Chapter 5
Structured Query Language (SQL)

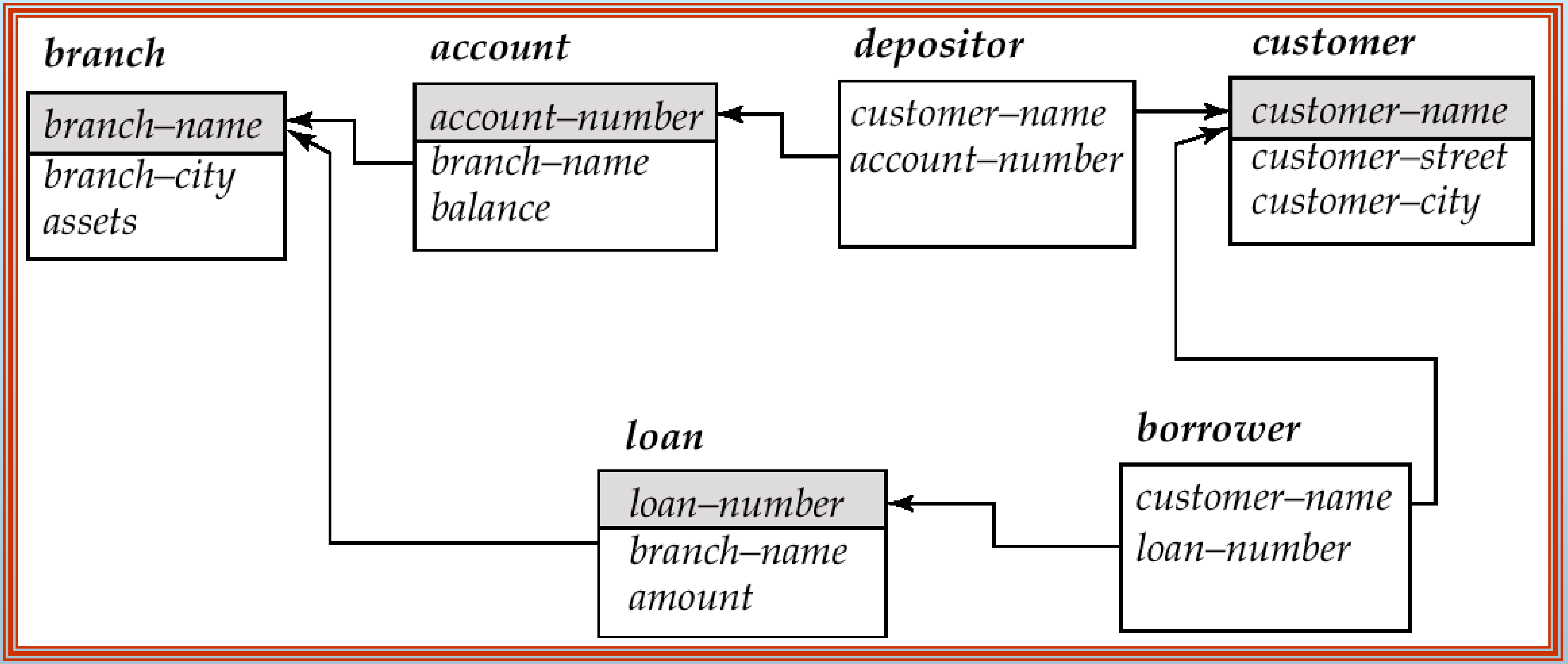
1. Introduction

- SQL (Structured Query Language) is the most popular and most user friendly query language. SQL uses a combination of *relational-algebra* and *relational-calculus* constructs.
- Although we refer to the SQL language as a “*query language*”, it can be used for defining the structure of the data, modifying the data in the database, and specifying security constraints.
- The SQL has the following parts:
  - **Data-definition language(DDL): -** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
  - **Interactive data-manipulation language(DML): -** The SQL DML includes a query language based on both the relational algebra and tuple relational calculus. It also includes commands to insert, delete, and modify tuples.

Introduction (Cont.)

- **View definition: -** The SQL DDL includes commands for defining views.
- **Transaction control: -** SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL and dynamic SQL: -** Embedded SQL and dynamic SQL defines how SQL statements can be embedded within general purpose programming languages, such as C, C++, Java etc.
- **Integrity: -** The SQL DDL includes commands for specifying integrity constraints.
- **Authorization: -** The SQL DDL includes commands for specifying access rights to relations and views.

Schema Used in Examples



2. Basic Structure

- The basic structure of SQL expression consists of three clauses: **select**, **from** and **where**.
  - The **select** clause corresponds to the projection operation of the *relational-algebra*. It is used to list the attributes desired in the result of a query.
  - The **from** clause corresponds to the Cartesian-product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
  - The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

Basic Structure (Cont.)

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i*s* represent attributes
  - r_i*s* represent relations
  - P is a predicate.

- This query is equivalent to the relational algebra expression.

$$\Pi_{A1,\ A2,\ ...,\ An}(\sigma_P\,(r_1 \times r_2 \times ... \times r_m))$$

- The result of an SQL query is a relation.
- SQL forms the Cartesian-product of the relations named in the **from** clause, performs a *relational-algebra* selection using the **where** clause predicate, and then projects the result onto the attributes of the **select** clause.

2.1. The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- E.g. find the names of all branches in the *loan* relation

  select *branch-name*
  from *loan*
- In the "pure" relational algebra syntax, the query would be:

$$\Pi_{\text{branch-name}}(loan)$$

- NOTE: SQL does not permit the '-' character in names,
  - Use, e.g., *branch_name* instead of *branch-name* in a real implementation.
- NOTE: SQL names are case insensitive, i.e. you can use capital or small letters.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after **select.**
- Find the names of all branches in the *loan* relations, and remove duplicates

  select distinct *branch-name*
  from *loan*

- The keyword **all** specifies that duplicates not be removed.

  select all *branch-name*
  from *loan*

The select Clause (Cont.)

- An asterisk in the select clause denotes "all attributes"

  select *

  from *loan*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, $*$, and /, and operating on constants or attributes of tuples.
- The query:

  select *loan-number, branch-name, amount* $*$ 100

  from *loan*

  would return a relation which is the same as the *loan* relations, except that the attribute *amount* is multiplied by 100.

2.2. The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

  select *loan-number*
  from *loan*
  where *branch-name* = 'Perryridge' **and** *amount* > 1200
- Comparison results can be combined using the logical connectives **and, or,** and **not**.
- Comparisons expressions involve comparison operators (<, <=, >, >=, = and <>).

The where Clause (Cont.)

- SQL includes a **between** comparison operator
- E.g. Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, ≥\$90,000 and ≤\$100,000), we write

select *loan-number*
from *loan*
where *amount* **between** 90000 **and** 100000

instead of

select *loan-number*
from *loan*
where *amount* >= 90000 **and** *amount* <= 100000

2.3. The from Clause

- The **from** clause lists the relations involved in the query
  - corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower x loan*

  select ∗
  from *borrower, loan*

- Find the name, loan number and loan amount of all customers who have loan from the bank.

  select *customer-name, borrower.loan-number, amount*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number*

The from Clause (Cont.)

- Find the name, loan number and loan amount of all customers who have loan from the Perryridge branch.

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number = loan.loan-number* **and** *branch-name =* 'Perryridge'

2.4. The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- The **as** can appear in both the **select** and **form** clauses.
- Find the name, loan number and loan amount of all customers; rename the column name *loan-number* as *loan-id.*

  select *customer-name, borrower.loan-number* **as** *loan-id, amount*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number*

2.5. Tuple Variables

- A tuple variable is associated with a particular relation.
- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

select *customer-name, T.loan-number, S.amount*
from *borrower* **as** *T, loan* **as** *S*
where *T.loan-number = S.loan-number*

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

select distinct *T.branch-name*
from *branch* **as** *T, branch* **as** *S*
where *T.assets > S.assets* **and** *S.branch-city* = 'Brooklyn'

2.6. String Operations

- The most commonly used operation on strings is pattern matching using the operator **like**. We describe patterns by using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Examples:
  - ‘Perry%’ matches any string beginning with “Perry”.
  - ‘%Perry’ matches any string ending with “Perry”.
  - ‘%Perry%’ matches any string containing “Perry” as a substring.
  - ‘---’ matches any string of exactly three characters.
  - ‘---%’ matches any string of at least three characters.
  - ‘%---’ matches any string of at most three characters.

String Operations (Cont.)

- Find the names of all customers whose street includes the substring “Main”.

  select *customer-name*
  from *customer*
  where *customer-street* **like ‘**%Main%’

- We define the escape character for a **like** comparison using the **escape** keyword as follows:
  - **like ‘**Main\%’ **escape** ‘\’ matches the string “Main%”
  - **like** ‘ab\%cd%’**escape** ‘\’ matches all strings beginning with “ab%cd”.
- Patterns are case sensitive.
- SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator.

String Operations (Cont.)

- SQL also supports a variety of string operations such as
  - Concatenation (using "||")
  - Extracting substrings
  - Converting from upper to lower case (and vice versa)
  - finding string length
  - It also provides a **similar to** operation which provides more powerful pattern matching than the **like** operation.

Note: A single quote character that is part of a string can be specified by using two single quote characters; for example "It's right" can be specified by 'It''s right'.

2.7. Ordering the Display of Tuples

- The **order by** clause causes the tuples in the result of a query to appear in sorted order.
- To list in alphabetic order the names of all customers having a loan in Perryridge branch

  select distinct *customer-name*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number* **and**
  branch-name = 'Perryridge'
  order by *customer-name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - E.g. **order by** *customer-name* **desc**

2.8. Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- We can define the duplicate semantics of an SQL query using multiset versions of the relational algebra.
- *Multiset* versions of some of the relational algebra operators – given multiset relations r_1 and r_2:
  1. $\boldsymbol{\sigma_\theta(r_1)}$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
  2. $\boldsymbol{\Pi_A(r_1)}$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
  3. $\boldsymbol{r_1 \times r_2}$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1.\ t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  $r_1 = \{(1, a)\ (2,a)\}$ $\quad$ $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1) \times r_2$ would be

  $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$

- SQL duplicate semantics:

  select $A_{1,}, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  is equivalent to the *multiset* version of the expression:

  $\Pi_{A1,, A2, ..., An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$

3. Set Operations

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup, \cap, -$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

Set Operations (Cont.)

- Find all customers who have a loan, an account, or both:

  (**select** *customer-name* **from** *depositor*)
  union
  (**select** *customer-name* **from** *borrower)*

- Find all customers who have both a loan and an account.

  (**select** *customer-name* **from** *depositor*)
  intersect
  (**select** *customer-name* **from** *borrower)*

- Find all customers who have an account but no loan.

  (**select** *customer-name* **from** *depositor*)
  except
  (**select** *customer-name* **from** *borrower)*

4. Aggregate Functions

- These functions operate on a collection (a set or multiset) of values of a column of a relation as input and return a single value. SQL offers the following five built-in aggregate functions:

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types.

Aggregate Functions (Cont.)

- For example, to find the average account balance at the Perryridge branch, we write

  select avg *(balance)*
  from *account*
  where *branch-name* = 'Perryridge'

The result of this query is a relation with a single attribute, containing a single tuple. We can also give name to the attribute of the result relation by using the **as** clause.

- We use **group by** clause if we want to apply the aggregate functions to a group of sets of tuples. The attribute or attributes given in the **group by** clause are used to form groups. For example, to find the average account balance at each branch, we write

  select *branch-name*, **avg** *(balance)*
  from *account*
  group by *branch-name*

Aggregate Functions (Cont.)

- To eliminate duplicates before computing an aggregate function, we use the keyword **distinct** in the aggregate expression. For example, to find the number of depositors for each branch, we write

  select *branch-name,* **count (distinct** *customer-name)*
  from *depositor, account*
  where *depositor.account-number = account.account-number*
  group by *branch-name*

- To state a condition that applies to groups rather than to tuples, we use the **having** clause. For example, to find the names of all branches where the average account balance is more than $1,200, we write

  select *branch-name,* **avg** *(balance)*
  from *account*
  group by *branch-name*
  having avg *(balance)* > 1200

Aggregate Functions (Cont.)

- If we wish to treat the entire relation as a single group, we do not use a **group by** clause. For example, to find the average balance for all accounts, we write

  select avg *(balance)*
  from *account*

- We use aggregate function **count** frequently to count the number of tuples in a relation. Thus to find the number of tuples in the customer relation, we write

  select count (*)
  from *customer*

- SQL does not allow the use of **distinct** with **count(*)**. It is also illegal to use distinct with **max** and **min**, even though the result does not change. We can use the keyword **all** in place of **distinct**, but **all** is the default.

Aggregate Functions (Cont.)

- If a **where** clause and **having** clause appear in the same query, SQL applies the predicate in the **where** clause first. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause. SQL then applies **having** clause, if it is present, to each group; it removes the groups that do not satisfy the **having** clause predicate. The **select** clause uses the remaining groups to generate tuples of the result of the query. For example, to find the average balance for each customer who lives in the Harrison and has at least three accounts, we write

select *depositor.customer-name,* **avg** (*balance)*
  from *depositor, account, customer*
  where *depositor.account-number = account.account-number* **and**
    depositor.customer-name = customer.customer-name **and**
    customer-city = 'Harrison'
  group by *depositor.customer-name*
  having count (**distinct** *depositor.account-number) >= 3*

Aggregate Functions (Cont.)

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

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

5. Null Values

- SQL allows the use of *null* values to indicate the absence of information about the value of an attribute.
- The keyword **is null** can be used to check for null values.
  - For example, to find all loan number which appear in the *loan* relation with null values for *amount,* we write

    select *loan-number*
    from *loan*
    where *amount* **is null**
- The keyword **is not null** tests the absence of a *null* value.
- The result of any arithmetic expression involving *null* is *null*
  - E.g. 5 + null returns null

Null Values (Cont.)

- Any comparison with *null* returns *unknown*
  - *E.g. 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown) = unknown, (false* **and** *unknown) = false, (unknown* **and** *unknown) = unknown*
  - NOT*:* (**not** *unknown) = unknown*
- We use the clauses **is unknown** and **is not unknown** to test whether the result of a comparison is unknown.

Null Values (Cont.)

- Aggregate functions ignore *null* values. For example, assume that some tuples in the *loan* relation have null value for *amount*. Consider the following query to total all loan amounts:

  select sum (*amount)*
  from *loan*

  - Above statement ignores null amounts
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

6. Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, make set comparisons, and determining set cardinality.

6.1. Set Membership

- The **in** connective tests for set membership, where set is the collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership. For example, to find all customers who have both a loan and an account, we write

  select distinct *customer-name*
  from *borrower*
  where *customer-name* **in (select** *customer-name*
  from depositor)

- Similarly, to find all customers who have a loan at the bank but do not have an account at the bank, we write

  select distinct *customer-name*
  from *borrower*
  where *customer-name* **not in (select** *customer-name*
  from *depositor)*

Set Membership (Cont.)

- The **in** and **not in** operators can also be used in enumerated sets. For example, to find the names of customers who have loan at the bank, and whose names are neither Smith nor Jones, we write

select distinct *customer-name*
from *borrower*
where *customer-name* **not in** (‘Smith’, ‘Jones’)

Set Membership (Cont.)

- We can also test set membership in an arbitrary relation. For example, to find all customers who have both an account and a loan at the Perryridge branch, we write

select distinct *customer-name*
from *borrower, loan*
where *borrower.loan-number = loan.loan-number* **and**
branch-name = "Perryridge" **and**
(branch-name, customer-name) **in**
(select *branch-name, customer-name*
from *depositor, account*
where *depositor.account-number =*
account.account-number)

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

6.2. Set Comparison

- We can use nested subquery to compare sets. For example, to find all branches that have greater assets than those of at least one branch located in Brooklyn, we write

select distinct *T.branch-name*
from *branch* **as** *T, branch* **as** *S*
where *T.assets > S.assets* **and**
S.branch-city = 'Brooklyn'

- We can also write the same query using **> some** clause. The **> some** clause is used to represent "greater than at least one"

select *branch-name*
from *branch*
where *assets* > **some**
(select *assets*
from *branch*
where *branch-city* = 'Brooklyn')

Set Comparison (Cont.)

- SQL also allows **<some**, **<=some**, **>=some**, **=some**, and **<>some** comparisons. The keyword **any** is synonymous to **some**.

$$(5 < \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$$

(read: 5 < some tuple in the relation)

$$(5 < \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 = \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$$

$$(5 \mathrel{<>} \textbf{some} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true (since } 0 \neq 5)$$

$(= \textbf{some}) \equiv \textbf{in}$

However, $(<> \textbf{some}) \not\equiv \textbf{not in}$

Set Comparison (Cont.)

- SQL also allows **>all**, **>=all**, **<all**, **<=all**, **=all** and **<>all** comparisons.

(5< **all** | 0 | 5 | 6 |) = false

(5< **all** | 6 | 10 |) = true

(5 = **all** | 4 | 5 |) = false

(5 <> **all** | 4 | 6 |) = true (since $5 \neq 4$ and $5 \neq 6$)

(<> **all**) $\equiv$ **not in**

However, (= **all**) $\not\equiv$ **in**

Set Comparison (Cont.)

- For example, to find the names of all branches that have greater assets than all branches located in Brooklyn, we write

  select *branch-name*
  from *branch*
  where *assets* **> all**
  (select *assets*
  from *branch*
  where *branch-city* = 'Brooklyn')

Set Comparison (Cont.)

- We can also use set comparisons with **having** clause. For example, to find the branch that has the highest average balance, we write

select *branch-name*
from *account*
group by *branch-name*
having avg(*balance*) **>= all** (**select avg**(*balance*)
from *account*
group by *branch-name*)

6.3. Test for Empty Relations

- SQL includes features for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty. For example, to find all customers who have both an account and loan at the bank, we can write

  select *customer-name*
  from *borrower*
  where exists (select *
  from *depositor*
  where *depositor.customer-name*
  = *borrower.customer-name*)

- We can test for the nonexistence of tuples in a subquery by using the **not exists** construct.

6.4.Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result. It returns true if the subquery contains no duplicate tuples.
- Find all customers who have at most one account at the Perryridge branch.

  select *T.customer-name*
  from *depositor* **as** *T*
  where unique (

  select *R.customer-name*
  from *account, depositor* **as** *R*
  where *T.customer-name* = *R.customer-name* **and**
  R.account-number = *account.account-number* **and**
  account.branch-name = 'Perryridge')

Test for Absence of Duplicate Tuples (Cont.)

- Find all customers who have at least two accounts at the Perryridge branch.

select distinct *T.customer-name*
from *depositor T*
where not unique (
  select *R.customer-name*
  from *account, depositor* **as** *R*
  where *T.customer-name = R.customer-name*
and
  R.account-number = account.account-number
and
  account.branch-name = 'Perryridge')

7. Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

create view *v* **as** <query expression>

where:

- <query expression> is any legal expression
- The view name is represented by *v*

Views (Cont.)

- A view consisting of branches and their customers

  create view *all-customer* **as**
  (**select** *branch-name, customer-name*
  from *depositor, account*
  where *depositor.account-number = account.account-number)*
  union
  (**select** *branch-name, customer-name*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number)*

- view names may appear in any place that a relation name may appear. For example, to find all customers of the Perryridge branch, we write

  select *customer-name*
  from *all-customer*
  where *branch-name* = 'Perryridge'

Views (Cont.)

- The attribute names of a view can be specified as follows

  create view *all-customer (bn, cn)* **as**
  (**select** *branch-name, customer-name*
  from *depositor, account*
  where *depositor.account-number = account.account-number)*
  union
  (**select** *branch-name, customer-name*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number)*

- To find all customers of the Perryridge branch, we write

  select *cn*
  from *all-customer*
  where *bn* = 'Perryridge'

8. Modification of the Database

8. 1. Deletion

- We can delete only whole tuples; we can not delete values on only particular attributes.
- SQL expresses a deletion by

  Delete from *r*

  where *P*

  Where *P* represents a predicate and *r* a relation. The **delete** statement first finds all tuples *t* in *r* for which *P(t)* is true, then deletes them from *r*. The where clause can be omitted, in which case all tuples in *r* are deleted.
- **Note:** A **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.

Deletion (Cont.)

- To delete all tuples from loan relation, we write

  delete from *loan*

- To delete all account records at the Perryridge branch, we write

  delete from *account*
  where *branch-name* = 'Perryridge'

- To delete all loans with loan amounts between \$1300 and \$1500, we write

  delete from *loan*
  where *amount* **between** *1300* **and** *1500*

- Delete all accounts at every branch located in Needham city.

  delete from *account*
  where *branch-name* **in** (**select** *branch-name*
  from *branch*
  where *branch-city* = 'Needham')

Deletion (Cont.)

delete from *depositor*
where *account-number* **not in** (**select** *account-number*
from *account*)

8.2. Insertion

- To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.
- The simple **insert** statement is a request to insert one tuple.
- Add a new tuple to *account*

  insert into *account*
  values (‘A-9732’, ‘Perryridge’,1200)

  or equivalently

  insert into *account (branch-name, balance, account-number)*
  values (‘Perryridge’, 1200, ‘A-9732’)
- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values (‘A-777’,‘Perryridge’, *null*)

Insertion (Cont.)

- More generally, we might want to insert tuples on the basis of the result of a query. For example, to provide as a gift for all loan customers of the Perryridge branch, a $200 savings account (let the loan number serve as the account number for the new savings account), we write

  insert into *account*
  select *loan-number, branch-name,* 200
  from *loan*
  where *branch-name* = 'Perryridge'
  insert into *depositor*
  select *customer-name, loan-number*
  from *loan, borrower*
  where branch-name = 'Perryridge'
  and *loan.account-number* = *borrower.account-number*

8.3. Updates

- To change a value in a tuple without changing all values in the tuple, we use **update** statement.
- To increase all balances by 5 percent we write

  update *account*
  set *balance* = *balance* $*$ 1.05

- If interest is to be paid only to accounts with a balance of \$1000 or more, we write

  update *account*
  set *balance* = *balance* $*$ 1.05
  where *balance* > 10000

Updates (Cont.)

- To pay 5% interest on accounts whose balance is greater than average, we write

update *account*
set *balance* = *balance* ∗ 1.05
where *balance* > (**select avg**(*balance*)
from *account*)

Updates (Cont.)

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

    update *account*
    set *balance* = *balance* $*$ 1.06
    where *balance* > 10000

    update *account*
    set *balance* = *balance* $*$ 1.05
    where *balance* ≤ 10000
  - The order is important
  - Can be done better using the **case** statement (next slide)

Updates (Cont.)

- Same query as before: Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

  update *account*
  set *balance* = **case**
  when *balance* <= 10000 **then** *balance* *1.05
  else *balance* * 1.06
  end

- The general form of the case statement is:

  case
  when $pred_1$ **then** $result_1$
  when $pred_2$ **then** $result_2$
  …….
  when $pred_n$ **then** $result_n$
  else $result_0$
  end

8.4. Update of a View

- Create a view of all loan data in *loan* relation, hiding the *amount* attribute

  create view *branch-loan* **as**
  select *branch-name, loan-number*
  from *loan*

- Add a new tuple to *branch-loan*

  insert into *branch-loan*
  values ('Perryridge', 'L-307')

  This insertion must be represented by the insertion of the tuple

  ('L-307', 'Perryridge', *null*)

  into the *loan* relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.

9. Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- Each of the variants of the join operations consists of a *join type* and a *join condition*.
  - **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
  - **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join Types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using (A_1, A_2, ..., A_n)

Joined Relations (Cont.)

- Relation *loan*

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

- Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations (Cont.)

- *loan* **inner join** *borrower* **on**
  loan.loan-number = borrower.loan-number

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan* **left outer join** *borrower* **on**
  loan.loan-number = borrower.loan-number

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations (Cont.)

- *loan* **natural inner join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- loan **natural right outer join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

Joined Relations (Cont.)

- *loan* **full outer join** *borrower* **using** *(loan-number)*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	null	null	Hayes

- Find all customers who have either an account or a loan (but not both) at the bank.

select *customer-name*
from (*depositor* **natural full outer join** *borrower*)
where *account-number* **is** *null* **or** *loan-number* **is** *null*

10. Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

10.1. Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits (plus a sign), with *d* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.

Domain Types in SQL

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. **date** '2001-7-27'
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. **time** '09:00:30'
- **timestamp**: date plus time of day
  - E.g. **timestamp** '2001-7-27 09:00:30'
- **Interval**: period of time
  - E.g. Interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Domain Types in SQL

- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person-name* **char**(20) **not null**

- We can extract values of individual fields from date/time/timestamp
  - E.g. **extract** (**year from** d)
- We can cast string types to date/time/timestamp
  - E.g. **cast** <string-valued-expression> **as date**

10.2. Creating a Table Construct

- An SQL relation is defined using the **create table** command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i
- Example:

  create table *branch*
  (*branch-name* char(15) **not null,**
  branch-city char(30),
  assets integer)

10.3. Integrity Constraints

Integrity constraints ensure that the changes made to the database by unauthorized users do not result in a loss of data consistency. Some are:

- **not null**
- **primary key** (A_1, ..., A_n)
- **check** *(P),* where *P* is a predicate

Example: Declare *branch-name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

create table *branch*
(branch-name char(15)**,**
branch-city char(30)
assets integer,
primary key *(branch-name),*
check *(assets >=* 0))

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89

10.4. Drop and Alter Table Constructs

- To remove a relation from database, we use the **drop table** command. It deletes all information about the dropped relation from the database. To delete a relation r, we write

  drop table r

- The **alter table** command is used to add attributes to an existing relation.

  alter table *r* **add** *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*

  - All tuples in the relation are assigned *null* as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation

  alter table *r* **drop** *A*

  where *A* is the name of an attribute of relation *r*

  - Dropping of attributes not supported by many databases

11. Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, Java and Cobol.
- A language to which SQL queries are embedded is referred to as a ***host* language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database.
- In embedded SQL, all query processing is performed by the database system, which then makes the result of the query available to the program one tuple at a time.
- An embedded SQL program must be processed by a special processor prior to compilation. EXEC SQL statement is used to identify embedded SQL request to the preprocessor. It has the form

  EXEC SQL <embedded SQL statement > END-EXEC

  Note: this varies by language. E.g. the Java embedding uses
  # SQL { …. } ;

Embedded SQL (Cont.)

- We place the statement SQL INCLUDE in the program to identify the place where the preprocessor should insert the special variables used for communication between the program and the database system.
- Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.
- To write a query, we use **declare cursor** statement. The program must use **open** and **fetch** commands to obtain the result tuples.

Embedded SQL (Cont.)

Example

From within a host language, find the names and cities of customers with more than the variable *amount* dollars in some account.

- Specify the query in SQL and declare a *cursor* for it

EXEC SQL

declare *c* **cursor for**
select *customer-name, customer-city*
from *depositor, customer, account*
where *depositor.customer-name = customer.customer-name*
        and *depositor account-number = account.account-number*
        and *account.balance > :amount*

END-EXEC

Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END-EXEC

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** :*cn, :cc* END-EXEC

  Repeated calls to **fetch** get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* END-EXEC

Note: above details vary with language. E.g. the Java embedding defines Java iterators to step through result tuples.

Embedded SQL (Cont.)

- We can also update tuples fetched by cursor by declaring that the cursor is for update

EXEC SQL
  declare *c* **cursor for**
    select *
    from *account*
    where *branch-name* = 'Perryridge'
   for update
  END-EXEC

- To update tuple at the current location of cursor

   update *account*
   set *balance* = *balance* + 100
   where current of *c*

- We can also modify databases by using embedded SQL. E.g.

  EXEC SQL <any valid update, insert or delete > END-EXEC

12. Dynamic SQL

- The dynamic SQL component of SQL allows programs to construct and submit SQL queries at run time.
- In contrast, embedded SQL statements must be completely present at compile time; they are compiled by the embedded SQL preprocessor.
- Using dynamic SQL, programs can create SQL queries as strings at run time and can either have them executed immediately or have them prepared for subsequent use.
- Example of the use of dynamic SQL from within a C program.

  char * *sqlprog* = "**update** *account*
  set *balance* = *balance* * 1.05
  where *account-number* = ?"
  EXEC SQL **prepare** *dynprog* **from** *:sqlprog;*
  char *account* [10] = "A-101";
  EXEC SQL **execute** *dynprog* **using** *:account;*

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.