MySQL

Monday, December 14, 2015 9:38 PM

12/14/2015
PRONUNCIATION: apparently the official pronunciation is "My S Q L", not "My Sequel".

Setup/basic usage:

- After installing on Windows, add "C:\Program Files\MySQL\MySQL Server 5.7\bin\" to the PATH so that you get access to mysql.exe.
  - **WARNING:** some IDIOT had the idea to name a folder "C:\Program Files (x86)\MySQL \MySQL Fabric 1.5 & MySQL Utilities 1.5\Doctrine extensions for PHP\", so every time you do something like "PATH %PATH%;" on Windows, it will try executing MySQL and say that it isn't found. I just removed this from my PATH entirely (it was there twice for some reason), but I suppose you could also just add quotes.
  - On Linux, you install it with "sudo apt-get install mysql-server".
- mysql -u root -p
  - Then type in your password.
- Make a new database
  - CREATE DATABASE test_db;
- Create a table
  - From scratch: CREATE TABLE hardware_unlock_history(user_id INT);
  - From an existing table:
    - SHOW CREATE TABLE hardware_unlock_history;
    - You'll get two columns as a result: one has the name of the table, and the other has the SQL statement used to create that table. You can just copy/paste the SQL into another database.
- You can show it with
  - SHOW DATABASES;
- To find the version of MySQL that you're using (reference)
  - SELECT @@version;
- Select relative dates
  - SELECT * FROM users WHERE last_login_date > UTC_TIMESTAMP() - INTERVAL 1 DAY;
    - (see the note below about using "NOW()")
- Select into a file:
  - SELECT journal FROM replays INTO OUTFILE ´journal.txt´
- Use the database
  - USE test_db;
- Show tables in the database:
  - SHOW tables;
- Make a new schema
  - CREATE SCHEMA IF NOT EXISTS test_db;
- To find the schema of a table:
  - DESCRIBE table_name;
- Check the existence of a row
  - SELECT 1 FROM users WHERE id = 5;
  - Note: the result will always be 1 if the row exists
- Select a bunch of rows at once
  - SELECT id, name FROM users WHERE name IN ('Adam', 'Adam2');
- Get rows similar in value to another row (reference)
  - SELECT id, skill FROM users ORDER BY ABS(skill - 500) ASC LIMIT 10;

        - Note that this sorts by absolute value difference the way I have it written, so the resulting table will look like this:

```
+-----+--------+
| id  | skill  |
+-----+--------+
| 500 | 500.00 | <-- difference of 0 (i.e. best result if comparing skills for matchmaking)
| 499 | 499.00 | <-- diff of 1
| 501 | 501.00 |
| 498 | 498.00 | <-- diff of 2
| 502 | 502.00 |
| 497 | 497.00 | <-- diff of 3
| 503 | 503.00 |
| 496 | 496.00 |
| 504 | 504.00 |
| 495 | 495.00 |
+-----+--------+
```

- Use TEXT instead of VARCHAR unless you want to limit the size of the data.
- In general, do not use root. Make a different user for each layer of access. Users belong to all of MySQL and they need to be added to schemas.
    - Keep in mind, in this command:
        - CREATE USER 'TomAto'@'localhost' IDENTIFIED BY 'password';

          The "@localhost" signifies the connecting machine, meaning I would only be able to connect while on the same machine as the database. If I tried connecting remotely, it wouldn't work. I could specify "@%" there so that I could connect from anywhere.
    - Also, you don't necessarily have to run this specific command in order to create a user. The "GRANT" command will create the user if it doesn't exist
        - GRANT ALL PRIVILEGES ON test_db.* TO 'TomAto'@'localhost' IDENTIFIED BY 'password';
- To list all users, do this:
    - use mysql;
    - select User from user;
- After making a user, you need to give them access.
    - GRANT ALL PRIVILEGES ON test_db.* TO 'TomAto'@'localhost';
      This grants TomAto access to all tables in test_db.
- Prefer DATETIME over DATE.
- In order for foreign keys to work, you first need to have the table in which the column is a primary key, so you MAY want to first create all your tables then prepare a script for the creation of foreign keys.
    - ALTER TABLE Orders ADD CONSTRAINT fk_PerOrders FOREIGN KEY (P_Id) REFERENCES Persons (P_Id)
    - EveryJuan: @Adam13531, Here is another gotcha, we actually don't use any foreign keys because we don't actually ever drop data, we have another field as a boolean which determines if it was deleted, we use it for archiving, then we do all the foreign key semantics in node
- Serializing data, e.g. an inventory, and storing it in a column can be a good idea if you don't need to query the data. It can be fine to store as JSON in cases like that.
- The JSON functions in MySQL are not really supposed to be used in production. I think it's just for obtaining NoSQL functionality from a SQL database. I.e. if you find yourself using these functions, you should probably format your data into proper relational tables.
- MySQL Workbench is great for viewing/editing quickly without having to type in queries manually

(which can be a pain on Windows when there's no autocomplete in CMD). To view tables quickly, use the Schemas view, choose <Your Database Name> --> Tables --> hover over any table and click the grid icon on the right. Then, to make edits, simply change any fields you want and click the Apply button at the lower right of the results pane. If you don't want confirmations every time, you can change the setting here: "Edit->preferences->SQL Queries: Confirm Data changes"

- To null out a column, right-click without having given the input the focus and you'll get a menu like this

Auto-updating timestamps (reference)

I've gotten suggestions from people that every single row in a database should have two values:

- created_at
- updated_at

Both of these can be initialized by default, and updated_at can be updated automatically by following the notes at the reference link.

Math.min and Math.max (reference)

These are just called GREATEST and LEAST and seem to take any number of arguments.

select GREATEST(5,3,6,2);

6

COALESCE (i.e. default values when null) (reference)

This returns the first non-null value in a list, so you can do something like this:

Suppose you have a table that has a "rating" column, and this can be null. You want to update the "rating" column to add 5. You can use this query:

```
UPDATE users SET rating = COALESCE(rating, 0) + 5;
```

I ran into a case where I wanted to UPSERT, and when running the INSERT side of things, set the value to "newValue", and in UPDATE, set it to the LEAST of the existing and new values. Here's

how I did it:

```
UPDATE users set rating = LEAST(COALESCE(rating, newValue), newValue);
```

JSON (Official reference, common usage reference)

"JSON" is the type just like how "INT" is a type. Simple usage:

```
CREATE TABLE json_test(numbers JSON);
INSERT INTO json_test VALUES (JSON_ARRAY(1, 2, 3));
INSERT INTO json_test VALUES ('[1, 2]');
```

Note: in Knex, when inserting array data into a JSON column, call "JSON.stringify()" on it first.

Views

A view is like a virtual table.

Here's an example that IAMABananaAMAA sent for protecting against leaking private info while streaming by making a view of "users" that only allows selecting ID and name:

```
    # Assume `botland`.`users` has: `id`, `username`, `password`, `email`
(and password/email are unencrypted, identifiable information)
    CREATE USER 'remote'@'%' IDENTIFIED BY 'remote'; # Create a new account
with your specifications (local/remote/etc)
    CREATE VIEW protected_users AS SELECT id, username FROM users; # Create a
new view (view == table copy with specified columns). You can do anything as
your SELECT here, so JOINing, WHERE, etc
    GRANT SELECT ON botland.protected_users TO 'remote'@'%'; # Grant
permissions to your new account
    # Now you can pretend `protected_users` is a table that only has access
to `id` and `username`. So hypothetically if you do get breached, the account
wouldn't have access to the `password` or `email` columns anyways
```

From <https://bpaste.net/raw/84a84df5af4f>

Outputting a SQL command directly to a file (reference)

```
mysql -user user -pass pass -e"COMMAND TO RUN;" > file.txt
```

For example:

```
mysql -D botland -u root --password=password --default-character-set=utf8mb4 -e"SELECT name
from users;" > a.txt
```

Global variables

To see the current value of a particular variable:

```
SHOW GLOBAL VARIABLES LIKE 'innodb_%prefix%';
```

Collations and charsets (reference)

Character sets determine which characters are valid in your database (e.g. UTF8 vs. UTF8MB4), and collations are just a way of comparing them (e.g. for sorting).

There's a character set for a database **and also one for a connection**. For example, if your database is utf8mb4 and you have emojis stored somewhere but your connection is, say, cp850, you will see "???":

```
mysql>  SHOW VARIABLES LIKE '%character_set%';
```

```
+--------------------------+------------------------------------------------------------+
| Variable_name            | Value                                                      |
+--------------------------+------------------------------------------------------------+
| character_set_client     | cp850                                                      |
| character_set_connection | cp850                                                      |
| character_set_database   | utf8mb4                                                    |
| character_set_filesystem | binary                                                     |
| character_set_results    | cp850                                                      |
| character_set_server     | utf8                                                       |
| character_set_system     | utf8                                                       |
| character_sets_dir       | C:\Program Files\MySQL\MySQL Server 5.7\share\charsets\ |
+--------------------------+------------------------------------------------------------+

mysql> select name from scripts;
+------+
| name |
+------+
| ???? |
+------+

mysql> charset utf8mb4
Charset changed
mysql> select name from scripts;
+------------------+
| name             |
+------------------+
| 🎁🎁🎁🎁          |
+------------------+
```

The above only changes your charset for that particular connection. You can specify this at startup with "--default-character-set":

```
mysql -D mydb -u user --password=password --default-character-set=utf8mb4
```

Note that I still cannot figure out how to get MySQL Workbench to display this correctly. It doesn't work with Chinese characters either, which makes me think it's just a mostly-ASCII Windows control that's being used or something.

Viewing current character set or collations

Database level:

```
SELECT @@character_set_database, @@collation_database;
```

"ci" === "case insensitive", e.g. utf8_general_ci

Table-level collations:

```
SHOW TABLE STATUS from botland;
```

Column-level collations:

```
SHOW FULL COLUMNS FROM users;
```

Choosing a character set

The future of character sets and collations is apparently this:

```
DEFAULT CHARACTER SET utf8mb4
DEFAULT COLLATE utf8mb4_unicode_ci
```

This allows for up to 4 bytes for unicode characters so that you can support something like emojis. For all "normal" characters, it will still take the usual 3 bytes.

12:22 Kfirba2: @Adam13531 1 IMPORTANT thing to note here is that your INDEXED columns under utf8mb4 charset CAN NOT exceed 191 characters UNLESS you enable innodb_large_prefix
12:27 syntonic8: I had an issue with this too. I upgraded MySQL and it turned this flag on. If you're using a new(est) version I believe it's on by default
12:27 syntonic8: Yeah version >= 5.7.7 is on by default

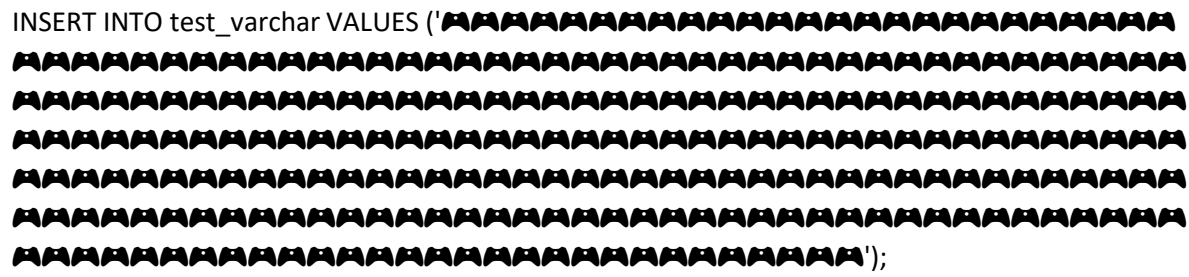
To check if it's on, you can do this:

```
SHOW GLOBAL VARIABLES LIKE 'innodb_%prefix%';
```

To create a database with this, use the following command:

```
CREATE DATABASE IF NOT EXISTS test_collation DEFAULT CHARACTER SET utf8mb4
DEFAULT COLLATE utf8mb4_unicode_ci;
```

I did a bunch of tests to see what would happen when you change character sets from utf8 to utf8mb4, and I couldn't find any errors. It seems that VARCHAR(X) will *always* be able to store X characters regardless of changing character sets. Here's some test SQL, but keep in mind that you can't just test all of this in CMD Prompt because it shows "?" in the database.

```
CREATE DATABASE IF NOT EXISTS test_collation DEFAULT CHARACTER SET utf8mb4 DEFAULT
COLLATE utf8mb4_unicode_ci;
USE test_collation;
CREATE TABLE test_varchar(str VARCHAR(255));
DESCRIBE test_varchar;
INSERT INTO test_varchar VALUES ('

');
ALTER TABLE test_varchar CONVERT TO CHARACTER SET utf8 COLLATE utf8_general_ci;
```

If you wanted to create the database as UTF8 first, you could do this:

```
CREATE DATABASE IF NOT EXISTS test_collation4 DEFAULT CHARACTER SET utf8 DEFAULT
COLLATE utf8_general_ci;
```

Bottom line: none of this makes any sense and I'm probably just testing this wrong thanks to charset-connection.

~~Based on what Chrizzmeister said, if you have a table with character set utf8 and a column with VARCHAR(255), you'll be able to save 255 utf8 characters (which makes sense). However, if you change the character set to utf8mb4, then that same column can only fit at most 191 characters since they can take up to 4 bytes (because the maximum key length if 767 bytes, so that allows 255 3-byte characters or 191 4-byte characters).~~

~~Note that VARCHAR is still telling you the number of characters you can store, *not* bytes, and that by default, just specifying "VARCHAR" alone will be 191 characters in utf8mb.~~

~~Looked into more information about this (reference).~~

~~InnoDB encodes fixed-length fields greater than or equal to 768 bytes in length as variable-length fields, which can be stored off-page. For example, a CHAR(255) column can exceed~~

~~768 bytes if the maximum byte length of the character set is greater than 3, as it is with utf8mb4.~~

NOW()

"NOW()" will return the current timestamp in the server's timezone, but you likely shouldn't store timezone-based data in the database. For example, in Bot Land, I store the creation_date of users in UTC. I am in PST/PDT which is 8 or 9 hours behind UTC, so if I create a user and run "select * from users where creation_date < NOW()" then I won't get the new user in the results.

Instead, either use UTC_TIMESTAMP() or call "SET time_zone = timezone;", which will change NOW(), CURTIME(), etc. but won't change DATE, TIME, etc.

Full instructions on how to set timezone:

- Make sure you have super privilege (e.g. log in as root)
- If you want to check your current offset:
  - SELECT @@global.time_zone;
- Change the timezone to UTC:
  - SET GLOBAL time_zone = '+00:00';
- Restart any sessions you have (e.g. command-line, Metabase, MySQL Workbench)

Alternatively, if you don't want to set the timezone globally, you can do so just for the session or via my.cnf:

> HiDeoo: Adam13531 On your prod server if you don't want to use the query to set the timezone, here's the config option to set it in my.cnf https://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_time_zone (you can even set it as a CLI argument with --default-time-zone)

Installation on Linux

- Install
  - sudo apt-get install -y mysql-server
  - You have to manually type a database root password in the installation.
- Configure MySQL to be accessible on all addresses
  - sudo vim /etc/mysql/my.cnf
  - Find "bind-address" and change it to "0.0.0.0".
- Restart MySQL service
  - sudo service mysql restart
- If you haven't created a user, then create one. Note: the '%' lets you access the database from any host.
  - CREATE USER 'TomAto'@'%' IDENTIFIED BY 'password';
  - GRANT ALL PRIVILEGES ON test_db.* TO 'TomAto'@'localhost';
- If you *have* created a user, then you need to make sure the host is set correctly.
  - Manual, easy way:
    - GRANT ALL PRIVILEGES ON test_db.* TO 'TomAto'@'localhost' IDENTIFIED BY 'password';
    - Note: this flushes privileges on its own.
    - Note: without typing "IDENTIFIED BY", you'll have no password set, but *only* on whichever host you specified. MySQL lets you set up users/passwords per-host, so you could have a weak password when on localhost and a strong password outside of it if you wanted.
  - Manual, hard way:
    - mysql -u root -p
    - use mysql;

- UPDATE user SET host='%' WHERE user='TomAto' AND host='192.168.1.17';
- FLUSH privileges;
- Without flushing privileges, you'll get an error message about: Host '192.168.1.17' is not allowed to connect to this MySQL server

Executing arbitrary commands from the command line

Just use "-e" and surround your command in quotation marks.

mysql -D botland -u root --password=password -e "DROP DATABASE IF EXISTS %localDbName%;"
mysql -D botland -u root --password=password -e "CREATE DATABASE %localDbName%;"
mysql -D botland -u root --password=password -e "GRANT ALL PRIVILEGES ON %localDbName%.* TO 'Adam'@'localhost';"

Saving/restoring a MySQL database (reference)

(keywords: export / import)

- Note: mysqldump is needed to back up the database and I don't know whether it comes from mysql-server or mysql-client.
- Backup: mysqldump -u root -p[root_password] [database_name] > dumpfilename.sql
- Restore: mysql -u root -p[root_password] [database_name] < dumpfilename.sql
    - Note: you may find it better to do this to ensure you don't have any tables that shouldn't be there:
        - DROP DATABASE [database_name];
        - CREATE DATABASE [database_name];

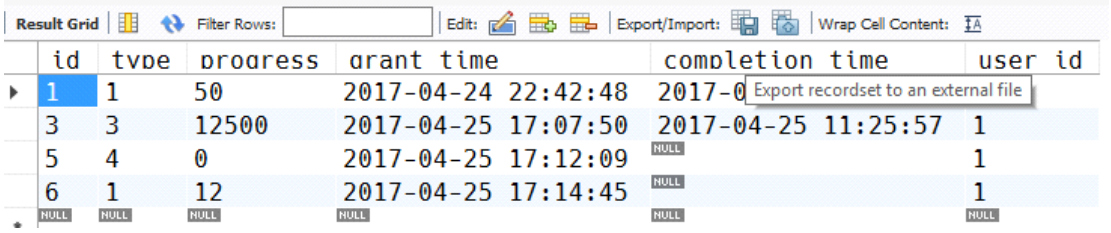
Example with Bot Land:
On Overseer: probably "delete from replays;" so that we don't get a bunch of journals that we don't care about
On Overseer: mysqldump -u root -h db.bot.land -p botland > dumpfilename.sql
On local machine: scp -i D:\Code\JavaScript\learning\aws\Firstkeypair.pem admin@50.112.22.133:/home/admin/dumpfilename.sql ./
On local MySQL: CREATE DATABASE botland2;
On local command line: mysql -u root --password=password botland2 < dumpfilename.sql

Saving/restoring a single MySQL table

I did this using MySQL Workbench.

Saving

- Click the "Export recordset to an external file" button



- Export to a ".sql" file

Restoring

- mysql -u root --password=password -D botland < file.sql

Basic commands

- Go look at the PostgreSQL note. If anything differs, I will list it here.
- Drop/delete a user: DROP USER 'Adam'@'localhost';
- Selecting when tables have tons of columns - just replace the semicolon at the end of a SELECT statement with "\G".
    - SELECT * FROM user\G
- Selecting columns concatenated together: use CONCAT
    - SELECT concat(name, ' (', id, ')') AS full_user FROM users;
- Altering a table:
    - Add a column:
        - ALTER TABLE users ADD COLUMN num_unopened_salvage INT UNSIGNED NOT NULL DEFAULT 5;
        - ALTER TABLE users ADD COLUMN copy_of_last_attacked_defense text DEFAULT NULL;
    - Remove a column
        - ALTER TABLE users DROP COLUMN testeroni;
            - You may need to remove any foreign key constraints on this first (the name of the foreign key can be found via "SHOW CREATE TABLE users") (reference)
                - ALTER TABLE mytable DROP FOREIGN KEY mytable_ibfk_1 ;
    - Change type of a column
        - ALTER TABLE defense_copies MODIFY journal LONGTEXT;
    - Change name of column (keyword: "rename") (reference)
        - ALTER TABLE `xyz` CHANGE `oldname` `newname` INT UNSIGNED NOT NULL DEFAULT 0;
            - Note: the type, "not null" (if you want that), and the default (if you want one) are all required since these properties aren't merged! I.e. there's no way to *just* rename a column; you have to specify everything about that column.
    - Drop the table
        - DROP TABLE users;
    - Changing character set and collation
        - ALTER TABLE Tablename CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;
- JOIN syntax

```
SELECT matches.attacker_id,
    matches.start_time,
    attacker.name AS attacker,
    defender.name AS defender
FROM `ongoing_matches`
INNER JOIN `matches` ON `ongoing_matches`.`match_id` = `matches`.`id`
INNER JOIN `users` AS attacker ON `matches`.`attacker_id` = `attacker`.`id`
INNER JOIN `users` AS defender ON `matches`.`defender_id` = `defender`.`id`;
```

- Replacing substring/text in the database. For example, I had to replace the word "towers" with "chips" in a JSON blob, so I ran this query:
    - UPDATE users SET serialized_defense = REPLACE(serialized_defense, 'towers', 'chips')
- Searching based on substrings
    - SELECT * FROM users WHERE name LIKE 'startswith%';
    - SELECT * FROM users WHERE name LIKE '%endswith';
    - SELECT * FROM users WHERE name LIKE '%substring%';
    - SELECT * FROM users WHERE name LIKE '%contains_backslash__%';
        - ^You need to write a backslash four times to have it resolve to a single backslash (reference)
- Getting the number of rows in a table
    - SELECT COUNT(*) FROM users;

- HAVING
  - This is basically a WHERE clause that works after the group has done its job.
  - Example

```
SELECT users.name,
    count(*) AS cnt
FROM `users`
INNER JOIN `bot_bays` ON `bot_bays`.`user_id` = `users`.`id`
GROUP BY users.id HAVING cnt > 2
ORDER BY cnt DESC;
```

- You can use variables to make some queries easier:
  - SET @amount = 5; SELECT * FROM users WHERE money > @amount;
- Find non-distinct/non-unique rows in a database

  This query finds all users who have the same name:

```
SELECT id,
    name,
    COUNT(name) AS the_count
FROM users
GROUP BY name HAVING the_count > 1;
```

Case statements (reference)

If you ever have something like a "status" column that's saved as an integer but represents something like "offline"/"online", you could use a "case" statement:

```
SELECT user_id,
    CASE
      WHEN status = 0 THEN 'offline'
      WHEN status = 1 THEN 'online'
      ELSE 'unrecognized'
    END AS status_name
FROM users;
```

Alternatively, if you really just have a couple of values, SUBSTRING_INDEX may be better:

```
SELECT SUBSTRING_INDEX('offline,online', ',', status) from users;
```

Truncating vs. deleting

Suppose you have two tables:

replays
- id
- journal

matches
- id
- replay_id (where this is set up with CONSTRAINT `matches_replay_id_foreign` FOREIGN KEY (`replay_id`) REFERENCES `replays` (`id`) ON DELETE SET NULL)

These are set up so that deleting a replay will null out the "replay_id" in "matches". However, if you try truncating the entire "replays" table, then you'll get this error:

```
ERROR 1701 (42000): Cannot truncate a table referenced in a foreign key constraint (`botland`.
`matches`, CONSTRAINT `matches_replay_id_foreign` FOREIGN KEY (`replay_id`) REFERENCES
`botland`.`replays` (`id`))
```

There are two ways you can deal with this (where one is clearly better):

1. [bad] Disable foreign key constraint checks or delete that particular foreign key constraint,

perform your truncation, then reinitialize the FK. This is bad because you will lose data integrity while the FKs are absent. For example, if you did that with the tables above, then the replay_id in "matches" would never get nulled out.

2. [good] "DELETE FROM replays;" - this is good because it abides by all of your constraints and maintains the next primary key value.

Procedures (reference)

Simple procedure with one argument:

```
DELIMITER //
CREATE PROCEDURE `getActiveUsers` (IN numDays INT)
BEGIN
SELECT name,
    last_login_date
FROM users
WHERE last_login_date > NOW() - INTERVAL numDays DAY;
END //
DELIMITER ;
```

Note: the DELIMITER statements change what string is used at the end of a line from a semicolon; you need this when you use multiple SQL statements in a procedure so that the parser doesn't end at the first semicolon it sees (which would be after the "WHERE" clause above). This is a feature of the MySQL *client*, not the server as far as I understand, so this doesn't work from something like knex or my-cli.

Calling the procedure:

```
CALL getActiveUsers(2);
```

Updating the procedure:

To update a procedure, I think you have to just drop it first and then update it.

```
DROP procedure IF EXISTS `getActiveUsers`;
```

Primary keys

Don't use IP addresses as primary keys! Your primary keys should almost always be incrementing integers or UUIDs.

You can have a primary key that consists of multiple other keys (AKA a primary composite key or primary compound key) (reference). If you do this, you don't necessarily need an autoincrementing "id" field for your table, but it can be helpful for manually deleting entries. If you *do* decide to have an autoincrementing "id" field, then you very likely shouldn't index it since that will end up taking much more space than just the 4-8 bytes that the integer itself needs.

Note that composite indexes can be used as long as you're selecting based on the first N columns, where N is any number up to the number of columns used in the composite index. E.g. if you have an index on "user_id" and then "item_id" and search just using "user_id", it will use the index.

According to this StackOverflow post, primary keys are always indexed.

Misc notes

Pagination

Read this: http://use-the-index-luke.com/no-offset

TL;DR: don't use OFFSET. Prefer something like this:

```
SELECT ...
 FROM ...
WHERE ...
  AND id < ?last_seen_id
ORDER BY id DESC
FETCH FIRST 10 ROWS ONLY
```

Transactions, locks, and isolation

Isolation Level - https://dev.mysql.com/doc/refman/5.5/en/set-transaction.html - note, you don't need to change this by default according to GsarGaming: "No you dont [need to modify the transaction isolation level]. Leave it at defaut. It is by default ACID." EveryJuan supports this by saying "The default isolation level is REPEATABLE READ, which requires a transaction commit for the lock to be broken, so you were right overall".

SELECT … FOR UPDATE (reference)
In short, this will lock the row so that nothing else can modify it. For more details, read the reference. What this means is that suppose you have this flow:

Transaction 1:
- SELECT FOR UPDATE * FROM users WHERE id = 1;
- <delay of 50 hours>
- <do something with the user>
- <commit or rollback transaction>

Transaction 2 (which runs AFTER the "SELECT" above but BEFORE the delay is finished)
- UPDATE users SET inventory = 'blah' WHERE id = 1;
- <commit or rollback>

Transaction 2 will wait for 50 hours before completing, the only caveat being that a timeout will likely stop it from finishing the whole 50-hour delay. The reason Transaction 2 waits is because there is no deadlock, otherwise it would error out immediately and need to be rerun at some point.

Furthermore, let's say you issued a plain old UPDATE outside of a transaction, e.g. "UPDATE users SET inventory = 'blah' WHERE id = 1;". That would respect the lock and wait until Transaction 1 is finished.

However, a "SELECT * FROM USERS;" or "SELECT * FROM USERS WHERE id = 1;" would work immediately. These are the consistent reads that the documentation refers to when it says "Consistent reads ignore any locks set on the records that exist in the read view.".

Do note, "SELECT * FROM USERS WHERE id = 1 LOCK IN SHARE MODE;" would wait for the lock to be freed.

Other notes about SELECT … FOR UPDATE:
- It is ONLY for transactions. It is not an error to put "SELECT * FROM users FOR UPDATE;", but it doesn't do anything.

SELECT … LOCK IN SHARE MODE
As mentioned above, this will try to acquire a lock that prevents updates (but allows reads still). I don't think I've run into a scenario where I've organically wanted to use this, but I imagine the scenario would be something like this contrived transaction:
- Read 'money' from the 'users' table (use LOCK IN SHARE MODE)
- Read 'inventory_size_remaining' from the 'items' table (use LOCK IN SHARE MODE)

This would let you know if you have enough money and inventory space to buy an item, but it wouldn't actually update anything. Note: when you actually go to buy an item, it would need to lock the appropriate rows FOR UPDATE.

Locks (reference)

Locks are acquired explicitly with "SELECT" when using "SELECT … FOR UPDATE" or "SELECT … LOCK IN SHARE MODE". However, they're implicitly obtained in "UPDATE … WHERE …" and "DELETE FROM … WHERE …" statements. This means that you're actually getting a lock if you do this:

```
START TRANSACTION;
INSERT INTO users(name) VALUES('Adam');    # Lock is obtained
INSERT INTO items(name) VALUES('Starter Sword'); # Lock is obtained
COMMIT;
```

However, I think that it doesn't really matter whether you got a lock or not above since it's a transaction, so it's treated as an atomic unit.

Table-level locks (reference)

If you want to lock an entire table, you can do something like this:

```
LOCK TABLES users WRITE;
-- Do stuff here that will now prevent other queries from being able to write to "users"
UNLOCK TABLES;
```

If locking multiple tables, make sure to do so in a single command:

```
LOCK TABLES users WRITE, accounts READ;
```

Note that this level of lock can affect transactions! I wrote a note about this here, and the official docs talk about it here.

Deadlocks (reference - actually very helpful)

Deadlocks can occur even from just inserting/deleting a single row (because of how they work underneath with obtaining locks). They are not fatal, and the documentation suggests that the application retry the transaction.

I am going to try to always acquire locks in the same order, that way I shouldn't run into a deadlock.

Subqueries

Using data from one table to populate another

```
INSERT INTO users (name, age, nickname)
SELECT name,
    18,
    name
FROM other_users;
```

Note: you can specify any immediate values you'd like in the subquery.

You can use this to duplicate rows in a table easily:

```
INSERT INTO users (name, age)
SELECT name,
    age
FROM users
WHERE id = 5;
```

An alternative to subqueries

This can be a helpful way to avoid using subqueries. For example, here I want to select the number of times outcome is 2 and divide it by the number of times outcome was 2.

```
SELECT
   computer_level,
   SUM(outcome = 2) AS def_wins,
   SUM(outcome = 0) AS def_losses,
   (SUM(outcome = 2) / SUM(outcome = 0)) AS def_win_ratio
FROM
   matches
GROUP BY computer_level;
```

Indexes

Basics

There's no real need to use an index until you've got thousands of entries (e.g. 10k+). Remember that there's no guarantee that the storage engine uses your index (because it can determine that sequential searching may be faster).

How to use them (reference)

```
CREATE TABLE users (name TEXT, INDEX(name(5)));
```

Note: the number in parentheses indicates how long of a prefix you're going to index on. It can be higher or lower than the number of characters in each individual name in the table. The lower it is, the less hard drive space the index will use, but the less performant it will be.

Suppose you created the index with length 1 and then added 'Adam' to your table 3 times. You could run "EXPLAIN" on your query as below:

```
mysql> EXPLAIN SELECT * FROM users WHERE name = 'adam';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	red	rows	filtered	Extra
1	SIMPLE	users	NULL	ref	name	name	6	const	3	100.00	Using where

For information on how to interpret the above, look at this reference.

Also, if you explicitly want to IGNORE a particular index, you can specify it just after the "SELECT" but before the "WHERE" clause like this:

```
SELECT *
FROM matches
IGNORE INDEX (matches_defender_and_replay_id_index)
WHERE defender_id = 2
  AND replay_id IS NOT NULL\G
```

Troubleshooting

Password expired (or you just want to change your password)

"ERROR 1820 (HY000): You must reset your password using ALTER USER statement before executing this statement."

Just set the password again using the root account:

```
SET PASSWORD FOR 'Adam'@'localhost' = PASSWORD('password');
```

Note that if this happens while you're on root, you can do this (reference):

```
SET PASSWORD = PASSWORD('password');
```

Warnings

If you ever run a query and you see something like "1 row in set, 2 warnings (0.00 sec)", then do "SHOW WARNINGS" to see the warnings.

Note: code 1003 is just the code for "EXPLAIN EXTENDED" (reference)