SQL: Schemas, Queries, Updates, Views

SQL

SQL = **S**tructured **Q**uery **L**anguage (sometimes called "sequel").

SQL is an ANSI/ISO standard language for querying and manipulating relational DBMSs.

Designed to be a "human readable" language comprising:

- data definition facilities
- database modification operations
- database query operations, including:
  - relational algebra, set operations, aggregation, grouping, ...

SQL (cont)

SQL was developed at IBM (San Jose Lab) during the 1970's, and standardised in 1986.

DBMSs typically implement the SQL2 standard (aka SQL-92).

Unfortunately, they also:

- implement a (large) subset of the standard
- extend the standard in various "useful" ways

SQL (in some form) looks likely to survive in the next generation of database systems.

In these slides, we try to use only *standard* (portable) SQL2.

SQL (cont)

Since SQL2, there have been three new proposed standards:

SQL:1999 added e.g.

- boolean and BLOB types, arrays/rows, ...
- procedures programming constructs, triggers

- recursive queries
- OO-like objects, inheritance, ...

SQL:2003 ...

- standardised some SQL:1999 extensions
- added a standard for meta-data (catalogues)
- standardised stored procedures (SQL/PSM)
- added a new **`MERGE`** statement ("upsert")
- defined interfaces to C, Java, XML, object systems, ...

SQL:2008 added additional support for XML.

SQL (cont)

Major DBMSs (Oracle, DB2, SQLServer, PostgreSQL MySQL):

- implement most/all of SQL2
- implement much of SQL:1999

- implement some of SQL:2003
- omit difficult-to-implement features e.g. assertions

PostgreSQL ...

- implements almost all of SQL2 (see documentation)
- does not implement: recursive queries, assertions
- provides non-standard mechanisms for: updatable views
- currently has PLpgSQL, will also have SQL/PSM soon

SQL (cont)

SQL provides high-level, declarative access to data.

However, SQL is not a Turing-complete programming language.

Applications typically embed evaluation of SQL queries into PL's:

- 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 and low-level library interfaces to DBMS engine
  (e.g. Oracle's OCI, PostgreSQL's libpq)

SQL (cont)

SQL's query sub-language is based on relational algebra.

Relational algebra:

- formal language of expressions mapping tables→tables
- comprising three basic operations ...
  - select: filter table rows via a condition on attributes
  - project: filter table columns by name
  - join: combines two tables via a condition

- along with set operations (union, intersection, difference)
- and a variety of aggregates (including min(), max(), count(), etc)

SQL (cont)

Example relational algebra operations:

[Diagram:Pic/sql/relalg.png]

Example Databases

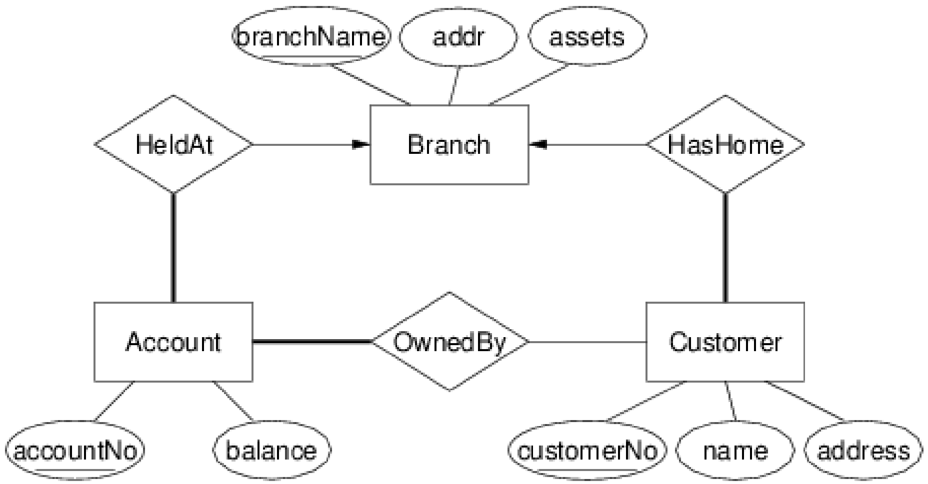
In order to demonstrate aspects of SQL, we use two databases:

- **`bank`**: customers, accounts, branches, ...
- **`beer`**: beers, bars, drinkers, ...

These databases are available for you to play with.

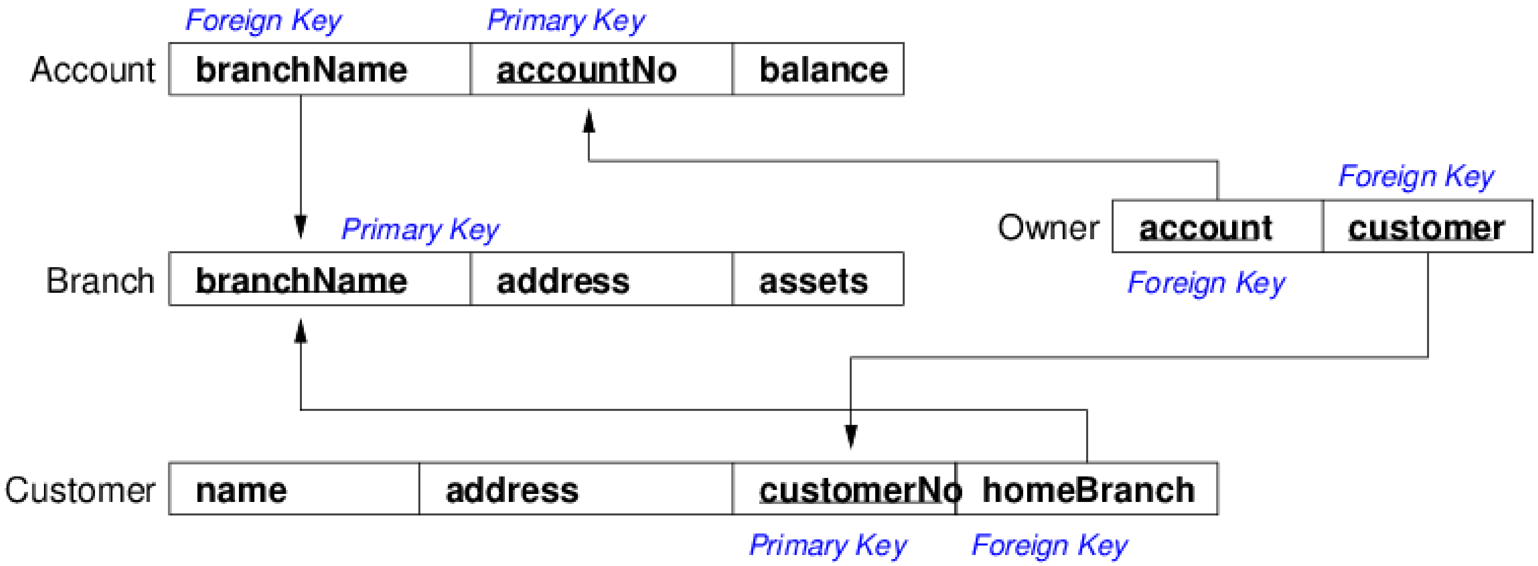
Example Database #1

ER design for a simple banking application:



Example Database #1 (cont)

Relational schema corresponding to the ER design:



Example Database #1 (cont)

We will use the following instance of this schema:

Branch relation/table instance:

```
 branchName |    address     | assets
------------+----------------+--------
 Clovelly   | Clovelly Rd.   |   1000
 Coogee     | Coogee Bay Rd. |  40000
 Maroubra   | Anzac Pde.     |  17000
 Randwick   | Alison Rd.     |  20000
 UNSW       | near Library   |   3000
```

Customer relation/table instance:

name	address	customerNo	homebranch
Adam	Belmore Rd.	12345	Randwick
Bob	Rainbow St.	32451	Coogee
Chuck	Clovelly Rd.	76543	Clovelly
David	Anzac Pde.	82199	UNSW
George	Anzac Pde.	81244	Maroubra
Graham	Malabar Rd.	92754	Maroubra
Greg	Coogee Bay Rd.	22735	Coogee
Jack	High St.	12666	Randwick

Example Database #1 (cont)

Account relation/table instance:

branchName	accountNo	balance
UNSW	U-245	1000
UNSW	U-291	2000
Randwick	R-245	20000
Coogee	C-123	15000
Coogee	C-124	25000

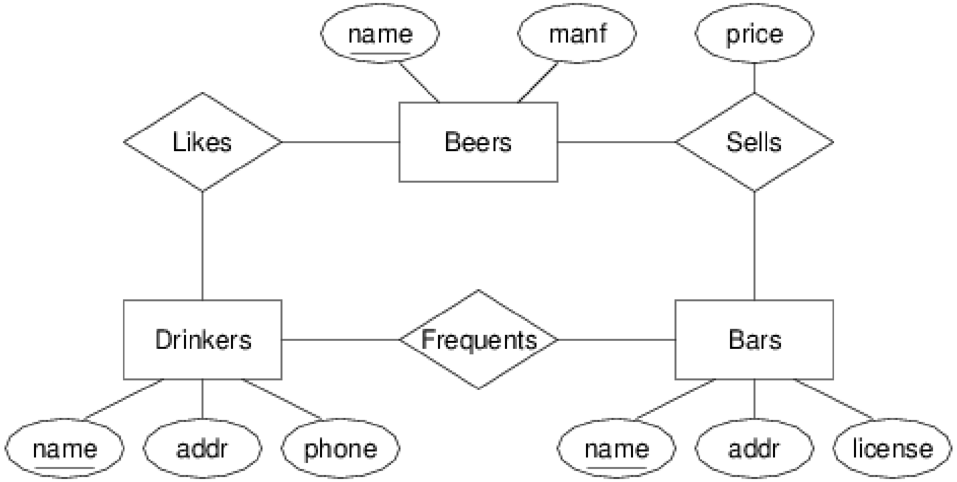
```
Clovelly    | Y-123       |    1000
Maroubra    | M-222       |    5000
Maroubra    | M-225       |   12000
```

Owner relation/table instance:

```
 account | customer
---------+----------
 U-245   |    12345
 U-291   |    12345
 U-291   |    12666
 R-245   |    12666
 C-123   |    32451
 C-124   |    22735
 Y-123   |    76543
 M-222   |    92754
 M-225   |    12345
```

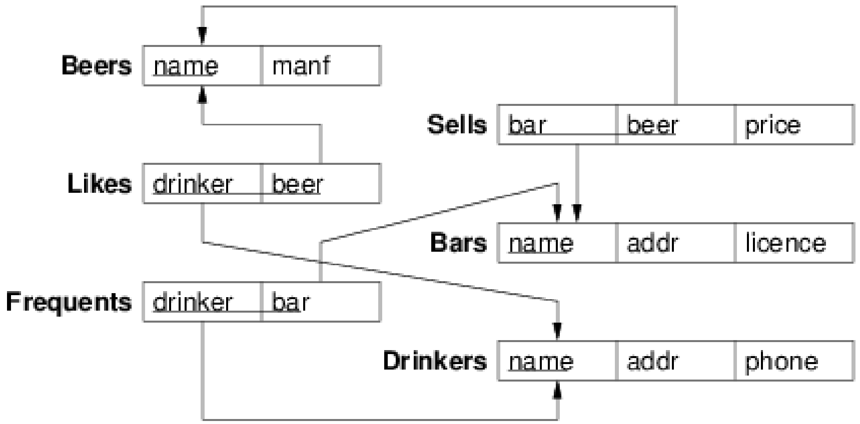
Example Database #2

ER design for beers/bars/drinkers database:



Example Database #2 (cont)

Relational schema corresponding to the ER design:



Example Database #2 (cont)

We will use the following instance of this schema:

Bars relation/table instance:

name	addr	license
Australia Hotel	The Rocks	123456
Coogee Bay Hotel	Coogee	966500
Lord Nelson	The Rocks	123888
Marble Bar	Sydney	122123
Regent Hotel	Kingsford	987654
Royal Hotel	Randwick	938500

Drinkers relation/table instance:

name	addr	phone
Adam	Randwick	9385-4444
Gernot	Newtown	9415-3378
John	Clovelly	9665-1234
Justin	Mosman	9845-4321

Example Database #2 (cont)

Beers relation/table instance:

```
         name          |     manf
-----------------------+---------------
 80/-                  | Caledonian
 Bigfoot Barley Wine   | Sierra Nevada
 Burragorang Bock      | George IV Inn
 Crown Lager           | Carlton
 Fosters Lager         | Carlton
 Invalid Stout         | Carlton
 Melbourne Bitter      | Carlton
 New                   | Toohey's
 Old                   | Toohey's
 Old Admiral           | Lord Nelson
 Pale Ale              | Sierra Nevada
 Premium Lager         | Cascade
 Red                   | Toohey's
 Sheaf Stout           | Toohey's
 Sparkling Ale         | Cooper's
 Stout                 | Cooper's
 Three Sheets          | Lord Nelson
 Victoria Bitter       | Carlton
```

Example Database #2 (cont)

Frequents relation/table instance:

```
 drinker |       bar
---------+------------------
 Adam    | Coogee Bay Hotel
 Gernot  | Lord Nelson
 John    | Coogee Bay Hotel
 John    | Lord Nelson
 John    | Australia Hotel
 Justin  | Regent Hotel
 Justin  | Marble Bar
```

Example Database #2 (cont)

Likes relation/table instance:

```
 drinker |        beer
---------+---------------------
 Adam    | Crown Lager
 Adam    | Fosters Lager
 Adam    | New
```

```
  Gernot    | Premium Lager
  Gernot    | Sparkling Ale
  John      | 80/-
  John      | Bigfoot Barley Wine
  John      | Pale Ale
  John      | Three Sheets
  Justin    | Sparkling Ale
  Justin    | Victoria Bitter
```

Example Database #2 (cont)

Sells relation/table instance:

```
       bar         |       beer        | price
-------------------+-------------------+-------
 Australia Hotel   | Burragorang Bock  |  3.50
 Coogee Bay Hotel  | New               |  2.25
 Coogee Bay Hotel  | Old               |  2.50
 Coogee Bay Hotel  | Sparkling Ale     |  2.80
 Coogee Bay Hotel  | Victoria Bitter   |  2.30
 Lord Nelson       | Three Sheets      |  3.75
 Lord Nelson       | Old Admiral       |  3.75
 Marble Bar        | New               |  2.80
 Marble Bar        | Old               |  2.80
 Marble Bar        | Victoria Bitter   |  2.80
 Regent Hotel      | New               |  2.20
```

```
Regent Hotel       | Victoria Bitter  |  2.20
Royal Hotel        | New              |  2.30
Royal Hotel        | Old              |  2.30
Royal Hotel        | Victoria Bitter  |  2.30
```

SQL Syntax

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`**, **`SELECT`**, **`TABLE`**)
- data types ... a small set (e.g. **`integer`**, **`varchar`**, **`date`**)
- operators ... similar to regular programming languages
- constants ... similar to regular programming languages

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

- **`'John'`**, **`'blue'`**, **`'it''s'`** are strings
- **`"Students"`**, **`"Really Silly!"`** are identifiers

SQL Syntax (cont)

While SQL identifiers and keywords are case-insensitive, we generally:

- write keywords in upper case (until it becomes annoying)
  e.g. **SELECT**, **FROM**, **WHERE**, **CREATE**, ...
- write relation names with an initial upper-case letter
  e.g. **Customers**, **Students**, **Owns**, **EnrolledIn**
- write attribute names in all lower-case
  e.g. **id**, **name**, **partNumber**, **isActive**

We follow the above conventions when writing programs.

We ignore the above conventions when typing in lectures.

SQL Keywords

A categorised list of frequently-used SQL92 keywords:

```
Querying        Defining Data   Changing Data
SELECT          CREATE          INSERT
FROM            TABLE           INTO
WHERE           INTEGER         VALUES
GROUP BY        REAL            UPDATE
HAVING          VARCHAR         SET
ORDER BY        CHAR            DELETE
DESC            KEY             DROP
EXISTS          PRIMARY         ALTER
IS NULL         FOREIGN
NOT NULL        REFERENCES
IN              CONSTRAINT
DISTINCT        CHECK
AS
```

There are 225 reserved words in SQL92 ... not a small language.

SQL Keywords (cont)

A list of PostgreSQL's SQL keywords:

```
ALL           DEFERRABLE     IS          OVERLAPS
ANALYSE       DESC           ISNULL      PRIMARY
ANALYZE       DISTINCT       JOIN        PUBLIC
AND           DO             LEADING     REFERENCES
```

```
ANY            ELSE          LEFT        RIGHT
AS             END           LIKE        SELECT
ASC            EXCEPT        LIMIT       SESSION_USER
BETWEEN        FALSE         NATURAL     SOME
BINARY         FOR           NEW         TABLE
BOTH           FOREIGN       NOT         THEN
CASE           FREEZE        NOTNULL     TO
CAST           FROM          NULL        TRAILING
CHECK          FULL          OFF         TRUE
COLLATE        GROUP         OFFSET      UNION
COLUMN         HAVING        OLD         UNIQUE
CONSTRAINT     ILIKE         ON          USER
CROSS          IN            ONLY        USING
CURRENT_DATE   INITIALLY     OR          VERBOSE
CURRENT_TIME   INNER         ORDER       WHEN
CURRENT_USER   INTERSECT     OUTER       WHERE
DEFAULT        INTO
```

Note that some SQL92 reserved words are not reserved words in PostgreSQL.

SQL Identifiers

Names are used to identify

- database objects such as tables, attributes, views, ...

- meta-objects such as types, functions, constraints, ...

Identifiers in SQL use similar conventions to programming languages i.e. a sequence of alpha-numerics, starting with an alphabetic.

Can create arbitrary indentifiers by enclosing in `"..."`

Example identifiers:

```
employee    student    Courses
last_name   "That's a Great Name!"
```

Oracle SQL also allows unquoted hash (`#`) and dollar (`$`) in identifiers.

SQL Identifiers (cont)

Since SQL does not distinguish case, the following are all treated as being the same identifier:

```
employee   Employee   EmPlOyEe
```

Most RDBMSs will let you give the same name to different kinds of objects (e.g. a table called **`Beer`** and an attribute called **`Beer`**).

Some common naming conventions:

- name tables representing entitites via plural nouns (e.g. **`Drinkers`**, **`TheDrinkers`**, **`AllDrinkers`**, ...)
- name foreign key attributes after the table they refer to (e.g. **`beer`** in the **`Sells`** relation)

Constants in SQL

Numeric constants have same syntax as programming languages, e.g.

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

String constants are written in single quotes, e.g.

```
'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'
```

Boolean constants: **`TRUE`** and **`FALSE`**

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

Constants in SQL (cont)

Other kinds of constants are typically written as strings.

Dates: **`'2008-04-13'`**, Times: **`'13:30:15'`**

Timestamps: **`'2004-10-19 10:23:54'`**

PostgreSQL also recognises: **`'January 26 11:05:10 1988 EST'`**

Time intervals: **`'10 minutes'`**, **`'5 days, 6 hours'`**

PostgreSQL also has IP address, XML, etc. data types.

SQL Data Types

All attributes in SQL relations are typed (i.e. have domain specified)

SQL supports a small set of useful built-in data types:
text string, number (integer,real), date, boolean, binary

Various type conversions are available (e.g. date to string, string to date, integer to real) and applied automatically "where they make sense".

Basic domain (type) checking is performed automatically.

The **`NULL`** value is treated as a member of all data types.

No structured data types are available (in SQL2).

SQL Data Types (cont)

Various kinds of number types are available:

- **INTEGER** (or **INT**), **SMALLINT** ... 32/16-bit integers
- **REAL**, **DOUBLE PRECISION** ... 32/64-bit floating point
- **NUMBER(***d*,*p***)** ... fixed-point reals (*d* digits, *p* after dec.pt.)

PostgreSQL also provides ...

- **serial**: auto-generated integer values for primary keys
- **currency**: fixed-point reals, displayed as strings **$1,000.00**

SQL Data Types (cont)

Two string types are available:

- **CHAR(***n***)** ... uses *n* bytes, left-justified, blank-padded
- **VARCHAR(***n***)** ... uses 0..*n* bytes, no padding

String types can be coerced by blank-padding or truncation.

```
'abc'::CHAR(2) = 'ab'      'abc'::CHAR(4) = 'abc '
```

PostgreSQL also provides **`TEXT`** for arbitrary strings

- convenient; no need to worry "how long is a name?"
- efficient (different to some other DBMSs)
- but not part of SQL standard

SQL Data Types (cont)

Dates are simply specially-formatted strings, with a range of operations to implement date semantics.

Format is typically **`YYYY-MM-DD`** , e.g. **`'1998-08-02'`**

Accepts other formats (and has format-conversion functions), but beware of two-digit years (year 2000)

Comparison operators implement *before* (**<**) and *after* (**>**).

Subtraction counts number of days between two dates.

Etc. etc. ... consult your local SQL Manual

SQL Data Types (cont)

PostgreSQL also supports several non-standard data types.

- generic text string data i.e. **`text`**
- arbitrary binary data (BLOBs) i.e. **`bytea`**
- geometric data types e.g. **`point`**, **`circle`**, **`polygon`**, ...

Also, extends relational model so that a single attribute can contain an array/matrix of values, e.g.

```
CREATE TABLE Employees (
       empid      integer primary key,
       name       text,
       pay_rate   float[]
);
INSERT INTO Employees VALUES
       (1234, 'John', '{35.00,45.00,60.00}');
SELECT pay_rate[2] FROM Employees ...
```

Tuple and Set Literals

Tuple and set constants are both written as:

$$(\; val_1, \; val_2, \; val_3, \; \ldots \;)$$

The correct interpretation is worked out from the context.

Examples:

```
INSERT INTO Student(stude#, name, course)
VALUES (2177364, 'Jack Smith', 'BSc')
       -- tuple literal

CREATE TABLE Academics (
       id   integer,
       name varchar(40),
       job  varchar(10) CHECK
                job IN ('Lecturer', 'Tutor');
                -- set literal
```

Tuple and Set Literals (cont)

SQL data types provide coarse-grained control over values.

If more fine-grained control over values is needed:

- constraints can express more precise conditions
- new "data types" can be defined

Examples:

```
CREATE DOMAIN PositiveInt AS INTEGER
   CHECK (VALUE > 0);
CREATE DOMAIN Colour AS
   CHECK (VALUE IN ('red','yellow','green','blue','violet'));
CREATE TABLE T (
   x Colour,
   y PositiveInt,
   z INTEGER CHECK (z BETWEEN 10 AND 20)
);
```

SQL Operators

Comparison operators are defined on all types:

```
<    >    <=    >=    =    <>    (or !=)
```

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 (cont)

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 (like `.*`)
- `_` matches any single char (like `.`)

SQL Operators (cont)

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`** matches 'John'

PostgreSQL also supports case-insensitive match: **`ILIKE`**

SQL Operators (cont)

Most Unix-based DBMSs utilise the regexp library

- to provide full POSIX regular expression matching

PostgreSQL uses the ~ operator for this:

```
Attr ~ 'RegExp'
```

PostgreSQL also provides full-text searching (see doc)

SQL Operators (cont)

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** matches 'John'

SQL Operators (cont)

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 chars 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||' '||c)`** ... **`NULL`** if any of **`a`**, **`b`**, **`c`** are null

---

SQL Operators (cont)

Arithmetic operations:

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

Aggregations apply to 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.

SQL Operators (cont)

`NULL` in arithmetic operation always yields **`NULL`**, e.g.

```
3 + NULL = NULL        1 / NULL = NULL
```

`NULL` in aggregations is ignored (treated as unknown), e.g.

```
sum(1,2,3,4,5,6)         = 21
sum(1,2,NULL,4,NULL,6) = 13
avg(1,2,3,4,5)           = 3
avg(NULL,2,NULL,4)       = 3
```

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 (cont)

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
```

The **NULL** Value (cont)

Other ways PostgeSQL 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

`nullif(`Val_1**`,`** Val_2**`)`**

- returns null if Val_1 is equal to Val_2
- can be used to provide inverse of **`coalesce()`**

SQL: Schemas

Relational Data Definition

In order to give a relational data model, we need to:

- describe tables
- describe attributes that comprise tables
- describe any constraints on the data

A relation schema defines an individual table.

A database schema is a collection of relation schemas that defines the structure of and constraints on an entire database.

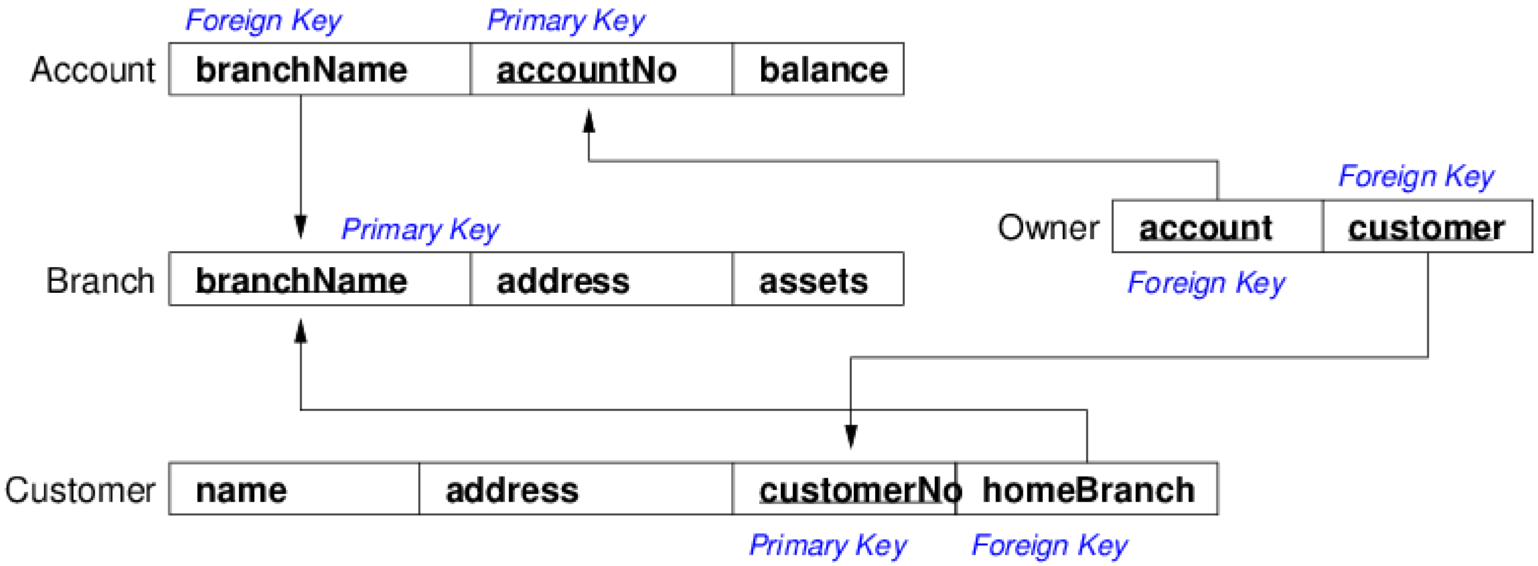
Relational Data Definition (cont)

So far, we have given relational schemas informally, e.g.

- individual relation schemas

```
Account(accountNo, branchName, balance)
Branch(branchNo, address, assets)
Customer(customerNo, name, address, homeBranch)
Owner(customer,branch)
```

- database schemas



SQL Data Definition Language

SQL is normally considered to be a query language.

However, it also has a data definition sub-language (DDL) for describing database schemas.

The SQL DDL allows us to specify:

- names of tables
- names and domains for attributes

- various types of constraints (e.g. primary/foreign keys)

It also provides mechanisms for performance tuning (see later).

Defining a Database Schema

Relations (tables) are described using:

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

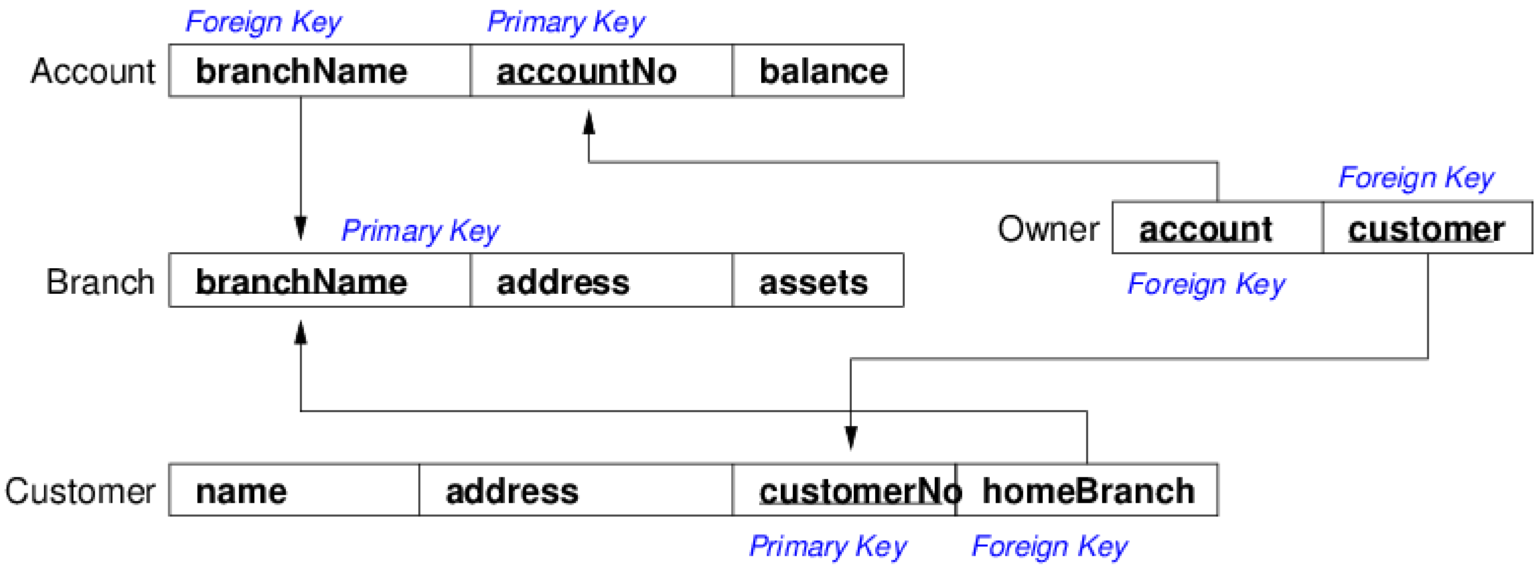
where *constraints* can include details about primary keys, foreign keys, default values, and constraints on attribute values.

This not only defines the table schema but also creates an empty instance of the table.

Tables are removed via **DROP TABLE** *RelName***;**

Defining a Database Schema (cont)

Consider the relational diagram for the example schema:



This shows explicitly the connection between foreign key attributes and their corresponding key attributes.

The SQL DDL provides notation for expressing this in the table definition.

Defining a Database Schema (cont)

SQL DDL for the example schema:

```
CREATE TABLE Branch (
    name            varchar(30),
    address         varchar(50),
    assets          float,
    PRIMARY KEY     (name)
);
```

Note: **`name`** is required to be unique and not null

Defining a Database Schema (cont)

More SQL DDL for the example schema:

```
CREATE TABLE Customer (
    customerNo      integer,
    name            varchar(40),
    address         varchar(50),
    homeBranch      varchar(30) not null,
    PRIMARY KEY     (customerNo),
```

```
    FOREIGN KEY   (homeBranch)
                  REFERENCES Branch(name)
);
```

Note: the **`not null`** captures total participation, i.e. every customer has a home branch.

Defining a Database Schema (cont)

More SQL DDL for the example schema:

```
CREATE TABLE Account (
    accountNo     char(5),
    heldAtBranch  varchar(30) not null,
    balance       float,
    PRIMARY KEY   (accountNo),
    FOREIGN KEY   (heldAtBranch)
                  REFERENCES Branch(name)
);
```

Note: the **`not null`** captures total participation, i.e. every accountis held at some branch.

Defining a Database Schema (cont)

More SQL DDL for the example schema:

```
CREATE TABLE OwnedBy (
    account        char(5),
    customer       integer,
    PRIMARY KEY    (account,customer),
    FOREIGN KEY    (account)
                   REFERENCES Account(accountNo),
    FOREIGN KEY    (customer)
                   REFERENCES Customer(customerNo)
);
```

Note: it is not possible in SQL to capture the semantics that Accounts are required to be owned by some Customer.

Declaring Keys

Primary keys:

- if a single attribute, declare with attribute, e.g.

```
accountNo char(5) PRIMARY KEY,
```

- if several attributes, declare with table constraints, e.g.

```
name    varchar(40),
address varchar(50),
...
PRIMARY KEY (name,address)
```

Declaring Keys (cont)

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

- generating a squence of unique values

- ensuring that tuples don't get assigned the same value

PostgreSQL's version:

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

Declaring Keys (cont)

Foreign keys:

- if a single attribute, specify *Relation(Attribute)*, e.g.

```
customer integer
            REFERENCES Customer(customerNo)
-- or
customer integer REFERENCES Customer
-- or
FOREIGN KEY (customer)
            REFERENCES Customer(customerNo)
```

Declaring Keys (cont)

Foreign keys: (cont)

- if several attributes, specify in table constraints, e.g.

```
name   varchar(40),
addr   varchar(50),
...
FOREIGN KEY (name,addr)
        REFERENCES Person(name,address)
```

If defining foreign keys with table constraints, must use **`FOREIGN KEY`** keywords.

Declaring Keys (cont)

Declaring foreign keys assures **referential integrity**.

Example:

`Account.branchName` refers to primary key of **`Branch`**

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/Oracle default behaviour)
- **set-NULL** the foreign key attributes in **`Account`** records
- **cascade** the deletion and remove **`Account`** records

Declaring Keys (cont)

Can force the alternative delete behaviours via e.g.

```
-- to cascade deletes
customer integer
          REFERENCES Customer(customerNo)
          ON DELETE CASCADE

-- to set foreign keys to NULL
customer integer
          REFERENCES Customer(customerNo)
          ON DELETE SET NULL
```

Declaring Keys (cont)

Example of different deletion strategies:

Branch

branchName	address	assets
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
~~Round Hill~~	~~Horseneck~~	~~8000000~~
North Town	Rye	3700000
Brighton	Brooklyn	7100000

Account *Original relation*

branchName	accountNo	balance
Downtown	A-101	500
Round Hill	A-215	700
Perryridge	A-102	400
Round Hill	A-305	350
Brighton	A-201	900
Redwood	A-222	700

Account *After deletion with SET NULL*

branchName	accountNo	balance
Downtown	A-101	500
NULL	A-215	700
Perryridge	A-102	400
NULL	A-305	350
Brighton	A-201	900
Redwood	A-222	700

Account *After deletion with CASCADE*

branchName	accountNo	balance
Downtown	A-101	500
Perryridge	A-102	400
Brighton	A-201	900
Redwood	A-222	700

Other Attribute Properties

Can specify that an attribute must have a non-null value, e.g.

```
barcode varchar(20) NOT NULL,
price   float NOT NULL
```

Can specify that an attribute must have a unique value, e.g.

```
barcode varchar(20) UNIQUE,
isbn    varchar(15) UNIQUE NOT NULL
```

Primary keys are automatically **`UNIQUE NOT NULL`**.

Other Attribute Properties (cont)

Can specify a **`DEFAULT`** value for an attribute

- will be assigned to attribute if no value is supplied during insert

Example:

```
CREATE TABLE Account (
```

```
    accountNo  char(5) PRIMARY KEY,
    branchName varchar(30)
                 REFERENCES Branch(name)
                 DEFAULT 'Central',
    balance    float DEFAULT 0.0
);

INSERT INTO Account(accountNo) VALUES ('A-456')
-- produces the tuple
Account('A-456','Central',0.0)
```

Attribute Value Constraints

In fact, **`NOT NULL`** is a special case of a constraint on the value that an attribute is allowed to take.

SQL has a more general mechanism for specifying such constraints.

```
attrName  type  CHECK ( condition )
```

The *Condition* can be arbitrarily complex, and may even involve other attributes, relations and **`SELECT`** queries.

(but many RDBMSs (e.g. Oracle and PostgreSQL) don't allow **`SELECT`** in **`CHECK`**)

Attribute Value Constraints (cont)

Example:

```
CREATE TABLE Example
(
    gender CHAR(1) CHECK (gender IN ('M','F')),
    Xvalue INT     NOT NULL,
    Yvalue INT     CHECK (Yvalue > Xvalue),
    Zvalue FLOAT   CHECK (Zvalue >
                          (SELECT MAX(price)
                           FROM   Sells)
                   )
);
```

Named Constraints

Any constraint in an SQL DDL can be named via

CONSTRAINT *constraintName* *constraint*

Example:

```
CREATE TABLE Example
(
   gender CHAR(1) CONSTRAINT GenderCheck
                  CHECK (gender IN ('M','F')),
   Xvalue INT     NOT NULL,
   Yvalue INT     CONSTRAINT XYOrder
                  CHECK (Yvalue > Xvalue),
);
```

SQL: Building Databases

Creating Databases

Mechanism for creating databases is typically DBMS-specific.

Many implement a (non-standard) SQL-like statement:

```
CREATE DATABASE DBname;
```

Many provide an external command, e.g PostgreSQL's

```
$ createdb DBname
```

Produces an empty database (no tables, etc) called *DBname*

Creating Databases (cont)

A database can be completely removed (no backup) via

```
$ dropdb DBname
```

This removes all tuples, all tables, all traces of *DBname*

Tables can be removed from a database schema via:

```
DROP TableName
```

All tuples can be removed from a table via:

```
DELETE FROM TableName
```

Creating Databases (cont)

Loading a schema with PostgreSQL:

```
$ createdb mydb
$ psql mydb
...
mydb=# \i schema.sql
...
```

or

```
$ psql -f schema.sql mydb
```

Running the above as:

```
$ psql -a -f schema.sql mydb
```

intersperses messages with the schema definition.

Useful for debugging, since errors appear in context.

Creating Databases (cont)

Re-loading schemas is not well-supported in PostgreSQL.

Simplest approach is:

```
$ dropdb mydb
$ createdb mydb
$ psql -f schema.sql mydb
```

An alternative is to leave DB but drop all tables:

```
$ psql mydb
...
mydb=# drop Table1;
mydb=# drop Table2;
etc. etc. in correct order
mydb=# \i schema.sql
...
```

Later, we'll see how to write functions to automate this.

Creating Databases (cont)

The entire contents of a database may be dumped:

```
$ pg_dump mydb > mydb.dump
```

Dumps all definitions needed to re-create entire DB

- table definitions (**`create table`**)
- constraints, including PKs and FKs
- all data from all tables
- domains, stored procedures, triggers, etc.

Some things change appearance, but mean the same thing
(e.g. **`varchar(30)`** becomes **`character varying(30)`**, etc.)

Creating Databases (cont)

Dumps may be used for backup/restore or copying DBs

```
$ pg_dump mydb > mydb.dump  -- backup
$ createdb newdb
$ psql newdb -f mydb.dump    -- copy
```

Result: **newdb** is a snapshot/copy of **mydb**.

- however, different object identifiers
- as changes are made, the two DBs will diverge

Data Modification in SQL

SQL provides mechanisms for modifying data (tuples) in tables:

- **INSERT** ... add a new tuple into a table
- **DELETE** ... remove tuples from a table (via condition)
- **UPDATE** ... modify values in exiting tuples (via condition)

Constraint checking is applied automatically on any change.

(See description of relational model for details of which checking applied when)

Data Modification in SQL (cont)

Also provides mechanisms for modifying table meta-data:

- **`CREATE TABLE`** ... create a new empty table
- **`DROP TABLE`** ... remove table from database (incl. tuples)
- **`ALTER TABLE`** ... change properties of existing table

Analogous operations are available on other kinds of database objects, e.g.

- **`CREATE VIEW`**, **`CREATE FUNCTION`**, **`CREATE RULE`**, ...
- **`DROP VIEW`**, **`DROP FUNCTION`**, **`DROP RULE`**, ...
- no **`UPDATE`** on these; use **`CREATE OR REPLACE`**

Insertion

Accomplished via the **`INSERT`** operation:

```
INSERT INTO RelationName
VALUES (val1, val2, val3, ...)

INSERT INTO RelationName(Attr1, Attr2, ...)
VALUES (valForAttr1, valForAttr2, ...)
```

Each form adds a single new tuple into *RelationName*.

Insertion (cont)

INSERT INTO *R* **VALUES** **(**v_1**,**v_2**,...)**

- values must be supplied for all attributes of *R*
- in same order as appear in **CREATE TABLE** statement

INSERT INTO *R*(*A*$_1$,*A*$_2$,...) **VALUES** **(**v_1**,**v_2**,...)**

- can specify any subset of attributes of *R*
- values must match attribute specification order
- unspecified attributes are assigned default or null

Insertion (cont)

Example: Add the fact that Justin likes 'Old'.

```
INSERT INTO Likes VALUES ('Justin','Old');
-- or --
INSERT INTO Likes(drinker,beer)
       VALUES('Justin', 'Old');
-- or --
INSERT INTO Likes(beer,drinker)
       VALUES('Old','Justin');
```

Example: Add a new drinker with unknown phone number.

```
INSERT INTO Drinkers(name,addr)
       VALUES('Frank','Coogee');
-- which inserts the tuple ...
('Frank', 'Coogee', null)
```

Insertion (cont)

Example: insertion with default values

```
ALTER TABLE Likes
   ALTER COLUMN beer SET DEFAULT 'New';
ALTER TABLE Likes
   ALTER COLUMN drinker SET DEFAULT 'Joe';

INSERT INTO Likes(drinker)
       VALUES('Fred');
INSERT INTO Likes(beer)
       VALUES('Sparkling Ale');

-- inserts the two new tuples ...
('Fred', 'New')
('Joe', 'Sparkling Ale')
```

Insertion (cont)

Example: insertion with insufficient values.

E.g. specify that drinkers' phone numbers cannot be **NULL**.

```
ALTER TABLE Drinkers
   ALTER COLUMN phone SET NOT NULL;
```

And then try to insert a new drinker whose phone number we don't know:

```
INSERT INTO Drinkers(name,addr)
       VALUES ('Zoe', 'Manly');

ERROR: ExecInsert: Fail to add null value
       in not null attribute phone
```

Insertion from Queries

Can use the result of a query to perform insertion of multiple tuples at once.

```
INSERT INTO Relation ( Subquery );
```

Tuples of *Subquery* must be projected into a suitable format (i.e. matching the tuple-type of *Relation*).

Insertion from Queries (cont)

Example: Create a relation of potential drinking buddies (i.e. people who go to the same bars as each other).

```
CREATE TABLE DrinkingBuddies (
   drinker varchar(20) references Drinkers(name),
   buddy   varchar(20) references Drinkers(name),
   primary key (drinker,buddy)
);

INSERT INTO DrinkingBuddies (
   SELECT a.drinker AS drinker,
          b.drinker AS buddy
   FROM   Frequents a, Frequents b
   WHERE  a.bar = b.bar AND a.drinker <> b.drinker
);
```

Note: this is better done as a view (treat this as a materialized view).

Bulk Insertion of Data

Tuples may be inserted individually:

```
insert into Stuff(x,y,s) values (2,4,'green');
insert into Stuff(x,y,s) values (4,8,null);
insert into Stuff(x,y,s) values (8,null,'red');
...
```

but this is tedious if 1000's of tuples are involved.

It is also inefficient, because all relevant constraints are checked after insertion of each tuple.

Bulk Insertion of Data (cont)

Most DBMSs provide non-SQL methods for bulk insertion:

- using a compact representation for each tuple
- loading all tuples without constraint checking
- doing all constraint checks at the end

Downside: if even one tuple is buggy, none are inserted.

Example: PostgreSQL's **copy** statement:

```
copy Stuff(x,y,s) from stdin;
2       4       green
4       8       \N
8       \N      red
\.
```

Can also copy from a named file.

Deletion

Accomplished via the **DELETE** operation:

```
DELETE FROM Relation
WHERE  Condition
```

Removes all tuples from *Relation* that satisfy *Condition*.

Example: Justin no longer likes Sparkling Ale.

```
DELETE FROM Likes
WHERE drinker = 'Justin'
      AND beer = 'Sparkling Ale';
```

Special case: Make relation *R* empty.

```
DELETE FROM R;
```

Deletion (cont)

Example: remove all expensive beers from sale.

```
DELETE FROM Sells
WHERE price >= 3.00;
```

Example: remove all drinkers with no fixed address.

```
DELETE FROM Drinkers
WHERE addr IS NULL;
```

This fails if such Drinkers are referenced in other tables.

Semantics of Deletion

Method A for **DELETE FROM** *R* **WHERE** *Cond* :

```
FOR EACH tuple T in R DO
    IF T satisfies Cond THEN
        remove T from relation R
    END
END
```

Method B for **DELETE FROM** *R* **WHERE** *Cond* :

```
FOR EACH tuple T in R DO
    IF T satisfies Cond THEN
        make a note of this T
    END
END
FOR EACH noted tuple T DO
    remove T from relation R
END
```

Does it matter which method is used?

Semantics of Deletion (cont)

Example: Delete all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS
      (SELECT name
       FROM   Beers
       WHERE  manf = b.manf
              AND name <> b.name);
```

Does the query result in ...

- deletion of all beers by brewers who make multiple beers
- deletion of all but the "last beer" by such brewers

Note: PostgreSQL disallows deletions with correlated subqueries (the **`FROM`** clause can be only a table name).

Semantics of Deletion (cont)

Example continued ...

Different results come from different evaluation methods ..

- Method A: iterate and evaluate condition for each beer
  - consider a manufacturer *M* who makes two beers *A* and *B*
  - when we reach *A*, there are two beers by *M*, so delete *A*
  - when we reach *B*, there are no other beers by *M*, so not deleted
- Method B: evalute condition and then do all deletions
  - both *A* and *B* test positive, and so both are deleted

Most RDBMSs use Method B, which matches natural semantics of **`DELETE`**.

Updates

An update allows you to modify values of specified attributes in specified tuples of a relation:

```
UPDATE R
SET    list of assignments
WHERE  Condition
```

Each tuple in relation *R* that satisfies *Condition* has the assignments applied to it.

Assignments may:

- assign constant values to attributes,
  e.g. **`SET price = 2.00`**
- use existing values in the tuple to compute new values,
  e.g. **`SET price = price * 0.5`**

Updates (cont)

Example: Adam changes his phone number.

```
UPDATE Drinkers
SET    phone = '9385-2222'
WHERE  name = 'Adam';
```

Example: John moves to Coogee.

```
UPDATE Drinkers
SET    addr = 'Coogee',
       phone = '9665-4321'
WHERE  name = 'John';
```

Updates (cont)

Can update many tuples at once (all tuples that satisfy condition)

Example: Make $3 the maximum price for beer.

```
UPDATE Sells
SET    price = 3.00
WHERE  price > 3.00;
```

Example: Increase beer prices by 10%.

```
UPDATE Sells
SET    price = price * 1.10;
```

Changing Tables

Accomplished via the **`ALTER TABLE`** operation:

```
ALTER TABLE Relation Modifications
```

Some possible modifications are:

- add a new column (attribute) (set value to **`NULL`** unless default given)
- change properties of an existing attribute (e.g. constraints)
- remove an attribute

Changing Tables (cont)

Example: Add phone numbers for hotels.

```
ALTER TABLE Bars
   ADD phone char(10) DEFAULT 'Unlisted';
```

This appends a new column to the table and sets value for this attribute to '**Unlisted**' in every tuple.

Specific phone numbers can subsequently be added via:

```
UPDATE Bars
SET    phone = '9665-0000'
WHERE  name = 'Coogee Bay Hotel';
```

If no default value is given, new column is set to all **NULL**.

For More Details ...

Full details are in the PostgreSQL Reference Manual.

See the section "SQL Commands", which has entries for

- **INSERT**, **DELETE**, **UPDATE**
- **CREATE** *X*, **DROP** *Y*, **ALTER** *Z*

You will become very familiar with some of these commands by end of session.

SQL: Queries

Queries

A query is a declarative program that retrieves data from a database.

Analogous to an expression in relational algebra.

But SQL does not implement relational algebra precisely.

Queries are used in two ways in RDBMSs:

- interactively (e.g. in **`psql`**)
  - the entire result is displayed in tabular format on the output
- by a program (e.g. in a PLpgSQL function)
  - the result tuples are consumed one-at-a-time by the program

---

Queries in SQL

The most common kind of SQL statement is the `SELECT` query:

```
SELECT attributes
FROM   relations
WHERE  condition
```

The result of this statement is a relation, which is typically displayed on output.

The **`SELECT`** statement contains the functionality of select, project and join from the relational algebra.

SELECT Example

The question "What beers are made by Toohey's?", can be phrased:

```
SELECT Name FROM Beers WHERE Manf = 'Toohey''s';
```

This gives a subset of the **`Beers`** relation, displayed as:

```
    name
-------------
 New
 Old
 Red
 Sheaf Stout
```

Notes:

- upper- and lower-case are not distinguished, except in strings.
- quotes are escaped by doubling them (`''''` is like C `'\''`)

Semantics of **SELECT**

For SQL **SELECT** statement on a single relation:

```
SELECT Attributes
FROM   R
WHERE  Condition
```

Formal semantics (relational algebra):

Proj[Attributes](Sel[Condition](R))

Semantics of **SELECT** (cont)

Operationally, we think in terms of a *tuple variable* ranging over all tuples of the relation.

Operational semantics:

```
FOR EACH tuple T in R DO
    check whether T satisfies the condition
                           in the WHERE clause
    IF it does THEN
        print the attributes of T that are
            specified in the SELECT clause
    END
END
```

Projection in SQL

For a relation R and attributes $X \subseteq R$, the relational algebra expression

$\pi_X(R)$ is implemented in SQL as:

```
SELECT X FROM R
```

Example: Names of drinkers = $\pi_{Name}(Drinkers)$

```
SELECT Name FROM Drinkers;

  name
--------
 Adam
 Gernot
 John
 Justin
```

Projection in SQL (cont)

Example: Names/addresses of drinkers = $\pi_{Name,Addr}(Drinkers)$

```
SELECT Name, Addr FROM Drinkers;

  name   |   addr
--------+----------
 Adam    | Randwick
 Gernot  | Newtown
 John    | Clovelly
 Justin  | Mosman
```

Projection in SQL (cont)

The symbol `*` denotes a list of all attributes.

Example: All information about drinkers = *(Drinkers)*

```
SELECT * FROM Drinkers;

  name   |   addr     |   phone
--------+-----------+------------
 Adam    | Randwick  | 9385-4444
 Gernot  | Newtown   | 9415-3378
```

```
 John    | Clovelly | 9665-1234
 Justin | Mosman   | 9845-4321
```

Renaming via AS

SQL implements renaming (ρ) via the **AS** clause within **SELECT**.

Example: rename Beers(name,manf) to Beers(beer,brewer)

```
SELECT name AS beer, manf AS Brewer
FROM Beers;

        beer          |    brewer
----------------------+---------------
 80/-                 | Caledonian
 Bigfoot Barley Wine | Sierra Nevada
 Burragorang Bock     | George IV Inn
 Crown Lager          | Carlton
 Fosters Lager        | Carlton
...
```

Expressions as Values in Columns

AS can also be used to introduce computed values (generalised projection)

Example: display beer prices in Yen, rather than dollars

```
SELECT bar, beer, price*120 AS PriceInYen  FROM Sells;

       bar         |       beer        |    priceinyen
-------------------+-------------------+------------------
 Australia Hotel   | Burragorang Bock  |              420
 Coogee Bay Hotel  | New               |              270
 Coogee Bay Hotel  | Old               |              300
 Coogee Bay Hotel  | Sparkling Ale     | 335.999994277954
 Coogee Bay Hotel  | Victoria Bitter   | 275.999994277954
 Lord Nelson       | Three Sheets      |              450
 Lord Nelson       | Old Admiral       |              450
 ...
```

Text in Result Table

Trick: to put specific text in output columns

- use string constant expression with **AS**

Example: using Likes(drinker, beer)

```
SELECT drinker, 'likes Cooper''s' AS WhoLikes
FROM   Likes
WHERE  beer = 'Sparkling Ale';

 drinker |    wholikes
---------+----------------
 Gernot  | likes Cooper's
 Justin  | likes Cooper's
```

Selection in SQL

The relational algebra expression $\sigma_{Cond}(Rel)$ is implemented in SQL as:

```
SELECT * FROM Rel WHERE Cond
```

Example: All about the bars at The Rocks

```
SELECT * FROM Bars WHERE Addr='The Rocks';
```

```
      name        |    addr     | license
------------------+-------------+---------
 Australia Hotel  | The Rocks   |  123456
 Lord Nelson      | The Rocks   |  123888
(2 rows)
```

The condition can be an arbitrarily complex boolean-valued expression using the operators mentioned previously.

Selection in SQL (cont)

Example: Find the price that The Regent charges for New

```
SELECT price
FROM   Sells
WHERE  bar = 'Regent Hotel' AND beer = 'New';

 price
-------
   2.2
```

This can be formatted better via **`to_char`**, e.g.

```
SELECT to_char(price,'$99.99') AS price
FROM   Sells
WHERE  bar = 'Regent Hotel' AND beer = 'New';

  price
---------
 $  2.20
```

`to_char()` supports a wide range of conversions.

Multi-relation **`SELECT`** Queries

Syntax is similar to simple **`SELECT`** queries:

```
SELECT Attributes
FROM   R1, R2, ...
WHERE  Condition
```

Difference is that **`FROM`** clause contains a list of relations.

Also, the condition typically includes cross-relation (join) conditions.

Multi-relation **SELECT** Queries (cont)

Example: Find the brewers whose beers John likes.

```
SELECT Manf as brewer
FROM   Likes, Beers
WHERE  beer = name AND drinker = 'John';

    brewer
---------------
 Caledonian
 Sierra Nevada
 Sierra Nevada
 Lord Nelson
```

Note: duplicates could be eliminated by using **DISTINCT**.

Multi-relation **SELECT** Queries (cont)

The above example corresponds to a relational algebra evaluation like:

```
BeerDrinkers = Likes Join[beer=name] Beers
JohnsBeers   = Sel[drinker=John](BeerDrinkers)
Brewers      = Proj[manf](JohnsBeers)
Result       = Rename[manf->brewer](Brewers)
```

The SQL compiler knows how to translate tests

- involving attributes from two relations into a join
- involving attributes from one relations into a selection

Semantics of Multi-Relation **SELECT**

For SQL **SELECT** statement on several relations:

```
SELECT Attributes
FROM   R1, R2, ... Rn
WHERE  Condition
```

Formal semantics (relational algebra):

Proj[Attributes](Sel[Condition](R1 × R2 × ... Rn))

Semantics of Multi-Relation **SELECT** (cont)

Operational semantics of **SELECT**:

```
FOR EACH tuple T1 in R1 DO
    FOR EACH tuple T2 in R2 DO
        ...
            check WHERE condition for current
               assignment of T1, T2, ... vars
            IF holds THEN
               print attributes of T1, T2, ...
                           specified in SELECT
            END
        ...
    END
END
```

Requires one tuple variable for each relation, and nested loops over relations. This is not how it's actually computed!

Name Clashes in Conditions

If a selection condition

- refers to two relations
- the relations have attributes with the same name

use the relation name to disambiguate.

Example: Which hotels have the same name as a beer?

```
SELECT Bars.name
FROM   Bars, Beers
WHERE  Bars.name = Beers.name;
```

(The answer to this query is empty, but there is nothing special about this)

Name Clashes in Conditions (cont)

Can use such qualified names, even if there is no ambiguity:

```
SELECT Sells.beer
FROM   Sells
WHERE  Sells.price > 3.00;
```

Advice:

- qualify attribute names only when absolutely necessary

Note:

- SQL's **`AS`** operator is only for renaming output
- it provides no help with disambiguation

Explicit Tuple Variables

The relation-dot-attribute convention doesn't help if we happen to use the same relation twice in a **`SELECT`**.

To handle this, we need to define new names for each "instance" of the relation in the **`FROM`** clause.

Syntax:

```
SELECT r1.a, r2.b
FROM   R r1, R r2
```

```
WHERE  r1.a = r2.a
```

Explicit Tuple Variables (cont)

Example: Find pairs of beers by the same manufacturer.

```
SELECT b1.name, b2.name
FROM   Beers b1, Beers b2
WHERE  b1.manf = b2.manf AND b1.name < b2.name;

        name          |       name
---------------------+------------------
 Crown Lager          | Fosters Lager
 Crown Lager          | Invalid Stout
 Crown Lager          | Melbourne Bitter
 Crown Lager          | Victoria Bitter
 Fosters Lager        | Invalid Stout
 Fosters Lager        | Melbourne Bitter
 ...
```

The second part of the condition is used to avoid:

- pairing a beer with itself e.g. **`(New,New)`**
- same pairs with different order e.g. **`(New,Old)`** **`(Old,New)`**

Explicit Tuple Variables (cont)

A common alternative syntax for

```
SELECT r1.a, r2.b
FROM   R r1, R r2
WHERE  r1.a = r2.a
```

uses the **as** keyword

```
SELECT r1.a, r2.b
FROM   R as r1, R as r2
WHERE  r1.a = r2.a
```

Explicit Joins

SQL supports syntax for explicit joins:

```
SELECT...FROM A natural join B
SELECT...FROM A join B using (A1,...,An)
SELECT...FROM A join B on Condition
```

The **`natural join`** and **`join using`** forms assume that the join attributes are named the same in each relation.

Explicit Joins (cont)

Example: Find the beers sold at bars where John drinks

```
SELECT Sells.bar, beer, price
FROM   Sells, Frequents
WHERE  drinker = 'John'
       AND Sells.bar = Frequents.bar;
```

could also be expressed as

```
SELECT bar, beer, price
FROM   Sells natural join Frequents
WHERE  drinker='John';
       -- joins on the only common attribute: bar
```

Explicit Joins (cont)

The example could also be expressed as

```
SELECT bar, beer, price
FROM   Sells join Frequents using (bar)
WHERE  drinker='John';
       -- only one bar attribute in join result
```

or

```
SELECT Sells.bar, beer, price
FROM   Sells join Frequents
           on Sells.bar = Frequents.bar
WHERE  drinker='John';
       -- bar attribute occurs twice in join result
```

Outer Join

Join only produces tuples where there are matching values in both of the relations involved in the join.

Often, it is useful to produce results for all tuples in one relation, even if it has no matches in the other.

Consider the query: for each region, find out who drinks there.

Outer Join (cont)

A regular join only gives results for regions where people drink.

```
SELECT B.addr, F.drinker
FROM   Bars as B join Frequents as F
       on (bar = name)
ORDER BY addr;

   addr    | drinker
-----------+---------
 Coogee    | Adam
 Coogee    | John
 Kingsford | Justin
 Sydney    | Justin
 The Rocks | John
```

But what if we want a result that shows all regions, even if there are no drinkers there?

Outer Join (cont)

An outer join solves this problem.

For *R* **OUTER JOIN** *S*

- all "tuples" in *R* have an entry in the result
- if a tuple from *R* matches a tuple in *S*, we get the normal join result tuple
- if a tuple from *R* has no matches in *S*, the attributes supplied by *S* are **NULL**

This outer join variant is called **LEFT OUTER JOIN**.

Outer Join (cont)

Solving the example query with an outer join:

```
SELECT B.addr, F.drinker
FROM   Bars as B
          left outer join
       Frequents as F
          on (bar = name)
ORDER BY B.addr;

    addr     | drinker
------------+---------
 Coogee      | Adam
 Coogee      | John
 Kingsford   | Justin
 Randwick    |
 Sydney      | Justin
 The Rocks   | John
```

Note that Randwick is now mentioned (because of the Royal Hotel).

Outer Join (cont)

Many RDBMSs provide three variants of outer join:

- *R* **LEFT OUTER JOIN** *S*
    - behaves as described above
- *R* **RIGHT OUTER JOIN** *S*
    - includes all tuples from *S* in the result
    - **NULL**–fills any *S* tuples with no matches in *R*
- *R* **FULL OUTER JOIN** *S*
    - includes all tuples from *R* and *S* in the result
    - those without matches in other relation are **NULL**–filled

---

Subqueries

The result of a **SELECT-FROM-WHERE** query can be used in the **WHERE** clause of another query.

Simplest Case: Subquery returns a single, unary tuple

Can treat the result as a single constant value and use in expressions.

Syntax:

```
SELECT *
FROM   R
WHERE  R.a = (SELECT x FROM S WHERE Cond)
             -- assume only one result
```

Subqueries (cont)

Example: Find bars that serve New at the same price as the Coogee Bay Hotel charges for VB.

```
SELECT bar
FROM   Sells
WHERE  beer = 'New' AND
       price =
          (SELECT price
           FROM   Sells
           WHERE  bar = 'Coogee Bay Hotel'
                  AND beer = 'Victoria Bitter');

     bar
-------------
 Royal Hotel
```

The inner query finds the price of VB at the CBH, and uses this as an argument to a test in the outer query.

Subqueries (cont)

Note the potential ambiguity in references to attributes of **Sells**

```
SELECT bar
FROM   Sells
WHERE  beer = 'New' AND
       price =
          (SELECT price
           FROM   Sells
           WHERE  bar = 'Coogee Bay Hotel'
                  AND beer = 'Victoria Bitter');
```

This introduces notions of scope: an attribute refers to the most closely nested relation with that attribute.

Parentheses around the subquery are required (and set the scope).

Subqueries (cont)

Note also that the query could be answered via:

```
SELECT s1.bar
FROM   Sells as s1, Sells as s2
WHERE  s1.beer = 'New'
       AND s1.price = s2.price
       AND s2.bar = 'Coogee Bay Hotel'
       AND s2.beer = 'Victoria Bitter';
```

In general, expressing a query via joins will be much more efficient than expressing it with sub-queries.

Subqueries (cont)

Complex Case: Subquery returns multiple unary tuples.

Treat it as a list of values, and use the various operators on lists/sets (e.g. **`IN`**).

Complex Case: Subquery returns a relation.

Most of the "list operators" also work on relations.

The **IN** Operator

Tests whether a specified tuple is contained in a relation.

tuple **IN** *relation* is true iff the tuple is contained in the relation.

Conversely for *tuple* **NOT IN** *relation*.

Syntax:

```
SELECT *
FROM   R
WHERE  R.a IN (SELECT x FROM S WHERE Cond)
              -- assume multiple results
```

The **IN** Operator (cont)

Example: Find the name and brewer of beers that John likes.

```
SELECT *
FROM   Beers
```

```
WHERE  name IN
          (SELECT beer
           FROM   Likes
           WHERE  drinker = 'John');

        name         |     manf
---------------------+---------------
 80/-                | Caledonian
 Bigfoot Barley Wine | Sierra Nevada
 Pale Ale            | Sierra Nevada
 Three Sheets        | Lord Nelson
```

The subexpression answers the question "What are the names of the beers that John likes?"

The **`IN`** Operator (cont)

Note that this query can be answered equally well without using **`IN`**.

```
SELECT Beers.name, Beers.manf
FROM   Beers, Likes
WHERE  Likes.drinker = 'John' AND
```

```
        Likes.beer = Beers.name;

         name          |     manf
-----------------------+---------------
 80/-                  | Caledonian
 Bigfoot Barley Wine   | Sierra Nevada
 Pale Ale              | Sierra Nevada
 Three Sheets          | Lord Nelson
```

The version with the subquery corresponds more closely to the way the original query was expressed, and is probably "more natural".

The subquery version is, however, potentially less efficient.

The `EXISTS` Function

`EXISTS(`*relation***`)`** is true iff the relation is non-empty.

Example: Find the beers that are the unique beer by their manufacturer.

```
SELECT name, manf
FROM   Beers b1
```

```
WHERE  NOT EXISTS
          (SELECT *
           FROM   Beers
           WHERE  manf = b1.manf
                  AND name != b1.name);
```

Note the scoping rule: to refer to outer **Beers** in the inner subquery, we need to define a named tuple variable (in this example **b1**).

A subquery that refers to values from a surrounding query is called a **correlated subquery**.

Quantifiers

ANY and **ALL** behave as existential and universal quantifiers respectively.

Example: Find the beers sold for the highest price.

```
SELECT beer
FROM   Sells
```

```
WHERE  price >=
          ALL(SELECT price FROM sells);
```

Beware: in common use, "any" and "all" are often synonyms.

E.g. "I'm better than any of you" vs. "I'm better than all of you".

Union, Intersection, Difference

SQL implements the standard set operations on "union-compatible" relations:

R1 **UNION** *R2*	set of tuples in either *R1* or *R2*
R1 **INTERSECT** *R2*	set of tuples in both *R1* and *R2*
R1 **EXCEPT** *R2*	set of tuples in *R1* but not *R2*

Oracle deviates from the SQL standard and uses **MINUS** for **EXCEPT**; PostgreSQL follows the standard.

Union, Intersection, Difference (cont)

Example: Find the drinkers and beers such that the drinker likes the beer and frequents a bar that sells it.

```
(SELECT * FROM Likes)
INTERSECT
(SELECT drinker,beer
 FROM   Sells natural join Frequents);
```

```
 drinker |       beer
---------+------------------
 Adam    | New
 John    | Three Sheets
 Justin  | Victoria Bitter
```

Bag Semantics of SQL

An SQL relation is really a **bag** (**multiset**):

- it may contain the same tuple more than once

- unlike lists, there is no specified order on the elements
- example: *{1, 2, 1, 3}* is a bag and is not a set

This changes the semantics of the "set" operators **`UNION`**, **`INTERSECT`** and **`MINUS`**.

Bag Semantics of SQL (cont)

Bag Union

Sum the times an element appears in the two bags

- example: *{1,2,1} ∪ {1,2,3} = {1,1,1,2,2,3}*

Bag Intersection

Take the minimum number of occurrences from each bag.

- example: *{1,2,1} ∩ {1,2,3} = {1,2}*

Bag Difference

Proper-subract the number of occurrences in the two bags.

- example: *{1,2,1}* – *{1,2,3}* = *{1}*

Forcing Bag/Set Semantics

Default result for **SELECT-FROM-WHERE** is a bag.

Default result for **UNION**, **INTERSECT**, **MINUS** is a set.

Why the difference?

A bag can be produced faster because no need to worry about eliminating duplicates (which typically requires sorting).

Can force set semantics with **SELECT DISTINCT**.

Can force bag semantics with **UNION ALL**, ...

Forcing Bag/Set Semantics (cont)

Example: What beer manufacturers are there?

```
SELECT DISTINCT manf FROM Beers;

     manf
---------------
 Caledonian
 Carlton
 Cascade
 Cooper's
 George IV Inn
 Lord Nelson
 Sierra Nevada
 Toohey's
```

Note that the result is sorted.

If we omit **`DISTINCT`**, we get 18 unsorted tuples in the result.

Division

Not all SQL implementations provide a divide operator, but the same effect can be achieved by combination of existing operations.

Example: Find bars that each sell all of the beers Justin likes.

```
SELECT DISTINCT a.bar
FROM   Sells a
WHERE  NOT EXISTS (
           (SELECT beer FROM Likes
            WHERE drinker = 'Justin')
           EXCEPT
           (SELECT beer FROM Sells b
            WHERE bar = a.bar)
       );
```

Selection with Aggregation

Selection clauses can contain aggregation operations.

Example: What is the average price of New?

```
SELECT AVG(price)
FROM   Sells
WHERE  beer = 'New';

       avg
------------------
 2.38749998807907
```

Note:

- the bag semantics of SQL gives the correct result here
- the price for New in all hotels will be included, even if two hotels sell it at the same price
- if we used set semantics, we'd get the average of all the different prices for New.

Selection with Aggregation (cont)

If we want set semantics, we can force using **`DISTINCT`**.

Example: How many different bars sell beer?

```
SELECT COUNT(DISTINCT bar)
FROM   Sells;

 count
-------
     6
```

Without **`DISTINCT`**, the result is 15 ... the number of entries in the **`Sells`** table.

Aggregation operators

The following operators apply to a list (bag) of numeric values in one column of a relation:

```
SUM    AVG    MIN    MAX    COUNT
```

The notation **`COUNT(*)`** gives the number of tuples in a relation.

Example: How many different beers are there?

```
SELECT COUNT(*) FROM Beers;

 count
-------
    18
```

Grouping

SELECT-FROM-WHERE can be followed by **GROUP BY** to:

- partition result relation into groups
  (according to values of specified attribute)
- summarise some (several) aspects of each group
- output relation contains one tuple per group

Example: How many beers does each brewer make?

There is one entry for each beer by each brewer in the **Beers** table ...

Grouping (cont)

The following gives us a list of brewers:

```
SELECT manf FROM Beers;
```

The number of occurrences of each brewer is the number of beers that they make.

Ordering the list makes it much easier to work out:

```
SELECT manf FROM Beers ORDER BY manf;
```

but we still need to count length of runs by hand.

Grouping (cont)

If we group the runs, we can **`count(*)`** them:

```
SELECT   manf, COUNT(manf)
FROM     Beers
```

```
GROUP BY manf;

     manf       | count
----------------+-------
 Caledonian     |     1
 Carlton        |     5
 Cascade        |     1
 Cooper's       |     2
 George IV Inn  |     1
 Lord Nelson    |     2
 Sierra Nevada  |     2
 Toohey's       |     4
```

Grouping (cont)

GROUP BY is used as follows:

```
SELECT   attributes/aggregations
FROM     relations
WHERE    condition
GROUP BY attribute
```

Semantics:

1. apply product and selection as for **`SELECT-FROM-WHERE`**
2. partition result into groups based on values of *attribute*
3. apply any aggregation separately to each group

Grouping (cont)

The query

```
select manf,count(manf) from Beers group by manf;
```

first produces a partitioned relation and then counts the number of tuples in each partition:

Name	Manf	
80–	Caledonian	1
Crown Lager	Carlton	
Fosters Lager	Carlton	
Invalid Stout	Carlton	5
Melbourne Bitter	Carlton	
Victoria Bitter	Carlton	
Premium Lager	Cascade	1
Sparkling Ale	Coopers	2
Stout	Coopers	

Name	Manf	
Burragorang Bock	George IV Inn	1
Old Admiral	Lord Nelson	2
Three Sheets	Lord Nelson	
Bigfoot Barley Wine	Sierra Nevada	2
Pale Ale	Sierra Nevada	
New	Toohey's	
Old	Toohey's	4
Red	Toohey's	
Sheaf Stout	Toohey's	

Grouping (cont)

Grouping is typically used in queries involving the phrase "for each".

Example: For each drinker, find the average price of New at the bars they go to.

```
SELECT   drinker, AVG(price) as "Avg.Price"
FROM     Frequents, Sells
WHERE    beer = 'New'
         AND Frequents.bar = Sells.bar
```

```
GROUP BY drinker;

 drinker | Avg.Price
---------+-----------
 Adam    |      2.25
 John    |      2.25
 Justin  |       2.5
```

Restrictions on **SELECT** Lists

When using grouping, every attribute in the **SELECT** list must:

- have an aggregation operator applied to it   OR
- appear in the **GROUP-BY** clause

Incorrect Example: Find the hotel that sells 'New' cheapest.

```
SELECT bar, MIN(price)
FROM   Sells
WHERE  beer = 'New';
```

PostgreSQL's response to this query:

```
ERROR: Attribute sells.bar must be GROUPed
        or used in an aggregate function
```

Restrictions on **SELECT** Lists (cont)

How to answer the query: Which bar sells 'New' cheapest?

```
SELECT bar
FROM   Sells
WHERE  beer = 'New' AND
       price <= (SELECT MIN(price)
                 FROM   Sells
                 WHERE  beer = 'New');

     bar
--------------
 Regent Hotel
```

Restrictions on **SELECT** Lists (cont)

Also, cannot use grouping to simply re-order results.

Incorrect Example: Print beers grouped by their manufacturer.

```
SELECT name, manf FROM Beers
GROUP BY manf;

ERROR: Attribute beers.name must be GROUPed
       or used in an aggregate function
```

Restrictions on **SELECT** Lists (cont)

How to print beers grouped by their manufacturer?

```
SELECT name, manf FROM Beers
ORDER BY manf;

         name           |     manf
------------------------+---------------
 80/-                   | Caledonian
 Crown Lager            | Carlton
 Fosters Lager          | Carlton
 Invalid Stout          | Carlton
 Melbourne Bitter       | Carlton
```

```
 Victoria Bitter      | Carlton
 Premium Lager        | Cascade
...
```

ORDER BY can be applied to multiple attributes.

Eliminating Groups

In some queries, you can use the **WHERE** condition to eliminate groups.

Example: Average beer price by suburb excluding hotels in The Rocks.

```
SELECT   Bars.addr, AVG(Sells.price)
FROM     Sells, Bars
WHERE    Bars.addr != 'The Rocks'
         AND Sells.bar = Bars.name
GROUP BY Bars.addr;
```

For more complex conditions on groups, use the **HAVING** clause.

Eliminating Groups (cont)

HAVING is used to qualify a **GROUP-BY** clause:

```
SELECT   attributes/aggregations
FROM     relations
WHERE    condition (on tuples)
GROUP BY attribute
HAVING   condition (on group);
```

Semantics of **HAVING**:

1. generate the groups as for **GROUP-BY**
2. eliminate groups not satisfying **HAVING** condition
3. apply aggregations to remaining groups

Note: **HAVING** condition can use relations/variables from **FROM** just like **WHERE** condition, but variables range over each group.

Eliminating Groups (cont)

Example: Find the average price of common beers (i.e. those that are served in more than one hotel).

```
SELECT   beer,
         to_char(AVG(price),'9.99')
           as "$$$"
FROM     Sells
GROUP BY beer
HAVING   COUNT(bar) > 1;

      beer       |  $$$
-----------------+-------
 New             |  2.39
 Old             |  2.53
 Victoria Bitter |  2.40
```

Eliminating Groups (cont)

The **`HAVING`** condition can have components that do not use aggregation.

Example: Find the average price of beers that are either commonly served (in more than one hotel) or are manufactured by Cooper's.

```
SELECT beer, AVG(price)
FROM   Sells
```

```
GROUP BY beer
HAVING COUNT(bar) > 1
       OR beer in
          (SELECT name
           FROM   beers
           WHERE  manf = 'Cooper''s');

      beer        |        avg
------------------+-------------------
 New              | 2.38749998807907
 Old              | 2.53333330154419
 Sparkling Ale    | 2.79999995231628
 Victoria Bitter  | 2.39999997615814
```

Eliminating Groups (cont)

GROUP-BY and HAVING also provide an alternative formulation for division.

Example: Find bars that each sell all of the beers Justin likes.

```
SELECT DISTINCT S.bar
FROM   Sells S, Likes L
WHERE  S.beer = L.beer
       AND L.drinker = 'Justin'
GROUP BY bar
HAVING count(S.beer) =
           (SELECT count(beer) FROM Likes
            WHERE drinker = 'Justin');
```

Partitions and Window Functions

Sometimes it is useful to

- partition a table into groups
- compute results that apply to each group
- use these results with individual tuples in the group

Comparison with **`GROUP-BY`**

- **`GROUP-BY`** produces one tuple for each group

- **PARTITION** augments each tuple with group-based value(s)
- can use other functions than aggregates (e.g. ranking)
- can use attributes other than the partitioning ones

Partitions and Window Functions (cont)

Syntax for **PARTITION**:

```
SELECT attr1, attr2, ...,
       aggregate1 OVER (PARTITION BY attri),
       aggregate2 OVER (PARTITION BY attrj), ...
FROM   Table
WHERE  condition on attributes
```

Note: the *condition* cannot include the *aggregate* value(s)

Partitions and Window Functions (cont)

Example: show each city with daily temperature and temperature range

Schema: *Weather(city,date,temperature)*

```
SELECT  city,  date,  temperature as temp,
        min(temperature) OVER (PARTITION BY city) as lowest,
        max(temperature) OVER (PARTITION BY city) as highest
FROM    Weather;
```

Output: *Result(city, date, temp, lowest, highest)*

Partitions and Window Functions (cont)

Example showing **GROUP BY** and **PARTITION** difference:

```
SELECT city, min(temperature) max(temperature)
FROM   Weather  GROUP BY city
```

Result: one tuple for each city *Result(city,min,max)*

```
SELECT city,  date,  temperature as temp,
       min(temperature) OVER (PARTITION BY city),
       max(temperature) OVER (PARTITION BY city)
FROM   Weather;
```

Result: one tuple for each temperature measurement.

Partitions and Window Functions (cont)

Example: get a list of low-scoring students in each course
(low-scoring = mark is less than average mark for class)

Schema: *Enrolment(course,student,mark)*

Approach:

- generate tuples containing *(student,mark,classAvg)*
- select just those tuples satisfying *(mark < classAvg)*

Implementation of first step via window function

```
SELECT course, student, mark,
       avg(mark) OVER (PARTITION BY course)
FROM   Enrolments;
```

We now look at several ways to complete this data request ...

Complex Queries

For complex queries, it is often useful to

- break the query into a collection of smaller queries
- define the top-level query in terms of these

This can be accomplished in three ways in SQL:

- views (discussed in detail below)
- subqueries in the **FROM** clause
- subqueries in a **WITH** clause

Note that we cannot "correlate" such subqueries in the same way as we can subqueries in the **WHERE** clause.

Complex Queries (cont)

Defining complex queries using views:

```
CREATE VIEW CourseMarksAndAverages(course,student,mark,avg)
AS
SELECT course, student, mark,
       avg(mark) OVER (PARTITION BY course)
FROM   Enrolments;

SELECT course, student, mark
FROM   CourseMarksAndAverages
WHERE  mark < avg;
```

Complex Queries (cont)

In the general case:

```
CREATE VIEW View1(a,b,c,d) AS Query1;
CREATE VIEW View2(e,f,g) AS Query2;
...
SELECT a,f FROM View1, View2 WHERE c = e;
```

Notes:

- look like tables ("virtual" tables)
- exist as objects in the database (stored queries)
- useful if specific query is required frequently

Complex Queries (cont)

Defining complex queries using **FROM** subqueries:

```
SELECT course, student, mark
FROM   (SELECT course, student, mark,
               avg(mark) OVER (PARTITION BY course)
        FROM   Enrolments) AS CourseMarkAndAverages
WHERE  mark < avg;
```

Avoids the need to define views.

Complex Queries (cont)

In the general case:

```
SELECT attributes
FROM   (Query1) AS X,
       (Query2) AS Y,
       ...
WHERE  X.a = Y.b AND other conditions
```

Notes:

- must provide name for each subquery, even if never used
- subquery table inherits attribute names from query
  (e.g. in the above, we assume that *Query*$_1$ returns an attribute called **`a`**)

Complex Queries (cont)

Defining complex queries using **`WITH`**:

```
WITH CourseMarksAndAverages AS
     (SELECT course, student, mark,
             avg(mark) OVER (PARTITION BY course)
      FROM   CourseEnrolments)
SELECT course, student, mark, avg
```

```
FROM   CourseMarksAndAverages
WHERE  mark < avg;
```

Avoids the need to define views.

Complex Queries (cont)

In the general case:

```
WITH    Name1(a,b,c) AS (Query1),
        Name2 AS (Query1), ...
SELECT  attributes
FROM    Name1, Name2, ...
WHERE   conditions with attributes of Name1 and Name2
```

Notes:

- *Name*$_1$, etc. are like temporary tables
- named tables inherit attribute names from query

Recursive Queries

WITH also provides the basis for recursive queries.

Recursive queries are structured as:

```
WITH RECURSIVE Recurs(attributes) AS (
     SELECT ... not involving Recurs
   UNION
     SELECT ... FROM Recurs, ...
)
SELECT attributes
FROM   Recurs, ...
WHERE  condition involving Recurs attributes
```

Useful for scenarios in which we need to traverse multi-level relationships.

Recursive Queries (cont)

Simple example involving a "virtual" table.

Sum the numbers from 1 to 100:

```
WITH RECURSIVE t(n) AS (
     SELECT 1
   UNION
     SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Recursive Queries (cont)

In the general case:

```
WITH RECURSIVE Recurs(attributes) AS (
     Q1 (non-recursive query)
   UNION
     Q2 (recursive query)
)
SELECT * FROM Recurs;
```

Requires the use of several temporary tables:

- *Result* is the final result of evaluating the query
- *Working*, *Temp* hold intermediate results

Recursive Queries (cont)

How recursion works:

```
Working =  Result = evaluate Q1
while (Working table is not empty) {
    Temp = evaluate Q2, using Working in place of Recurs
    Temp = Temp - Result
    Result = Result UNION Temp
    Working = Temp
}
```

I.e. generate new tuples until we see nothing not already seen.

Recursive Queries (cont)

Example: count number of each sub-part in a given part.

Schema: *Parts(part, sub_part, quantity)*

```
WITH RECURSIVE IncludedParts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity
    FROM   Parts WHERE part = GivenPart
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM   IncludedParts i, Parts p
    WHERE  p.part = i.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM   IncludedParts
GROUP  BY sub_part
```

SQL: Views

Views

A view is like a "virtual relation" defined via a query.

View definition and removal:

```
CREATE VIEW ViewName AS Query

CREATE VIEW ViewName [ (AttributeNames) ]
AS Query

DROP VIEW ViewName
```

The *Query* may be any SQL query, involving

- other views (*intensional relations*)
- stored tables (*extensional relations*)

Views (cont)

The stored tables in a view are referred to as base tables.

Views are defined only after their base tables are defined.

A view is valid only as long as its underlying query is valid.

Dropping a view has no effect on the base tables.

Views (cont)

Example: An avid Carlton drinker might not be interested in any other kinds of beer.

```
CREATE VIEW  MyBeers  AS
   SELECT name, manf
   FROM   Beers
   WHERE  manf = 'Carlton';

SELECT * FROM MyBeers;

       name        |  manf
-------------------+---------
 Crown Lager       | Carlton
 Fosters Lager     | Carlton
 Invalid Stout     | Carlton
 Melbourne Bitter  | Carlton
 Victoria Bitter   | Carlton
```

Views (cont)

A view might not use all attributes of the base relations.

Example: We don't really need the address of inner-city hotels.

```
CREATE VIEW  InnerCityHotels  AS
   SELECT name, license
   FROM   Bars
   WHERE  addr in ('The Rocks','Sydney');

SELECT * FROM InnerCityHotels;

      name        | license
-----------------+---------
 Australia Hotel |  123456
 Lord Nelson     |  123888
 Marble Bar      |  122123
```

Views (cont)

A view might use computed attribute values.

Example: Number of beers produced by each brewer.

```
CREATE VIEW BeersBrewed AS
   SELECT manf as brewer,
          count(*) as nbeers
   FROM   beers GROUP BY manf;

SELECT * FROM BeersBrewed;

    brewer      | nbeers
---------------+--------
 Caledonian     |      1
 Carlton        |      5
 Cascade        |      1
 ...
```

Renaming View Attributes

This can be achieved in two different ways:

```
CREATE VIEW  InnerCityHotels  AS
   SELECT name AS pub, license AS lic
   FROM   Bars
   WHERE  addr IN ('The Rocks', 'Sydney');
```

```
CREATE VIEW  InnerCityHotels(pub,lic)  AS
   SELECT name, license
   FROM   Bars
   WHERE  addr IN ('The Rocks', 'Sydney');
```

Using Views

Views can be used in queries as if they were stored relations.

However, they differ from stored relations in two important respects:

- their "value" can change without being explicitly modified
  (i.e. a view may change whenever one of its base tables is updated)
- they may not be able to be explicitly modified (updated)
  (only a certain simple kinds of views can be explicitly updated)

Using Views (cont)

Example: of view changing when base table changes.

```
SELECT * FROM InnerCityHotels;
      name        | license
-----------------+---------
 Australia Hotel |  123456
 Lord Nelson     |  123888
 Marble Bar      |  122123

-- then the Lord Nelson goes broke
DELETE FROM Bars WHERE name = 'Lord Nelson';

-- no explict update has been made to InnerCityHotels
SELECT * FROM InnerCityHotels;
      name        | license
-----------------+---------
 Australia Hotel |  123456
 Marble Bar      |  122123
```

Updating Views

Explicit updates are allowed on views satisfying the following:

- the view involves a single relation **R**

- the **WHERE** clause does not involve **R** in a subquery
- the **WHERE** clause only uses attributes from the **SELECT**

Attributes not in the view's **SELECT** will be set to **NULL** in the base relation after an insert into the view.

Updating Views (cont)

Example: Our **InnerCityHotel** view is not updatable.

```
INSERT INTO InnerCityHotels
VALUES ('Jackson''s on George', '9876543');
```

creates a new tuple in the **Bars** relation:

```
(Jackson's on George,  NULL,  9876543)
```

when we **SELECT** from the view, this new tuple does not satisfy the view condition:

```
addr IN ('The Rocks', 'Sydney')
```

Updating Views (cont)

If we had chosen to omit the **license** attribute instead, it would be updatable:

```
CREATE VIEW CityHotels AS
   SELECT name,addr FROM Bars
   WHERE  addr IN ('The Rocks', 'Sydney');

INSERT INTO CityHotels
VALUES ('Jackson''s on George', 'Sydney');
```

creates a new tuple in the **Bars** relation:

```
(Jackson's on George, Sydney, NULL)
```

which would appear in the view after the insertion.

Updating Views (cont)

Updatable views in PostgreSQL require us to specify explicitly how updates are done:

```
CREATE RULE InsertCityHotel AS
   ON INSERT TO CityHotels
   DO INSTEAD
      INSERT INTO Bars VALUES
         (new.name, new.addr, NULL);

CREATE RULE UpdateCityHotel AS
   ON UPDATE TO CityHotels
   DO INSTEAD
      UPDATE Bars
      SET    addr = new.addr
      WHERE  name = old.name;
```

Evaluating Views

Two alternative ways of implementing views:

- re-writing rules (or macros)

  - when a view is used in a query, the query is re-written
  - after rewriting, becomes a query only on base relations
- explicit stored relations (called materialized views)
  - the view is stored as a real table in the database
  - updated appropriately when base tables are modified

The difference: underlying query evaluated either at query time or at update time.

Evaluating Views (cont)

Example: Using the **InnerCityHotels** view.

```
CREATE VIEW  InnerCityHotels  AS
   SELECT name, license
   FROM   Bars
   WHERE  addr IN ('The Rocks', 'Sydney');

SELECT name
FROM   InnerCityHotels
WHERE  license = '123456';
```

```
--is rewritten into the following form before execution

SELECT name
FROM   Bars
WHERE  addr IN ('The Rocks', 'Sydney')
       AND license = '123456';
```

Evaluating Views (cont)

Demonstrate the rewriting process via relational algebra.

Some abbreviations

- **`n`** = **`name`**, **`l`** = **`license`**
- **`L`** = **`license = ''123456'`**
- **`A`** = **`addr IN ('The Rocks', 'Sydney')`**

View definition in RA:

$$\texttt{InnerCityHotels} = \pi_{(n,l)}(\sigma_{(A)}(\texttt{Bars}))$$

Evaluating Views (cont)

Rewriting of query involving a view:

```
= SELECT name from InnerCityHotels
  WHERE license = '123456'
```

$$= \pi_{(n)}(\sigma_{(L)}(\texttt{InnerCityHotels}))$$

$$= \pi_{(n)}(\sigma_{(L)}(\pi_{(n,l)}(\sigma_{(A)}(\texttt{Bars}))))$$

$$= \pi_{(n)}(\pi_{(n,l)}(\sigma_{(L)}(\sigma_{(A)}(\texttt{Bars}))))$$

$$= \pi_{(n)}(\sigma_{(L)}(\sigma_{(A)}(\texttt{Bars})))$$

$$= \pi_{(n)}(\sigma_{(L\ \&\ A)}(\texttt{Bars}))$$

$$= \pi_{(n)}(\sigma_{(A\ \&\ L)}(\texttt{Bars}))$$

```
= SELECT name FROM Bars
  WHERE addr IN ('The Rocks', 'Sydney')
        AND license = '123456'
```

Materialized Views

Naive implementation of materialized views:

- replace view table by re-evaluating query after each update

Clearly this costs space and makes updates more expensive.

However, in a situation where

- updates are infrequent compared to queries on the view
- the cost of "computing" the view is expensive

this approach provides substantial benefits.

Materialized views are used extensively in data warehouses.

Produced: 13 Sep 2020