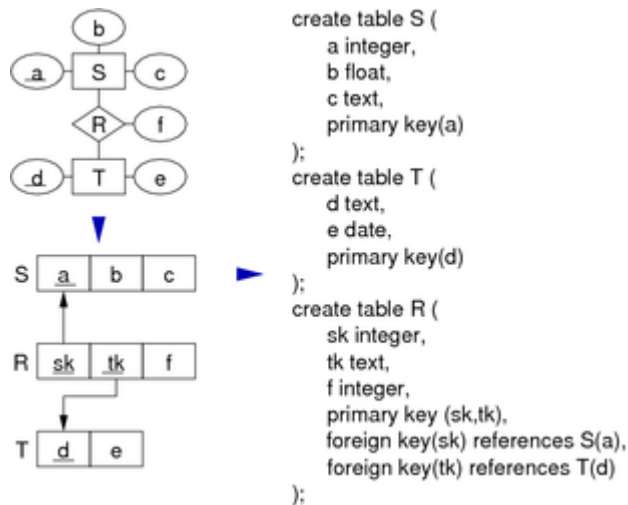


COMP3311 Week 03 Lecture (cont)

Exercise: Mapping ER to Rel to SQL

1/39



SQL

2/39

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*.

SQL Syntax

3/39

SQL definitions, queries and statements are composed of:

- *comments* ... `--` comments to end of line
- *identifiers* ... similar to regular programming languages
- *keywords* ... a large set (e.g. CREATE, DROP, TABLE)
- *data types* ... small set of basic types (e.g. integer, date)
- *operators* ... similar to regular programming languages
- *constants* ... similar to regular programming languages

Similar means "often the same, but not always" ...

... SQL Syntax

4/39

How SQL syntax differs from regular programming languages ...

- single-quotes are used for strings
- double-quotes used for "non-standard" identifiers

Identifiers are case-insensitive (unless "double-quoted")

(Staff = staff = STAFF = "staff" ≠ "Staff" ≠ "StAff")

Variations in identifier syntax:

- Oracle also allows unquoted hash (#) and dollar (\$) in identifiers.

- MySQL uses non-standard back-quote (`) instead of double-quote (").

... SQL Syntax

5/39

Identifiers denote:

- database objects such as tables, attributes, views, ...
- meta-objects such as types, functions, constraints, ...

Naming conventions that I (try to) use in this course:

- relation names: e.g. Branches, Students, ...
- attribute names: e.g. name, code, firstName, ...
- foreign keys: named after either or both of
 - table being referenced e.g. staff, ...
 - relationship being modelled e.g. teaches, ...

We initially write SQL keywords in all upper-case in slides.

Types/Constants in SQL

6/39

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

7/39

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, XML, objectIDs, ...
- non-standard types typically have string literals ('...') (except OIDs)

... Types/Constants in SQL

8/39

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: Defining domains

9/39

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)

[\[Solution\]](#)

Exercise: Enumerated types

10/39

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

11/39

Tuple and set constants are both written as:

```
( val1, val2, val3, ... )
```

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

12/39

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

String comparison:

- $str_1 < str_2$... compare using dictionary order
- $str \text{ LIKE } pattern$... matches string to pattern

Pattern-matching uses SQL-specific pattern expressions:

- % matches anything (cf. regexp .*)
- _ matches any single char (cf. regexp .)

... SQL Operators

14/39

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

15/39

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

PostgreSQL uses [~](#) and [!~](#) operators for this:

$Attr \sim 'RegExp'$ or $Attr !\sim 'RegExp'$

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

$Attr \sim^* 'RegExp'$ or $Attr !\sim^* 'RegExp'$

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

... SQL Operators

16/39

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

17/39

String manipulation:

- $str_1 || str_2$... return concatenation of str_1 and str_2
- $\text{lower}(str)$... return lower-case version of str
- $\text{substring}(str, start, count)$... extract substring from str

Etc. etc. ... consult your local SQL Manual (e.g., [PostgreSQL Section 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

18/39

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

19/39

Expressions containing NULL generally yield NULL.

However, boolean expressions use three-valued logic:

<i>a</i>	<i>b</i>	<i>a AND b</i>	<i>a OR b</i>
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

20/39

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

21/39

Other ways that SQL provides for dealing with NULL:

`coalesce(val1, val2, ... valn)`

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

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

`nullif(val1, val2)`

- returns NULL if *val₁* is equal to *val₂*
- can be used to implement an "inverse" to coalesce

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

... Conditional Expressions

22/39

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: Schemas

SQL Data Definition

24/39

Relations (tables) are declared using:

```
CREATE TABLE RelName (
  attribute1 domain1 constraints1,
  attribute2 domain2 constraints2,
  ...
  table-level constraints, ...
);
```

where *constraints* can include details about primary keys, foreign keys, default values, and constraints on attribute values.

Defines table schema *and* creates empty instance of table.

Tables are removed via `DROP TABLE RelName;`

... SQL Data Definition

25/39

Example table definition:

```
create table Students (
  id          integer, -- e.g. 3123456
  familyName  text,    -- e.g. 'Smith'
  givenName   text,    -- e.g. 'John'
  birthDate   date,    -- e.g. '1-Mar-1984'
  degree      integer, -- e.g. 3648
  wam         float,   -- e.g. 84.75 (derived)
  primary key (id),
  foreign key (degree) references Degrees(id)
);
```

Primary key ⇒ unique not null

Primary Keys

26/39

If we want to define a numeric primary key, e.g.

```
CREATE TABLE R ( id INTEGER PRIMARY KEY, ... );
```

we still have the problem of generating unique values.

Most DBMSs provide a mechanism to

- generate a sequence of unique values
- ensure that two tuples don't get assigned the same value

PostgreSQL's version:

```
CREATE TABLE R ( id SERIAL PRIMARY KEY, ... );  
INSERT INTO R VALUES ( DEFAULT, ... );
```

Referential Integrity

27/39

Declaring foreign keys assures **referential integrity**.

E.g. Account.branch text references Branch(name)

Every Account tuple must contain an existing Branch name.

If we want to delete a tuple from Branch, and there are tuples in Account that refer to it, we could ...

- **reject** the deletion (PostgreSQL default behaviour)
 - **set-NULL** the foreign key attributes in Account records
 - **cascade** the deletion and remove Account records
-

Exercise: Data Insertion

28/39

Consider the following schema:

```
create table R (  
    id integer primary key,  
    s char(1) references S(id)  
);  
create table S (  
    id char(1) primary key,  
    r integer references R(id)  
);
```

Devise a method to:

- load the schema
- INSERT data into the tables

Advanced: what if both foreign keys were NOT NULL.

[\[Solution\]](#)

Other Attribute Properties

29/39

Example (the red constraint is invalid):

```
CREATE TABLE Example (  
    gender char(1) CHECK (gender IN ('M','F')),  
    Xval integer NOT NULL,  
    Yval integer CONSTRAINT isPos CHECK (Yval > 0),  
    Zval real DEFAULT 100.0,  
    CONSTRAINT XgtY CHECK (Xval > Yval),  
    CONSTRAINT Zcondition CHECK  
        (Zval >  
         (SELECT MAX(price) FROM Sells)  
        )  
);
```

SQL Query Language

31/39

SQL provides powerful, high-level manipulation of data.

However, SQL is *not* a complete programming language.

Applications typically embed SQL into programming languages:

- Java and the JDBC API
- PHP/Perl/Tcl and their various DBMS bindings
- RDBMS-specific programming languages
(e.g. Oracle's PL/SQL, PostgreSQL's PLpgSQL)
- C-level library interfaces to DBMS engine
(e.g. Oracle's OCI, PostgreSQL's libpq)

... SQL Query Language

32/39

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

33/39

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

34/39

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*)
-

Request: description of required information from database.

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

36/39

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

37/39

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

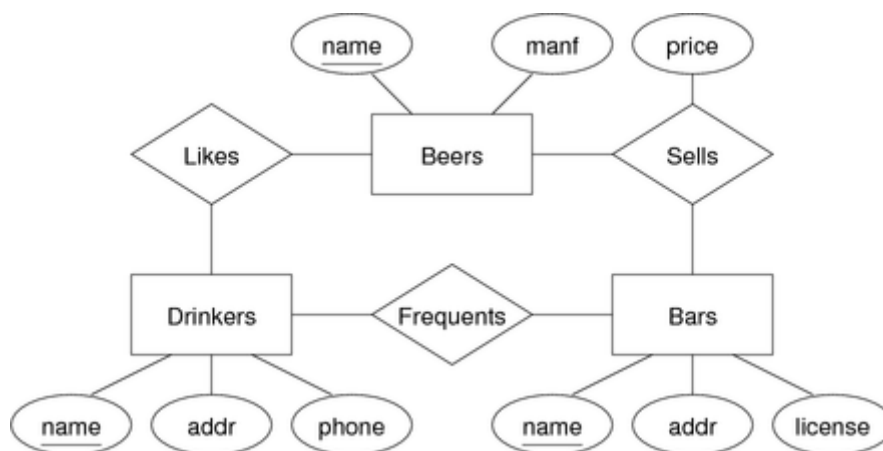
A view can be treated as a "virtual table".

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

Exercise: Queries on Beer Database

38/39

ER design for Beer database:

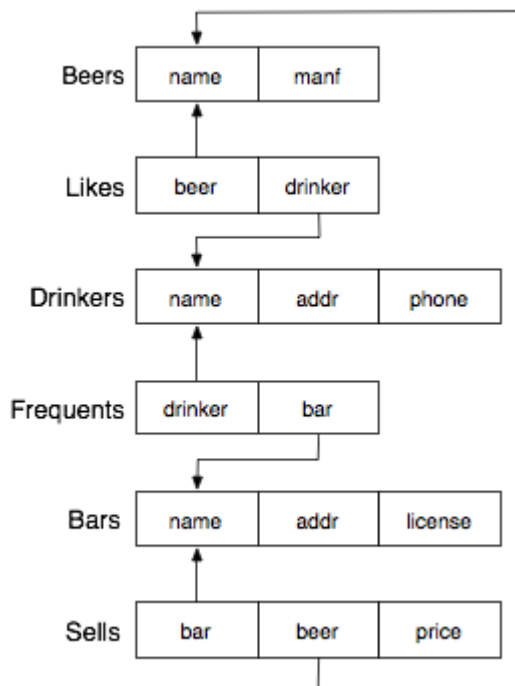


... Exercise: Queries on Beer Database

39/39

Answer these queries on the Beer database:

1. What beers are made by Toohey's?
2. Show beers with headings "Beer", "Brewer".
3. Find the brewers whose beers John likes.



4. Find pairs of beers by the same manufacturer.
5. Find beers that are the only one by their brewer.
6. Find the beers sold at bars where John drinks.
7. How many different beers are there?
8. How many different brewers are there?

[\[Solution\]](#)