COMP3311 Week 03 Lectures

Relational DBMSs

What is an RDBMS?

2/44

A *relational database management system* (RDBMS) is

- software designed to support large-scale data-intensive applications
- allowing high-level description of data (tables, constraints)
- with high-level access to the data (relational model, SQL)
- providing efficient storage and retrieval (disk/memory management)
- supporting multiple simultaneous users (privilege, protection)
- doing multiple simultaneous operations (transactions, concurrency)
- maintaining reliable access to the stored data (backup, recovery)

Note: databases provide *persistent* storage of information

RDBMSs in COMP3311

3/44

PostgreSQL

- full-featured, client-server DBMS, resource intensive
- applications communicate via server to DB
- can run distributed and replicated
- follows SQL standard closely, but not totally
- extra data types (e.g. JSON), multiple procedural languages

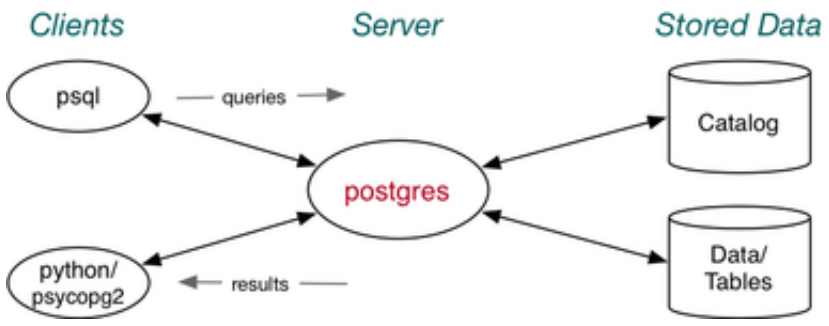
SQLite

- full-featured, serverless DBMS, light user of resources
- intended to be embedded in applications
- follows SQL standard closely, but not totally
- no stored procedures, JSON, add functions via PLs

PostgreSQL Architecture

4/44

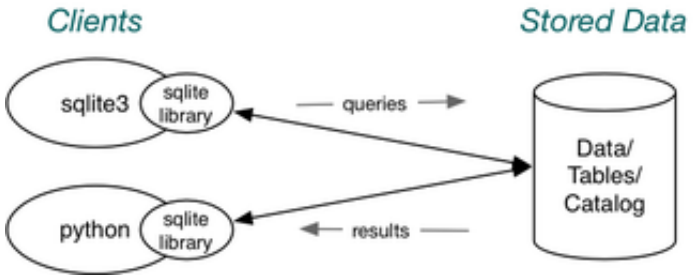
PostgreSQL's client-server architecture:



SQLite Architecture

5/44

SQLite's serverless architecture:



Using PostgreSQL in CSE

6/44

Using your PostgreSQL server in CSE (once installed):

- login to `grieg`, set up environment, start server
- use `psql`, etc. to manipulate databases
- stop server, log off `grieg`

```
wagner$ ssh YOU@grieg
grieg$ priv srvr
grieg$ source /srvr/YOU/env
grieg$ pg start
grieg$ psql mydb
... do stuff with your database ...
grieg$ pg stop
grieg$ exit
```

... Using PostgreSQL in CSE

7/44

PostgreSQL files (helps to understand state of server)

- PostgreSQL home directory ... `/srvr/YOU/pgsql/data/`
- under the home directory ...
  - `postgresql.conf` ... main configuration file
  - `base/` ... subdirectories containing database files
  - `postmaster.pid` ... process ID of server process
  - `.s.PGSQL.5432` ... socket for clients to connect to server
  - `.s.PGSQL.5432.lock` ... lock file for socket
- PostgreSQL environment settings ... `/srvr/YOU/env`

Building/Maintaining Databases

Managing Databases

9/44

Shell commands to create/remove databases:

- **`createdb`** *dbname* ... create a new totally empty database
- **`dropdb`** *dbname* ... remove *all* data associated with a DB

(If no *dbname* supplied, assumes a database called *YOU*)

Shell commands to dump/restore database contents:

- **`pg_dump`** *dbname* **`>`** *dumpfile*
- **`psql`** *dbname* **`-f`** *dumpfile*

(Database *dbname* is typically created just before restore)

SQL statements (used in *dumpfile*):

- **`CREATE TABLE`**, **`ALTER TABLE`**, **`COPY`**

Managing Tables

10/44

SQL statements:

- **`CREATE TABLE`** *table* **`(`** *Attributes+Constraints* **`)`**
- **`ALTER TABLE`** *table TableSchemaChanges*
- **`DROP TABLE`** *table(s)* [**`CASCADE`**]
- **`TRUNCATE TABLE`** *table(s)* [**`CASCADE`**]

(All conform to SQL standard, but all also have extensions)

`DROP..CASCADE` also drops objects which depend on the table

- objects could be tuples or views, but *not* whole tables

`TRUNCATE..CASCADE` truncates tables which refer to the table

Managing Tuples

11/44

SQL statements:

- **`INSERT INTO`** *table* **`(`** *Attrs* **`)`** **`VALUES`** *Tuple(s)*
- **`DELETE FROM`** *table* **`WHERE`** *condition*
- **`UPDATE`** *table* **`SET`** *AttrValueChanges* **`WHERE`** *condition*

Attrs = **`(`** $attr_1$**`,`** ... $attr_n$ **`)`** *Tuple* = **`(`** val_1**`,`** ... val_n **`)`**

AttrValueChanges is a comma-separated list of:

- *attrname* **`=`** *expression*

Each list element assigns a new value to a given attribute.

Exercise 1: Creating/Populating Databases

12/44

Do the following:

- create a database called `ex1`
- create a table `T` with two integer fields `x` and `y`
- examine the catalog definition of table `T`
- use `insert` statements to load some tuples (from a file)
- use `pg_dump` to make a copy of the database contents
- remove the `ex1` database, then restore it from the dump

Exercise 2: Inserting tuples

13/44

Create a database and define the following table:

```
create table p1 (
        x integer,
        y integer
);
create table p2 (
        x integer primary key,
        y integer
);
```

- insert the tuples (1,2),(2,3),(1,3) into each table
- how are the tables different in the catalog?

Exercise 3: Generating IDs

14/44

Consider the following schema:

```
create table Students (
    id      serial,
    name    text,
    address text,
    primary key (id)
);
```

Managing Other DB Objects

15/44

Databases contain objects other than tables and tuples:

- views, functions, sequences, types, indexes, roles, ...

Most have SQL statements for:

- **CREATE** *ObjectType name* ...
- **DROP** *ObjectType name* ...

Views and functions also have available:

- **CREATE OR REPLACE** *ObjectType name* ...

See PostgreSQL documentation Section IV, Chapter I for SQL statement details.

SQL

SQL

17/44

SQL = Structured Query Language (standards 1982, 1992, 1999, 2011)

SQL has several sub-languages:

- meta-data definition language (e.g. CREATE TABLE)
- meta-data update language (e.g. ALTER TABLE)

- data update language (e.g. INSERT, UPDATE, DELETE)
- query language (SQL) (e.g. SELECT)

Meta-data languages manage the *schema*.

Data languages manipulate (sets of) *tuples*.

Query languages are based on *relational algebra* (see later in course)

Types/Constants in SQL

Numeric types: `INTEGER`, `REAL`, `NUMERIC(w,d)`

```
10    -1    3.14159    2e-5    6.022e23
```

String types: `CHAR(n)`, `VARCHAR(n)`, `TEXT`

```
'John'   'some text'   '!%#%!$'   'O''Brien'
'"'   '[A-Z]{4}\d{4}'   'a VeRy! LoNg String'
```

PostgreSQL provides extended strings containing \ escapes, e.g.

```
E'\n'   E'O\'Brien'   E'[A-Z]{4}\\d{4}'   E'John'
```

Type-casting via *Expr*`::`*Type* (e.g. `'10'::integer`)

... Types/Constants in SQL

Logical type: `BOOLEAN`, `TRUE` and `FALSE` (or `true` and `false`)

PostgreSQL also allows `'t'`, `'true'`, `'yes'`, `'f'`, `'false'`, `'no'`

Time-related types: `DATE`, `TIME`, `TIMESTAMP`, `INTERVAL`

```
'2008-04-13'  '13:30:15'  '2004-10-19 10:23:54'
'Wed Dec 17 07:37:16 1997 PST'
'10 minutes'  '5 days, 6 hours, 15 seconds'
```

Subtraction of timestamps yields an interval, e.g.

```
now()::TIMESTAMP - birthdate::TIMESTAMP
```

PostgreSQL also has a range of non-standard types, e.g.

- geometric (point/line/...), currency, IP addresses, JSON, XML, objectIDs, ...
- non-standard types typically use string literals (`'...'`) which need to be interpreted

---

... Types/Constants in SQL

Users can define their own types in several ways:

```
-- domains: constrained version of existing type

CREATE DOMAIN Name AS Type CHECK ( Constraint )

-- tuple types: defined for each table

CREATE TYPE Name AS ( AttrName AttrType, ... )
```

```
-- enumerated type: specify elements and ordering

CREATE TYPE Name AS ENUM ( 'Label', ... )
```

Exercise 4: Defining domains

21/44

Give suitable domain definitions for the following:

- positive integers
- a person's age
- a UNSW course code
- a UNSW student/staff ID
- colours (as used in HTML/CSS)
- pairs of integers *(x,y)*
- standard UNSW grades (FL,PS,CR,DN,HD)

Exercise 5: Enumerated types

22/44

How are the following different?

```
create domain SizeValues1 AS
   text CHECK (value in ('small','medium','large'));

create type SizeValues2 AS
   enum ('small','medium','large');
```

Tuple and Set Literals

23/44

Tuple and set constants are both written as:

(val_1, val_2, val_3, ...)

The correct interpretation is worked out from the context.

Examples:

```
INSERT INTO Student(studeID, name, degree)
   VALUES (2177364, 'Jack Smith', 'BSc')
          -- tuple literal

CONSTRAINT CHECK gender IN ('male','female')
                           -- set literal
```

SQL Operators

24/44

Comparison operators are defined on all types:

```
<    >    <=    >=    =    <>
```

In PostgreSQL, `!=` is a synonym for `<>` (but there's no ==)

Boolean operators `AND`, `OR`, `NOT` are also available

Note `AND`,`OR` are not "short-circuit" in the same way as C's `&&`, `||`

Most data types also have type-specific operations available

See PostgreSQL Documentation Chapter 8/9 for data types and operators

---

... SQL Operators

String comparison:

- *str*$_1$ < *str*$_2$ ... compare using dictionary order
- *str* `LIKE` *pattern* ... matches string to pattern

Pattern-matching uses SQL-specific pattern expressions:

`%` matches anything (cf. regexp `.*`)

`_` matches any single char (cf. regexp `.`)

... SQL Operators

Examples (using SQL92 pattern matching):

`name LIKE 'Ja%'`	`name` begins with 'Ja'
`name LIKE '_i%'`	`name` has 'i' as 2nd letter
`name LIKE '%o%o%'`	`name` contains two 'o's
`name LIKE '%ith'`	`name` ends with 'ith'
`name LIKE 'John'`	`name` equals 'John'

PostgreSQL also supports case-insensitive match: `ILIKE`

... SQL Operators

Many DBMSs also provide *regexp*-based pattern matching
(*regexp* = *regular expression*; the POSIX regexp library is widely available)

PostgreSQL uses `~` and `!~` operators for this:

Attr `~ '`*RegExp*`'` or *Attr* `!~ '`*RegExp*`'`

Also provides case-insensitive matching (makes some regexps shorter)

Attr `~* '`*RegExp*`'` or *Attr* `!~* '`*RegExp*`'`

PostgreSQL also provides full-text searching (see Chapter 12)

... SQL Operators

Examples (using POSIX regular expressions):

`name ~ '^Ja'`	`name` begins with 'Ja'
`name ~ '^.i'`	`name` has 'i' as 2nd letter
`name ~ '.*o.*o.*'`	`name` contains two 'o's

`name ~ 'ith$'` `name` ends with 'ith'

`name ~ 'John'` `name` contains 'John'

... SQL Operators

String manipulation:

- *str*$_1$ || *str*$_2$ ... return concatenation of *str*$_1$ and *str*$_2$
- `lower(`*str*`)` ... return lower-case version of *str*
- `substring(`*str*,*start*,*count*`)` ... extract substring from *str*

Etc. etc. ... consult your local SQL Manual (e.g. PostgreSQL Sec 9.4)

Note that above operations are null-preserving (strict):

- if any operand is `NULL`, result is `NULL`
- beware of `(a||' '||b)` ... `NULL` if either of `a` or `b` is `NULL`

... SQL Operators

Arithmetic operations:

`+ - * / abs ceil floor power sqrt sin` *`etc.`*

Aggregations "summarize" a column of numbers in a relation:

- `count(`*attr*`)` ... number of rows in *attr* column
- `sum(`*attr*`)` ... sum of values for *attr*
- `avg(`*attr*`)` ... mean of values for *attr*
- `min/max(`*attr*`)` ... min/max of values for *attr*

Note: `count` applies to columns of non-numbers as well.

The `NULL` Value

Expressions containing `NULL` generally yield `NULL`.

However, boolean expressions use three-valued logic:

a	*b*	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

... The `NULL` Value

Important consequence of `NULL` behaviour ...

These expressions do not work as (might be) expected:

`x = NULL` `x <> NULL`

Both return `NULL` regardless of the value of *x*

Can only test for `NULL` using:

`x IS NULL` `x IS NOT NULL`

Conditional Expressions

33/44

Other ways that SQL provides for dealing with `NULL`:

`coalesce(`val_1`,`val_2`,...`val_n`)`

- returns first non-null value val_i
- useful for providing a "displayable" value for nulls

E.g. `select coalesce(mark,'??') from Marks ...`

`nullif(`val_1`,`val_2`)`

- returns `NULL` if val_1 is equal to val_2
- can be used to implement an "inverse" to `coalesce`

E.g. `nullif(mark,'??')`

... Conditional Expressions

34/44

SQL also provides a generalised conditional expression:

```
CASE
   WHEN test1 THEN result1
   WHEN test2 THEN result2
   ...
   ELSE resultn
END
```

E.g. `case when mark>=85 then 'HD' ... else '??' end`

Tests that yield `NULL` are treated as `FALSE`

If no `ELSE`, and all tests fail, `CASE` yields `NULL`

SQL: Queries

SQL Query Language

36/44

An SQL *query* consists of a sequence of clauses:

```
SELECT   projectionList
FROM     relations/joins
WHERE    condition
```

```
GROUP BY groupingAttributes
HAVING   groupCondition
```

`FROM`, `WHERE`, `GROUP BY`, `HAVING` clauses are optional.

Result of query: a relation, typically displayed as a table.

Result could be just one tuple with one attribute (i.e. one value) or even empty

---

... SQL Query Language

37/44

Schema:

- *Students(id, name, ...)*
- *Enrolments(student, course, mark, grade)*

Example SQL query:

```
SELECT   s.id, s.name, avg(e.mark) as avgMark
FROM     Students s, Enrolments e
WHERE    s.id = e.student
GROUP BY s.id, s.name
-- or --
SELECT   s.id, s.name, avg(e.mark) as avgMark
FROM     Students s
           JOIN Enrolments e on (s.id = e.student)
GROUP BY s.id, s.name
```

... SQL Query Language

38/44

How the example query is computed:

- produce all pairs of *Students,Enrolments* tuples
  which satisfy condition *(Students.id = Enrolments.student)*
- each tuple has *(id,name,...,student,course,mark,grade)*
- form groups of tuples with same *(id,name)* values
- for each group, compute average mark
- form result tuples *(id,name,avgMark)*

Problem-solving in SQL

39/44

Starts with an information request:

- (informal) description of the information required from the database

Ends with:

- a list of tuples that meet the requirements in the request

Pre-req: *know your schema*

Look for keywords in request to identify required data :

- tell me the names of all students...
- how many students failed ...
- what is the highest mark in ...
- which courses are ... (course codes?)

... Problem-solving in SQL

Developing SQL queries ...

- relate required data to *attributes* in schema
- identify which *tables* contain these attributes
- combine data from relevant tables (*FROM*, *JOIN*)
- specify conditions to select relevant data (*WHERE*)
- [optional] define grouping attributes (*GROUP BY*)
- develop expressions to compute output values (*SELECT*)

Views

A *view* associates a name with a query:

- **CREATE VIEW** *viewName* [**(** *attributes* **)**] **AS** *Query*

Each time the view is invoked (in a FROM clause):

- the *Query* is evaluated, yielding a set of tuples
- the set of tuples is used as the value of the view

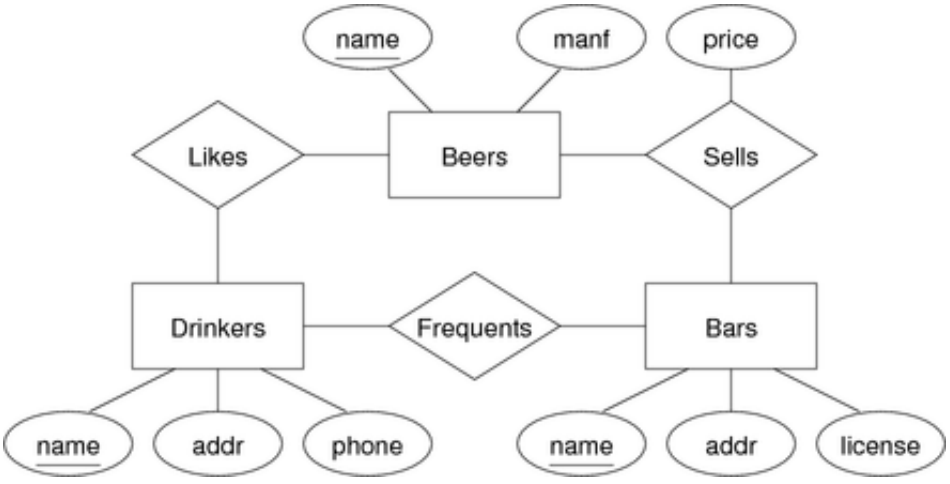
A view can be treated as a "virtual table".

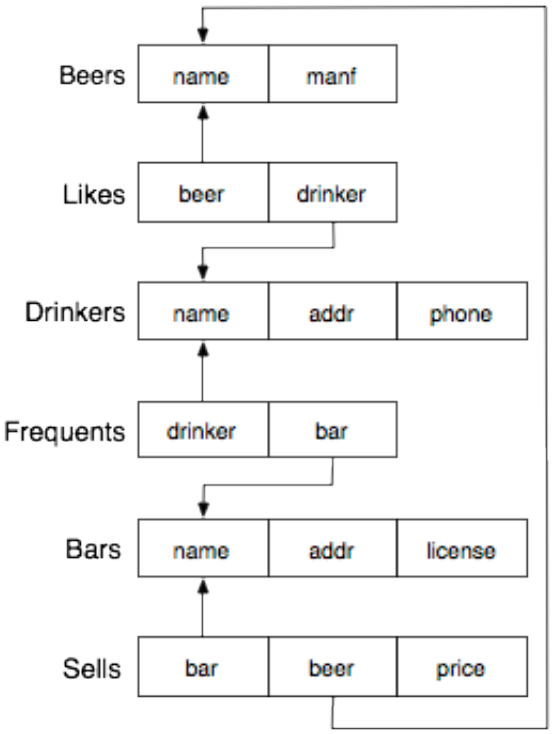
Views are useful for "packaging" a complex query to use in other queries.

cf. writing functions to package computations in programs

Exercise 6: Queries on Beer Database

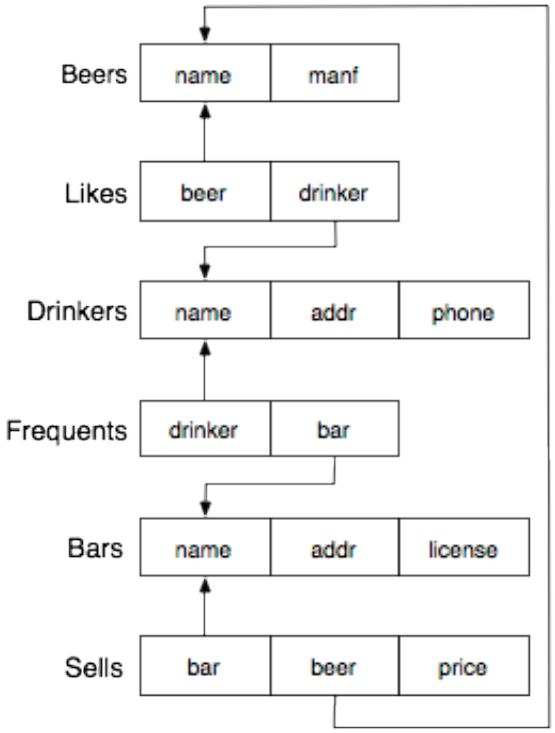
ER design for Beer database:





Answer these queries on the Beer database:

1. What beers are made by Toohey's?
2. Show beers with headings "Beer", "Brewer".
3. How many different beers are there?
4. How many different brewers are there?
5. (a) Which beers does John like?
   (b) Find the brewers whose beers John likes.
6. Find pairs of beers by the same manufacturer.
7. (a) How many beers does each brewer make?
   (b) Which brewers make only one beer?
   (c) Find beers that are the only one by their brewer.
8. Find the beers sold at bars where John drinks.



More queries on the Beer database:

9. Which brewer makes the most beers?
10. Bars where either Gernot or John drink.
11. Bars where both Gernot and John drink.
12. Find bars that serve New at the same price as the Coogee Bay Hotel charges for VB.
13. Find the average price of common beers (i.e. served in more than two hotels).
14. Which bar sells 'New' cheapest?

Produced: 25 Feb 2020