Computer Science 202 2003

Introduction to the
Structured Query Language (SQL)

Objectives

- To learn the SQL data definition: CREATE
- To learn the basic SQL data management language: INSERT, UPDATE, and DELETE
- To learn the basic SQL data querying language: SELECT

SQL Basics

- What is SQL?
  - Structured Query Language
- SQL Syntax
- Using SQL

SQL Syntax

- SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (";"). Which tokens are valid depends on the syntax of the particular command.
- A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or constant), or a special character symbol. Tokens are normally separated by white space (space, tab, new line)
- Additionally, *comments* can occur in SQL input. They are not tokens, they are effectively equivalent to white space.

Key Words & Identifiers

- *key words* - Tokens such as SELECT, UPDATE, or VALUES are examples of, words that have a fixed meaning in the SQL language.
- *Identifiers* - They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called “names”.

Key Words & Identifiers (II)

- Identifier and key word names are case insensitive. The following statements are all equivalent
  - UPDATE MY_TABLE SET A = 5;
  - uPDaTE my_TabLE SeT a = 5;
- A convention often used is to write key words in upper case and names in lower case, e.g:
  - UPDATE my_table SET a = 5;

Key Words & Identifiers (III)

- Quoting
- *quoted identifier* – an identifier created when the token is enclosed in quotes (“”)
- Can be used to use built in RESERVED words as identifiers
- SELECT “select” FROM my_table;

Comments

- Two type of comments can be used , much as in C and other similar languages
- For single line comments, a double dash can be used. This treats everything following the dashes until the end of line as a comment
  - **`-- This is a standard SQL92 comment`**
- Alternatively C-style block comments can be used for commenting multiple lines.
  -
    ```
    /* multiline comment
       begins with /* and extends
       to the matching
       occurrence of */. */
    ```
  - Nesting is allowed in the SQL99 spec.

SQL - TABLES

- Tables hold the Data within a relational database
- Tables are used to build the relationships
- Objectives:
  - Creating Tables
  - Modifying Tables
  - Deleting Tables

Creating Tables

CREATE TABLE -- define a new table

CREATE TABLE *table_name* (
{ *column_name data_type* [DEFAULT *default_expr*]
[CONSTRAINT [NOT NULL | NULL | UNIQUE | PRIMARY KEY]]
[, ...]]
}

CREATE TABLE example

```
CREATE TABLE films
( code CHARACTER(5) PRIMARY KEY,
  title CHARACTER VARYING(40) NOT NULL,
  date_prod DATE,
  kind CHAR(10),
  length INTEGER );
```


Inserting Information

- INSERT -- create new rows in a table

INSERT INTO *table* [(*column* [, ...])]
VALUES ({ *expression* [, ...])

Example Using INSERT

INSERT INTO
student (studID, lastname, firstname)
VALUES (95I0563 , ‘Irwin’ , ‘Barry’);

Removing Information

- DELETE -- delete rows of a table

DELETE FROM *table* [WHERE *condition*]

DELETE FROM
student WHERE studID=‘95I0563’ ;

Updating information

- UPDATE -- update rows of a table
- UPDATE [ONLY] *table* SET *column* = *expression* [, ...] [FROM *fromlist*] [WHERE *condition*]

UPDATE films SET kind = 'Dramatic'
WHERE kind = 'Drama';

Retrieving Information

- SELECT -- retrieve rows from a table or view
- **Command Summary**
  - SELECT * | *expression* FROM *from_item* [, ...]
    [WHERE *condition*]
    [GROUP BY *expression*]
    [ORDER BY *expression* [ASC | DESC]
    [LIMIT { *count* | ALL }]

Using the SELECT statement

- Basic Use
- SELECT * FROM EMP;

Using SELECT (2)

- Table aliasing

- SELECT f.title, f.did, d.name, f.date_prod, f.kind FROM distributors d, films f WHERE f.did = d.did

SQL Arithmetic

- One can use arithmetic functions within a SQL SELECT statement

Logical Functions within SQL

- Standard Boolean logic functions can be used within queries
- NOT is a particularly useful function and can be used to negate other operations
- OPERATIORS
  - AND
  - OR
  - NOT

Logical Functions within SQL

A	B	A **AND** B	A **OR** B
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
TRUE	NULL	NULL	TRUE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

See PostgreSQL Function Ref Section 6.1

Logical Functions within SQL

A	NOT A
TRUE	FALSE
FALSE	TRUE
NULL	NULL

See PostgreSQL Function Ref Section 6.1

Comparison Operators

Operator	Description
<	Less Than
>	Greater Than
<=	Less Than or Equal
>=	Greater Than or Equal
=	Equal
!= or <>	Not Equal

See PostgreSQL Function Ref Section 6.2

Comparison Operators

- In addition to the comparison operators, the BETWEEN construct is available.

a BETWEEN *x* AND *y*

is equivalent to

a >= x AND a <= y

a NOT BETWEEN x AND y

is equivalent to

a < x OR a > y

Comparison Operators

- Comparison operations are BINARY
  - Operate on TWO operands
  - a < B --- is valid
  - A < B < C --- is NOT valid
    - A Boolean cannot be > or < than C
- To check whether a value is or is not null, use the constructs
  - *expression* IS NULL
  - *expression* IS NOT NULL

Testing Boolean Conditions

- Boolean values can also be tested using the constructs
  - *expression* IS TRUE
  - *expression* IS NOT TRUE
  - *expression* IS FALSE
  - *expression* IS NOT FALSE

Aggregate Function Operators

- MIN()
  - minimum value of *expression* across all input values
- MAX()
  - maximum value of *expression* across all input values
- SUM()
- COUNT(*) - counts ALL input values
- COUNT(expression) – counts only non NULL
- AVG()

Pattern Matching

- **LIKE**
  - *string* LIKE *pattern*
  - *string* NOT LIKE *pattern*
- *If pattern does **not** contain wildcard signs then LIKE acts like the equals operator.*
- *An underscore (_) in pattern matches any single character*
- *A percent sign (%) matches any string of zero or more characters.*

Pattern Matching Examples

Some examples:

- 'abc' LIKE 'abc' *true*
- 'abc' LIKE 'a%' *true*
- 'abc' LIKE '_b_' *true*
- 'abc' LIKE 'c' *false*
- 'abc' LIKE 'c%' *false*
- 'abc' LIKE '%b_' *true*

Advanced Pattern Matching

- **SIMILAR TO**
- **Part of the newer SQL99 specification. LIKE is definded in SQL92**
- *string* SIMILAR TO *pattern* [ESCAPE *escape-character*]
- *string* NOT SIMILAR TO *pattern* [ESCAPE *escape-character*]

Advanced Pattern Matching II

Examples:

- 'abc' SIMILAR TO 'abc' *true*
- 'abc' SIMILAR TO 'a' *false*
- 'abc' SIMILAR TO '%(b|d)%' *true*
- 'abc' SIMILAR TO '(b|c)%' *false*

SQL arithmetic

- Examples

SELECT studentid, pracmark, testmark, (pracmark +testmark) AS classmark FROM
studentdata

ORDER BY

- ORDER BY *expression* [ASC | DESC | USING *operator*] [, ...]
- An ORDER BY item can be the name a column or it can be an arbitrary expression formed from input-column values.
- SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;

GROUP BY

- GROUP BY specifies a grouped table derived by the application of this clause:
- GROUP BY *expression* [, ...]
- GROUP BY will condense into a single row all selected rows that share the same values for the grouped columns.
- Aggregate functions are computed across all rows making up each group, producing a separate value for each group

LIMIT

- LIMIT { *count* | ALL } OFFSET *start*
  - where *count* specifies the maximum number of rows to return, and *start* specifies the number of rows to skip before starting to return rows.
- LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query.
- If a limit count is given, no more than that many rows will be returned.
- If an offset is given, that many rows will be skipped before starting to return rows.
- When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows
  - Eg. you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify ORDER BY.

SQL Data Types

Type Name	Aliases	Description
boolean	Bool, logical	Boolean (T/F)
bytea		binary data
character	varying(n) varchar(n)	variable-length character string
character(n)	char(n)	fixed-length char string
date		calendar date (year, month, day)
integer	int, int4	signed four-byte integer
numeric [(p, s)]	decimal [(p,s)]	exact numeric with selectable precision
smallint	int2	signed two-byte integer
text		variable-length character string
serial	serial4	autoincrementing four-byte integer

See PostgreSQL User Guide Ch 5

Esoteric Data Types

box	rectangular box in 2D plane
cidr	IP network address
circle	circle in 2D plane
inet	IP host address
macaddr	MAC address
polygon	closed geometric path in 2D plane

See PostgreSQL User Guide Ch 5

SQL Compatibility

- The following generic types are generally supported across all SQL implementations
- *bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time, timestamp*

Numeric Data Types

- Integer
  - store **whole** numbers, without fractional components
  - The type ***integer*** is the usual choice
  - best balance between range, storage size, and performance.
  - The ***smallint*** type is generally only used if disk space is at a premium.
  - The ***bigint*** type should only be used if the integer range is not sufficient

Numeric Data Types

- Arbitrary Precision Numbers
  - The scale of a numeric is the count of decimal digits in the fractional part, to the right of the decimal point.
  - The precision of a numeric is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point.
  - So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Numerical Data Types

- Numeric
  - The type ***numeric*** can store numbers with up to 1,000 digits
  - It is especially recommended for storing monetary amounts and other quantities where exactness is required.
  - ***numeric*** type is very slow compared to the floating-point types
  - Both the precision and the scale of the ***numeric*** type can be configured.

  `NUMERIC(precision, scale)`

  `NUMERIC(precision )` -- selects a scale of 0.

Numerical Data Types

- Floating Point
  - Data types ***real*** and ***double*** precision are inexact, variable-precision numeric types.
  - Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies.
    - If you require exact storage and calculations (such as for monetary amounts), use the ***numeric*** type instead.
    - If you want to do complicated calculations with these types for anything important, you should evaluate the implementation carefully.
    - Comparing two floating-point values for equality may or may not work as expected

Numerical Data Types

- Serials
  - Not a true type, but convenient for setting up identifier columns (similar to the AUTO_INCREMENT property supported by some other databases).

```
CREATE TABLE tablename (
    colname SERIAL
);
```

- This creates an integer column and arranged for its default values to be assigned from a sequence generator.
- A NOT NULL constraint is applied to ensure that a null value cannot be explicitly inserted
- In most cases you would also want to attach a UNIQUE or PRIMARY KEY constraint to prevent duplicate values from being inserted by accident, but this is not automatic.

Character Data Types

- SQL defines two primary character types:
  - character varying(n)
  - character(n)
- Both types store strings up to *n* characters in length.
- An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.
- If the string to be stored is shorter than the declared length, values of type character will be space-padded; values of type character varying will simply store the shorter string.

Date/Time Data Types

January 8, 1999	unambiguous
1999-01-08	ISO-8601 format
1/8/1999 ; 8/1/1999	U.S. vs European mode
1999.008 ; 99008	year and day of year
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

Date/Time Data Types

- Timestamp
  - Holds both date and time
  - 4713 BC → 1465001 AD
  - 1 microsecond / 14 digits
- Date
  - Holds dates only
  - 4713 BC → 32767 AD
  - 1 day
- Time
  - times of day only
  - 00:00:00.00 → 23:59:59.99
  - 1 microsecond

Boolean Data Types

- TRUE
- 't'
- 'true'
- 'y'
- 'yes'
- '1'

- FALSE
- 'f'
- 'false'
- 'n'
- 'no'
- '0'

Queries across Multiple Tables

- Joins
  - Inner
  - Outer
  - Left/Right

Example of a Join

- SELECT f.title, f.did, d.name, f.date_prod, f.kind FROM distributors d, films f WHERE f.did = d.did

Altering Tables

- Add columns
- Remove columns
- Add constraints
- Remove constraints
- Change default values
- Rename columns
- Rename tables

Altering Tables: ALTER command

- ALTER TABLE *table* [*] ADD [COLUMN] *column type* [*column_constraint* [...]]
- ALTER TABLE *table* [*] DROP [COLUMN] *column* [RESTRICT | CASCADE]
- ALTER TABLE *table* [*] ALTER [COLUMN] *column* { SET | DROP } NOT NULL
- ALTER TABLE *table* [*] RENAME [COLUMN] *column* TO *new_column*
- ALTER TABLE *table* RENAME TO *new_table*

ALTER COMMAND

- ALTER TABLE products ADD COLUMN description text;
- ALTER TABLE products DROP COLUMN description;
- ALTER TABLE products RENAME COLUMN product_no TO product_number;
- ALTER TABLE products RENAME TO items;

Creating Tables (2)

- SELECT INTO -- create a new table from the results of a query

SELECT *expression* [AS *output_name*] [, ...]
INTO TABLE *new_table*

[FROM *from_item* [, ...]] [WHERE *condition*] [GROUP BY *expression*

Views

- What are views ?
  - Views are not separate copies of the data in the table(s) or view(s) from which they're derived. In fact, views are called virtual tables because they do not exist as independent entities in the database as "real" tables do. (The ANSI term for a view is a viewed table; a native database table is a base table.) You can query views much as you query tables. Modifying data through views is restricted, however
- How are Views used?
  - Creating a view based on a SELECT statement gives you an easy way to examine and handle just the data you (or others) need—no more, no less. In effect, a view "freezes" a SELECT statement.

Creating views

- CREATE [OR REPLACE] VIEW view
  [(column name list)] AS SELECT query
- View
  - The name (optionally schema-qualified) of a view to be created.
- column name list
  - An optional list of names to be used for columns of the view. If given, these names override the column names that would be deduced from the SQL query.
- Query
  - An SQL query (that is, a SELECT statement) which will provide the columns and rows of the view.

Refer to SELECT for more information about valid arguments.

Select Revisited

SELECT * | *expression* FROM *from_item* [, ...]
[WHERE *condition*]
[GROUP BY *expression*]
[ORDER BY *expression* [ASC | DESC]
[LIMIT { *count* | ALL }]

CREATE VIEW

- CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, a query rewrite rule (an ON SELECT rule) is automatically generated to support SELECT operations on views.

CREATE OR REPLACE VIEW

- CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced. You can only replace a view with a new query that generates the identical set of columns (i.e., same column names and data types).

More on Views

- Currently, views are read only: the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. For more information see CREATE RULE.
- Use the DROP VIEW statement to drop views.

View Example

Create a view consisting of all Comedy films:

```
CREATE VIEW kinds AS
   SELECT *
   FROM films
   WHERE kind = 'Comedy';

SELECT * FROM kinds;
```

code	title	did	date	kind	len
UA502	Bananas	105	1971-07-13	Comedy	01:22
C_701	There's a Girl in my Soup	107	1970-06-11	Comedy	01:36

Removing Views

- DROP VIEW name [, ...]
  [CASCADE | RESTRICT]

- Name
  - The name (optionally schema-qualified) of an existing view.
- CASCADE
  - Automatically drop objects that depend on the view (such as other views).
- RESTRICT
  - Refuse to drop the view if there are any dependent objects. This is the default.

Indexes

- Why Use indexes ?
- Benefits of Indexes?
  - Faster Queries
  - Can be used to help enforce constraints
- Disadvantages
  - Require additional system resources
  - Can take time to update

Index Creation

Given a table with the following definition

```
CREATE TABLE emp (
    empno integer,
    ename character(30),
    job character(30),
    mgr integer,
    hiredate date NOT NULL,
    sal numeric(6,2),
    comm numeric(4,0),
    dept integer
);
```


Index Creation

- User Application requires a lot of queries of the form:

```
SELECT ename, job FROM emp
   WHERE mgr = constant;
```

- Create an Index

```
CREATE INDEX emp_mgr_index
  ON emp (mgr);
```


Index Removal

- To remove an index, use the DROP INDEX command.
  - DROP INDEX emp_mgr_index ;
- Indexes can be added to and removed from tables at any time.

Indexes and the DBMS

- Once the index is created, no further intervention is required
- DMBS will use the index when it thinks it would be more efficient than a sequential table scan.
- When an index is created, the DBMS has to keep it synchronized with the table. This adds overhead to data manipulation operations.
- Indexes that are non-essential or do not get used at all should be removed.
- A query or data manipulation command can use at most one index per table.

Indexing Performance

- Indexes can benefit UPDATEs and DELETEs with search conditions.
- Indexes can also be used in join queries.
- An index defined on a column that is part of a join condition can significantly speed up queries with joins.

Multicolumn Indexes

- An index can be defined on more than one column.
- For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

- SELECT name FROM test2 WHERE major = constant AND minor = constant;
- CREATE INDEX test2_mm_idx ON test2 (major, minor);
- Multicolumn indexes can only be used if the clauses involving the indexed columns are joined with **AND**.
- SELECT name FROM test2 WHERE major = constant OR minor = constant;

Unique Indexes

- Indexes may also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.
- CREATE UNIQUE INDEX name ON table (column [, ...]);
- When an index is declared unique, multiple table rows with equal indexed values will not be allowed. NULL values are not considered equal.
- PostgreSQL automatically creates unique indexes when a table is declared with a unique constraint or a primary key, on the columns that make up the primary key or unique columns (a multicolumn index, if appropriate), to enforce that constraint.
- A unique index can be added to a table at any later time, to add a unique constraint.
- **Note**: The preferred way to add a unique constraint to a table is ALTER TABLE ... ADD CONSTRAINT. The use of indexes to enforce unique constraints could be considered an implementation detail that should not be accessed directly.

Index Performance Testing

- Difficult to formulate a general procedure for determining which indexes to set up.
- A good deal of experimentation will be necessary in most cases.

Index Performance Testing

- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.
- It is especially fatal to use proportionally reduced data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows will probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.
- Be careful when making up test data, which is often unavoidable when the application is not in production use yet. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

Advanced Indexes

- PostgreSQL provides several index types:
  - B-tree
  - R-tree
  - GiST
  - Hash.
- Each index type is more appropriate for a particular query type because of the algorithm it uses.
- By default, the CREATE INDEX command will create a B-tree index
- The PostgreSQL query optimizer will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:
  - <, <=, =, >=, >
- R-tree indexes are especially suited for spatial data. The PostgreSQL query optimizer will consider using an R-tree index whenever an indexed column is involved in a comparison using one of these operators:
  - <<, &<, &>, >>, @, ~=, &&
- The query optimizer will consider using a hash index whenever an indexed column is involved in a comparison using the = operator.

Specifying Index Types

- To create an R-tree index:
  - CREATE INDEX name ON table USING RTREE (column);
- Creating a hash index:
  - CREATE INDEX name ON table USING HASH (column);

Database Backups

- Why not use standard filesystem tools?
- The database server must be shut down in order to get a usable backup.
- Needless to say that you also need to shut down the server before restoring the data.
- Temptation to try to back up or restore only certain individual tables or databases from their respective files or directories.
  - This will not work because only a portion of the information is contained here.
  - The other half is in the commit log files which contain the commit status of all transactions.
- Also note that the file system backup will not necessarily be smaller than an SQL dump.
- FS backup will most likely be larger. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.)

Backups

- SQL Dump
- The idea behind the SQL-dump method is to generate a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump.
- PostgreSQL provides the utility program pg_dump for this purpose.

  pg_dump dbname > outfile

Restore

- The text files created by pg_dump are intended to be read in by the psql program.

  psql dbname < infile
- infile is produced by the pg_dump command.
- he database dbname will not be created by this command
- If the objects in the original database were owned by different users, then the dump will instruct psql to connect as each affected user in turn and then create the relevant objects. This way the original ownership is preserved.
- All users must already exist, and furthermore that you must be allowed to connect as each of them.

Plumbing

- pg_dump -h host1 dbname | psql -h host2 dbname
- Use compressed dumps. Use your favorite compression program, for example gzip.
  - pg_dump dbname | gzip > filename.gz
- Reload with:

  createdb dbname

  gunzip -c filename.gz | psql dbname

  or

  cat filename.gz | gunzip | psql dbname