysqladmin`.)

Fedora 25: The above is not necessary. By default, there's no password for `root` user in MySQL. It's a good idea to change the password to `root` – see later section.

**STEP 4.**

Now to test run the MySQL shell.

Fedora 18: Do one of the following. (You do not need to login as root in your linux. You can be logged in as student in your linux and access your MySQL as root.)

```
mysql -u root
```

or

```
mysql -u root -p
```

or

```
mysql --user=root -p
```

or

```
mysql --user=root --password=root
```

Fedora 25: Use the first command unless if you somehow changed the root password.

```
mysql -u root
```

You'll get a prompt:

```
mysql>
```

or

```
MariaDB [(none)]>
```

You are now ready to enter MySQL commands at the prompt. To exit, either do `ctrl-d` or type `quit`.

That's it. You're now ready to learn SQL.

ASIDE. If for some reason you do not want MySQL to run in the background anymore, you do this:

```
service mysqld stop
```

Note that if you restart the system, MySQL will be running again. If you want to prevent MySQL from running when the system starts, you do

```
chkconfig mysqld off
```

For Fedora 25, do

```
systemctl stop mariadb
systemctl disable mariadb
```

This stop the mariadb server and stops it from running when you restart your F25.

**Exercise 2.1.**

- Find MySQL's website.
- Find and download MySQL's (free!) reference manual.

File: reset-password.tex

# 3 Reset Password

You can skip this section.

In the bash shell, as root, stop the mysql/maria service, and then execute

```
mysqld_safe --skip-grant-tables &
mysql -u root
```

In the mysql shell do

```
use mysql
update user set Password=PASSWORD("ANEWPASSWORD") where User='root';
flush privileges;
```

where `"ANEWPASSWORD"` is changed to your favorite password and exit your mysql/mariadb shell.

In bash shell, stop and start your mysqldb/mariadb (see previous section if you forgot how to do it.)

**Exercise 3.1.** Test the new password.

**Exercise 3.2.** When you run `mysqld_safe` as in the above, it's possible for people to login to the database system without entering passwords. Check the MySQL manual on how to prevent a remote user from getting into MySQL through the network while you're in the middle of changing your root password.

File: db.tex

# 4 Database Operations

To list all databases in your MySQL:

```
mysql> show databases;
```

By the way the keywords are not case sensitive. So you can do this:

```
mysql> SHOW DATABASES;
```

Here's how to create a database:

```
mysql> create database test;
```

To delete a database do this:

```
mysql> drop database test;
```

**Exercise.** Recreate the test database and go on.

To go "into" the test database do this:

```
mysql> use test;
```

At this point, within the `test` database, you can add, delete, modify tables and add, delete, modify data within the tables.

File: script.tex

# 5 Script

You don't want to type MySQL commands into the console window, except possibly for the simplest experiments you want to do. If you made a mistake it would be painful to correct. So save your commands, say

```
drop database test;
create database test;
```

into a text file, say `x.sql`, and then run it this way from your bash terminal shell:

```
mysql --user=root --password=root < x.sql
```

(Leave out the password option if your left the root password empty.) If you made a mistake you can correct `x.sql` and run it again.

If you like, you can save the output to another text file, say `output.txt` like this:

```
mysql --user=root --password=root < x.sql > output.txt
```

This is helpful if the output is long.

File: sql.tex

# 6 SQL

SQL stands for structured query language. It's the language for managing a relational database. In the next section, we will begin with the command to create database tables.

This is a 4th generation declarative language (4GL). (Never heard of declarative? 4GL? Google!)

We'll be using MySQL which has its extension of SQL. So be aware that some keywords are specific to MySQL. Also, not all SQL keywords are implemented in MySQL. In general, the query language of almost all RDBMS differ from the standard SQL. (The current standard is ANSI SQL 2011 ISO/IEC 9075:2011.) Most RDBMS also contains commands to administer the database (example: creating users and setting permissions for different users). However once you know a vendor version of SQL, you can easily learn another.

File: create-table.tex

# 7 Create Table

Now you can create tables in test:

```
create table Person (
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT
);
```

This means that each row of data in this table has three values. The name of these three pieces of data (for each row) is `fname`, `lname`, and `fingers`. Their types are `VARCHAR(100)`, `VARCHAR(100)`, and `INT`.

`VARCHAR(100)` means "string of length at most 100". Of course `INT` means integer. Specifically `INT` means "32-bit signed integer". There are many other numeric types.

There are many other types (example: `DATE`, `DATETIME`, ...)

You can view the structure of your `Person` table like this:

```
describe Person;
```

You can delete the table:

```
drop table Person;
```

**Exercise.** Recreate the `Person` table and go on.

**Exercise.** Use google to find out how to delete a field (or column) to a table. Delete the `fingers` field from `Person`.

**Exercise.** Use google to find out how to add a field to a table. Add the `fingers` field back to `Person`.

**Exercise.** Use google to find out how to modify a field in a table. Change the type of `fname` from `VARCHAR(100)` to `VARCHAR(200)`.

**Exercise.** Use google to find out the difference between `VARCHAR(100)`, `CHAR(100)`, and `TEXT`.

File: adding-rows.tex

# 8 Adding Rows

You can add rows to the `Person` table:

```
insert Person set fname='John', lname='Doe', fingers=10;
```

This is the same as

```
insert into Person set fname='John', lname='Doe', fingers=10;
```

or this:

```
insert Person (fname, lname, fingers)
values ('John', 'Doe', 10);
```

You can use either single or double quotes as string delimiters.

You can also specify values for only certain fields:

```
insert Person set fname='Tom', lname='Smith';
```

Go ahead and insert a few more rows.

The second syntax

```
insert Person (fname, lname, fingers)
values ('John', 'Doe', 10);
```

allows you insert multiple rows:

```
insert Person (fname, lname, fingers)
values ('John', 'Doe', 10),
       ('Tom', 'Smith', 9);
```

File: query.tex

# 9 Query

Next, list the contents of the `Person` table like this:

```
select * from Person;
```

and you will see this

```
+-------+-------+---------+
| fname | lname | fingers |
+-------+-------+---------+
| John  | Doe   |      10 |
| Tom   | Smith |    NULL |
+-------+-------+---------+
```

(If you ran the command through a script then the boundary lines are not printed.)

You can also choose what columns to display. Try this:

```
select fname from Person;
```

and this:

```
select lname, fingers from Person;
```

The SQL statement

```
select lname, fingers from Person;
```

is the same as

```
select Person.lname, Person.fingers from Person;
```

At this point we are only working on one table. So there's no need to specify `Person.` to indicate where the column comes from. Later, we'll be working with multiple tables. In that situation, you might have two tables with common column names. In that case, you will need to specify the table name when you refer to a column.

You can also filter:

```
select * from Person where fingers >= 10;
```

including boolean expressions with `AND` and `OR`:

```
select * from Person
where fname = 'John' and (fingers <= 5 or fingers >=10);
```

You can sort too:

```
select * from Person group order by fname;
```

Try this too:

```
select * from Person group order by lname, fname;
```

Of course you can combine filtering and sorting:

```
select lname, fname from Person
where fname='John' and (fingers <= 5 or fingers >=10)
group order by fname desc;
```

where `desc` is order by descending order. `asc` (ascending) is the default.

**Exercise 9.1.** The previous section shows you how to add one row at a time. Find out how to add 3 rows of data in one command. Verify with select.

**Exercise 9.2.** Try this

```
select * from Person where fingers in (0, 5, 10);
```

and this

```
select * from Person where fingers not in (0, 5, 10);
```

Add some rows and test the above queries. It should exhaust too many grey cells to figure out what they do.

**Exercise 9.3.** Find out about `LIMIT` and `OFFSET` and try them out. Make sure you know them well.

**Exercise 9.4.** You are trying to tidy up your comic book library which is in a complete mess and your friends keep borrowing them. Create a table ComicBook with columns title, issue, borrower.Populate it with the following:

```
Batman: Two Faces             1
Batman: Unseen                1
Batman: Unseen                3
Batman: Unseen                4   David
Batman Villains Secret Files  1
Batman: Widening Gyre         1   Brandy
Batman: Widening Gyre         2
Batman/Wildcat                1   Sam
Battle Classics               1
Shadow of the Batman          4
Catwoman: When in Rome        3
Beast Boy                     2   Brandy
Captain Storm                 9
Captain Storm                 13  Ujjwal
Captain Storm                 14
Captain Storm                 16
The Legend of Wonder Woman    2
```

Put empty strings where the book in the appropriate column if the book is not out on loan. Run the following SQL statements:

```
select * from ComicBook where title like 'Bat'
select * from ComicBook where title like 'Bat%'
select * from ComicBook where title like '%at'
select * from ComicBook where title like '%at%'
```

Check the web of the MySQL reference for the keyword LIKE and the %.

- Write an SQL statement that lists all rows in the table.
- Write an SQL statement that lists comics which are on loan.
- Write an SQL statement that lists all Batman comics.
- Write an SQL statement that lists the first 5 Batman comics; write a second SQL statement that lists the next 5 (or less) Batman comics.
- List all Batman comics which are out on loan.
- List all Batman comics which are not out on loan.
- To list the borrowers, we can do

  ```
  select borrower from ComicBook
  ```

  However there are repeats. Using the web of MySQL reference manual, look for the SELECT syntax and find a way to select without printing repeating rows. (ANSWER: select distinct borrower from ComicBook)

File: default-value.tex

# 10 Default Values

Notice that the `Person` table from the previous section looks like this:

```
+-------+-------+---------+
| fname | lname | fingers |
+-------+-------+---------+
| John  | Doe   |      10 |
| Tom   | Smith |    NULL |
+-------+-------+---------+
```

The `fingers` value of the second row is `NULL`. The reason is that when we inserted the data for the second row we did not specify the value for `fingers`.

Get rid of the `Person` table and recreate it with this:

```
create table Person (
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT default 10
);
```

and redo the previous two inserts. Perform a select and you'll see that Tom Smith has 10 fingers even if the insert does not specify a value.

File: deleting-rows.tex

# 11 Deleting Rows

Try this:

```
delete from Person where fname = 'John';
```

**Exercise.** Suppose Person table has 5 rows with `fname` of `'John'`. Is it possible to delete just one of the 5 rows?

**Exercise.** How would you delete all rows from a table in one command?

File: update.tex

# 12 Update Rows

Changing the value(s) in a row is easy. Do this:

```
update Person set fname='George' where fname='Tom';
```

**Exercise.** Perform a single update on two fields of the same row.

**Exercise.** Perform a single update to a column of multiple rows. [Example: Create 3 rows with the same `fname` value and then change the `fname` value.]

File: data-types.tex

# 13 Data Types

Recall that when you create a table you have to describe the type of data for the columns. You have seen some of them. Here they are together with some more:

First here are the basic numeric types for integers. For the following, the `[M]` refers to the maximum *display* length. This is a common misconception: I know some people thought `[M]` refers to the maximum length of the integer value for the column. So for instance the `10` in `INT[10]` has nothing to do with the range of values you can store in the column. Note that the `[10]` is optional – it's only for display purposes.

```
BOOL or BOOLEAN    TRUE, FALSE (This is actually TINYINT[1])
TINYINT            -128, ..., 127
SMALLINT           -32768, ..., 32767
MEDIUMINT          -8388608, ..., 8388607
INT                -2147483648 to 2147483647
BIGINT             -9223372036854775808 to 9223372036854775807
```

There are also unsigned versions:

```
TINYINT UNSIGNED   0, ..., 255
SMALLINT UNSIGNED  0, ..., 65535
MEDIUMINT UNSIGNED 0, ..., 16777215
INT UNSIGNED       0, ..., 4294967295
BIGINT UNSIGNED    0, ..., 18446744073709551615
```

**Exercise 13.1.** For a column with data type of `SMALLINT`, what happens when you insert a value of 1000000000?

**Exercise 13.2.** For a column with data type of `INT UNSIGNED`, what happens when you insert a value of -1?

The following are numeric types that can represent numbers with fractional parts. The `FLOAT` and `DOUBLE` is what you've seen in C/C++. For `FLOAT` and `DOUBLE`, the `(M,D)` refers to a display of maximum `M` digits with `D` digits for the fractional part. As you know, operations on `FLOAT` and `DOUBLE` might have rounding errors. One numeric type that you might not be familiar with is the `DECIMAL`. For the `DECIMAL`, the `(M,D)` does refer to both the display and the maximum storage of the value. For instance `DECIMAL(5,2)` means that you can have at most 5 digits 2 of which are for the fractional part. For `DECIMAL`, if you don't

specify (M,D), then by default M = 65 and D = 0. Operations on DECIMAL is exact as long as there is no overflow or underflow.

```
FLOAT              IEEE754 Single precision floating point numbers
DOUBLE             IEEE754 Double precision floating point numbers
DECIMAL(M,D)       Exact representation of a number with M digits with
                   D digits for the fractional part
NUMERIC            Same as DECIMAL
```

**Exercise 13.3.** What happens when you use FLOAT(2,5) (i.e., the M is too small)? What about FLOAT(2,5) and DECIMAL(2,5)?

**Exercise 13.4.** For a column with data type of DECIMAL(2,1), what happens when you insert a value of 123.456?

For string data types the most basic ones are the following. The last section below are for binary data stored as arrays of bytes. It seems to be similar to array of characters. In what way are they different? The main difference that you *must* be aware of is that strings (such as first names, last names, addresses, etc.) are frequently viewed with some kind of ordering. For instance if you are looking at a phone list, the list is probably listed by lastname,firstname. Depending on the character set you're presenting, the ordering might be different for different languages. Another thing to note is that for some character sets, certain characters require more than one byte for storage. The (M) refers to the maximum number of characters for the string.

```
CHAR(M)          M <= 255
VARCHAR(M)       M <= 65,535
```

You see that even among string data types, there are quite a few. So what are the differences and why are there so many?

The different string types represent different implementations in order to maximum I/O performance and effective use of storage in different ways. The main difference between CHAR and VARCHAR is that CHAR is the faster in terms of performance. However CHAR *might* waste storage space. For more details, you should refer to the MySQL reference manual or use the web. If you know that a column stores strings which are always, say around 100 characters, then you should use CHAR(100). Suppose you have a lastname column and you want to accommodate really long lastnames such as:

```
Wolfeschlegelsteinhausenbergerdorffvoralternwarengewissenhaf
tschaferswesenchafewarenwholgepflegeundsorgfaltigkeitbeschut
zenvonangereifenduchihrraubgiriigfeindewelchevorralternzwolf
tausendjahresvorandieerscheinenbanderersteerdeemmeshedrraums
chiffgebrauchlichtalsseinursprungvonkraftgestartseinlangefah
rthinzwischensternartigraumaufdersuchenachdiesternwelshegeha
btbewohnbarplanetenkreisedrehensichundwohinderneurassevanver
standigmenshlichkeittkonntevortpflanzenundsicherfreunanleben
slamdlichfreudeundruhemitnichteinfurchtvorangreifenvonandere
rintlligentgeschopfsvonhinzwischensternartigraum
```

which has a length of 597 ... no kidding ... see http://everything2.com/index.pl?node_id=1534419. It was so long that this person shortened his lastname to

```
Wolfeschlegelsteinhausenbergerdorff
```

of length 35. But you know that most lastnames are not that long. You might want to use `VARCHAR(50)`.

Next, let's talk about are the binary data type, i.e., `TEXT` and `BLOB` types. They are slower than `CHAR` and `VARCHAR`.

```
TINYTEXT, TINYBLOB      <= 255 B
MEDIUMTEXT, MEDIUMBLOB  <= 65,535 B        (64KB)
TEXT, BLOB              <= 16,777,215 B    (16MB)
LONGTEXT, LONGBLOB      <= 4,294,967,295 B ( 4GB)
```

The `BLOB` versions are used to stored binary data (image files, video files, etc.) while the `TEXT` versions are used to store (huge) textual data.

Finally, here are the data and time data types.

```
DATE      '1000-01-01', ..., '9999-12-31' (in YYYY-MM-DD format)
TIME      '-838:59:59', ..., '838:59:59' (in HHH:MM:SS or HH:MM:SS format)
DATETIME  '1000-01-01 00:00:00', ..., '9999-12-31 23:59:59'
TIMESTAMP '1970-01-01 00:00:01', ..., '2038-01-19 03:14:07'
YEAR(4)   1901, ..., 2155 or '1991', ..., '2155'
```

By the way, the reason `TIME` is allowed to have such a huge hour value is because it can be used to denote not just time of the day but also time interval. Instead of using `TIMESTAMP`, you should use `DATETIME` (do you see why?)

There are two other data types (which are not in the SQL ISO standard): `ENUM` and `SET`. Let me show you examples.

```
create table customer (
    fname varchar(100),
    lname varchar(100),
    gender enum('m','f')
);

insert into customer (fname, lname, gender) values ('john', 'doe', 'm');
insert customer (fname, lname, gender) values ('mary', 'smith', 'f');
insert customer (fname, lname, gender) values ('yoda', 'master', 'huh?');

select * from customer;
select * from customer where gender = 'f';
```

Get it?

Here's an example for SET:

```
create table applicant (
    fname varchar(100),
    lname varchar(100),
    skills set('c++', 'java', 'python', 'telepathy')
);

insert into applicant (fname, lname, skills)
values ('john', 'doe', 'c++,java');

insert into applicant (fname, lname, skills)
values ('tom', 'doe', 'c++, java'); -- note space after comma
                                       in last value --

insert applicant (fname, lname, skills)
values ('mary', 'smith', 'java,c++');

insert applicant (fname, lname, skills)
values ('yoda', 'master', 'c++,python,telepathy');

select * from applicant;
select * from applicant where skills = 'c++,java';
select * from applicant where skills = 'java,c++';
select * from applicant where skills = 'python';
select * from applicant where skills = 'c++,python';
select * from applicant where find_in_set('python', skills) > 0;
```

**Exercise 13.5.** What's the point of each SQL statement in the above SQL program?

**Exercise 13.6.** What's the main difference between `ENUM` and `SET`?

**Exercise 13.7.** Using the `applicant` table, write an SQL statement that prints all the applicants with C++ but not Java skill.

You can easily get the same effect using a separate table. Note that an `ENUM` can have at most 65535 values. It's even worse for `SET`: you can have at most 16 values.

Because `ENUM` and `SET` is not in the SQL ISO standard and their very strict size constraint, you have to be careful about using them.

File: references.tex

# 14 References

If you do this:

```
create table Person (
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT,
email VARCHAR(100)
);

insert Person set fname='John', lname='Doe', email='jdoe@gmail.com';
```

then each row in Person can have only one email. To allow each person to have more than one email you can do this:

```
create table Person (
id INT,
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT
);

create table Email (
Person_id INT,
email VARCHAR(100)
);

insert Person set id=42, fname='John', lname='Doe';
insert Email set Person_id=42, email='jdoe@gmail.com';
insert Email set Person_id=42, email='jdoe@yahoo.com';
```

Now do this:

```
select * from Person, Email
where Email.Person_id = Person.id;
```

and this:

```
select fname,lname,email from Person, Email
where Email.Person_id = Person.id;
```

and this:

```
select Person.fname, Person.lname, Email.email from Person, Email
where Email.Person_id = Person.id;
```

Of course a row in `Email` refers to a row in `Person` through the condition

```
        Email.Person_id = Person.id.
```

File: primary-key-foreign-key.tex

# 15 Primary Key and Foreign Key

In the previous section, we make the rows of a table refer to another. We had this:

```
create table Person (
id INT,
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT
);

create table Email (
Person_id INT,
email VARCHAR(100)
);
```

If we're not careful we might enter some garbage:

```
insert Person set id=42, fname='John', lname='Doe';
insert Email set Person_id=9999, email='jdoe@gmail.com';
insert Email set Person_id=9999, email='jdoe@yahoo.com';
```

You will see that

```
select * from Person, Email
where Email.Person_id = Person.id;
```

gives you nothing.

Also, the process of joining up the rows from two tables would be extremely slow.

We can make sure that the value of `Email.Person_id` is valid, i.e., is a `Person.id` value.

```
create table Person (
id INT,
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT,
primary key (id)
) engine=innodb;
```

```
create table Email (
Person_id INT,
email VARCHAR(100),
foreign key (Person_id) references Person(id)
) engine=innodb;
```

Now try:

```
insert Person set id=42, fname='John', lname='Doe';
insert Email set Person_id=9999, email='jdoe@gmail.com';
```

You will get an error.

The constraint

```
primary key (id)
```

means that there is at most one row in `Person` for a given `id` value. Also, given a value, MySQL can find very quickly the row with the `id` value matching the given value. That's because an indexing structure is built for the `id` column of this table. The primary key value of a row cannot be `NULL`.

The constraint

```
foreign key (Person_id) references Person(id)
```

Says that `Email.Person_id` must have a value in the `Person.id` column. There is however one exception. You can inserting a row in `Email` without specifying a `Person_id` value. Try this:

```
insert Email set email='jdoe@gmail.com';
select * from Email;
```

You will see that the `Person_id` is `NULL`.

In summary, with the above tables, either `Person_id` is a value in `Person.id` or `Person_id` is `NULL`.

By the way we say that `Person` is a parent and `Email` is a child since `Email` referenced `Person`. The row with `email='jdoe@gmail.com'`, since it does not refer to a row in `Person`, is said to be an orphan child.

If you want to use `foreign key ... references ...`, the field referenced in the parent table must be a primary field.

**Exercise.** You can also have multiple fields referring to multiple fields. Check how this is done and try it out yourself.

[ADDED 2019/01/23: Frequently if a primary key is an integer and does not correspond to a real world concept, i.e., the purpose is just to be an primary key for referencing and users never see the value of the primary key, then you can get your RDBMS to auto–generate them. For MySQL you can get the system to produce integer values that increments by 1 each time you do an insert:

```
create table Person (
id INT AUTO_INCREMENT,
...
primary key (id)
) engine=innodb;
```

The first auto-generated value for `id` is 1. Instead of starting with 1 as the first integer, you can also tell MySQL to start with a different integer. You just execute (for instance):

```
alter table Person AUTO_INCREMENT=42;
```

In that case, the next auto-generated integer value is 42. If the new integer value for `id` is already used, MySQL will try a new one that is one larger, etc. until it finds one that is not in use. You can manually insert a value for `id` if you wish instead of using the auto-generated value.

Another type of auto-generated id values are GUID (Global Unique Identifier) or UUID (Unique Universal Identifier). The range of representation of GUID is extremely huge: there are $3.4 \times 10^{38}$ possible GUIDs. For comparison, there are $\approx 5.6 \times 10^{21}$ grains of sand on earth. Therefore each grain of sand can hold $\approx 60,000,000,000,000,000$ GUIDs. The chance of two auto-generated GUID being the same is almost zero. So while an auto-increment primary key integer values in a table is meant uniquely identify a row *within a table*, an auto-generated GUIDs are meant to distinguish between any two things in the world. Some people also hash the GUID, for instance using MD5.

There's a lot of debate on the pros and cons of the above options. For instance using sequentially generated integers as primary key value and the values are exposed to users, then sequential timing information is leaked and that can be a security issue. Google to find out more.

For our class, assume that hidden primary keys should be auto-generated sequential integers and for the case of MySQL, this means using `auto_increment`.]

File: not-null.tex

# 16 NOT NULL Constraint

In the previous section, we have a child table referencing parent table (the foreign key column of the child refers to the primary key column of the parent):

```
create table Person (
id INT,
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT,
primary key (id)
) engine=innodb;

create table Email (
Person_id INT,
email VARCHAR(100),
foreign key (Person_id) references Person(id)
) engine=innodb;
```

We allow a row in the child table to have the foreign key value of `NULL`, i.e., we allow orphans.

If we prefer *not* to allow orphans, i.e., if we do not want `Person_id` to be `NULL`, we add another constraint to the `Person_id` of the `Email` table:

```
create table Email (
Person_id INT not null,
email VARCHAR(100),
foreign key (Person_id) references Person(id)
) engine=innodb;
```

**Exercise.** Attempt to add a row in `Email` by setting the value of `Person_id` to `NULL` (or by omitting to specify a value for this field) and check that MySQL will catch this violation of the constraint.

File: phpmyadmin.tex

# 17 MySQL Tool: phpMyAdmin

There are many GUI tools for MySQL. One of the most popular is probably phpmyadmin.

Since phpMyAdmin is a web-based application, you will need to have a web server running. To install and run Apache/httpd do the following as root in your bash shell:

```
dnf -y install httpd
/sbin/chkconfig httpd on
apachectl start
```

To check that httpd is running, open a web browser and go to http://localhost. You should see a test web page.

To install phpMyAdmin do this as root in your bash shell:

```
dnf install -y phpmyadmin
```

At this point you should be able to use phpMyAdmin. Note that phpmyadmin will require a non-null password. So if you want to use phpmyadmin, you would need to change your root password (in MySQL) to non-null. (See previous sections on how to change the root password.)

To run phpMyAdmin, open a web browser and go to http://localhost/phpmyadmin. To login, you should use a user and password from MySQL.

NOTES. The above should work for most cases. Depending one the version of linux, apache, phpmyadmin, something might fail. Read on ...

- If your browser flags the following error (or something similar to it)

  > *#2002 - The server is not responding (or local MySQL server's socket is not correctly configured).*

  you can allow httpd to connect to the MySQL server by executing the following command as root:

  ```
  setsebool -P httpd_can_network_connect_db on
  ```

  It takes some time for the command to complete.

- If you get the following error while running phpmyadmin in the browser:

```
mysqli_real_connect(): (HY000/2002): No such file or directory
```

then, as root, you should add this line

```
$cfg['Servers'][$i]['host'] = '127.0.0.1';
```

near the top of the file `/usr/share/phpMyAdmin/setup/frames/config.inc.php`. Your linux might have this file in a different location. So if you can't find the above file, do this to search for the file

```
find / -name 'config.inc.php' -print 2>/dev/null
```

**Exercise 17.1.** Use phpMyAdmin to create a database, add some tables to it, add some rows to the tables, delete some tables, delete the database.

References:

- http://www.linux-answered.com/2012/04/23/centos-6-2-linux-apache-php-and-mysql-server-installation-guide/
- http://tag1consulting.com/blog/stop-disabling-selinux
- http://www.tela-group.com/configure-fedora-web-server

File: joins.tex

# 18 Joins

The 4 types of joins:

- inner join
- left (outer) join
- right (outer) join
- full (outer) join

Since joining tables requires comparing values, it's important to know that the NULL value when compared with any value will always give you FALSE. Even NULL = NULL gives you FALSE.

Suppose you have two tables `L` and `R` (memory aid: `L` = left table, `R` = right table). Suppose `L` has a column called `x` and `R` has a column called `y`. A join `J` is a table with columns taken from `L` and `R`. When you specify a join to create a table, you have to say how to match rows in `L` with rows in `R`. Let's say we're matching the values in `L.x` with `R.y`. The condition can be `L.x = L.y`, i.e., equality comparison. It can also be `L.x < L.y`. Etc.

In the examples below I will use the following tables.

```
mysql> select * from L;
+------+------+
| foo  | x    |
+------+------+
| a    |    1 |
| g    |    3 |
| h    |    4 |
| z    |    7 |
+------+------+
4 rows in set (0.01 sec)
```

and here's table `R`:

```
mysql> select * from R;
+------+------+
| y    | bar  |
+------+------+
|    1 | P    |
|    2 | W    |
|    6 | M    |
|    7 | C    |
|    7 | K    |
+------+------+
5 rows in set (0.00 sec)
```

**Exercise 18.1.** Create tables `L` and `R` in the test database.

The rows in `J` can be formed in several ways:

CASE 1. Only when the join condition is TRUE. This is called an **inner join**. The result of joining the above `L` and `R` gives us

```
mysql> select * from L join R on L.x = R.y;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| z    |    7 |    7 | C    |
| z    |    7 |    7 | K    |
+------+------+------+------+
3 rows in set (0.00 sec)
```

You don't really need the JOIN syntax since you get the same result with the SELECT-WHERE syntax:

```
mysql> select * from L, R where L.x = R.y;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| z    |    7 |    7 | C    |
| z    |    7 |    7 | K    |
+------+------+------+------+
3 rows in set (0.00 sec)
```

CASE 2. The **left join** is where *every* row in the *left* table will appear. Here's how you should remember the construction of this table: For each row in `L`, match with rows in `R` (this is like the inner join). *In addition*, if you cannot find a match this left row, then force it to match a dummy right row where all the `R` column values are NULL.

```
mysql> select * from L left join R on L.x = R.y;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| g    |    3 | NULL | NULL |
| h    |    4 | NULL | NULL |
| z    |    7 |    7 | C    |
| z    |    7 |    7 | K    |
+------+------+------+------+
```

CASE 3. If you understand the left join, then the **right join** is analagous. In other words for each row in `R`, you match all rows in `L`. However if there's no match, you force a match with a dummy row where all `L` column values are NULL.

```
mysql> select * from L right join R on L.x = R.y;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| NULL | NULL |    2 | W    |
| NULL | NULL |    6 | M    |
| z    |    7 |    7 | C    |
| z    |    7 |    7 | K    |
+------+------+------+------+
5 rows in set (0.00 sec)
```

CASE 4. A **full join** is a combination of *both* left and right join. Again you match the left rows with the right rows. If a left row cannot be matched, create a dummy right row with all NULL values for this left row to match. If a right row cannot be matched, create a dummy left row with all NULL values for this right row to match. The syntax for doing this is

```
select * from L full join R on L.x = R.y
```

and the expected output is

```
L.foo   L.x   R.x   R.bar
a       1     1     P
g       3     NULL  NULL
h       4     NULL  NULL
z       7     7     C
z       7     7     K
NULL    NULL  2     W
NULL    NULL  6     M
```

However, MySQL currently does not support the FULL JOIN syntax. This is not a problem. Since the full join is just a combination of left and right, you can emulate the FULL JOIN using the UNION of the LEFT and RIGHT joins.

```
mysql> select * from L left join R on L.x = R.y
-> union
-> select * from L right join R on L.x = R.y;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| g    |    3 | NULL | NULL |
| h    |    4 | NULL | NULL |
| z    |    7 |    7 | C    |
| z    |    7 |    7 | K    |
| NULL | NULL |    2 | W    |
| NULL | NULL |    6 | M    |
+------+------+------+------+
7 rows in set (0.04 sec)
```

By the way, the UNION operator is used to create a set of rows by combining the results of two SELECT statements. The syntax is (obviously):

```
select ...
union
select ...
```

Note that UNION will remove duplicate rows. It's obvious that UNION operates on two results with the same column names and types.

Part of the reason why joins are confusing is because there are way too many names. For instance there's also the left outer join. Well ... left outer join is just left join. Also, right outer join is right join.

Yet another useless name is the equi-join. This is a join where the join condition uses =.

Remember that the join condition need not use equality comparison.

Here's another useless name: a self-join is a join where the left and right tables are the same table. Also, left and right and full joins and collectively called outer joins. If you need to do a self join, then you need to know that you need to give the table two names like this:

```
mysql> select * from
-> L L1 left join L L2 on L1.x = L2.x;
+------+------+------+------+
| foo  | x    | foo  | x    |
+------+------+------+------+
| a    |    1 | a    |    1 |
| g    |    3 | g    |    3 |
| h    |    4 | h    |    4 |
| z    |    7 | z    |    7 |
+------+------+------+------+
4 rows in set (0.00 sec)
```

Do you see the `L1` and `L2` The above SQL statement joins the left table `L1` (which is just an alias for `L`) with the right table `L2` (which is just an alias for `L`). The above is the same as the following using the word AS for alias:

```
mysql> select * from
-> L as L1 left join L as L2 on L1.x = L2.x;
```

One last join: a **cross join** of `L` and `R` is a join where every row in `L` is forced to matches every row in `R`. The reason for not including this join in the above list is because you can get such a join without the JOIN keyword. The SQL statement is just

```
select * from L, R
```

For our above `L` and `R` tables, since `L` has 4 rows and `R` has 5, the cross join will give us a table of 20 rows.

```
mysql> select * from L,R;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| g    |    3 |    1 | P    |
| h    |    4 |    1 | P    |
| z    |    7 |    1 | P    |
| a    |    1 |    2 | W    |
| g    |    3 |    2 | W    |
| h    |    4 |    2 | W    |
| z    |    7 |    2 | W    |
| a    |    1 |    6 | M    |
| g    |    3 |    6 | M    |
| h    |    4 |    6 | M    |
| z    |    7 |    6 | M    |
| a    |    1 |    7 | C    |
| g    |    3 |    7 | C    |
| h    |    4 |    7 | C    |
| z    |    7 |    7 | C    |
| a    |    1 |    7 | K    |
| g    |    3 |    7 | K    |
| h    |    4 |    7 | K    |
| z    |    7 |    7 | K    |
+------+------+------+------+
20 rows in set (0.00 sec)
```

**Exercise 18.2.** If you look at the left join example above:

```
mysql> select * from L left join R on L.x = R.y;
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| a    |    1 |    1 | P    |
| g    |    3 | NULL | NULL |
| h    |    4 | NULL | NULL |
| z    |    7 |    7 | C    |
| z    |    7 |    7 | K    |
+------+------+------+------+
```

you see that there are two row in `L` that does not match exactly with any rows in `R`, resulting in the two rows matching NULL values. Write an SQL statement that will give the following

output:

```
+------+------+------+------+
| foo  | x    | y    | bar  |
+------+------+------+------+
| g    |    3 | NULL | NULL |
| h    |    4 | NULL | NULL |
+------+------+------+------+
```

[HINT: Add a where clause.]
(SOLUTION: select * from L left join R on L.x = R.y where R.y is NULL)

**Exercise 18.3.** Select the non-matching rows for the right join case.

**Exercise 18.4.** You're designing a database for HyVee. Create a department table with two columns: id and name. In this table insert three rows for the produce, meat, and pharmacy departments. Create an employee table with three columns: firstname, lastname, department that this employee works in (this obvious should refer to the id column of the department table). In the employee table insert 5 rows: Christian Bale and Nicholas Cage working in the produce department, Tom Hanks and Arnold Schwarzenegger working in the meat department, and the new employee Adam Sandler who has not been assigned a department yet. Write SQL statements to do the following:

- List all employees (firstname and lastname) and department names.
- List all employees with assigned department.
- List all departments with no employees.
- HyVee is organizing a doubles tennis tournament. List all possible pairs of employees. (I hope it's obvious that (Christian Bale, Christian Bale) is *not* a valid double.)
- Oops ... new employees without an assigned department has to go to the training/orientation program which, unfortunately, is held at the same time as the tennis tournament. Redo the previous select ignoring employees without assigned departments.

[HINT: The SELECT can be applied to a table which you have seen before:

SELECT * FROM sometable ...

It can also be applied to any collection of rows such as a select:

SELECT * FROM (SELECT * FROM sometable) ...

```
select * from
(select from employee where departmentid != NULL) L
left join
(select from employee where departmentid != NULL) R
on L.firstname != R.firstname or L.lastname != R.lastname
```

You can also do this:

select * from table1 where column1 in (select column2 from table2)

There are 2 ways to write the query: Form the join and then filter out some rows or filter out rows and then do the join. ]

File: subquery.tex

# 19 Subquery

## 19.1 Alias

First set you this test table:

```
drop table Persons;
create table Persons
(
  id int,
  fname varchar(100),
  lname varchar(100),
  fingers int
);

insert into Persons (id, fname, lname, fingers) values
  (1, 'john', 'doe', 10),
  (2, 'mary', 'smith', 11),
  (3, 'felix', 'cat', 12),
  (3, 'mary', 'jones', 13);
```

Next do this simple query:

```
select * from Persons;
```

You can have an alias for your `Persons` table:

```
select * from Persons as P;
```

You can use the alias name with your column name:

```
select P.id, P.fname from Persons as P;
```

Aliases is helpful when you're doing a join on the same table:

```
select * from Persons as P0 join Persons as P1;
select * from Persons as P0 join Persons as P1 where P0.id <> P1.id;
```

You can have an alias for a column name too:

```
select P0.id as id0, P1.id as id1, P0.fname as fname0, P1.fname as fname1
from Persons as P0 join Persons as P1 where P0.id <> P1.id;
```

## 19.2 Subqueries

Instead of selecting from a stored table, you can select from a table that is created on the fly:

```
select *
  from
    (select id, fname from Persons) as T;
```

Try this too:

```
select *
  from
    (select id, fname from Persons) as T0
    join
    (select id, lname from Persons) as T1
;
```

You can reference the columns of the join in the `WHERE` clause:

```
select *
  from
    (select id, fname from Persons) as T0
    join
    (select id,lname from Persons) as T1
  where
    T0.id <> T1.id
;
```

Of course a subquery so it can include a `WHERE` clause:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
;
```

Recall that you can restrict values of a column in a collection of values using `IN`:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
    and T0.id in (1, 2, 3)
;
```

You can also use `IN` using the values from a column obtained from a subquery:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
    and T0.id in (select id from Persons)
;
```

Note that in the `WHERE` clause, you do not need to give a select table an alias.

And of course the subquery in the `WHERE` clause is a complete query and can itself have a `WHERE` clause too:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
    and T0.id in (select id from Persons where id < 3)
;
```

And the subquery in the where clause can use column name from the `FROM`:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
    and T0.id in (select id from Persons where id < T1.id)
;
```

It's also works if the subquery is a single value:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
    and T0.id in (select MAX(id) from Persons where id < T1.id)
;
```

If the subquery in the `WHERE` clause is a single row with a single column, you can treat is as a value:

```
select *
  from
    (select id, fname from Persons where id > 1) as T0
    join
    (select id, lname from Persons) as T1
  where
    T0.id <> T1.id
    and T0.id = (select MAX(id) from Persons where id < T1.id)
;
```

File: aggregate.tex

# 20 Operators and aggregate operators

There are operators on columns. First create this table with rows:

```
create table Person (
fname VARCHAR(100),
lname VARCHAR(100),
fingers INT,
eyes INT
);
insert Person (fname, lname, fingers, eyes)
values ('John', 'Doe', 10, 2),
       ('Jane', 'Doe', 9, 3),
       ('Tom', 'Smith', 11, 4),
       ('Sue', 'Jones', 11, 3),
       ('Harry', 'Doe', 12, 2);
```

OK. Ready? Try this:

```
select fname, lname, fingers + eyes from Person;
```

Get it?

```
+-------+-------+----------------+
| fname | lname | fingers + eyes |
+-------+-------+----------------+
| John  | Doe   |             12 |
| Jane  | Doe   |             12 |
| Tom   | Smith |             15 |
| Sue   | Jones |             14 |
| Harry | Doe   |             14 |
+-------+-------+----------------+
```

Notice the operator is executed for *each row*. You know what to do next: Practice active learning. Go ahead and try all the different numeric operators you know ... without looking at books or references.

(And did you try operators that give you boolean values?)

Of course there are string operators too. Of course we must have string concatenation (duh):

```
select concat(fname, lname), (fingers + eyes)/2 from Person;
```

and then this

```
select concat(fname, ' ', lname), (fingers + eyes)/2 from Person;
```

For columns with `DATE` datatype, the function `TO_DAYS` will give you the number of dates from `0000-00-00`. This allows you to, for instance, compute the difference between two date columns.

**Exercise 20.1.** Create a table with two date columns. Add some rows and compute the different between the two dates for each row.

**Exercise 20.2.** Do the same but now for seconds. What do you think the name of the function that converts TIME to seconds since 00:00:00?

You can also operate on a collection of rows. Try this:

```
select SUM(fingers) from Person;
```

This function sums up over all the rows of the table. So this is different from the functions/operators on single rows. `SUM` is an example of an aggregate function.

**Exercise 20.3.** Can you include functions/operators on rows inside SUM? Try it.

**Exercise 20.4.** Make sure you try `COUNT`, `AVG`, `MAX`, `MIN`. It's also not surprising that there is standard deviation (`STDDEV`) and variance (`VARIANCE`).

Instead of summing over the whole table, you can create groups of rows and sum over the rows of each group. Try this:

```
select lname, SUM(fingers) from Person group by lname;
```

and you'll get

```
+-------+--------------+
| lname | SUM(fingers) |
+-------+--------------+
| Doe   |           31 |
| Jones |           11 |
| Smith |           11 |
+-------+--------------+
```

Read the data – it should be clear what's happening. You can include `fname` in the above query, but it will only list one of the `fname` from each group:

```
select fname, lname, SUM(fingers) from Person group by lname;
```

gives us

```
+-------+-------+--------------+
| fname | lname | SUM(fingers) |
+-------+-------+--------------+
| John  | Doe   |           31 |
| Sue   | Jones |           11 |
| Tom   | Smith |           11 |
+-------+-------+--------------+
```

You can have groups within groups:

```
select lname, eyes, SUM(fingers) from Person group by lname, eyes;
```

gives us

```
+-------+------+--------------+
| lname | eyes | SUM(fingers) |
+-------+------+--------------+
| Doe   |    2 |           22 |
| Doe   |    3 |            9 |
| Jones |    3 |           11 |
| Smith |    4 |           11 |
+-------+------+--------------+
```

You can see that the `Doe` group (by `lname`) contains rows with `2` eyes and another group with `3` eyes. That's why you see `Doe` twice.

It's not surprising that you can have multiple aggregate functions:

```
select lname, eyes, COUNT(*), SUM(fingers) from Person group by lname, eyes;
```

gives us

```
+-------+------+----------+--------------+
| lname | eyes | COUNT(*) | SUM(fingers) |
+-------+------+----------+--------------+
| Doe   |    2 |        2 |           22 |
| Doe   |    3 |        1 |            9 |
| Jones |    3 |        1 |           11 |
| Smith |    4 |        1 |           11 |
+-------+------+----------+--------------+
```

Note for the various numeric functions, NULL values are skipped. If the table is empty or the values in the relevant column(s) are NULL, then the value computed by the aggregate function is NULL.

**Exercise 20.5.** Create a table with column `a` of `INT` data type. Sum the whole table. What do you get? Insert a single row with `a` set to `NULL`. Sum the whole table – what do you get? Finally insert a single row with `a` set to `42`. Sume the whole table – you should get 42.

You can also remove duplicate rows. If you do this:

```
select eyes from Person;
```

you will get repeats. To remove the repeats (in a group), you do this:

```
select distinct eyes from Person;
```

You can do that on multiple columns. First try this:

```
select lname, eyes from Person;
```

then try this:

```
select distinct lname, eyes from Person;
```

The above uses `distinct` to select duplicates in whole table. It's not too surprising that the distinct keyword can be used to remove duplicated in a group. Try these:

```
select lname, eyes from Person;
select distinct lname from Person;
select count(distinct lname) from Person;
select count(distinct lname, eyes) from Person;
select count(distinct lname, eyes) from Person group by lname, eyes;
```

Of course you can also do this:

```
select count(distinct lname, eyes), sum(eyes) from Person
group by lname, eyes;
```

or even this

```
select count(distinct lname, eyes), sum(distinct eyes) from Person
group by lname, eyes;
```

**Exercise 20.6.** Find out how to extract the year from a `DATE` value. What about month? What about day?

**Exercise 20.7.** You have a table of sales. Each row contains `customer_id` (an integer), `product_id` (an integer), `number_of_units`, `date_of_purchase`. For instance `(13234, 434354, 45, '2000-10-23')` represents the customer with id `13234` bought `45` of a product with id `434354` on `'2000-10-23'`. The `product_id` of the sales table references the `product` table that contains `id` of the product and the `unit_cost` (the other columns are

not relevant). Write, if possible, queries that give the following:

- the total expenses for each customer
- customers who spends the most
- total expenses of all customer
- customers who bought all products
- customers who did not buy all products
- customers who did not any products
- products not purchased by anyone
- products that brought in the maximum revenue
- for each product, number of customers who bought that product
- for each product that brought in the maximum revenue, list customers who did not buy that product
- for each customer, the product(s) that he/she spends most money on
- All the above queries are also relevant if the queries are "by year".
- You can also ask "seasonal" versions of the above queries. For instance which month of each year generates maximum revenue?
- You can also ask questions related to time intervals. For instance for each product, you can measure the difference in dates between two consecutive purchases. You can then ask what is the maximum date difference for each product. You can also ask which product(s) has the maximum date difference among all products.

Note that of course you can also fetch all the relevant data and perform the computations using a programming language. For very complex computations, you probably want to do that.

However, there's a benefit to doing computations in your MySQL server. If for instance the computation results in a very small piece of data, then the network communication between your program with your MySQL server is minimal. Think for instance of a sum computation on a column – the return is just a number. If you write a program to fetch the numeric data for a column in a table with 1000 rows, your program is fetching 1000 integers from the MySQL server. Of course if you do the sum, MySQL server still has to

run over the 1000 integers. But that is done within the server. And RDBMS are highly optimized for their basic aggregate operations. In this case, there's no doubt that doing the sum in MySQL going to be better than writing a progam to fetch the data and sum in your program.

File: acid.tex

# 21 ACID

ACID starts for
1. atomicity
2. consistency
3. isolation
4. durability

First let's talk about a transaction. You can think of a transaction as a bunch of things that you want to do to a database. For instance you might want to add a row in a table $X$, delete a row from table $Y$, and modify a column value in table $Z$. So ... what's the point of bunching up all the three commands? or three pieces of work?

Well ... there are times when you want to consider all three really as one scenario and you want to either do all three or not. So although they are physically changing the memory or storage of the database (in the physical sense), you want to think of this bunch of operations as (in the logical sense) a single unit of work.

Here's an example: Suppose you have two bank accounts. You want to transfer $1000 from one to another. This involves decrementing the amount in one account by $1000 and incrementing the amoung in another. What if, for some reason, the RDBMS was only able to perform the first operation of decrementing the first account? That would be bad right? For instance what if the system crashed after the first operation? It does happen you know.

The property that a transaction (a bunch of work) is guaranteed to be either completely done or not at all is the idea of **atomicity**. The atomicity is the "all or none" rule. The technical term for "the transaction is done or guaranteed" is that the "the transaction is **committed**."

**Consistency** means that if the database is consistent before the transaction, then it will be consistent after the transaction. If at the end of a transaction, some constraint is violated, then the database must have the ability to undo the transaction to go back to the state of the database before the transaction (in order to remain consistent.) For instance suppose at a bank you can withdraw up by $1000 but only deposit up to $500 at a time. If you try to move $1000 from one account $X$ to another account $Y$, then the first decrement from account $X$ would work, but the increment for $Y$ would fail. In this case, the constraint violation should reset the account $X$ back to its original state ... or you would be an

unhappy customer.

**Isolation** means that the system must protect the data that is currently in use by a transaction from another transaction. For instance imagine that you are trying to deposit $1000 to your bank account and another institution $X$ is depositing $500 to your account (maybe you own some lucky draw). If you originally had $200. Your transaction might read $200. The transaction from institution $X$ might read $200. Your transaction increments $200 by $1000 and write to your account. The transaction from insitution $X$ increments $200 by $500 and write to your account. Guess what? You have $700 and not $(200 + 1000 + 500). Tough.

**Durability** refers to the fact that once a transaction is executed, the results cannot disappear or be changed unless another transaction performs such a changed. This includes the case of power failure: If the database software says that the transaction is successfully executed, then the database software vendor must ensure that the result must be stored even if there's a power failure immediately after the system reports the success of the transaction. Power failure is not the only problem that can happen. For instance the success message from the database system might be buffered and queued and the transaction might also be buffered and queued. It's entirely possible for the success message to be sent to the client before the power failure while the transaction is still in the buffer for updating the database tables on storage. In this case, the problem is a combination of a power failure and the operating system. But again, it's entirely the responsibility of the database software vendor to make sure that the results are indeed committed to the storage. We don't have to care about how the vendor does it.

The concept of transaction applies not only to database systems. This concept is crucial to all critical and concurrent systems.

The above are not available to you if you use regular files (provided by your OS) for the storage of your data.

File: commit-rollback.tex

# 22 Commit and Rollback

Recall that because of ACID, we want a higher level of definition of a unit of work. You can think of the execution a single SQL statement as a low level unit of work. For the sake of ACID, our concept of higher level of work is a transaction which is a bunch of low level SQL statements.

Because we want a transaction to be either completely done or not at all, this means we need to know if a group of SQL executions has a failure. Furthermore for the purpose of ACID, associated with the concept of transaction we need to have two commands: commit and rollback.

A commit is a command that tells the RDBMS that the changes to the database does not cause any violation of ACID and therefore we want the intended changes to be made to the database.

A rollback is a command that tells the RDBMS that, because something has gone wrong during the execution of the transaction (i.e., one of the SQL execution causes a violation of ACID), we want to throw away this transaction. This means that we want the database to revert back to the state just before the transaction.

Note that for MySQL, you can choose to have a table to be transactional or non-transactional. For a transactional table, you have to create the table like this:

```
create table ... (
...
) engine=innodb;
```

i.e., with the `engine=innodb` after the create statement.

To execute and commit a bunch of SQL statements as a transaction, you do this:

```
start transaction;
sql1
sql2
sql3
commit;
```

The changes made in sql1, sql2, sql3 will then be reflected in the database.

To execute and rollback a bunch of SQL statements as a transaction, you do this:

```
start transaction;
sql1
sql2
sql3
rollback;
```

In this case, the changes made in sql1, sql2, sql3 will *not* be reflected in the database.

Let's try an experiment. Get rid of the `test` database, recreate it, and create the `Person` table (with `engine=innodb`) with the following structure:

```
mysql> desc Person;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| fname   | varchar(100) | YES  |     | NULL    |       |
| lname   | varchar(100) | YES  |     | NULL    |       |
| fingers | int(11)      | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Open two MySQL console clients. In the first client do this:

```
mysql> use test;
Database changed
mysql> select * from Person;
Empty set (0.00 sec)
```

No surprises. Continue in the first client with this:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert Person set fname='John';
Query OK, 1 row affected (0.01 sec)

mysql> select * from Person;
+-------+-------+---------+
| fname | lname | fingers |
+-------+-------+---------+
| John  | NULL  |    NULL |
+-------+-------+---------+
1 row in set (0.01 sec)

mysql>
```

The first client says that the `Person` table has one row.

Now in the second client, if you display the row in `Person`, you will see that there is nothing:

```
mysql> use test;
Database changed
mysql> select * from Person;
Empty set (0.00 sec)
```

The point is that the insert in the first client has not been committed to the database. Only the first client can see the change. No other client sees it.

Now go back to the first client and do this:

```
mysql> commit;
```

and when you display all the rows in `Person`, whether it's in the first *or second client*, you will see one row of data.

**Exercise.** Repeat the above experiment with a small change:

```
CLIENT1                                          CLIENT2
use test;
                                                 use test;
delete from Person;
select * from Person;
                                                 select * from Person;
insert into Person set fname='John';
select * from Person;
                                                 select * from Person;
start transaction;
insert Person set fname='Tom';
select * from Person;
                                                 select * from Person;
rollback;
select * from Person;
                                                 select * from Person;
```

What is the point of this experiment?

Let's do another experiment and see what happens when the first client *updates* ôw without committing followed by the second client updating to the same row.

```
CLIENT1                                          CLIENT2
use test;

                                                 use test;
delete from Person;
insert into Person set fname='John';
select * from Person;

                                                 select * from Person;
start transaction;
update Person
  set fname='Tom'
  where fname='John';
select * from Person;

                                                 update Person
                                                   set fname='George'
                                                   where fname='John';
```

After some time, you will see that the second client's update statement will fail – there's a conflict. It does not fail immediately because MySQL gives the second client a certain amount of time to wait for the first client to commit. If the first client commits fast enough, the second client can still execute its update, but MySQL will report no row has been changed since (of course) the row with `fname = 'John'` has it's `fname` already changed by the first client.

**Exercise.** Open two MySQL console clients and start a transaction on both. Perform an insert into the `Person` table for each client without committing. Display the contents of `Person` for each client. Next, commit both clients and display then display the contents of `Person` for each client. In *this* case, the clients are not blocked even though there are pending transactions for each clients – there is no conflict.

File: cursor.tex

# 23 Cursor

When you execute an SQL query such as a select statement, the rows are stored in a container (this is called a result set). You should think of a cursor as something that points to the first row of your container. With the cursor, you can run through the rows in the container, with the cursor moving to the next row after each row operation on the container. You can think of the cursor as an iterator of a container.

By the way, you recall that SQL syntax allows you to write SQL statement that operates on a whole result set – there's no concept of a for loop in the SQL syntax. You do realize that when you took the database class, right? The SQL language is completely different from a 3rd generation language like C++ or Python or Java.

It's a lot more important to know how to access data programmatically using a programming language (C++/Java/Python/PHP/etc.). Once you see some examples, the concept of a cursor will be clear.

```
File:  python.tex
```

# 24 Programmatic Access: Python

In this section, I'll show you how to access MySQL using the Python programming language. I assume (of course) you know Python.

To allow Python access to MySQL, do this at your bash shell as root:

```
dnf install -y MySQL-python
```

or this if the above does not work, do this:

```
pip install MySQL-python
```

Instead of the MySQL-python library, you can also use the pymysql library. As root do this:

```
dnf install -y python2-pymysql
```

For Fedora 29:

```
dnf install -y python2-PyMySQL
dnf install -y python3-PyMySQL
```

for python2 and python3.

Get rid of the `test` database, recreate it, and create the `Person` table (with `engine=innodb`) with the following structure:

```
mysql> desc Person;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| fname   | varchar(100) | YES  |     | NULL    |       |
| lname   | varchar(100) | YES  |     | NULL    |       |
| fingers | int(11)      | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Next, add data to the `Person` so that you get this:

```
mysql> select * from Person;
+-------+-------+---------+
| fname | lname | fingers |
+-------+-------+---------+
| John  | Doe   |      10 |
| Tom   | Smith |    NULL |
+-------+-------+---------+
```

Now we are ready to begin accessing MySQL using Python.

Run the Python interpreter and enter the following:

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(user='root', passwd='root')
>>> cursor = conn.cursor()
>>> cursor.execute("use test")
0L
>>> cursor.execute("select * from Person")
2L
>>> records = cursor.fetchall()
>>> print records
(('John', 'Doe', 10L), ('Tom', 'Smith', None))
>>>
```

If you are using the pymysql library, make sure you replace `MySQLdb` by `pymysql`.

The above shows you how to make a network connection to the MySQL server, how to create a cursor for the connection, how to execute, and read the results of SQL statements.

You can see that, after the select statement, the cursor contains a tuple of tuples of the result of executing the select statement. The order of the values in the tuple (of course) corresponds to the ordering of the column names in your `Person`.

Instead of fetching all the tuples, you can fetch them one at a time:

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(user='root', passwd='root')
>>> cursor = conn.cursor()
>>> cursor.execute('use test')
0L
>>> cursor.execute('select * from Person')
2L
>>> print cursor.fetchone()
('John', 'Doe', 10L)
>>> print cursor.fetchone()
('Tom', 'Smith', None)
>>> print cursor.fetchone()
None
>>>
```

**Exercise 24.1.** Instead of printing all the records (i.e. rows) in the cursor, you can print them one at a time using a loop. Redo the above in the following way:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root')
cursor = conn.cursor()
cursor.execute("use test")
cursor.execute("select * from Person")
for x in cursor.fetchall():
    print x
```

**Exercise 24.2.** Redo the above so that the output is

```
First name: John, Last name: Doe, Number of fingers: 10
First name: Tom, Last name: Smith, Number of fingers: None
```

**Exercise 24.3.** Redo the above so that the output is

```
First name  Last name  Number of fingers
----------  ---------  -----------------
John        Doe        10
Tom         Smith      None
```

Your code should work for very long first names and very long last names. For instance if `Tom` was changed to `Elizabethiana`, the output should be:

```
First name     Last name  Number of fingers
-------------  ---------  -----------------
John           Doe        10
Elizabethiana  Smith      None
```

## 24.1 Dictionary Cursor

It's actually more convenient to use a dictionary cursor. Try this:

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(user='root', passwd='root')
>>> cursor = conn.cursor(MySQLdb.cursors.DictCursor)
>>> cursor.execute("use test")
0L
>>> cursor.execute("select * from Person")
2L
>>> print cursor.fetchall()
({'lname': 'Doe', 'fingers': 10L, 'fname': 'John'}, {'lname': 'Smith',
'fingers': None, 'fname': 'Tom'})
>>>
```

**Exercise 24.4.** Using a loop, we can rewrite the code to print all results like this:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root')
cursor = conn.cursor(MySQLdb.cursors.DictCursor)
cursor.execute("use test")
cursor.execute("select * from Person")
for row in cursor.fetchall():
    print row['fname'], row['lname'], row['fingers']
```

Why is the dictionary cursor better than the regular cursor?

**Exercise 24.5.** Write a python program that prompts the user for fname, lname, fingers and insert a row in the Person table in the test database.

**Exercise 24.6.** Modify your program from the previous exercise and write a function `add_person` that does the obvious.

## 24.2 Other Parameters During Connection

Instead of executing "use test", you can also specify the database you want to use during the connection:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root', db='test')
```

**Exercise.** By default MySQLdb assumes that your MySQL server is on the same machine, i.e. localhost. What if the machine is somewhere else? [HINT: Google.]

## 24.3 Parameter Substitution

In many cases, you want to perform an SQL statement based on some input. (For instance the input is from a web form.) You can build the SQL statement using string methods:

```
fname = 'John'
sql = "select * from Person where fname='%s'" % fname
cursor.execute(sql)
```

or

```
fname = 'John'
sql = "select * from Person where fname='" + fname + "'"
cursor.execute(sql)
```

You should *not* do that. Instead you should let MySQLdb substitute the value(s) for you like this:

```
fname = 'John'
sql = "select * from Person where fname=%s"
cursor.execute(sql, (fname,))
```

(Do not put quotes around the `%s`.) Or if you have multiple values to substitute:

```
fname = 'John'
lname = 'Doe'
sql = "select * from Person where fname=%s and lname=%s"
cursor.execute(sql, (fname, lname))
```

(Use `and` instead of a comma.)

## 24.4 Transaction

Since you already know SQL, there's not a whole lot left to talk about. There's however the issue of transactions. If your tables are built with `engine=innodb`, then you can perform commits and rollbacks of transactions. Transactions are connection-specific – when you start a connection and get a cursor, you have started a transaction. This means the following:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root', db='test')
cursor = conn.cursor()
... EXECUTE SOME SQL HERE ...
conn.commit()
```

Note that the last line is a call to the commit method in the connection object.

Here's an experiment on transactions. Open two bash shells. In the first shell, run the MySQL console client, clear all the rows in the `Person` table in the `test` database. When you select everything from the `Person` table of course you should see this:

```
mysql> select * from Person;
Empty set (0.01 sec)
```

In your second bash shell, run Python and execute the following:

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(user='root', passwd='root', db='xxx')
>>> c = conn.cursor()
>>> c.execute("insert Person set fname='john'")
1L
```

You have just added one row to `Person`. Now go back to the first terminal that is still running the MySQL console client and view the contents of the `Person` table:

```
mysql> select * from Person;
Empty set (0.01 sec)
```

Again, there's nothing. The point is that the results from the second bash shell has not been committed yet. So go back to the second bash shell that is still running Python and continue with this extra command:

```
>>> conn.commit()
```

The new row should now be updated. So we go back to the first bash shell running the MySQL console client and do this to verify:

```
mysql> select * from Person;
+-------+-------+---------+
| fname | lname | fingers |
+-------+-------+---------+
| john  | NULL  |    NULL |
+-------+-------+---------+
1 row in set (0.00 sec)
```

Now for an experiment on rollback. First delete all rows from `Person`. Next do this:

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(user='root', passwd='root', db='xxx')
>>> c = conn.cursor()
>>> c.execute("insert Person set fname='john'")
```

At this point check, using your MySQL console client, that the `Person` is still empty. Continuing above, do this:

```
>>> c.rollback()
```

and exit Python. Now check, using your MySQL console client, that the `Person` is still empty. This of course means that the insert in Python was not committed.

If you keep your connection object alive, and perform a commit, the SQL commands committed is from the last commit or rollback. For instance, in the following:

```
cursor.execute(sql1)
cursor.execute(sql2)
conn.rollback()
cursor.execute(sql3)
cursor.execute(sql4)     # sql3, sql4 executed and committed
conn.commit()
```

only the changes due to sql3 and sql4 are actually committed to the database. The results of sql1 and sql2 are rolled back. Likewise we have this:

```
cursor.execute(sql1)
cursor.execute(sql2)
conn.commit()            # sql1, sql2 executed and committed
cursor.execute(sql3)
cursor.execute(sql4)
conn.commit()            # sql3, sql4 executed and committed
```

Remember that a transaction is a bunch of work (i.e., SQL commands for our case). So I

still need to tell you how to detect if one of the SQL statements in the transaction failed. You detect that through exceptions like this:

```
try:
    cursor.execute(sql1)
    cursor.execute(sql2)
    cursor.execute(sql3)
    cursor.execute(sql4)
    conn.commit()
except Exception as e:
    print "exception!!!"
    print e
    conn.rollback()
```

You are warned not to let threads share a connection object.

## 24.5 Correct use of connections

All the above examples we ran above assume that we are the only ones using the database server. To get a perspective on real world issues, in the case of web applications, a company's database server might be hit by many browsers requesting for data in a short period of time. To make this issue concrete, run the following:

**Exercise 24.7.** Run the following program (changing the root password to whatever you're using):

```
import MySQLdb
connectionpool = []
for i in range(10000):
    print i
    connectionpool.append(MySQLdb.connect(user='root', passwd='root'))
```

What does the output tell you?

The following are two rules that you should follow:

- You should have only one cursor for each connection. However it's OK to have multiple connections.
- You should close a connection as soon as possible in order to share resources fairly.

First this is how you close a cursor and a connection:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root', db='test')
cursor = conn.cursor()
... EXECUTE SQL1 ...
... EXECUTE SQL2 ...
... EXECUTE SQL3 ...
cursor.close()
conn.commit()
conn.close()
```

And if you need to perform bunches of SQL statements, you obviously do something like this:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root', db='test')
cursor = conn.cursor()
... EXECUTE SQL1 ...
... EXECUTE SQL2 ...
... EXECUTE SQL3 ...
cursor.close()
conn.commit()
conn.close()

... DO SOME COMPUTATIONS ...

conn = MySQLdb.connect(user='root', passwd='root', db='test')
cursor = conn.cursor()
... EXECUTE SQL4 ...
... EXECUTE SQL5 ...
... EXECUTE SQL6 ...
cursor.close()
conn.commit()
conn.close()
```

Of course you have to think about the above carefully: SQL1, SQL2, SQL3 should be a complete transaction and SQL4, SQL5, SQL6 should be another transaction.

It would be a terrible idea to execute some SQL statements, do some other work for *one whole hour*, and then execute another unrelated SQL statement, and then close the connection (unless of course you already know that there will be enough connections to go around.) You want to release the resource as soon as you can. Of course if the SQL statements are related and forms a transaction, then you should execute all of them,

commit the transaction, and then close everything.

Taking into account the possibility of exceptions, the above becomes:

```
import MySQLdb
conn = MySQLdb.connect(user='root', passwd='root', db='test')
cursor = conn.cursor()
try:
    ... EXECUTE SQL1 ...
    ... EXECUTE SQL2 ...
    ... EXECUTE SQL3 ...
    conn.commit()
except Exception as e:
    ... POSSIBLY PERFORM SOME CLEAN UP OPERATION ...
    conn.rollback()
cursor.close()
conn.close()

... DO SOME COMPUTATIONS ...

conn = MySQLdb.connect(user='root', passwd='root', db='test')
cursor = conn.cursor()
try:
    ... EXECUTE SQL4 ...
    ... EXECUTE SQL5 ...
    ... EXECUTE SQL6 ...
    conn.commit()
except Exception as e:
    ... POSSIBLY PERFORM SOME CLEAN UP OPERATION ...
    conn.rollback()
cursor.close()
conn.close()
```

Of course doing a commit is necessary only if the SQL statements made some modification to the database.

**Exercise 24.8.**

- Using the web, figure out how to retrieve the maximum number of concurrent connections for your MySQL server.
- Next, figure out how to change this configuration setting. Test it.

[At the mysql prompt, execute `select variables like "max_connections"`. To set it to 200, execute `set global max_connections=200`.]

## 24.6 Exceptions

For finer control you can do something like the following:

```
try:
    ...
except MySQLdb.Error as e:
    ...
except MySQLdb.Warning as e:
    ...
except SomeOtherExceptionClass as e:
    ...
```

For more details, you can refer to http://www.python.org/dev/peps/pep-0249/.

**Exercise 24.9.** What happens when you use python to execute a badly formed SQL statement? Which exception is caught? Here's an example of a badly formed SQL statement: `select * frommmm Person` What about the case where you attempt to add a table when the name already exists? What if you attempt to add a row to a table with a column name that is not in the table?

**Exercise 24.10.** Write a program that prints the following menu

```
[a] add person
[d] delete person
[u] update person
[f] find person
```

and then prompts the user for one of the above options in the menu. If the user selects 'a', prompt him/her for fname, lname, fingers and add a row to the Person table in the test database. If the user select 'd', prompt him/her for fname, lname and deletes all rows with the given fname, lname. If the usr select 'u', prompt him/her for fname, lname, and then new fname, new lname, new fingers and update all rows with the supplied fname, lname with the given new fname, lname, fingers. If the user enters 'f', prompt the user for fname, lname and print all rows matching the given fname and lname; note that if the user enters an empty string for fname, then you should only match by lname, etc. If the user enters empty strings for fname and lname, you should print all rows in the table. Your program should release MySQL connnections as soon as your each query is completed. Use phpMyAdmin to browse the table to verify the success of each query.

File: cpp.tex

# 25 Programmatic Access: C/C++

You need to install the following:

```
yum -y install mysql-devel
yum -y install mysql++
```

With your experience with accessing connecting to the MySQL server using python, the following should be easy to understand.

```
#include <stdio.h>
#include <mysql.h>

int main(int argc, char **args)
{
    MYSQL_RES *result;
    MYSQL_ROW row;
    MYSQL *connection, mysql;
    int state;

    mysql_init(&mysql);
    connection = mysql_real_connect(&mysql, "127.0.0.1",
                                    "root", "root", // user, password
                                    "test", // database
                                    0,0,0);
    if (connection == NULL)
    {
        printf(mysql_error(&mysql));
        return 1;
    }
    state = mysql_query(connection, "SELECT * FROM applicant");
    if (state != 0)
    {
        printf(mysql_error(connection));
        return 1;
    }

    result = mysql_store_result(connection);
    printf("Rows:%d\n", mysql_num_rows(result));

    while ((row=mysql_fetch_row(result)) != NULL)
    {
        printf("%s, %s\n", (row[0] ? row[0] : "NULL"),
                           (row[1] ? row[1] : "NULL" ));

    }
    mysql_free_result(result);
    mysql_close(connection);
    return 0;
}
```

You compile the program like this:

```
g++ main.cpp -I/usr/include/mysql /usr/lib/mysql/libmysqlclient.so
```