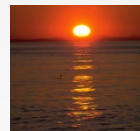




Chapter 3: SQL

Database System Concepts, 5th Ed.

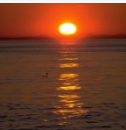
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use





Chapter 3: SQL

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





History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999 (language name became Y2K compliant!)
 - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
 - Not all examples here may work on your particular system.

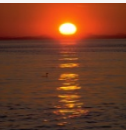




Data Definition Language

Allows the specification of:

- The schema for each relation, including attribute types.
- Integrity constraints
- Authorization information for each relation.
- Non-standard SQL extensions also allow specification of
 - The set of indices to be maintained for each relations.
 - The physical storage structure of each relation on disk.





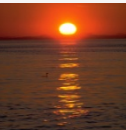
Create Table Construct

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

```
create table  $r$  ( $A_1 D_1, A_2 D_2, \dots, A_n D_n,$   
                (integrity-constraint1),  
                ...,  
                (integrity-constraintk))
```

- 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 attribute A_i
- Example:

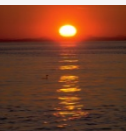
```
create table branch  
  (branch_name char(15),  
   branch_city char(30),  
   assets integer)
```





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.
- More are covered in Chapter 4.





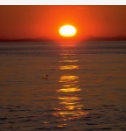
Integrity Constraints on Tables

- **not null**
- **primary key** (A_1, \dots, A_n)

Example: Declare *branch_name* as the primary key for *branch*

```
create table branch
    (branch_name char(15),
     branch_city char(30) not null,
     assets integer,
     primary key (branch_name))
```

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

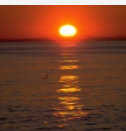




Basic Insertion and Deletion of Tuples

- Newly created table is empty
- Add a new tuple to *account*
insert into *account*
values ('A-9732', 'Perryridge', 1200)
 - Insertion fails if any integrity constraint is violated
- Delete *all* tuples from *account*
delete from *account*

Note: Will see later how to delete selected tuples





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

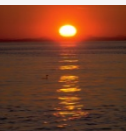
where A is the name of the attribute to be added to relation r and D is the domain of A .

- 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





Basic Query Structure

- A typical SQL query has the form:

select A_1, A_2, \dots, A_n
from r_1, r_2, \dots, r_m
where P

- A_i represents an attribute
 - R_i represents a relation
 - P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\Pi_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation.

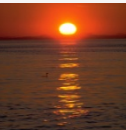




Figure 3.1: Database Schema

branch (*branch_name*, *branch_city*, *assets*)

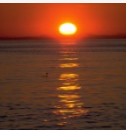
customer (*customer_name*, *customer_street*, *customer_city*)

loan (*loan_number*, *branch_name*, *amount*)

borrower (*customer_name*, *loan_number*)

account (*account_number*, *branch_name*, *balance*)

depositor (*customer_name*, *account_number*)





The select Clause

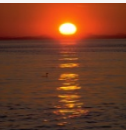
- The **select** clause list the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:

select *branch_name*
from *loan*

- In the relational algebra, the query would be:

$\Pi_{branch_name}(loan)$

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
 - E.g. *Branch_Name* \equiv *BRANCH_NAME* \equiv *branch_name*
 - Some people use upper case wherever we use bold font.





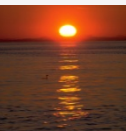
The select Clause (Cont.)

- 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.
- E.g.:

```
select loan_number, branch_name, amount * 100  
from loan
```

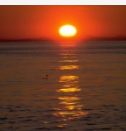




Figure 3.1: Database Schema

branch (*branch_name*, *branch_city*, *assets*)

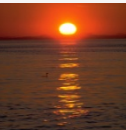
customer (*customer_name*, *customer_street*, *customer_city*)

loan (*loan_number*, *branch_name*, *amount*)

borrower (*customer_name*, *loan_number*)

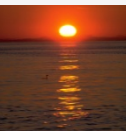
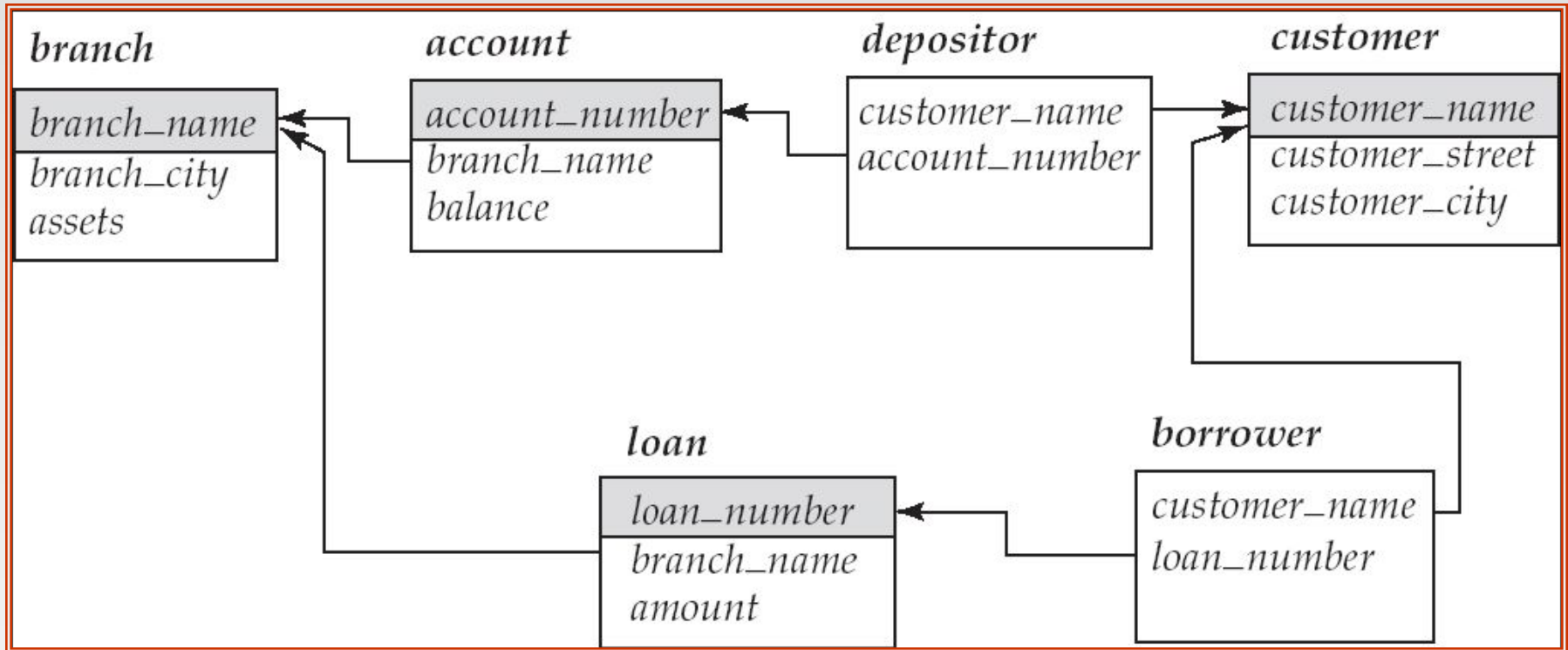
account (*account_number*, *branch_name*, *balance*)

depositor (*customer_name*, *account_number*)





Schema Diagram





The where Clause

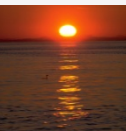
- The **where** clause specifies conditions that the result must satisfy
 - Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than \$1200.

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**.





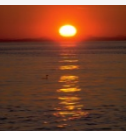
The from Clause

- 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'
```

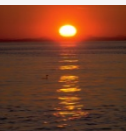




The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:
old-name as new-name
- E.g. 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 and amount 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
```

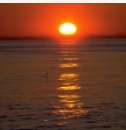
- Find the names of all branches that have greater assets than some branch located in Brooklyn.

```
select distinct T.branch_name
      from branch as T, branch as S
      where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

- Keyword **as** is optional and may be omitted

```
borrower as T  $\equiv$  borrower T
```

- Some database such as Oracle *require* **as** to be omitted



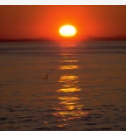


String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that 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”.

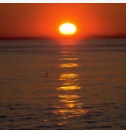
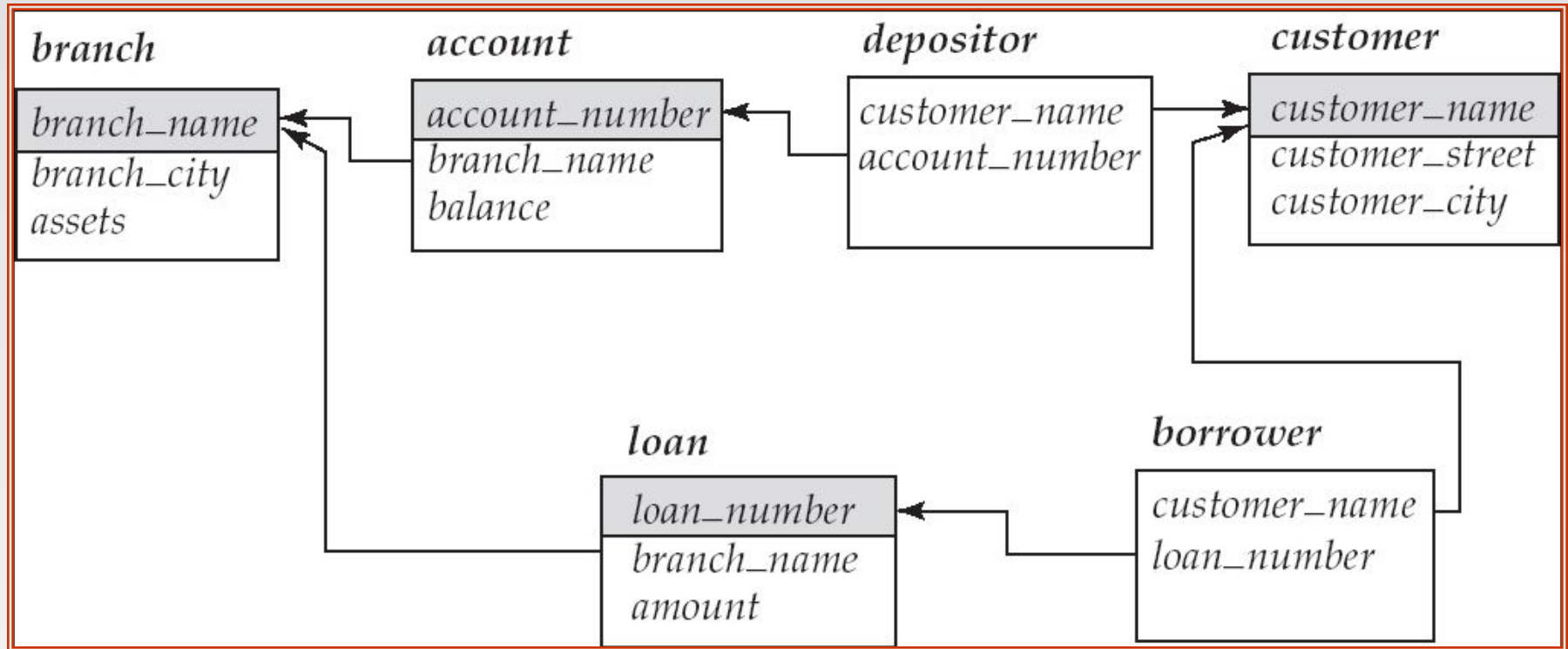
```
select customer_name  
from customer  
where customer_street like '% Main%'
```
- Match the name “Main%”

```
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.





Schema Diagram



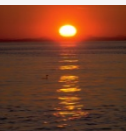


Ordering the Display of Tuples

- 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.
 - Example: **order by** *customer_name* **desc**



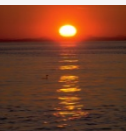


Set Operations

- 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**.

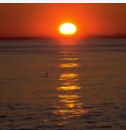
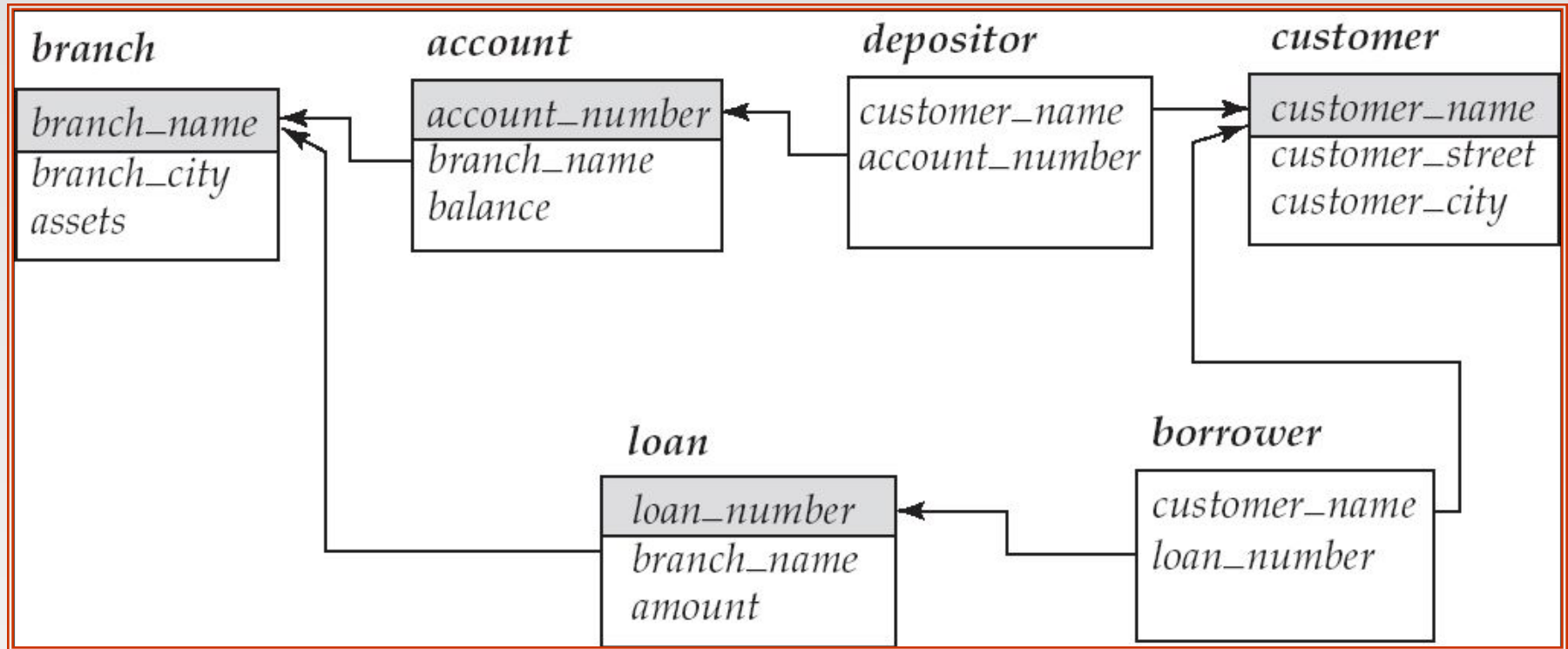
Suppose a tuple occurs m times in r and n times in s , then, it occurs:

- $m + n$ 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





Schema Diagram





Set Operations

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

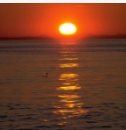
```
(select customer_name from depositor)  
union  
(select customer_name from borrower)
```

- 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.

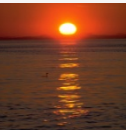
```
(select customer_name from depositor)  
except  
(select customer_name from borrower)
```





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.

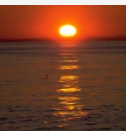
```
select avg (balance)  
  from account  
  where branch_name = 'Perryridge'
```

- 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
```



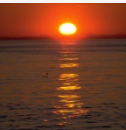


Aggregate Functions – Group By

- Find the number of depositors for each branch.

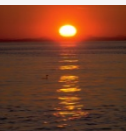
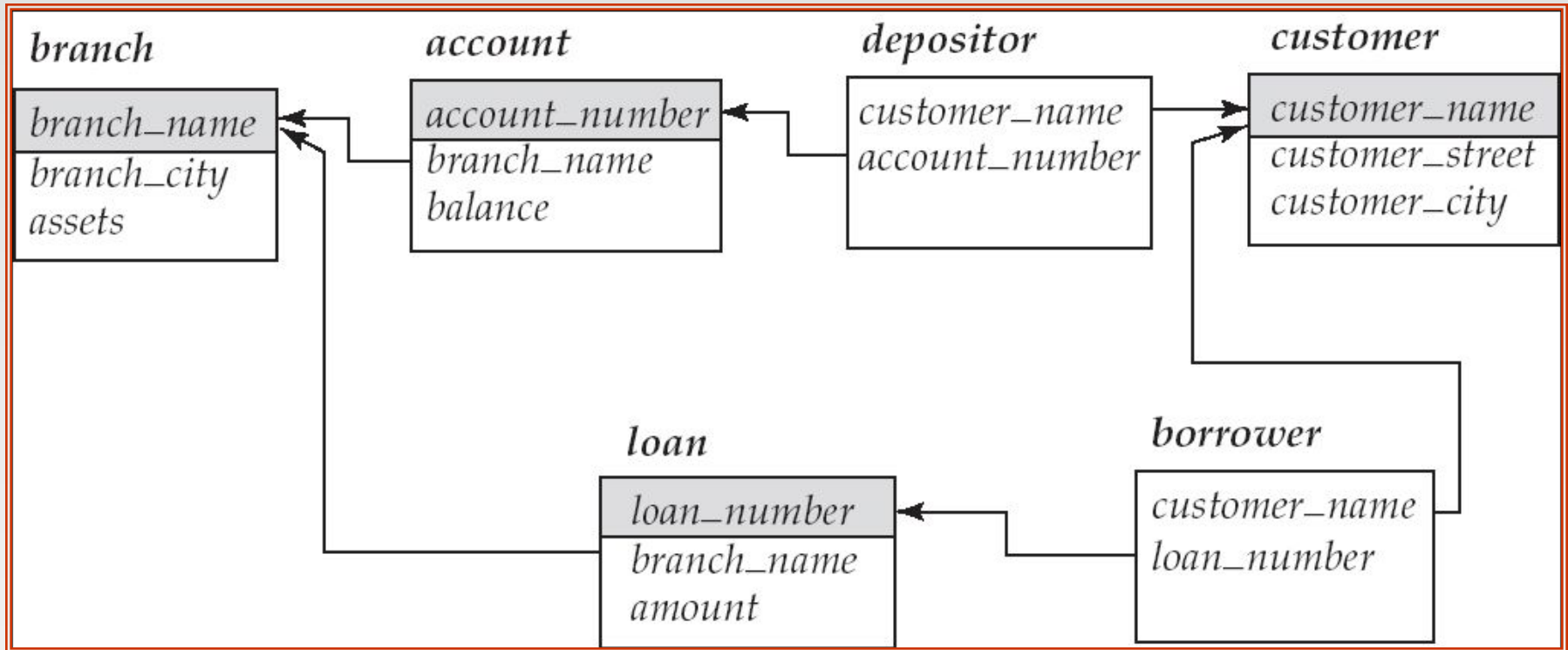
```
select branch_name, count (distinct customer_name)  
  from depositor, account  
 where depositor.account_number = account.account_number  
 group by branch_name
```

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





Schema Diagram



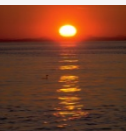


Aggregate Functions – Having Clause

- 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

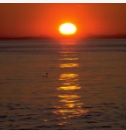




Nested Subqueries

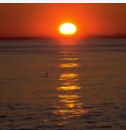
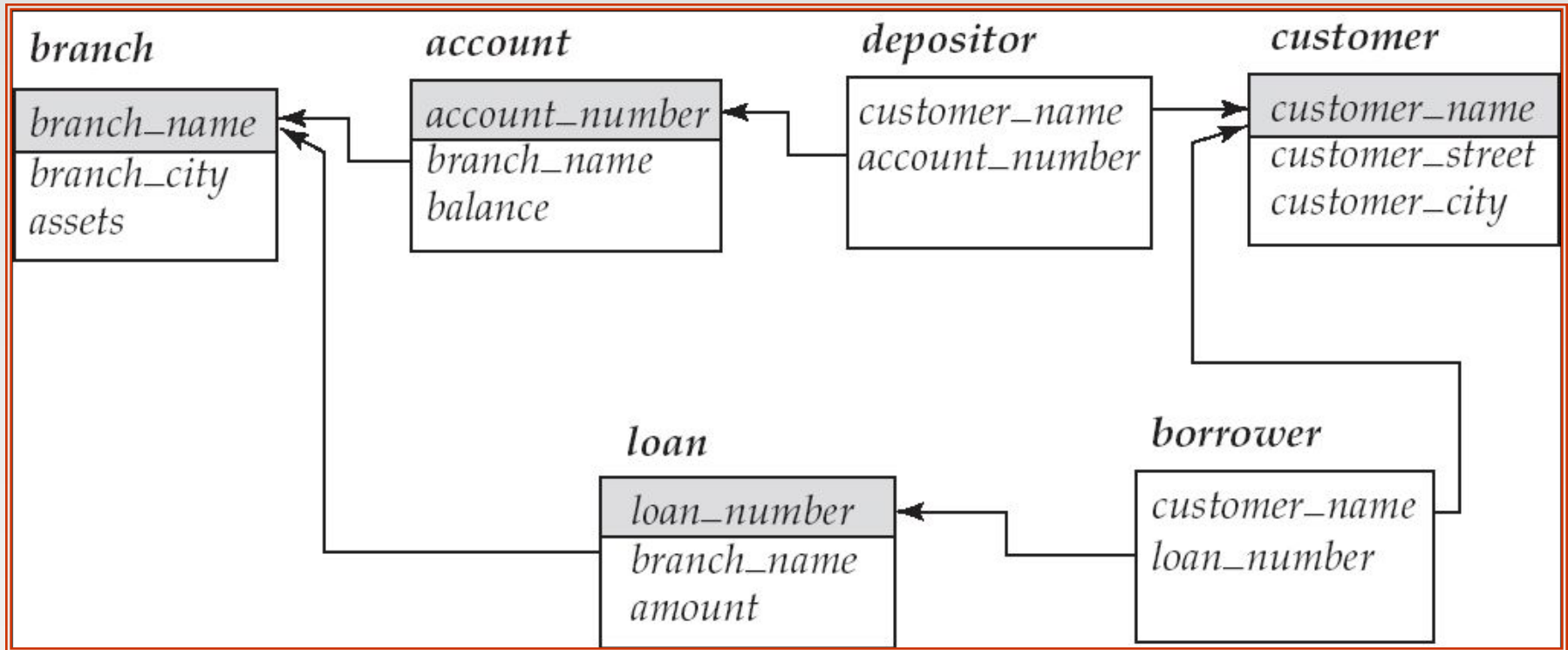
- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Select
from
where operator (Select
 from
 where)





Schema Diagram





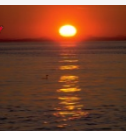
not in

- ```
-select distinct customer_name
from borrower .
where customer_name in (select customer_name
 from depositor)
```

- ```
select distinct customer_name  
from borrower  
where customer_name not in (select customer_name  
                                from depositor)
```

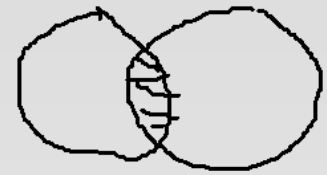
O/P
acCount

A handwritten diagram in red ink. It features a vertical line on the right side. To the left of this line, there are several horizontal lines of varying lengths, some ending in 'x' marks. The label 'a m' is written in the top left corner.





Example Query

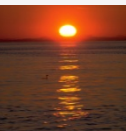


- Find all customers who have both an account and a loan at the Perryridge branch

~~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 =~~
~~(name, branch_name))~~

account depositor
 1)
 2) loan, borrower
 Perryridge

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





Scott schema

- Find the Nth highest salary using the Correlated subquery.

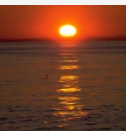
```
SELECT name, sal
```

```
FROM emp e1
```

```
WHERE N-1 = (SELECT COUNT(DISTINCT sal)  
             FROM emp e2
```

```
WHERE e2.sal > e1.sal)
```

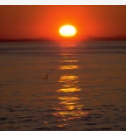
Note: The distinct keyword is there to deal with duplicate salaries in the table.





Example of Third largest salary(N=3)

```
SELECT name, sal
FROM emp e1
WHERE 3-1 = (SELECT COUNT (DISTINCT sal)
FROM emp e2
WHERE e2.sal > e1.sal)
```







SubQuery

Subquery is a SELECT statement that is embedded in a clause of another SELECT statement. i.e. nesting of queries or query within query.

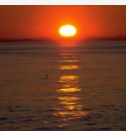
Types of subqueries

1. Single row subqueries
2. Multiple row subqueries
3. Multiple column subqueries

Single row subqueries

A Single row subquery is one that returns one row from the inner SELECT

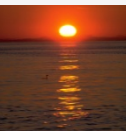
statement. This type of subquery uses a single row operator that are listed below:





Operator

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to





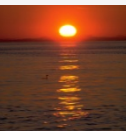
Multiple-row subqueries

Subqueries that return more than one row are called multiple-row subqueries. We use a multiple-row operator instead of a subquery, with a multiple row subquery. These operators are listed below:





Operator	Meaning
IN	Equal to any member in the list.
ANY	Compare value to each value returned by the subquery.
ALL	Compare value to every value returned by the subquery.





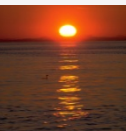
“Some” Construct

- 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')
```

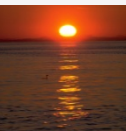




“All” Construct

- 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')
```



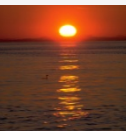


“Exists” Construct

- 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 that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note:* Cannot write this query using = **all** and its variants

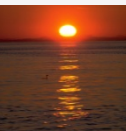




Absence of Duplicate Tuples

- 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')
```



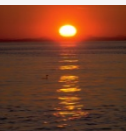


Example Query

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

```
select distinct T.customer_name
from depositor as 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')
```

- Variable from outer level is known as a **correlation variable**





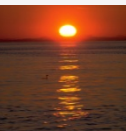
Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

```
delete from account  
where branch_name = 'Perryridge'
```

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

```
delete from account  
where branch_name in (select branch_name  
                        from branch  
                        where branch_city = 'Needham')
```



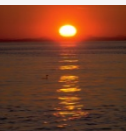


Example Query

- 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

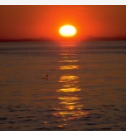
insert into *account* (*branch_name*, *balance*, *account_number*)

values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

insert into *account*

values ('A-777', 'Perryridge', *null*)



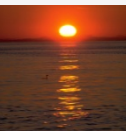


Modification of the Database – Insertion

- 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 evaluated fully before any of its results are inserted into the relation
 - Motivation: **insert into table1 select * from table1**





Modification of the Database – Updates

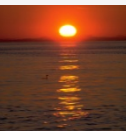
- Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

- 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)

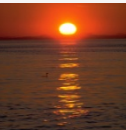




Case Statement for Conditional Updates

- 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
```

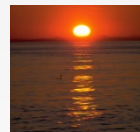




More Features

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

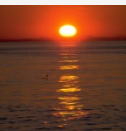




Joined Relations**

- **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.

<i>Join types</i>	<i>Join Conditions</i>
inner join left outer join right outer join full outer join	natural on <predicate> using (A_1, A_1, \dots, A_n)



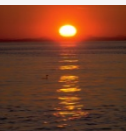


Joined Relations – Datasets for Examples

- Relation *loan*
- Relation *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	Hayes	L-155
<i>loan</i>			<i>borrower</i>	

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





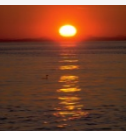
Joined Relations – Examples

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

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
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

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	<i>null</i>	<i>null</i>





Joined Relations – Examples

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

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

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

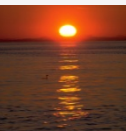
<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	<i>null</i>	<i>null</i>	Hayes

- 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**

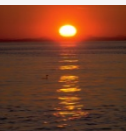




Joined Relations – Examples

- Natural join can get into trouble if two relations have an attribute with same name that should not affect the join condition
 - e.g. an attribute such as *remarks* may be present in many tables
- *Solution:*
 - *loan* **full outer join** *borrower* **using** (*loan_number*)

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes



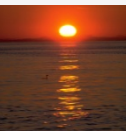


Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- 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 branch_avg ( branch_name, avg_balance )
where avg_balance > 1200
```

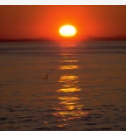
Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.





View Definition

- A relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.
- A view is defined using the **create view** statement which has the form
create view v as \langle query expression \rangle
where \langle query expression \rangle 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.





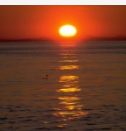
Example Queries

- 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'
```

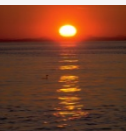




Uses of Views

- Hiding some information from some users
 - Consider a user who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount.
 - Define a view

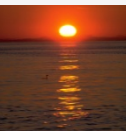
```
(create view cust_loan_data as
select customer_name, borrower.loan_number, branch_name
from borrower, loan
where borrower.loan_number = loan.loan_number )
```
 - Grant the user permission to read *cust_loan_data*, but not *borrower* or *loan*
- Predefined queries to make writing of other queries easier
 - Common example: Aggregate queries used for statistical analysis of data





Processing of Views

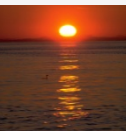
- When a view is created
 - the query expression is stored in the database along with the view name
 - the expression is substituted into any query using the view
- Views definitions containing views
 - One view may be used in the expression defining another view
 - A view relation v_1 is said to *depend 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

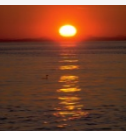




With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- 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
```



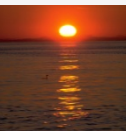


Complex Queries using With Clause

- 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
```

- Note: the exact syntax supported by your database may vary slightly.
 - E.g. Oracle syntax is of the form
with *branch_total* **as** (**select** ..),
 branch_total_avg **as** (**select** ..)
select ...





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
```

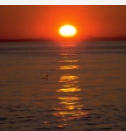
- Add a new tuple to *loan_branch*

```
insert into loan_branch  
  values ('L-37', 'Perryridge')
```

This insertion must be represented by the insertion of the tuple

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

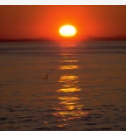
into the *loan* relation





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')
 - 4 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



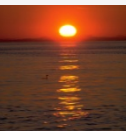


Null Values

- 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.
 - Example: Find all loan number which appear in the *loan* relation with null values for *amount*.

```
select loan_number  
from loan  
where amount is null
```

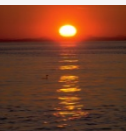
- The result of any arithmetic expression involving *null* is *null*
 - Example: $5 + \text{null}$ returns null
- However, aggregate functions simply ignore nulls
 - More on next slide





Null Values and Three Valued Logic

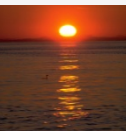
- Any comparison with *null* returns *unknown*
 - Example: $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$
- Three-valued logic using the truth value *unknown*:
 - OR: $(\text{unknown} \text{ or } \text{true}) = \text{true}$,
 $(\text{unknown} \text{ or } \text{false}) = \text{unknown}$
 $(\text{unknown} \text{ or } \text{unknown}) = \text{unknown}$
 - AND: $(\text{true} \text{ and } \text{unknown}) = \text{unknown}$,
 $(\text{false} \text{ and } \text{unknown}) = \text{false}$,
 $(\text{unknown} \text{ and } \text{unknown}) = \text{unknown}$
 - NOT: $(\text{not unknown}) = \text{unknown}$
 - “*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.

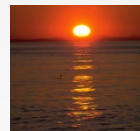




End of Chapter 3

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use





The where Clause (Cont.)

- SQL includes a **between** comparison operator
- Example: Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, \geq \$90,000 and \leq \$100,000)

```
select loan_number  
from loan  
where amount between 90000 and 100000
```

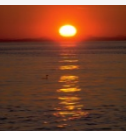




Figure 3.1: Database Schema

branch (*branch_name*, *branch_city*, *assets*)

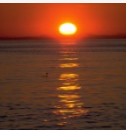
customer (*customer_name*, *customer_street*, *customer_city*)

loan (*loan_number*, *branch_name*, *amount*)

borrower (*customer_name*, *loan_number*)

account (*account_number*, *branch_name*, *balance*)

depositor (*customer_name*, *account_number*)



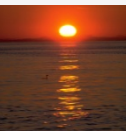


Definition of Some Clause

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

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

- $(= \text{some}) \equiv \text{in}$
- However, $(\neq \text{some})$ is not equivalent to **not in**





Definition of all Clause

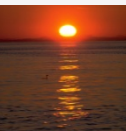
$(5 < \text{all } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$

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

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

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

- $(\neq \text{all}) \equiv \text{not in}$
- However, $(= \text{all})$ is not equivalent to **in**





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** $r \Leftrightarrow r = \emptyset$

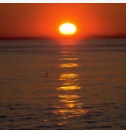




Figure 3.3: Tuples inserted into *loan* and *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-11	Round Hill	900	Adams	L-16
L-14	Downtown	1500	Curry	L-93
L-15	Perryridge	1500	Hayes	L-15
L-16	Perryridge	1300	Jackson	L-14
L-17	Downtown	1000	Jones	L-17
L-23	Redwood	2000	Smith	L-11
L-93	Mianus	500	Smith	L-23
<i>null</i>	<i>null</i>	1900	Williams	L-17
<i>loan</i>			Johnson	<i>null</i>
			<i>borrower</i>	

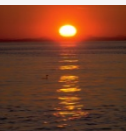




Figure 3.4: The *loan* and *borrower* relations

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	Hayes	L-155
<i>loan</i>			<i>borrower</i>	

