**Class**: Introduction to Databases with SQL

**Official notes:** <https://cs50.harvard.edu/sql/notes/0/>

Querying

* Why databases?
  + Good way to organize data
  + Advantages over spreadsheets: Scale, update capacity (many updates a second), speed
* Which DBMS (database management system) should we choose?
  + Consider cost, amount of support, weight
  + This class uses SQLite and MySQL and PostgreSQL
* Functions
  + Sqlite3 database\_name.db – open database
  + SELECT .. FROM ..
  + LIMIT
  + WHERE
    - != is not equal
    - IS NULL/ IS NOT NULL
    - BETWEEN … AND …
    - LIKE “%pattern%” (unknown length) or “t\_\_” (known length)
  + ORDER BY ASC/DESC
  + AVG, MAX, MIN, SUM, COUNT
  + .tables – show all tables
  + .schema – show tables/triggers/views/indices in database

Relating

* Relational database – contains tables with relationships
* Entity relationship diagrams



* Keys
  + Primary key – unique identifier
  + Foreign key – primary key of another table, useful for relations
* Subqueries – can stack queries inside queries (using parentheses)
* Functions
  + IN (desired list)
  + JOIN … ON … (must use common ID)
    - LEFT JOIN, RIGHT JOIN, FULL JOIN
  + GROUP BY – useful for finding counts, averages, etc.
* Sets
  + For getting INTERSECT, UNION, EXCEPT from multiple (selects)

Designing

* Normalizing – reduce redundancies in the database by separating entities into their own tables
* CREATE TABLE table\_name (var1 TYPE, …)
  + SQLite TYPE: Null, Integer, Real, Text, Blob (large picture/video/audio)
  + SQLite has type affinity – will try to convert ‘wrong’ type to column preference
* Table constraints
  + PRIMARY KEY(var)
  + FOREIGN KEY(var) REFERENCES table2(var2)
* Column constraints
  + CHECK (ex. CHECK(amount > 0) )
  + DEFAULT (ex. DEFAULY CURRENT\_TIMESTAMP)
  + NOT NULL
  + UNIQUE
* Altering tables
  + ALTER TABLE table RENAME TO table2
  + ALTER TABLE table ADD COLUMN var1 TEXT;
  + ALTER TABLE table RENAME COLUMN var1 TO var2
  + ALTER TAVLE table DROP COLUMN var1

Writing

* Inserting data
  + INSERT INTO table (var1, var2, var3) VALUES (‘1’,’2’,’3’)
  + Multiple rows
  + INSERT INTO table (var1, var2, var3)   
     VALUES   
     (‘1’,’2’,’3’),

(1’,’2’,’3’);

* Working with CSVs
  + .import --csv --skip 1 filename.csv table\_name
    - skip 1 – remove header
    - May be useful to save to temporary table to deal with primary keys
* Deleting data
  + DELETE FROM table WHERE …
  + If deleting foreign key, can run into errors
  + Specify in table:
    - FOREIGN KEY() REFERENCES table() ON DELETE \*\*\*
    - \*\*\* = RESTRICT (error)
    - \*\*\* = NO ACTION
    - \*\*\* - SET NULL (set to null)
    - \*\*\* Set DEFAULT (set default instead of NULL)
    - \*\*\* CASCADE (deletes referenced id)
* Updating data
  + UPDATE table SET var = ‘value’ WHERE …
  + Good for typos

Viewing

* Views allow for easier visualization of separated data as a ‘virtual table’
* Creating views
  + CREATE VIEW view\_name AS SELECT … FROM …
* Temporary views
  + CREATE TEMPORARY VIEW view\_name AS …
  + Lasts for connection
* Common Table Expression (CTE)
  + Lasts just for one query
  + WITH cte\_name AS () SELECT …;
* Uses
  + Simplifying: Use joins to simplify separated data
  + Aggregating: Compute statistics (ex. Average) to look at overall data
  + Partitioning: look at subsets of data using ‘where’
  + Securing: limit access to certain data by limiting what others can see (ex. Analysts)
* Soft deletions
  + Instead of permanently deleting data, add a ‘deleted’ column to table
    - ALTER TABLE table\_name ADD COLUMN deleted INTEGER DEFAULT 0
  + Perform soft delete
    - UPDATE table\_name   
      SET deleted = 1 WHERE…
  + Create view for current items only
    - CREATE VIEW view\_name   
      AS SELECT (vars) FROM table WHERE deleted = 0;
  + Create a trigger that auto-“deletes” when you try to delete on the view
    - CREATE TRIGGER delete   
      INSTEAD OF DELETE ON view  
      FOR EACH ROW  
      BEGIN  
       UPDATE table SET deleted = 1  
       WHERE id = OLD.id  
      END
  + Can also create a trigger to insert data into a view (when id already exists)
    - CREATE TRIGGER insert\_when\_exists  
      INSTEAD OF INSERT ON view  
      FOR EACH ROW  
      WHEN NEW.id\_num IN (SELECT id\_num FROM table)  
      BEGIN  
       UPDATE table  
       SET deleted = 0  
       WHERE id\_num = NEW.id\_num  
      END
  + Create trigger to insert data into a view (when is is new)
    - CREATE TRIGGER insert\_when\_new  
      INSTEAD OF INSERT ON view  
      FOR EACH ROW  
      WHEN NEW.id\_num NOT IN (SELECT id\_num FROM table)  
      BEGIN  
       INSERT INTO table (var1, var2, var3)  
       VALYES (NEW.var1, NEW.var2, NEW.var3)  
      END

Optimizing

* Optimizing runtime and space
  + .timer on (look at runtime)
* If you run a query very often, you can use an index to speed it up
  + CREATE INDEX index\_name ON table (var);
  + CREATE INDEX index\_name ON (var1, var2);
  + Covering index – all info for query can be found in index
* Indexes occupy more space (balance tree or B-Tree)
  + With many nodes: root with children (branches, leaves)
  + Creates a copy that you can order/sort and search more efficiently
    - Since copy is large, breaks up copy into sections
* Partial index
  + Index with subset (hopefully the most queried data, ex. Most recent)
* Vaccum
  + Run VACUUM to free up unused space (check with du -b database\_name.db)
* Concurrency
  + Handling multiple queries at the same time
  + Some transactions/changes are multi-part and you do not want them broken up by another user
* Transactions
  + Atomicity (can’t be broken down)
  + Consistency (cannot break db constraints)
  + Isolation (multiple user transactions cannot interfere with one another)
  + Durability (if db fails, all data changes by transctions will remain)
* Transaction function
  + BEGIN TRANSACTION;  
    UPDATE table SET balance = balance + 10 WHERE…;  
    UPDATE table Set balance = balance – 10 WHERE…;  
    COMMIT
  + If there is an issue, you can ROLLBACK a transaction
* Race conditions
  + Race conditions occur when multiple entities try to simultaneously access and change database, causing inconsistencies -> need to process each transaction in isolation
  + You can lock
    - UNLOCKED default
    - SHARED multiple users
    - EXCLUSIVE only one action (even reading) at a time
    - Syntax: BEGIN EXCLUSIVE TRANSACTION;

Scaling

* MySQL and PostgreSQl are database servers – run on dedicated hardware – resulting in faster queries
* Connecting to MySQL
  + mysql -u root -h 127.0.0.1 -P 3306 -p
  + User -u, want to connect to root (admin connection)
  + 127.0.0.1 is the address of own computer
  + 3306 is the port you want to connect to (default)
  + -p you want a password
* Navigating MySQL
  + SHOW DATABASES;
  + CREATE DATABASES `my\_db`;
  + USE `my\_db`;
* Data types
  + Integers
    - Many integer types: TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
  + Text
    - CHAR(2) of set length, and VARCHAR(32) of variable length (with max)
    - TEXT is for longer chunks of text : TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT, BLOB (binary)
  + Text, other
    - ENUM – predefined options ex. ENUM(‘red’, ‘green’, ‘orange’) NOT NULL
    - SET – like enum, but can have multiple values per row
  + Date
    - DATE, YEAR, TIME, DATETIME, TIMESTAMP
  + Real
    - FLOAT, DOUBLE PRECISION, DECIMAL(5,2)
      * Decimal is of defined precision: first number is number of positions, second number is numbers after the decimal
* Using tables
  + CREATE TABLE `table` (  
     `id` INT AUTO\_INCREMENT   
     PRIMARY KEY(`id`)  
    );
  + SHOW TABLES;
  + DESCRIBE `table`;
* Altering tables
  + ALTER TABLE `table` MODIFY `var` ENUM(…) NOT NULL;
  + ALTER TABLE `table` ADD COLUMN …;
* Stored procedures
  + Ways to automate SQL statements
  + Useful to change delimiter so we can use ; inside our procedure: delimiter //
  + CREATE PROCEDURE `my\_procedure`  
    BEGIN  
     SELECT … FROM … WHERE …   
    END//
  + After, change delimiter back: delimiter ;
  + Now, call the procedure: CALL my\_procedure();
* Stored procedures with parameters
  + Can have procedure with an input (parameter) – like a function
  + Can have multiple actions within procedure
  + delimiter //  
    CREATE PROCEDURE `sell` (IN `soldIid` INT)  
    BEGIN  
     UPDATE `collections` SET `deleted` = 1  
     WHERE `id` = `sold\_id`;  
     INSERT INTO `transactions` (`title`,`action`)  
     VALUES (  
     (SELECT `title` FROM `collections` WHERE `id`=`sold\_id`),   
     ‘sold)  
    END //  
    delimiter ;
  + Can now call eith CALL `sell`(2);
  + Stored procedures are compatible with if statements, loops (for/while), etc.
* PostgreSQL
  + Connecting
    - psql postgresql://postgres@127.0.0.1:5432/postgres
  + View databases
    - \l
  + Creating databases
    - CREATE DATABASE “my\_db”;
  + Connecting to database
    - \c “my\_db”
  + Look at tables
    - \dt
    - \d “table\_name”
  + Creating tables
    - CREATE TABLE “my\_table” (“id” SERIAL, PRIMARY KEY(“id));
    - Serial means auto-increment integer
  + Datatypes
    - Integers
      * SMALLINT, INT, BIGINT
      * Serial
    - Text
      * VARCHAR(32)
    - ENUM
      * CREATE TYPE “swipe\_type” AS ENUM (‘enter’, ‘exit’)
    - Time
      * TIMESTAMP, DATE, TIME, INTERVAL
      * Get current timestamp using now()
    - Numbers
      * MONEY
      * NUMERIC(precision, scale) - instead of MySQL’s decimal
  + Exiting
    - \q
* Scaling techniques
  + Optimizing queries
  + Vertical scaling – increasing a server’s computing power
  + Horizontal scaling – distributing load across more servers
    - Replication – keeping copies of a database on multiple servers
      * Single-leader: single server handles incoming writes and copies to other servers
      * Multi-leader: multiple servers receive updates (higher complexity)
      * Leaderless..
    - Read replica – copy of a database from which data may only be read (follower)
  + Synchronous vs asynchronous
    - Synchronous – lead will wait for follower to get/process data before another action (slower)
    - Asynchronous – leader sends data to follower, resumes next action (less redundancy)
  + Sharding
    - Splitting up data across multiple servers in an organized manner (ex. Alphabetical)
    - Want to have about equal distribution of action (no hotspot)
    - No/less replication – if one system goes down, all do – need to incorporate forms of replication
* Access control with MySQL
  + Create a new user
    - CREATE USER ‘myname’ IDENTIFIED BY ‘password’
    - Can log-in as previous using out new password
  + Show databases
    - SHOW DATABASES;
    - Likely can’t see anything, since not admin
  + Grant access (from admin)
    - GRANT SELECT ON database.table To myname;
    - REVOKE SELECT ON database.table FROM myname;
  + Now, access our (virtual) table
    - USE database;
    - We can only see what is permitted
* SQL injection attack
  + Maliciously inserting SQL database to access data/features that should be inaccessible
  + Prepared statements cleans up input to prevent against injection attacks