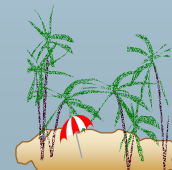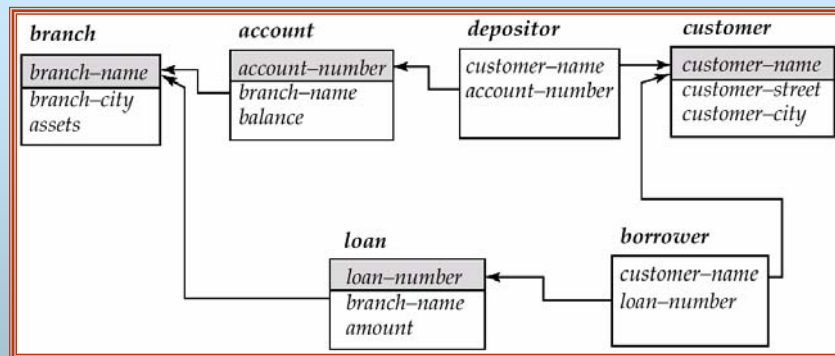# Chapter 4: SQL

- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Derived Relations
- Views
- Modification of the Database
- Joined Relations
- Data Definition Language
- Embedded SQL, ODBC and JDBC

Schema Used in Examples

Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

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

  - ☞ A_is represent attributes
  - ☞ r_is 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.

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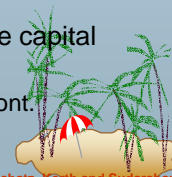
$$\Pi_{\text{branch-name}}(loan)$$

- NOTE: SQL does not permit the '-' character in names,
  - ☞ Use, e.g., *branch_name* instead of *branch-name* in a real implementation.
  - ☞ We use '-' since it looks nicer!
- NOTE: SQL names are case insensitive, i.e. you can use capital or small letters.
  - ☞ You may wish to use upper case where-ever we use bold font.

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.
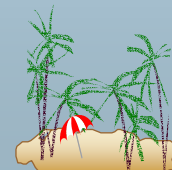
The where Clause

- 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 can be applied to results of arithmetic expressions.

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)

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

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 having a loan at the Perryridge branch.

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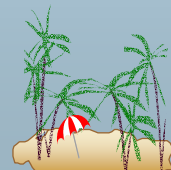   *branch-name* = 'Perryridge'

The Rename Operation

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

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

- 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*

Tuple Variables

- 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*'*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
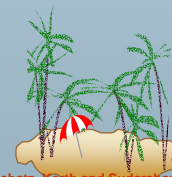- Find the names of all customers whose street includes the substring "Main".

  select *customer-name*
  from *customer*
  where *customer-street* **like** '%Main%'

- Match the name "Main%"

  like 'Main\%' **escape** '\'

- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- 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**

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- *Multiset* versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

  1. $\sigma_\theta (r_1)$: 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. $\Pi_A(r_1)$: 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. $r_1 \times r_2$: 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 $r_1 \times r_2$
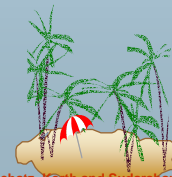
Duplicates (Cont.)

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

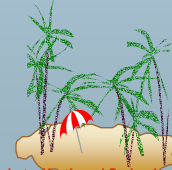$$r_1 = \{(1, a)\ (2,a)\} \qquad r_2 = \{(2), (3), (3)\}$$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x 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:

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

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.**

  Suppose a tuple occurs m times in r and n times in s, then, it occurs:

  - $m + n$ times in r **union all** s
  - min(m,n) times in r **intersect all** s
  - max(0, $m - n$) times in r **except all** s
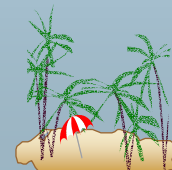
Set Operations

- 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)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

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

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

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

- Find the number of tuples in the *customer* relation.

    select count (*)
    from *customer*

- Find the number of depositors in the bank.

    select count (distinct *customer-name)*
    from *depositor*
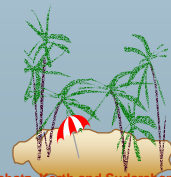

Aggregate Functions – Group By

- Find the number of depositors for each branch.

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

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

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than $1,200.

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

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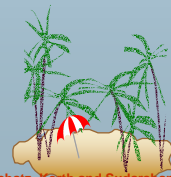

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - E.g. Find all loan number which appear in the *loan* relation with null values for *amount.*

    > **select** *loan-number*
    > **from** *loan*
    > **where** *amount* **is null**

- The result of any arithmetic expression involving *null* is *null*
  - E.g. 5 + null returns null
- However, aggregate functions simply ignore nulls
  - more on this shortly

Null Values and Three Valued Logic

- 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*
  - "*P* **is unknown"** evaluates to true if predicate *P* evaluates to *unknown*
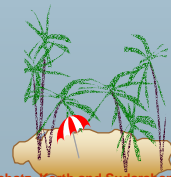- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount)*
  from *loan*
  - Above statement ignores null amounts
  - result is null if there is no non-null amount
  - All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

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, set comparisons, and set cardinality.

Example Query

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

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

- Find all customers who have a loan at the bank but do not have an account at the bank

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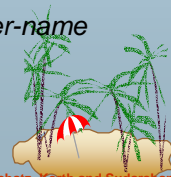
Example Query

- Find all customers who have both an account and a loan at the Perryridge branch

  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.

(Schema used in this example)

Set Comparison

- Find 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'

- Same query using > **some** clause

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

Definition of Some Clause

- F \<comp\> **some** $r \Leftrightarrow \exists\ t \in r$ *s.t. (F \<comp\> t)*
  Where \<comp\> can be: $<, \leq, >, =, \neq$

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

                          (read: 5 \< some tuple in the relation)

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

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

(5 $\neq$ **some** $\begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 $\neq$ 5)

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

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

Definition of all Clause

- F \<comp\> **all** $r \Leftrightarrow \forall\ t \in r$ (F \<comp\> t)

(5\< **all** $\begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = false

(5\< **all** $\begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}$) = true

(5 = **all** $\begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}$) = false

(5 $\neq$ **all** $\begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}$) = true (since 5 $\neq$ 4 and 5 $\neq$ 6)

$(\neq \textbf{all}) \equiv \textbf{not in}$

However, $(= \textbf{all}) \not\equiv \textbf{in}$

Example Query

■ Find the names of all branches that have greater assets than all branches located in Brooklyn.

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

Test for Empty Relations

■ The **exists** construct returns the value **true** if the argument subquery is nonempty.

■ **exists** $r \Leftrightarrow r \neq \emptyset$

■ **not exists** $r \Leftrightarrow r = \emptyset$

Example Query

- Find all customers who have an account at all branches located in Brooklyn.

  select distinct *S.customer-name*
        from *depositor* **as** *S*
        where not exists (
                      (select *branch-name*
                      from *branch*
                      where *branch-city* = 'Brooklyn')
                except
                      (select *R.branch-name*
                      from *depositor* **as** *T, account* **as** *R*
                      where *T.account-number* = *R.account-number* **and**
                              S.customer-name = *T.customer-name))*

- (Schema used in this example)
- Note that $X - Y = \varnothing \iff X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

---

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- 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')

- (Schema used in this example)
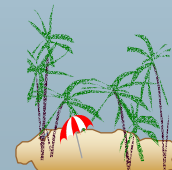
Example Query

- 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*')

- (Schema used in this example)
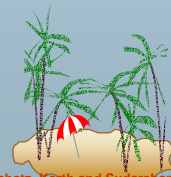

Views

- 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*

Example Queries

- 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*)

- Find all customers of the Perryridge branch

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

Derived Relations

- Find the average account balance of those branches where the average account balance is greater than $1200.

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

  Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *result* in the **from** clause, and the attributes of *result* can be used directly in the **where** clause.
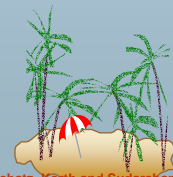
With Clause

- With clause allows views to be defined locally to a query, rather than globally. Analogous to procedures in a programming language.

- Find all accounts with the maximum balance

    with *max-balance*(*value*) **as**
        select max (*balance*)
        from *account*
    select *account-number*
    from *account, max-balance*
    where *account.balance = max-balance.value*

Complex Query using With Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

    with *branch-total* (*branch-name, value*) **as**
        select *branch-name*, **sum** (*balance*)
        from *account*
        group by *branch-name*
    with *branch-total-avg*(*value*) **as**
        select avg (*value*)
        from *branch-total*
    select *branch-name*
    from *branch-total, branch-total-avg*
    where *branch-total.value >= branch-total-avg.value*
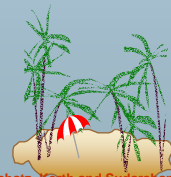
Modification of the Database – Deletion

- Delete all account records at the Perryridge branch

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

- 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′)
    delete from *depositor*
    where *account-number* **in**
            (**select** *account-number*
            from *branch, account*
            where *branch-city* = ′Needham′
              and *branch.branch-name* = *account.branch-name)*

- (Schema used in this example)
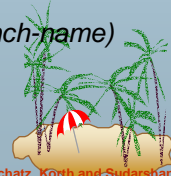
Example Query

- Delete the record of all accounts with balances below the average at the bank.

    delete from *account*
        where *balance* < (**select avg** *(balance)*
            from *account)*

- Problem: as we delete tuples from *deposit,* the average balance changes
- Solution used in SQL:
1. First, compute **avg** balance and find all tuples to delete
2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- 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*)

Modification of the Database – Insertion

- 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

  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*

- The select from where statement is fully evaluated before any of its results are inserted into the relation (otherwise queries like
  insert into *table*1 **select** * **from** *table*1
  would cause problems

Modification of the Database – Updates

- 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
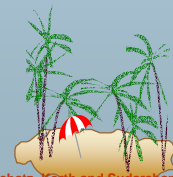   **where** *balance* ≤ 10000

    - The order is important
    - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- 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

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.

- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

---

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - **commit work:** makes all updates of the transaction permanent in the database
    - **rollback work:** undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by **rollback work.**
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within
    begin atomic
    …
    end

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
- 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	Join Conditions
inner join	**natural**
left outer join	**on** <predicate>
right outer join	**using** $(A_1, A_2, ..., A_n)$
full outer join	

Joined Relations – Datasets for Examples

- 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 – Examples

- *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 – Examples

■ *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 – Examples

■ *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*

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.

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, with *n* 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.
- 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**
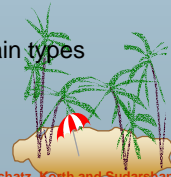
Date/Time Types in SQL (Cont.)

- **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'   **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - E.g. **timestamp** '2001-7-27 09:00:30.75'
- **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
- Can extract values of individual fields from date/time/timestamp
  - E.g. **extract** (**year from** r.starttime)
- Can cast string types to date/time/timestamp
 - E.g. **cast** <string-valued-expression> **as date**

Create Table Construct

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

  create table r $(A_1\ D_1,\ A_2\ D_2,\ ...,\ A_n\ D_n,$
  $(integrity\text{-}constraint_1),$
  $...,$
  $(integrity\text{-}constraint_k))$

  - 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)
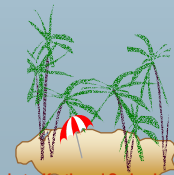
Integrity Constraints in Create Table

- **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

Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.
- 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

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, 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.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- EXEC SQL statement is used to identify embedded SQL request to the preprocessor

    EXEC SQL <embedded SQL statement > END-EXEC

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

Example Query

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.

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

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

- To update tuple at the current location of cursor

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

Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- 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.

ODBC

- Open DataBase Connectivity(ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
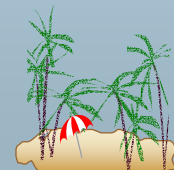
ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

- int ODBCexample()
  {
  RETCODE error;
  HENV   env;   /* environment */
  HDBC   conn; /* database connection */
  SQLAllocEnv(&env);
  SQLAllocConnect(env, &conn);
  SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS, "avipasswd", SQL_NTS);
  { …. Do actual work … }

  SQLDisconnect(conn);
  SQLFreeConnect(conn);
  SQLFreeEnv(env);
  }

ODBC Code (Cont.)

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
    - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
    - Arguments to SQLBindCol()
        - ODBC stmt variable, attribute position in query result
        - The type conversion from SQL to C.
        - The address of the variable.
        - For variable-length types like character arrays,
            - » The maximum length of the variable
            - » Location to store actual length when a tuple is fetched.
            - » Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.


ODBC Code (Cont.)

- Main body of program

```
char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;

SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
                from account
                group by branch_name";

error = SQLExecDirect(stmt, sqlquery, SQL_NTS);

if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR,   branchname , 80, &lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,        0 , &lenOut2);

    while (SQLFetch(stmt) >= SQL_SUCCESS) {
        printf (" %s  %g\n", branchname, balance);
    }
}
SQLFreeStmt(stmt, SQL_DROP);
```
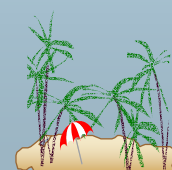
More ODBC Features

- Prepared Statement
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- Metadata features
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
  - transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requires support for metadata querying
  - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.
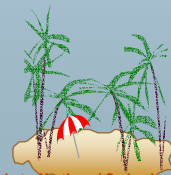
JDBC

- JDBC is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
            … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(  "insert into account values
                                    ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery( "select branch_name, avg(balance)
                                from account
                                group by branch_name");

while (rset.next()) {
    System.out.println(
            rset.getString("branch_name") + "  " + rset.getFloat(2));

}
```
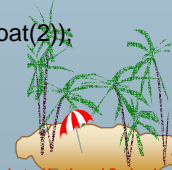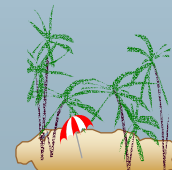

JDBC Code Details

- Getting result fields:
  - rs.getString("branchname") and rs.getString(1) equivalent if branchname is the first argument of select result.
- Dealing with Null values

```
int a = rs.getInt("a");
if (rs.wasNull()) Systems.out.println("Got null value");
```

Prepared Statement

- Prepared statement allows queries to be compiled and executed multiple times with different arguments

  PreparedStatement pStmt = conn.prepareStatement(

  "insert into account values(?,?,?)");

  pStmt.setString(1, "A-9732");

  pStmt.setString(2, "Perryridge");

  pStmt.setInt(3, 1200);

  pStmt.executeUpdate();

  pStmt.setString(1, "A-9733");

  pStmt.executeUpdate();

- Beware: If value to be stored in database contains a single quote or other special character, prepared statements work fine, but creating a query string and executing it directly would result in a syntax error!

Other SQL Features

- SQL sessions
  - client *connects* to an SQL server, establishing a session
  - executes a series of statements
  - *disconnects* the session
  - can *commit* or *rollback* the work carried out in the session
- An SQL environment contains several components, including a user identifier, and a *schema,* which identifies which of several schemas a session is using.
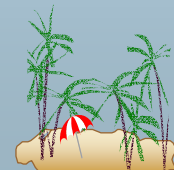
Schemas, Catalogs, and Environments

- Three-level hierarchy for naming relations.
  - Database contains multiple **catalogs**
  - each catalog can contain multiple **schemas**
  - SQL objects such as relations and views are contained within a schema
- e.g. catalog5.bank-schema.account
- Each user has a default catalog and schema, and the combination is unique to the user.
- Default catalog and schema are set up for a connection
- Catalog and schema can be omitted, defaults are assumed
- Multiple versions of an application (e.g. production and test) can run under separate schemas

Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
  - more in Chapter 9
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- These features are covered in Chapter 9 (Object Relational Databases)
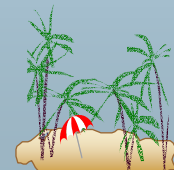
Extra Material on JDBC and Application Architectures

Transactions in JDBC

- As with ODBC, each statement gets committed automatically in JDBC
- To turn off auto commit use
  conn.setAutoCommit(false);
- To commit or abort transactions use
  conn.commit() or conn.rollback()
- To turn auto commit on again, use
  conn.setAutoCommit(true);

Procedure and Function Calls in JDBC

- JDBC provides a class CallableStatement which allows SQL stored procedures/functions to be invoked.

 CallableStatement cs1 = conn.prepareCall("{call proc (?,?)}") ;

  CallableStatement cs2 = conn.prepareCall("{? = call func (?,?)}");

Result Set MetaData

- The class ResultSetMetaData provides information about all the columns of the ResultSet.
- Instance of this class is obtained by getMetaData( ) function of ResultSet.
- Provides Functions for getting number of columns, column name, type, precision, scale, table from which the column is derived etc.

  ResultSetMetaData rsmd = rs.getMetaData ( );

  for (int i = 1; i <= rsmd.getColumnCount( ); i++) {

      String name = rsmd.getColumnName(i);

      String typeName = rsmd.getColumnTypeName(i);

  }
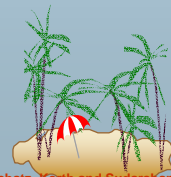
Database Meta Data

- The class DatabaseMetaData provides information about database relations
- Has functions for getting all tables, all columns of the table, primary keys etc.
- E.g. to print column names and types of a relation

  DatabaseMetaData dbmd = conn.getMetaData( );

  ResultSet rs = dbmd.getColumns(null, "BANK-DB", "account", "%");
    //Arguments: catalog, schema-pattern, table-pattern, column-pattern
    // Returns: 1 row for each column, with several attributes such as
    //   COLUMN_NAME, TYPE_NAME, etc.

  while (rs.next( )) {
      System.out.println(rs.getString("COLUMN_NAME") ,
                          rs.getString("TYPE_NAME");
  }

- There are also functions for getting information such as
  - Foreign key references in the schema
  - Database limits like maximum row size, maximum no. of connections, etc

Application Architectures

- Applications can be built using one of two architectures
  - Two tier model
    - Application program running at user site directly uses JDBC/ODBC to communicate with the database
  - Three tier model
    - Users/programs running at user sites communicate with an application server. The application server in turn communicates with the database
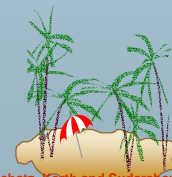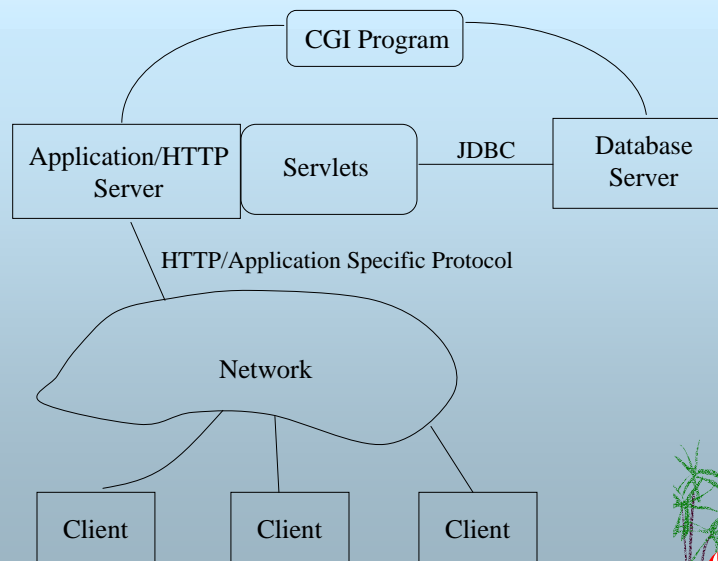
Two-tier Model

- E.g. Java code runs at client site and uses JDBC to communicate with the backend server
- Benefits:
  - flexible, need not be restricted to predefined queries
- Problems:
  - Security: passwords available at client site, all database operation possible
  - More code shipped to client
  - Not appropriate across organizations, or in large ones like universities

Three Tier Model

CGI Program

Application/HTTP Server — Servlets — JDBC — Database Server

HTTP/Application Specific Protocol

Network

Client   Client   Client

Three-tier Model (Cont.)

- E.g. Web client + Java Servlet using JDBC to talk with database server
- Client sends request over http or application-specific protocol
- Application or Web server receives request
- Request handled by CGI program or servlets
- Security handled by application at server
  - Better security
  - Fine granularity security
- Simple client, but only packaged transactions
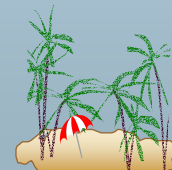
End of Chapter

The *loan* and *borrower* Relations

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	Hayes	L-155

loan   *borrower*

The Result of *loan* inner join *borrower* on *loan.loan-number = borrower.loan-number*

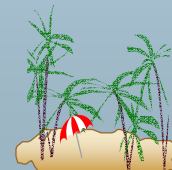| loan-number	branch-name	amount	customer-name	loan-number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

The Result of *loan* left outer join *borrower* on *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*

The Result of *loan* natural inner join *borrower*

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

Join Types and Join Conditions

Join types	Join Conditions
inner join left outer join right outer join full outer join	natural on < predicate> using $(A_1, A_1, \ldots, A_n)$
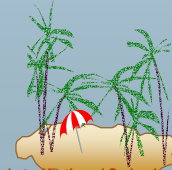
The Result of *loan* natural right outer join *borrower*

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

The Result of *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

SQL Data Definition for Part of the Bank Database

```
create table customer
    (customer-name    char(20),
     customer-street  char(30),
     customer-city    char(30),
     primary key (customer-name))

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

create table account
    (account-number   char(10),
     branch-name      char(15),
     balance          integer,
     primary key (account-number),
     check (balance >= 0))

create table depositor
    (customer-name    char(20),
     account-number   char(10),
     primary key (customer-name, account-number))
```