The Relational Data Model and SQL

Lecture topics:

- basic concepts and operations of the relational model
- the SQL query language

References:

- text 3rd edition: Chapter 7; Chapter 8, sections 1–5; supplementary material – Chapter 22, sections 1-2
- text 4th edition: Chapter 5; Chapter 6, sections 1–4; Chapter 8, sections 1–6; Chapter 9, section 2; supplementary material – Chapter 23, sections 1-2
- DBMS vendor documents

Basic relational concepts

Relating to descriptions of data:

- **Attribute** (**column**): a name denoting a property or characteristic
- **Relation schema** (**table header**): a finite set of attributes and a mapping of each attribute to a domain (defined below)

...continued

Relating to data:

Domain: an “abstract data type” (i.e. a name, a set of values and a number of functions defined over the values)

- **Null value**: a special exception value (meaning “not known”, “not applicable”)
- **Tuple**: a set of attribute/value pairs, with each attribute occurring at most once
- **Relation (table)**: a relation schema, and a finite set of tuples
- **Relational database**: a finite set of relation names and a mapping of each relation name to a relation

...continued

Other:

- **Intention of a relation**: the associated relation schema
- **Extension of a relation**: the associated set of tuples

The relational model assumes no ordering of either rows or columns for any table.

Basic rules

- **Domain constraints**: the value associated with each attribute in a tuple must occur in the set of values associated with the domain of the attribute; or the value is *Null*
- **First normal form**: domain values cannot be tuples or relations
- **Completeness**: each tuple in a relation has an attribute/value pair for precisely the set of attributes in the associated relation schema
- **Closed world**: the database “knows of” all tuples in all relations
- **Unique rows**: no two distinct tuples in any given relation consist of the same set of attribute/value pairs

Keys

- **Relation superkey**: a subset of the associated relation schema for which no pair of distinct tuples in the relation ***will ever*** agree on the corresponding values.
- **Relation candidate key**: a minimal superkey
- **Relation primary key**: a distinguished candidate key of the relation
- **Foreign key**: primary key of one relation appearing as attributes of another relation
- Foreign keys enable capturing more complex entity structure

Integrity of primary and foreign keys

- **Entity integrity**: No component of a primary key value may be the null value, nor may be updated.

- **Referential integrity**: A tuple with a non-null value for a foreign key that does not match the primary key value of a tuple in the referenced relation is not allowed.

Relational algebra

- Proposed by E.F. Codd (1972) as basic means of manipulating data in a relational database
- A procedural query language, with fundamental operations:
  - reference
  - selection
  - projection
  - cross product
  - set union
  - set difference
  - renaming

Algebra: Set of operators mapping existing relations to new relations

Reference

Reference: referring to an existing relation

Notation: *Id*

Value: the relation named by *Id*

E.g. "The Vendors"

```
Vendor
```

Result: (a duplication of the Vendor table)

Vno	Vname	City	Vbal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05
3	Esso	Montreal	0.00
4	Esso	Waterloo	2.25

Selection

Selection: choosing some rows from a relation

Notation: $\sigma_{C}(R)$

where *R* is a relation and *C* is a condition on individual rows of *R*

Value: those rows of *R* for which condition *C* is true

E.g. “The vendors in Waterloo.”

$$\sigma_{\texttt{City = 'Waterloo'}}(\texttt{Vendor})$$

Result:

Vno	Vname	City	Vbal
4	Esso	Waterloo	2.25

...continued

Selection condition may:

- test for equality (=) between attributes and values, and
- invoke function calls on underlying domains

To ease writing queries, may:

- build more complicated conditions using logical connectives AND, OR and NOT

E.g. "Vendors that are in Toronto or have a balance exceeding 100."

$$\sigma_{(\texttt{City = 'Toronto' OR Vbal > 100})}(\texttt{Vendor})$$

Result:

Vno	Vname	City	Vbal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05

Projection

Projection: drop attributes from the result

Notation: $\pi_{\texttt{A1,...,An}}(\texttt{R})$

where A_i are attributes in the relational schema of R

Value: R restricted to attributes in A_i

E.g. “Vendor names.”

$$\pi_{\texttt{Vname}}(\texttt{Vendor})$$

Result:

Vname
Sears
Walmart
Esso

Because relations are sets, any resulting duplicate rows are removed

Cross product

Cross-product: pairing all possible combinations of tuples from two relations

Notation: $R_1 \times R_2$
where R_1 and R_2 are relations with disjoint relational schema

Value: every tuple in R_1 unioned ("matched") with every tuple in R_2

May generate a very large relation:

- Number of tuples in result = (number of tuples in R_1) X (number of tuples in R_2)
- Number of values in result tuple = (number of values in an R_1 tuple) + (number of values in an R_2 tuple)

…continued

E.g.

R_1

A	B
1	x
2	y

R_2

C	D
a	s
b	t
c	u

$R_1 \times R_2$

A	B	C	D
1	x	a	s
1	x	b	t
1	x	c	u
2	y	a	s
2	y	b	t
2	y	c	u

Set union

Union: merging two relations

Notation: $R_1 \cup R_2$

where R_1 and R_2 are relations with equivalent relational schema

Value: all tuples in R_1 or in R_2 (or in both)

E.g. "Vendors that are in Toronto or have a balance exceeding 100."

$\sigma_{\texttt{City = 'Toronto'}}$ (Vendor) $\cup$
$\sigma_{\texttt{Vbal > 100}}$ (Vendor)

Result:

Vno	Vname	City	Vbal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05

Set difference

Difference: excluding tuples of one relation

Notation: $R_1 - R_2$

where R_1 and R_2 are relations with equivalent relational schema

Value: Tuples in R_1 that are not in R_2.

E.g. "Vendor numbers for vendors with no transactions."

π_{Vno}(Vendor) – π_{Vno}(Transactions)

Result:

Vno
1

...continued

Another e.g. "Vendor names for vendors with no transactions."

$$\pi_{\texttt{Vname}}(\texttt{Vendor}) - \pi_{\texttt{Vname}}(\texttt{???})$$

Result:

Vname
Sears

Discussion:

- Why is the previous example easy, but the above more complex?
- Vno attribute is common to both relations, but Vname is not
- Use a cross-product

Cross-product and select

- Use a cross-product to form a table with the desired attributes added to the rows
- How many rows in the C.P.? Too many!
- Use a selection to choose the right rows
- Attribute name problem: need to rename one of the VNo attributes temporarily
- **E.g.** "Vendor names for vendors with no transactions."

$$\pi_{\texttt{Vname}}(\texttt{Vendor}) -$$
$$\pi_{\texttt{Vname}}(\sigma_{\texttt{Vno = V.Vno}}(\rho_{\texttt{Vno as V.Vno}}(\texttt{Vendor}) \times \texttt{Transaction}))$$

Result:

Vname
Sears

Step-by-step

$\pi_{Vname}(Vendor)$

Vname
Sears
Kmart
Esso

$\rho_{Vno\ as\ V.Vno}(Vendor)$

V.Vno	Vname	City	VBal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05
3	Esso	Montreal	0.00
4	Esso	Waterloo	2.25

Transaction

Tno	Vno	AccNum	Tdate	Amount
1001	2	101	20060115	13.25
1002	2	103	20060116	19.00
1003	3	101	20060115	25.00
1004	4	102	20060120	16.13
1005	4	103	20060125	33.12

…continued

$$\rho_{\texttt{Vno as V.Vno}}(\texttt{Vendor}) \times \texttt{Transaction}$$

1	Sears	Toronto	200	1001	2	101	20060115	13.25
1	Sears	Toronto	200	1002	2	103	20060116	19
1	Sears	Toronto	200	1003	3	101	20060115	25
1	Sears	Toronto	200	1004	4	102	20060120	16.13
1	Sears	Toronto	200	1005	4	103	20060125	33.12
2	Walmart	Ottawa	671.05	1001	2	101	20060115	13.25
2	Walmart	Ottawa	671.05	1002	2	103	20060116	19
2	Walmart	Ottawa	671.05	1003	3	101	20060115	25
2	Walmart	Ottawa	671.05	1004	4	102	20060120	16.13
2	Walmart	Ottawa	671.05	1005	4	103	20060125	33.12
3	Esso	Montreal	0	1001	2	101	20060115	13.25
3	Esso	Montreal	0	1002	2	103	20060116	19
3	Esso	Montreal	0	1003	3	101	20060115	25
3	Esso	Montreal	0	1004	4	102	20060120	16.13
3	Esso	Montreal	0	1005	4	103	20060125	33.12
4	Esso	Waterloo	2.25	1001	2	101	20060115	13.25
4	Esso	Waterloo	2.25	1002	2	103	20060116	19
4	Esso	Waterloo	2.25	1003	3	101	20060115	25
4	Esso	Waterloo	2.25	1004	4	102	20060120	16.13
4	Esso	Waterloo	2.25	1005	4	103	20060125	33.12

…continued

$\sigma_{\texttt{Vno = V.Vno}}(...etc...)$

V.Vno	Vname	City	Bal	Tno	Vno	AccNum	Tdate	Amount
2	Walmart	Ottawa	671.05	1001	2	101	20060115	13.25
2	Walmart	Ottawa	671.05	1002	2	103	20060116	19
3	Esso	Montreal	0	1003	3	101	20060115	25
4	Esso	Waterloo	2.25	1004	4	102	20060120	16.13
4	Esso	Waterloo	2.25	1005	4	103	20060125	33.12

$\pi_{\texttt{Vname}}(...etc...)$

Vname
Walmart
Esso

$\pi_{\texttt{Vname}}(\texttt{Vendor}) - (...etc...)$

Vname
Sears

- a cross-product followed by a select is called a **join** (also equijoin, nautral join)

Attribute renaming

- Renaming: temporarily changing names of attributes

- **Notation**: $\rho_{A1 \text{ as } B1, \ldots, An \text{ as } Bn}$ `(R)` where *R* is a relation and A_n are attributes of *R*

- **Value**: same as *R*, with attribute A_i replaced by attribute B_i (attribute name A_i replace by B_i)

Additional operators

Do not increase expressive power, but make life easier:

- set intersection: $\cap$
- join (cross-product & select): $\otimes$
- division: $\div$
- assignment: $\leftarrow$

Assignment

Notation: `NewR` $\leftarrow$ `R`

Value: creates a relation named *NewR* identical to *R*

E.g. "Vendor names for vendors with no transactions."

$$T1 \leftarrow \rho_{\texttt{Vno as V.Vno}}(\texttt{Vendor})$$
$$T2 \leftarrow T1 \times \texttt{Transaction}$$
$$T3 \leftarrow \pi_{\texttt{Vname}}(\sigma_{\texttt{Vno=V.Vno}}(T2))$$
$$T4 \leftarrow \pi_{\texttt{Vname}}(\texttt{Vendor}) - T3$$

The SQL query language

- Expressing the algebraic operators
- More examples of querying in SQL
- Expressiveness and limitations

Retrieving all information from a table

E.g. “The vendors.”

```
select * from Vendor
```

Selecting data

E.g. “The vendors in Waterloo.”

```
select * from Vendor
  where City = 'Waterloo'
```

E.g. “Vendors that are in Waterloo or have a balance exceeding 100.”

```
select * from Vendor
  where City = 'Waterloo'
  or    Vbal > 100
```

Projecting columns

E.g. "The names of vendors."

```
select distinct
Vname from Vendor
```

Vname
Sears
Walmart
Esso

But, note:

```
select Vname
from Vendor
```

Vname
Sears
Walmart
Esso
Esso

In SQL, a query returns a **multiset** of tuples; that is, the same row can appear more than once.

Table aliases

- column names appearing in several tables must be made unambiguous
- **alias**: a name for referring to a table.
- terminology: table aliases, tuple variables, correlation names

E.g. “Names of customers and vendors that have a common transaction.”

Solution 1:

```
select Vname, Cname
from Customer, Transaction, Vendor
where Transaction.AccNum =
      Customer.AccNum
and Transaction.Vno = Vendor.Vno
```

…continued [table aliases]

Solution 2:

```
select Vname, Cname
from Customer as C,
     Transaction as T,
     Vendor as V
where T.AccNum = C.AccNum
and T.Vno = V.Vno
```

Alternate syntax:

```
select Vname, Cname
from Customer C, Transaction T,
     Vendor V
where T.AccNum = C.AccNum
and T.Vno = V.Vno
```

Cross products and joins

E.g. “All combinations of vendors and transactions.”

```
select * from Vendor, Transaction
```

E.g. “Names of vendors and their transaction amounts.”

```
select Vname, Amount
  from Vendor V, Transaction T
  where V.Vno = T.Vno
```

Vname	Amount
Walmart	13.25
Walmart	19
Esso	25
Esso	16.13
Esso	33.12

Set difference

- not defined explicitly in earlier standards; standard in SQL92; some products do support it (EXCEPT)
- use **exists**, **subselects** to compute set difference

E.g. "Vendor numbers for vendors with no transactions."

```
select Vno from Vendor V
where not exists
(select * from Transaction T
 where T.Vno = V.Vno)
```

Subselects

- Select statements can be nested almost anywhere:
- in a `select` list:
  - lists vnames for each transaction
    ```
    select tno, (select vname from
    vendor v where v.vno = t.vno)
    from transaction t
    ```
  - subselect returns single attribute & row
- in a `from` clause:
  - list tno & vnames for Waterloo vendors
    ```
    select tno, v.vname
    from transaction t,
     (select * from vendor
      where city='Waterloo') as v
    where t.vno=v.vno
    ```
  - subselect returns a table with alias
  - similar to views (without the view def^{n}!)

…continued [subselects]

- In a `where` clause:

```
select * from transaction t
   where exists
   (select * from vendor v
    where city='Waterloo'
    and v.vno=t.vno)
```

  - useful with `exists`, `not exists`
  - also useful with `in` operator (discussion following)
  - can be used in place of any single value (see discussion on aggregate functions following)

Outer Join

- Consider the following schema:
  - F(fid, name, dean, budget, etc);
    foreign key dean references FM(eid);
  - FM(eid, name, rank, salary, etc);

- Query: list all FMs and the name of the faculty of which he/she is the dean

- Easy to do the other way: list all faculty and the name of the dean
  - following the FK connection “towards” the primary key is easy, but the opposite direction might not be
  - might not be any corresponding value

…continued [outer join]

- use `select` list subselect:

```
select eid, name, rank,
       (select F.name from F
        where F.dean = FM.eid)
from FM
```

- if no row results from the subselect, NULL is substituted
  - produces a column consisting of the name of the faculty the FM is dean of, or NULL
  - won't work if someone is dean of more that one faculty (why?)
- SQL defines a special operator to do this:

```
select eid, FM.name,
       rank, F.name
from FM left outer join F
on F.dean = FM.eid)
```

…continued [outer join]

- variations of outer join:
  - **`left outer join`**
  - **`right outer join`**
  - **`full outer join`**
- require use of on clause to identify foreign-key relationship
- basic operation:
  - preserves all the rows in one table, and supplies nulls for the other table when it does not meet the join condition

Computing a set union

E.g. “Vendors that are in Waterloo or have a balance exceeding 100.”

```
(select * from Vendor
 where City = 'Waterloo')
union
(select * from Vendor
 where Vbal > 100)
```

More on SQL Queries

- selecting rows based upon set membership
- **in**: set membership

E.g. "Vendor names for vendor numbers 1, 2 and 3."

```
select Vname from Vendor
where Vno in (1,2,3)
```

Result:

Vname
Sears
Walmart
Esso

…continued [in predicate]

- membership testing often useful with subqueries

E.g. “Names of vendors with no transactions on January 16, 2006.”

```
select Vname from Vendor
where Vno not in
(select Vno from Transaction
 where Tdate = 20060116)
```

Result:

Vname
Sears
Esso
Esso

Recall that SQL does not remove duplicates automatically.

…continued [in predicate, select distinct]

- avoiding duplicates: **distinct**

```
select distinct Vname from Vendor
where Vno not in
(select Vno from Transaction
 where Tdate = 20060116)
```

Result:

Vname
Sears
Esso

…continued [column aliasing]

E.g. “Names of vendors and customers.”

```
(select Vname as Name
 from Vendor)
union
(select Cname as Name
 from Customer)
```

...continued [column aliasing]

- terminology: column aliasing, expression aliasing
- can be used for column titles

E.g. “Transaction amounts for Esso.”

```
select Amount
as "Transaction Amounts"
from Vendor, Transaction
where Vendor.Vname = 'Esso'
and Vendor.Vno = Transaction.Vno
```

Result:

Transaction Amounts
25.00
16.14
33.12

…continued [exists predicate]

- testing for (non-)emptiness of a subquery
- **exists** *sub-query*: true if value of *sub-query* contains at least one tuple

E.g. “Names of customers with all transactions on vendors in the same city.”

```
select Cname from Customer C
where exists
(select * from Transaction T1,
               Vendor V1
 where T1.AccNum = C.AccNum
 and T1.Vno = V1.Vno
 and not exists
 (select * from Transaction T2,
                Vendor V2
  where T2.AccNum = C.AccNum
  and T2.Vno = V2.Vno
  and V1.City <> V2.City))
```

…continued [row ordering]

- tables are sets, order of rows indeterminate
- may want/need to order (sort) results

E.g. “Names of customers living in Ontario, in alphabetical order.”

```
select Cname from Customer
where Prov = 'Ont'
order by Cname
```

…continued

E.g. “Vendor cities, names and balances in alphabetical order of vendor names and in descending order of balances.”

```
select City, Vname, Vbal
from Vendor
order by Vname, Vbal desc
```

Result:

City	Vname	Vbal
Waterloo	Esso	2.25
Montreal	Esso	0.00
Ottawa	Walmart	671.05
Toronto	Sears	200

...continued [operators, string matching]

Additional operators for predicates:

- **`like`** *`pattern`*: string pattern matching

  % matches any string (including zero-length)

  _ (underscore) matches any single character

- *`Attr`* **`between`** *`Value1`* **`and`** *`Value2`*

  ≡ ((`Attr >= Value1)` **`and`**

  `(Attr <= Value2))`

…continued [operators, between]

E.g. “Employees whose name consists of ‘Wong’ preceded by five characters, and who live on Elm street.”

Employee:

Name	Street
A. Wong	123 Elm street
B.C. Wong	1 Elm street
E.F. Wong	456 Elm street
G.H.I. Wong	456 Elm street

```
select Name from Employee
where Name like '_____Wong'
and Street like '%Elm street'
```

E.g. “Names of vendors whose balance is between $100 and $500.”

```
select VName from Vendor
where VBal between 100 and 500
```

…continued [aggregate functions]

Aggregate functions:

- **`count(*)`**
  - number of tuples
- **`count(`*`column`*`)`**

  `count(distinct `*`column`*`)`
  - number of (nonduplicate) values
- **`sum(`*`expr`*`)`**

  `sum(distinct `*`expr`*`)`
  - sum of values
- **`avg(expr)`**

  `avg(distinct `*`expr`*`)`
  - average of values
- **`max(`*`expr`*`)`**
  - largest value
- **`min(`*`expr`*`)`**
  - smallest value

…continued [aggregate functions]

E.g. "Number of transactions."

```
select count(*) from transaction
```

E.g. "Number of vendors with transactions."

```
select count(distinct Vno) from
transaction
```

E.g. "Total vendor balances."

```
select sum(Vbal) from Vendor
```

E.g. "Average customer balance."

```
select avg(Cbal) from Customer
```

E.g. "Transactions of less than average amt"

```
select * from transaction
where amount < (select
avg(amount) from Transaction)
```

…continued [row grouping]

- grouping rows together, according to a common value
- Syntax:
  select *list* **group by** *columns*
- **list** contains only attributes used for grouping, or aggregate functions applied to the groups

E.g. “The total amount of transactions for each account.”

```
select AccNum, sum(Amount)
from Transaction
group by AccNum
```

Result:

AccNum	SUM(Amount)
101	38.25
102	16.13
103	52.12

…continued

- grouped **`select`** can be ordered, subject to the same restrictions on the select *`list`*

E.g. “The total amount of transactions for each account, in increasing order of amount.”

```
select AccNum, sum(Amount)
from Transaction
group by AccNum
order by sum(Amount)
```

Result:

AccNum	SUM(Amount)
102	16.13
101	38.25
103	52.12

…continued

- groups can be qualified using **`having`**

E.g. "The total amount of transactions for accounts that have more than one transaction."

```
select AccNum, sum(Amount)
from Transaction
group by AccNum
having count(*) > 1
```

Result:

AccNum	SUM(Amount)
101	38.25
103	52.12

Select statement syntax

- For all selects:

select [**<u>all</u>** | **distinct**] *exp* {, *exp*}
  from *table* [[as] *alias*]
    {, *table* [[**as**] *alias*] }
  [**where** *cond*]
  [**group by** *col* {, *col*}
  [**having** *cond*]]
  [**union** [**all**] *select*]

- For top-level queries:

select
  [**order by** *resultcol* [**<u>asc</u>** | **desc**]
    {, *resultcol* [**<u>asc</u>** | **desc**]}]

Semantics of an SQL query

- compute cross product of all tables in **`from`** clause
- eliminate rows not satisfying **`where`** condition
- group rows according to **`group by`** clause
- eliminate groups not satisfying **`having`** condition
- evaluate expressions in **`select`** target list
- eliminate duplicate rows if **`distinct`** specified
- compute **`union`** of each **`select`**
- sort rows according to **`order by`**

The power of the SQL query language

- can express anything in the relational algebra, and more:
  - result of a query can have duplicate tuples
  - result of a query can be ordered
  - can count
  - aggregate functions & grouping
- there are limitations:
  - other aggregate functions?
  - no aggregate functions on subqueries
  - no recursion or iteration
  - generalized constraints
  - not programmable like ordinary programming languages

More views

- Definition: a view is a derived table whose definition, not the table itself, is stored
  - the set of views and tables comprises the external schema
- Creating a view:
  `CREATE VIEW` *viewname*
  [(*column-name*) *[, column-name]*) *]*
  AS *select-statement* ;
- Example:

```
CREATE VIEW VTotals(vno,amt)
AS SELECT Vno, SUM(Amount)
FROM Transaction
GROUP BY Vno
```

- Removing views:
  `DROP VIEW` *viewname*
- Example:

```
DROP VIEW Vtotals
```

...continued

- A view is a *virtual table* that is computed dynamically (not stored explicitly)
- Any derivable table can be defined as a view (some minor restrictions on the SELECT)
- A table defined as a view can be used in the same way as a base table:
  - retrieval (SELECT)
  - view definition (view of view)
- **But**: updates can be performed only on certain views
  - views derived from a single base table
  - views with each row and attribute corresponding to a distinct, unique row and attribute in the base table

Pros & cons of views

- Views provide several advantages:
  - users are independent of DB growth
  - users are independent of DB restructuring (except for updating)
  - users’ perception can be simplified
  - the same data (base table) can be viewed in different ways by different users
  - security for hidden data
- Problem with views:
  - creating & view requires special permission (DBA or “resource”)
  - can use nested selects instead of view-name, i.e. use the select statement that defines the view
    - can be arbitrarily complex, including aggregates, `having`, `union`, etc

The “view update” problem

- Consider the previous view example:

```
CREATE VIEW VTotals(vno,amt)
AS SELECT Vno, SUM(Amount)
FROM Transaction
GROUP BY Vno
```

- An update to this view cannot be translated to a base-table operation
- Example:

```
UPDATE VTotals SET amt=amt+1
```

  - what rows in Transaction should be modified??
- There is no simple answer:
  - non-deterministic
  - still a research problem:
    - DBMS can try to guess
    - force the user/DBA to decide

Nulls in SQL

- Unknown: not yet known, but will be known eventually
- Not applicable: does not apply to a particular tuple
- Not the same as 0 or ‘’ (null string)
- “Not applicable” often used to simplify DB design
- Null values complicate expression evaluation. E.g.:

```
select average(vacation) from emps
select count(*) from emps
select name from emps
  where vacation <= 10
select name from emps
  where vacation > 10
```

- Solution: three-valued logic

Three-valued logic

- A **`where`** predicate returns **`unknown`** for any tuple that contains **`null`**
- `Null` also results from empty (sub)selects:

```
select name from emps
        where exists(select...)
```

- Relational operations =, <>, <, <=, >, >= yield `unknown` if either operand is `null`
- Cannot use =, <> to test `null`, use:

```
expr is null
expr is not null
```

- Test for `unknown` with:

```
expr is unknown
```

- Three-valued logic tables:

and	**T**	**F**	**U**
T	T	F	U
F	F	F	F
U	U	F	U

or	**T**	**F**	**U**
T	T	T	T
F	T	F	U
U	T	U	U

not	
T	F
F	T
U	U

Review of SQL statements

- DDL: {**create|drop} {table|view}**, **grant, revoke**
- DML: **insert**, **delete**, **update**, **select**
- more later (e.g. transaction processing)

Examples:

```
create table EssoVendors
(Vno      INTEGER not null,
 City     VARCHAR(10),
 Vbal     DECIMAL(10,2),
 primary key (Vno) );

insert into EssoVendors
 select Vno, City, VBal
 from Vendor
 where Vname like '%Esso%'
```

…continued

```
insert into EssoVendors
  values (5, 'Kitchener', 123.45)
insert into EssoVendors
  (Vbal, Vno, City)
  values(666.66, 6, 'Route 66')

update EssoVendors
  set Vbal = Vbal * 1.01
update EssoVendors
  set Vbal = Vbal * 1.02
  where Vbal < 50.00

delete from Transaction
  where Vno in
  (select Vno from EssoVendors)
```

The “last word” on SQL – for now

- Many, many details omitted
  - table-spaces, named schemas
  - table ownership
  - stored procedures & triggers
  - constraints (unique, check, …)
  - and others
- Most commercial products implement their own version of SQL
  - typically a cross between SQL89 and SQL92
  - lots of extra features
  - “your mileage may vary”
- The SQL vendor documents are essential to any realistic SQL project

Supplementary material: Security in SQL

- The GRANT and REVOKE statements are used to:
  - maintain users and user groups for a database
  - maintain DDL privileges for users and user groups
  - maintain DML privileges for users and user groups

E.g. “Create a new user called Grove, with password abc.”

```
GRANT CONNECT TO Grove IDENTIFIED BY "abc";
```

E.g. “Add a benefits group to the database with access to the employee table.”

```
GRANT CONNECT TO benefits;
GRANT GROUP TO benefits;
GRANT ALL PRIVILEGES ON Employee TO benefits;
```

...continued

E.g. “Make Grove a member of the benefits group.”

```
GRANT MEMBERSHIP IN GROUP benefits TO Grove;
```

E.g. “Create a new user called George, password xyz, with the authority to execute SQL DDL statements.”

```
GRANT CONNECT TO George IDENTIFIED BY xyz;
GRANT RESOURCE TO George;
```

E.g. “Make Mary the database administrator, with password ‘change quickly’.”

```
GRANT CONNECT TO Mary
  IDENTIFIED BY "change quickly";
GRANT DBA TO Mary;
```

...continued

E.g. “Have Mary change her password and revoke Grove’s membership in benefits, but still allow him to query the employee table.”

```
CONNECT Mary IDENTIFIED BY "change quickly";
GRANT CONNECT TO Mary IDENTIFIED BY "xvqmt";
REVOKE MEMBERSHIP IN benefits FROM Grove;
GRANT SELECT ON Employee TO Grove;
```

Privileges on databases

CONNECT	may create new users
DBA	may do anything (super-user)
RESOURCE	may create tables and views (DDL functions)
GROUP	may have members (i.e. the user is to be a group)

MEMBERSHIP IN GROUP userid [, userid...]
places users in a group (user inherits group's permissions)

Privileges on tables and views

ALTER may use ALTER TABLE to modify table schema

DELETE may delete existing tuples from named table or view

INSERT may insert new tuples in named table or view

REFERENCES may create a foreign key constraint to named table

SELECT may query existing tuples in named table or view

UPDATE [*column-name-list*] may update indicated columns of existing tuples in named table or view

ALL [**PRIVILEGES**] all of the above