CHAPTER 3
SQL

Chapter 3: SQL

- Data Definition
- Basic Structure of SQL Queries
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations

Create Table Construct

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

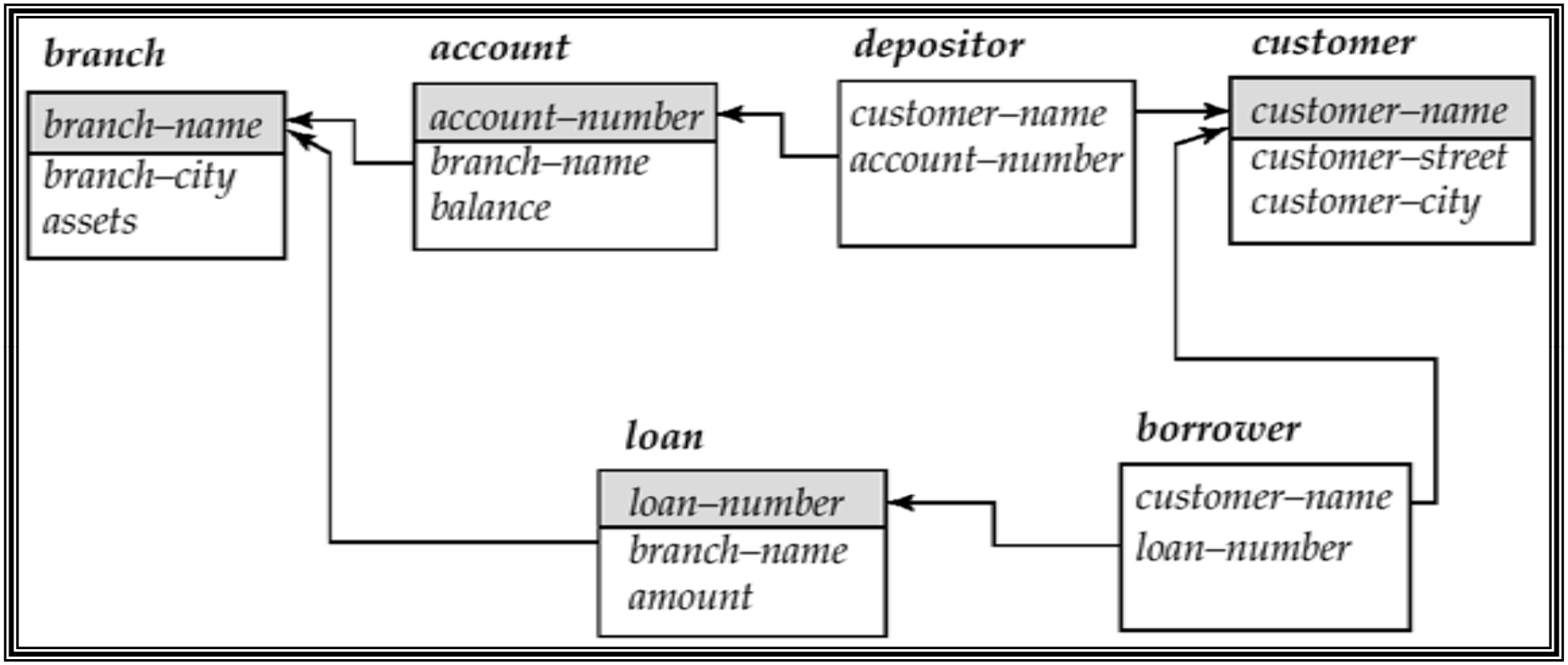
  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - 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)

Schema Used in Examples



Basic Structure of SQL Queries

- 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.

Tuple Variables

- Defined in the **from** clause by use of **as**.
- 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'

- '**as**' is optional

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*

- **desc** for descending order or **asc** for ascending order (default)
  - E.g. **order by** *customer-name* **desc**
  - E.g. **order by** *customer-name* **desc**, loan-number **asc**

Set Operations

- The set operations: **union, intersect, except** correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each set operation automatically eliminates duplicates
- to retain all duplicates use 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 (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)*

Aggregate 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 – *Examples*

- 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

- Tuples may have a null value (*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.

  Find all loan numbers which appear in the *loan* relation with null values for *amount*.

  select *loan-number*
  from *loan*
  where *amount* **is null** (why not '*amount*=**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*
  - *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

- A subquery is a **select-from-where** expression that is nested within another query.
- Common use of subqueries:

  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: This query can be written in a much simpler manner.

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))
```

Note: $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

Views

- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

  (**select** *customer_name, borrower.loan_number, branch_name*
  from *borrower, loan*
  where *borrower.loan_number = loan.loan_number*)

- A ***view*** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

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

  where <query expression> is any legal SQL expression. The view name is represented by *v*.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.

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'

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Update of a View

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

  create view *loan_branch* **as**
  select *loan_number*, *branch_name*
  from *loan*

- Add a new tuple to *branch_loan*

  insert into *branch_loan*
  values ('L-37', 'Perryridge')

  This insertion must be represented by the insertion of the tuple

  ('L-37', 'Perryridge', *null*)

  into the *loan* relation

Updates Through Views (Cont.)

- Some updates through views are impossible to translate into updates on the database relations

  create view *v* **as**
  select *loan_number, branch_name, amount*
  from *loan*
  where *branch_name* = 'Perryridge'

  insert into *v* **values ('**L-99*',*'Downtown*', '23'***)***

- Others cannot be translated uniquely

  insert into *all_customer* **values** *(*'Perryridge', 'John')

  - Have to choose loan or account, and create a new loan/account number!
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Materialized Views (section 14.5)

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  create view *branch_total_loan*(*branch_name*, *total_loan*) **as**
  select *branch_name*, **sum**(*amount*)
  from *loan*
  group by *branch_name*
- Materializing the above view would be very useful if the total loan amount is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - Changes to database relations are used to compute changes to the materialized view, which is then updated
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Above methods are directly supported by many database systems
    - Avoids manual effort/correctness issues

END OF CHAPTER 3