CS315: Principles of Database Systems
Structured Query Language (SQL)

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs315/

2nd semester, 2013-14

Tue, Fri 1530-1700 at CS101

Structured Query Language (SQL)

- A language to specify queries for a relational database
- Is a data manipulation language (DML)
  - Can access and manipulate data stored as a particular data model
- Declarative language
  - Specifies what to do, but not how to do
- Is a data definition language (DDL)
  - Defines database relations and schemas

Structured Query Language (SQL)

- A language to specify queries for a relational database
- Is a data manipulation language (DML)
  - Can access and manipulate data stored as a particular data model
- Declarative language
  - Specifies what to do, but not how to do
- Is a data definition language (DDL)
  - Defines database relations and schemas
- SQL has evolved widely after its first inception
  - Supports lots of extra operations, which are non-standard

Running example

- branch(bname, bcity, assets)
- customer(cname, cstreet, ccity)
- account(ano, bname, bal)
- loan(lno, bname, amt)
- depositor(cname, ano)
- borrower(cname, lno)

Creating a relation schema

- create table: create table $r(A_1\ D_1\ C_1, \ldots, A_n\ D_n\ C_n, (IC_1), \ldots, (IC_k))$
  - r is the name of the relation
  - Each A_i is an attribute name whose data type or domain is specified by D_i
  - C_i specifies constraints or settings (if any)
  - IC_j represents integrity constraints (if any)
- Example

```
create table branch (
   bname varchar(20) primary key,
   bcity varchar(20) not null,
   assets integer
)
```

Data types in SQL

- *char(n)*: fixed-length character string
- *varchar(n)*: variable-length character string, up to *n*
- *integer* or *int*: integer
- *smallint*: short integer
- *numeric(n,d)*: floating-point number with a total of *n* digits of which *d* is after the decimal point
- *real*: single-precision floating-point number
- *double precision*: double-precision floating-point number
- *float(n)*: floating-point number with at least *n* digits

Data types in SQL

- *char(n)*: fixed-length character string
- *varchar(n)*: variable-length character string, up to *n*
- *integer* or *int*: integer
- *smallint*: short integer
- *numeric(n,d)*: floating-point number with a total of *n* digits of which *d* is after the decimal point
- *real*: single-precision floating-point number
- *double precision*: double-precision floating-point number
- *float(n)*: floating-point number with at least *n* digits
- *date*: yyyy-mm-dd format
- *time*: hh:mm:ss format
- *time(i)*: hh:mm:ss:i. . . i format with additional *i* digits for fraction of a second
- *timestamp*: both date and time
- *interval*: relative value in either year-month or day-time format

Other data types

- User-defined data type

```
create type cgpa as numeric(3,1)
```

- Large objects such as images, videos, strings can be stored as
  - blob: binary large object
  - clob: character large object
  - A pointer to the object is stored in the relation, and not the object itself

Other data types

- User-defined data type

```
create type cgpa as numeric(3,1)
```

- Large objects such as images, videos, strings can be stored as
  - blob: binary large object
  - clob: character large object
  - A pointer to the object is stored in the relation, and not the object itself
- User-defined domain

```
create domain name as varchar(20) not null
```

Constraints

- Can be specified for each attribute as well as separately
  - *not null*: the attribute cannot be null
    - Requires some value while inserting as otherwise null is the default
  - *primary key* $(A_i, \ldots, A_j)$: automatically ensures not null
  - *default n*: defaults to *n* if no value is specified
  - *unique*: specifies that this is a candidate key
  - *foreign key*: specifies as a foreign key and the relation it references to
  - *check P*: predicate *P* must be satisfied

```
create table branch (
  bname varchar(20),
  bcity varchar(20) not null,
  assets integer default 0,
  primary key (bname)
  check (assets >= 0))
create table borrower (
  cname varchar(20),
  lno integer,
  foreign key (cname) references customer,
  foreign key (lno) references loan)
```

Deleting or modifying a relation schema

- drop table: drop table *r* deletes the table from the database
  - Must satisfy other constraints already applied
- Example

```
drop table borrower
```

Deleting or modifying a relation schema

- drop table: drop table *r* deletes the table from the database
  - Must satisfy other constraints already applied
- Example

```
drop table borrower
```

- alter table: alter table *r* add *A D C*
  - Adds attribute *A* with data type *D* at the end
  - *C* specifies constraints on *A* (if any)
  - Must satisfy other constraints already applied
- alter table: alter table *r* drop *A*
  - Deletes attribute *A* from all tuples
  - Must satisfy other constraints already applied
- Example

```
alter table branch add bcode integer not null

alter table branch drop assets
```

Basic query structure

- SQL is based on relational algebra, but has certain important modifications
- A typical SQL query is of the form

  select $A_1, \ldots, A_n$
  from $r_1, \ldots, r_m$
  where P
- Each r_i is a relation
- Each A_j is an attribute from one of $r_1, \ldots, r_m$
- P is a predicate involving attributes and constants
- where can be left out, which then means *true*
- Result is a relation with the schema $(A_1, \ldots, A_n)$

Basic query structure

- SQL is based on relational algebra, but has certain important modifications
- A typical SQL query is of the form

  select $A_1, \ldots, A_n$
  from $r_1, \ldots, r_m$
  where P
- Each r_i is a relation
- Each A_j is an attribute from one of $r_1, \ldots, r_m$
- P is a predicate involving attributes and constants
- where can be left out, which then means *true*
- Result is a relation with the schema $(A_1, \ldots, A_n)$
- Is equivalent to the relational algebra query

  $\Pi_{A_1,\ldots,A_n}(\sigma_P(r_1 \times \cdots \times r_m))$

Distinction with relational algebra

- SQL relations are *multi-sets* or *bags* of tuples and not sets
- Multi-sets
  - Example: {A, A, B}
  - It is distinct from {A, B} but equivalent to {A, B, A}
- Consequently, there may be two identical tuples

Distinction with relational algebra

- SQL relations are *multi-sets* or *bags* of tuples and not sets
- Multi-sets
  - Example: {A, A, B}
  - It is distinct from {A, B} but equivalent to {A, B, A}
- Consequently, there may be two identical tuples
- The set behavior can be enforced by the keyword unique
- In a query, keyword distinct achieves the same effect
- Opposite is keyword all, which is default

Select

- Lists attributes in the final output
- Example

```
select bname
from branch
where bcity = ''Kanpur''
```

- Case-insensitive
- select * chooses all attributes
- To eliminate duplicates, use select distinct . . .
- Otherwise, by default is select all . . .

Select

- Lists attributes in the final output
- Example

```
select bname
from branch
where bcity = ''Kanpur''
```

- Case-insensitive
- select * chooses all attributes
- To eliminate duplicates, use select distinct . . .
- Otherwise, by default is select all . . .
- Can contain arithmetic expressions

```
select bname, assets * 100
from branch
where bcity = ''Kanpur''
```

From

- Lists relations from where attributes will be listed
- Corresponds to Cartesian product of the relations
- Example

```
select *
from depositor, account
```

From

- Lists relations from where attributes will be listed
- Corresponds to Cartesian product of the relations
- Example

```
select *
from depositor, account
```

- When two relations contain attributes of the same name, qualification is needed to remove ambiguity

```
select cname, depositor.ano, bal
from depositor, account
where depositor.ano = account.ano
```

Where

- Specifies conditions that the result tuples must satisfy
- Example

```
select bname
from branch
where bcity = ''Kanpur''
```

- May use *and*, *or* and *not* to connect predicates

```
select *
from account
where bname = ''IIT'' and bal >= 10000
```

- Unused clause is equivalent to where true

Where

- Specifies conditions that the result tuples must satisfy
- Example

```
select bname
from branch
where bcity = ''Kanpur''
```

- May use *and*, *or* and *not* to connect predicates

```
select *
from account
where bname = ''IIT'' and bal >= 10000
```

- Unused clause is equivalent to where true
- SQL allows between operator

```
select *
from account
where bal between 1000 and 9999
```

Rename operation

- SQL allows renaming of relations and attributes to remove ambiguity
- Keyword as is used
- Example

```
select cname, depositor.ano as aid, bal
from depositor, account
where depositor.ano = account.ano
```

Rename operation

- SQL allows renaming of relations and attributes to remove ambiguity
- Keyword as is used
- Example

```
select cname, depositor.ano as aid, bal
from depositor, account
where depositor.ano = account.ano
```

- Renaming is necessary when the same relation needs to be used twice
- Example: Find names of all branches that have greater assets than the “IIT” branch

Rename operation

- SQL allows renaming of relations and attributes to remove ambiguity
- Keyword as is used
- Example

```
select cname, depositor.ano as aid, bal
from depositor, account
where depositor.ano = account.ano
```

- Renaming is necessary when the same relation needs to be used twice
- Example: Find names of all branches that have greater assets than the “IIT” branch

```
select T.bname
from branch as T, branch as S
where T.assets > S.assets and S.bname = ''IIT''
```

- as can be omitted by simply stating branch T

String operations

- Supports string matching other than equality of two strings
- Uses like to match patterns specified using special characters
  - _: matches any character
  - %: matches any substring
- Example: Find all branches having “IIT” in its name

```
select *
from branch
where bname like ''%IIT%''
```

String operations

- Supports string matching other than equality of two strings
- Uses like to match patterns specified using special characters
  - _: matches any character
  - %: matches any substring
- Example: Find all branches having “IIT” in its name

```
select *
from branch
where bname like ''%IIT%''
```

- Example: Find assets at the branch “IIT_Kanpur”
  - ‘\’ protects the next character

```
select assets
from branch
where bname = ''IIT\_Kanpur''
```

Ordering of tuples

- Tuples in the final relation can be ordered using order by

Ordering of tuples

- Tuples in the final relation can be ordered using order by
  - For display purposes only – has no actual effect
- Example: List all customers in alphabetic order of their names

```
select *
from customer
order by cname
```

- Use desc to obtain tuples in descending order
- Default is ascending order (asc)

```
select *
from customer
order by cname desc
```

Set operations

- Operators union, intersect and except correspond to $\cup$, $\cap$, –
- *Eliminates* duplicates
- For multiset operations, i.e., to retain duplicates, use all after the operations
- Example: Find customers having a loan or an account

```
(select cname from depositor)
union
(select cname from borrower)
```

- Example: Find customers having a loan but not an account

```
(select cname from depositor)
except
(select cname from borrower)
```

Aggregate functions

- Five operations that work on multisets: avg, min, max, sum, count
- Example: Find the average account balance at “IIT” branch

```
select avg(bal)
from account
where bname = ''IIT''
```

- For set operations, use distinct
- Example: Find the total number of depositors

```
select count(distinct cname)
from depositor
```

Grouping

- To apply aggregate operations on separate groups, use group by
- The aggregate operator is applied on *each* group separately
- Example: Find the number of depositors in each branch

```
select bname, count(distinct cname)
from depositor, account
where depositor.ano = account.ano
group by bname
```

- Attributes in select clause outside of aggregate functions *must* appear in group by list

Qualifying groups

- In order to select certain groups, use having clause
- Only those groups satisfying having clause appears in the result
- Example: Find names of all branches where the average account balance is more than 9999 and the corresponding average balance

```
select bname, avg(bal)
from account
group by bname
having avg(bal) > 9999
```

- The predicate in having is applied *after* forming groups whereas the predicate in where is applied *before* doing so

Qualifying groups

- In order to select certain groups, use having clause
- Only those groups satisfying having clause appears in the result
- Example: Find names of all branches where the average account balance is more than 9999 and the corresponding average balance

```
select bname, avg(bal)
from account
group by bname
having avg(bal) > 9999
```

- The predicate in having is applied *after* forming groups whereas the predicate in where is applied *before* doing so
- Example: Find names of all branches in “Kanpur” where the average account balance is more than 9999 and the corresponding balance

```
select account.bname, avg(bal)
from account, branch
where account.bname = branch.bname and bcity = ''Kanpur''
group by account.bname
having avg(bal) > 9999
```

Null

- *null* signifies missing or unknown value
- The predicates is null and is not null can be used to check for null values
- Example

```
select lno
from loan
where amt is not null
```

Null

- *null* signifies missing or unknown value
- The predicates is null and is not null can be used to check for null values
- Example

```
select lno
from loan
where amt is not null
```

- Result of expressions involving null evaluate to null
- Comparison with null returns *unknown*
- Uses same three-valued logic as relational algebra
- Aggregate functions ignore null
  - count(*) does not ignore nulls

Nested subqueries

- A query which occurs in where or from clause of another query is called a subquery
- Entire query is called outer query while the subquery is called inner query or nested query
- Used in tests for set membership, set cardinality, set comparisons

Set membership

- Keyword in is used for set membership tests
- Example: Find names of customers having both a loan and an account

```
select cname
from borrower
where cname in (
   select cname
   from depositor )
```

- Example: Find names of customers having a loan but not an account

```
select cname
from borrower
where cname not in (
   select cname
   from depositor )
```

Scoping of attributes

- It is a good practice to qualify attributes
- Also, it is better to rename relations if it is used in both the outer and the inner queries
- An unqualified attribute refers to the *innermost* query
- When a nested query refers to an attribute in the outer query, they are called correlated queries
- Example: Find names of all customers having the same account number and the loan number

```
select cname
from depositor as D, customer
where D.cname = customer.cname and cname in (
   select cname
   from borrower as B, customer
   where B.cname = customer.cname and B.lno = D.ano )
```

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r\ (F \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < some \{0, 5, 6\} =$

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r \ (F \langle \text{comp} \rangle \ t))$
- Examples:
  - $5 < some \{0, 5, 6\} = \text{true}$
  - $5 < some \{0, 5\} =$

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r \ (F \langle \text{comp} \rangle \ t))$
- Examples:
  - $5 < some\{0, 5, 6\} = \text{true}$
  - $5 < some\{0, 5\} = \text{false}$
  - $5 = some\{0, 5\} =$

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r \ (F \langle \text{comp} \rangle \ t))$
- Examples:
  - $5 < some\{0, 5, 6\} = \text{true}$
  - $5 < some\{0, 5\} = \text{false}$
  - $5 = some\{0, 5\} = \text{true}$
  - $5 \neq some\{0, 5\} =$

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r \ (F \langle \text{comp} \rangle \ t))$
- Examples:
  - $5 < some\{0, 5, 6\} = \text{true}$
  - $5 < some\{0, 5\} = \text{false}$
  - $5 = some\{0, 5\} = \text{true}$
  - $5 \neq some\{0, 5\} = \text{true}$

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r \ (F \langle \text{comp} \rangle \ t))$
- Examples:
  - $5 < \textit{some}\{0, 5, 6\} = \text{true}$
  - $5 < \textit{some}\{0, 5\} = \text{false}$
  - $5 = \textit{some}\{0, 5\} = \text{true}$
  - $5 \neq \textit{some}\{0, 5\} = \text{true}$
- $(= \text{ some}) \equiv (\text{in})$
- $(\neq \text{ some}) \not\equiv (\text{not in})$
- Example: Find names of branches that have assets greater than some branch in “Kanpur”

Set comparison: some

- $(F \langle \text{comp} \rangle \text{ some } r) \Leftrightarrow (\exists t \in r \ (F \langle \text{comp} \rangle \ t))$
- Examples:
  - $5 < \textit{some}\{0, 5, 6\} = \text{true}$
  - $5 < \textit{some}\{0, 5\} = \text{false}$
  - $5 = \textit{some}\{0, 5\} = \text{true}$
  - $5 \neq \textit{some}\{0, 5\} = \text{true}$
- $(= \text{some}) \equiv (\text{in})$
- $(\neq \text{some}) \not\equiv (\text{not in})$
- Example: Find names of branches that have assets greater than some branch in "Kanpur"

```
select bname
from branch
where assets > some (
   select assets
   from branch
   where bcity = ''Kanpur'' )
```

Set comparison: all

- $(F \langle \text{comp} \rangle \text{ all } r) \Leftrightarrow (\forall t \in r\ (F \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} =$

Set comparison: all

- $(F \langle \text{comp} \rangle \text{ all } r) \Leftrightarrow (\forall t \in r\ (F \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} = \text{false}$
  - $5 < all\{6, 9\} =$

Set comparison: all

- $(F \langle \text{comp} \rangle \text{ all } r) \Leftrightarrow (\forall t \in r\ (F \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} = \text{false}$
  - $5 < all\{6, 9\} = \text{true}$
  - $5 = all\{0, 5\} =$

Set comparison: all

- $(F\ \langle \text{comp} \rangle\ \text{all}\ r) \Leftrightarrow (\forall t \in r\ (F\ \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} = \text{false}$
  - $5 < all\{6, 9\} = \text{true}$
  - $5 = all\{0, 5\} = \text{false}$
  - $5 \neq all\{4, 6\} =$

Set comparison: all

- $(F\ \langle \text{comp} \rangle\ \text{all}\ r) \Leftrightarrow (\forall t \in r\ (F\ \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} = \text{false}$
  - $5 < all\{6, 9\} = \text{true}$
  - $5 = all\{0, 5\} = \text{false}$
  - $5 \neq all\{4, 6\} = \text{true}$

Set comparison: all

- $(F\ \langle \text{comp} \rangle\ \text{all}\ r) \Leftrightarrow (\forall t \in r\ (F\ \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} = \text{false}$
  - $5 < all\{6, 9\} = \text{true}$
  - $5 = all\{0, 5\} = \text{false}$
  - $5 \neq all\{4, 6\} = \text{true}$
- $(\neq \text{all}) \equiv (\text{not in})$
- $(= \text{all}) \not\equiv (\text{in})$
- Example: Find names of branches that have assets greater than all branches in “Kanpur”

Set comparison: all

- $(F \langle \text{comp} \rangle \text{ all } r) \Leftrightarrow (\forall t \in r\ (F \langle \text{comp} \rangle\ t))$
- Examples:
  - $5 < all\{0, 5, 6\} = \text{false}$
  - $5 < all\{6, 9\} = \text{true}$
  - $5 = all\{0, 5\} = \text{false}$
  - $5 \neq all\{4, 6\} = \text{true}$
- $(\neq \text{all}) \equiv (\text{not in})$
- $(= \text{all}) \not\equiv (\text{in})$
- Example: Find names of branches that have assets greater than all branches in "Kanpur"

```
select bname
from branch
where assets > all (
   select assets
   from branch
   where bcity = ''Kanpur'' )
```

Empty set

- exists tests if the relation is empty
- (exists r) $\Leftrightarrow$ ($r \neq \Phi$)
- (not exists r) $\Leftrightarrow$ ($r = \Phi$)
- Example: Find names of customers who have both an account and a loan

Empty set

- exists tests if the relation is empty
- (exists r) $\Leftrightarrow$ ($r \neq \Phi$)
- (not exists r) $\Leftrightarrow$ ($r = \Phi$)
- Example: Find names of customers who have both an account and a loan

```
select cname
from borrower as B
where exists (
   select *
   from depositor as D
   where D.cname = B.cname )
```

Duplication in sets

- unique tests if the relation contains duplicate tuples
- (unique r) $\Leftrightarrow$ ($\forall t, s \in r$ ($t \neq s$))
- (not unique r) $\Leftrightarrow$ ($\exists t, s \in r$ ($t = s$))
- Example: Find names of customers who have at most one account at the “IIT” branch

Duplication in sets

- unique tests if the relation contains duplicate tuples
- (unique r) $\Leftrightarrow$ ($\forall t, s \in r$ ($t \neq s$))
- (not unique r) $\Leftrightarrow$ ($\exists t, s \in r$ ($t = s$))
- Example: Find names of customers who have at most one account at the “IIT” branch

```
select D.cname
from depositor as D
where unique (
   select E.cname
   from account, depositor as E
   where D.cname = E.cname and E.ano = account.ano and
         account.bname = ''IIT'' )
```

- Example: Find names of customers who have at least two accounts at the “IIT” branch

Duplication in sets

- unique tests if the relation contains duplicate tuples
- (unique r) $\Leftrightarrow$ ($\forall t, s \in r$ ($t \neq s$))
- (not unique r) $\Leftrightarrow$ ($\exists t, s \in r$ ($t = s$))
- Example: Find names of customers who have at most one account at the "IIT" branch

```
select D.cname
from depositor as D
where unique (
   select E.cname
   from account, depositor as E
   where D.cname = E.cname and E.ano = account.ano and
         account.bname = ''IIT'' )
```

- Example: Find names of customers who have at least two accounts at the "IIT" branch

```
...
where not unique (
... )
```

Explicit sets

- Use set literals specified within brackets
- Example: Find names of customers having account numbers 11, 22 and 33

```
select cname
from depositor
where depositor.ano in (11, 22, 33)
```

Summary of SQL query format

- An SQL query may contain upto six clauses and may be nested, but only the first two—select and from—are mandatory
- Format (in order)
  select ⟨attribute list⟩
  from ⟨relation list⟩
  where ⟨tuple condition⟩
  group by ⟨group attribute list⟩
  having ⟨group condition⟩
  order by ⟨attribute list⟩
- Execution order
  1. from
  2. where
  3. group by
  4. having
  5. order by
  6. select

Derived relations

- In the from clause, a derived relation (result of a subquery) can be used
- Example: Find average account balance of branches where the average is greater than 999

```
select bname, avg_bal
from (
   select bname, avg(bal)
   from account
   group by bname )
   as branch_avg(bname, avg_bal)
where avg_bal > 999
```

- Avoids using having clause

With

- with clause defines a temporary relation
- This temporary relation is available *only* to the query using the with clause
- Example: Find all accounts with maximum balance

```
with max_bal(val) as
   select max(bal)
   from account
select ano
from account, max_bal
where bal = val
```

Insertion

- insert into . . . values statement

```
insert into account(ano, bname, bal)
values (12, ''IIT'', 100)
```

Insertion

- insert into . . . values statement

```
insert into account(ano, bname, bal)
values (12, ''IIT'', 100)
```

- May omit schema

```
insert into account
values (12, ''IIT'', 100)
```

Insertion

- insert into . . . values statement

```
insert into account(ano, bname, bal)
values (12, ''IIT'', 100)
```

- May omit schema

```
insert into account
values (12, ''IIT'', 100)
```

- If value is not known, specify null

```
insert into account
values (12, ''IIT'', null)
```

Insertion

- insert into . . . values statement

```
insert into account(ano, bname, bal)
values (12, ''IIT'', 100)
```

- May omit schema

```
insert into account
values (12, ''IIT'', 100)
```

- If value is not known, specify null

```
insert into account
values (12, ''IIT'', null)
```

- To avoid null, specify schema

```
insert into account(ano, bname)
values (12, ''IIT'')
```

Insertion (contd.)

- Example: Create an account with balance 20 at “IIT” branch for every loan with the same number

```
insert into account
select lno, bname, 20
from loan
where bname = ''IIT''

insert into depositor
select cname, lno
from loan, borrower
where bname = ''IIT'' and loan.ano = borrower.ano
```

- Query is evaluated fully before any tuple is inserted

Insertion (contd.)

- Example: Create an account with balance 20 at “IIT” branch for every loan with the same number

```
insert into account
select lno, bname, 20
from loan
where bname = ''IIT''

insert into depositor
select cname, lno
from loan, borrower
where bname = ''IIT'' and loan.ano = borrower.ano
```

- Query is evaluated fully before any tuple is inserted
- Otherwise, infinite insertion happens for queries like

```
insert into r
select * from r
```

Deletion

- delete from . . . where statement

```
delete from account
where ano = 12
```

- where selects tuples that will be deleted
- If where is empty,

Deletion

- delete from . . . where statement

```
delete from account
where ano = 12
```

- where selects tuples that will be deleted
- If where is empty, all tuples are deleted

Deletion

- delete from . . . where statement

```
delete from account
where ano = 12
```

- where selects tuples that will be deleted
- If where is empty, all tuples are deleted
- Delete all accounts at branches of “Kanpur”

```
delete from account
where bname in (
   select bname
   from branch
   where bcity = ‘‘Kanpur’’ )
```

Deletion (contd.)

- Example: Delete all accounts whose balance is less than the average balance

```
delete from account
where bal < (
   select avg(bal)
   from account )
```

- Average is computed before any tuple is deleted
- It is not re-computed

Deletion (contd.)

- Example: Delete all accounts whose balance is less than the average balance

```
delete from account
where bal < (
   select avg(bal)
   from account )
```

- Average is computed before any tuple is deleted
- It is not re-computed
- Otherwise, average balance keeps on changing
- Ultimately, all but the account with the largest balance will be deleted

Updating

- update . . . set . . . where statement

```
update account
set bal = bal * 1.05
where bal >= 1000
```

- where selects tuples that will be updated
- If where is empty,

Updating

- update . . . set . . . where statement

```
update account
set bal = bal * 1.05
where bal >= 1000
```

- where selects tuples that will be updated
- If where is empty, all tuples are updated with the new value

Updating

- update . . . set . . . where statement

```
update account
set bal = bal * 1.05
where bal >= 1000
```

- where selects tuples that will be updated
- If where is empty, all tuples are updated with the new value
- Example: Give 5% interest to all accounts with balance less than 1000 and 6% interest otherwise

```
update account
set bal = bal * 1.06
where bal >= 1000

update account
set bal = bal * 1.05
where bal < 1000
```

- Order is important

Updating (contd.)

- case statement handles conditional updates in a better manner
- Example: Give 5% interest to all accounts with balance less than 1000 and 6% interest otherwise

```
update account
set bal =
  case (bal)
    when bal < 1000 then bal * 1.05
    else bal * 1.06
  end
```

Join

- Join types: inner join, left (outer) join, right (outer) join, full (outer) join
- Join conditions: natural, on ⟨predicate⟩, using (⟨attribute list⟩)
- Examples

```
loan inner join borrower on loan.lno = borrower.lno
```

Join

- Join types: inner join, left (outer) join, right (outer) join, full (outer) join
- Join conditions: natural, on ⟨predicate⟩, using (⟨attribute list⟩)
- Examples

```
loan inner join borrower on loan.lno = borrower.lno
```

```
loan natural left join borrower
```

Join

- Join types: inner join, left (outer) join, right (outer) join, full (outer) join
- Join conditions: natural, on ⟨predicate⟩, using (⟨attribute list⟩)
- Examples

```
loan inner join borrower on loan.lno = borrower.lno
```

```
loan natural left join borrower
```

```
loan right outer join borrower using (lno)
```

- Find all customers who have either an account or a loan but not both

Join

- Join types: inner join, left (outer) join, right (outer) join, full (outer) join
- Join conditions: natural, on ⟨predicate⟩, using (⟨attribute list⟩)
- Examples

```
loan inner join borrower on loan.lno = borrower.lno

loan natural left join borrower

loan right outer join borrower using (lno)
```

- Find all customers who have either an account or a loan but not both

```
select cname
from (depositor natural full join borrower)
where ano is null or lno is null
```

Join

- Join types: inner join, left (outer) join, right (outer) join, full (outer) join
- Join conditions: natural, on ⟨predicate⟩, using (⟨attribute list⟩)
- Examples

```
loan inner join borrower on loan.lno = borrower.lno

loan natural left join borrower

loan right outer join borrower using (lno)
```

- Find all customers who have either an account or a loan but not both

```
select cname
from (depositor natural full join borrower)
where ano is null or lno is null
```

- Multiple joins: (*r* join *s*) join *t*, etc.

Views

- A relation that is not present physically but is made visible to the user is called a view
- A view is a *virtual* relation derived from other relations

Views

- A relation that is not present physically but is made visible to the user is called a view
- A view is a *virtual* relation derived from other relations
- It helps in query processing
  - If a sub-query is very common, obtain a view for it
- It helps in hiding certain data from a user
  - A view can leave out sensitive attributes
  - Example: Find all the loans of a customer but not the loan amount

```
create view v as
  select cname, borrower.lno, bname
  from (borrower natural inner join loan)
```

- A view can be deleted simply using drop

```
drop v
```

- A view has full query capabilities, but limited modification facilities

Storing views

- A view is *not* stored physically
- Only the query expression is stored
- Wherever a view is used, the query expression is substituted
- Example: Select loans only from “IIT” branch (but not loan amount)

```
select *
from v
where bname = ''IIT''
```

is expanded at runtime to

```
select *
from (
   select cname, borrower.lno, bname
   from (borrower natural inner join loan) )
where bname = ''IIT''
```

Storing views

- A view is *not* stored physically
- Only the query expression is stored
- Wherever a view is used, the query expression is substituted
- Example: Select loans only from “IIT” branch (but not loan amount)

```
select *
from v
where bname = ''IIT''
```

is expanded at runtime to

```
select *
from (
   select cname, borrower.lno, bname
   from (borrower natural inner join loan) )
where bname = ''IIT''
```

- This allows to capture all updates in the base relations

Storing views

- A view is *not* stored physically
- Only the query expression is stored
- Wherever a view is used, the query expression is substituted
- Example: Select loans only from “IIT” branch (but not loan amount)

```
select *
from v
where bname = ''IIT''
```

is expanded at runtime to

```
select *
from (
   select cname, borrower.lno, bname
   from (borrower natural inner join loan) )
where bname = ''IIT''
```

- This allows to capture all updates in the base relations
- If a view is materialized, it is stored physically
- To ensure consistency, database *must* update views once base relations are updated

Views using views

- A view may be defined using another view
- View v_1 *depends directly on* view v_2 if v_2 is used in the definition of v_1
- View v_1 *depends on* view v_2 if v_2 is there is a path of dependencies from v_1 to v_2
- View expansion is used in the reverse order
- A view is *recursive* if it depends on itself
  - Not allowed

Updating a view

- Updating a view causes many problems, and is, in general, not allowed
- Update must map to updates on the base relations
- If a view involves join or Cartesian product, update must map to updates on all the base relations
  - Not always possible
- Problems with insert or delete
  - Spurious tuple
  - Null
  - Non-uniqueness

Problems with updating a view

- Spurious tuple: “ShopC” tuple should not be inserted into “IIT” view

```
create view v as
   select lno, bname
   from loan
   where bname = ''IIT''

insert into v values(32, ''ShopC'')
```

Problems with updating a view

- Spurious tuple: “ShopC” tuple should not be inserted into “IIT” view

```
create view v as
   select lno, bname
   from loan
   where bname = ''IIT''

insert into v values(32, ''ShopC'')
```

- Null: Amount needs to set to null

```
insert into v values(32, ''IITK'')
```

Problems with updating a view

- Spurious tuple: “ShopC” tuple should not be inserted into “IIT” view

```
create view v as
  select lno, bname
  from loan
  where bname = ''IIT''

insert into v values(32, ''ShopC'')
```

- Null: Amount needs to set to null

```
insert into v values(32, ''IITK'')
```

- Non-uniqueness: Has to choose either borrower or depositor by deciding if this is an account number or a loan number

```
create view v as (
  select cname, ano as n
  from borrower )
  union (
  select cname, lno as n
  from depositor )
```

Updatable views

- A view is an updatable view if
  - It is a simple query
  - It involves only one relation
  - select does not use distinct, aggregates or expressions
  - Attributes not in select can be set to null
  - There is no group by or having
- Example:

```
create view v as
  select lno, bname, amt
  from loan
  where bname = ''IIT''
```

Updatable views

- A view is an updatable view if
  - It is a simple query
  - It involves only one relation
  - select does not use distinct, aggregates or expressions
  - Attributes not in select can be set to null
  - There is no group by or having
- Example:

```
create view v as
  select lno, bname, amt
  from loan
  where bname = ''IIT''
```

- To counter spurious tuples, with check option is used
  - Any tuple not satisfying where clause is rejected

```
create view v as
  select lno, bname, amt
  from loan
  where bname = ''IIT''
with check option
```

Triggers

- A trigger statement allows automatic (or *active*) management during database modifications
- It is invoked *only* by the database engine and not by the user

Triggers

- A trigger statement allows automatic (or *active*) management during database modifications
- It is invoked *only* by the database engine and not by the user
- It follows the event-condition-action (ECA) model
  - *Event*: Database modification
  - *Condition*: Only if true; if no condition, then assumed true
  - *Action*: Database action or any program

Triggers

- A trigger statement allows automatic (or *active*) management during database modifications
- It is invoked *only* by the database engine and not by the user
- It follows the event-condition-action (ECA) model
  - *Event*: Database modification
  - *Condition*: Only if true; if no condition, then assumed true
  - *Action*: Database action or any program
- It may not allow the full range of modification statements
- It can be called *before* or *after* the modification
- New and old values are referenced using new and old respectively
  - New refers to a inserted or new value of updated tuple
  - Old refers to a deleted or old value of updated tuple
- By default, it is for each row (i.e., tuple)

Example

- Created using a create trigger command
  - Update the branch name of account when the name of a branch in Kanpur is updated

```
create trigger update_bname
  after update of bname on branch
  for each row
  when bcity = ''Kanpur''
  begin
    update account set bname = new.bname where bname =
        old.bname;
  end
```

Example

- Created using a create trigger command
  - Update the branch name of account when the name of a branch in Kanpur is updated

```
create trigger update_bname
  after update of bname on branch
  for each row
  when bcity = ''Kanpur''
  begin
    update account set bname = new.bname where bname =
        old.bname;
  end
```

- A trigger can be deleted simply using drop

```
drop update_bname
```