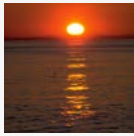
UNIT -9
Chapter 3: SQL

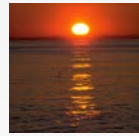
Database System Concepts, 5th Ed.

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



Chapter 3: SQL

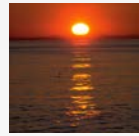
- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**



©Silberschatz, Korth and Sudarshan

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.

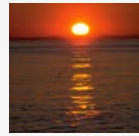


 ©Silberschatz, Korth and Sudarshan

Data Definition Language

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

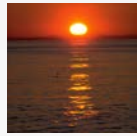
- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.



 ©Silberschatz, Korth and Sudarshan

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.



 ©Silberschatz, Korth and Sudarshan

Create Table Construct

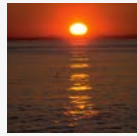
- An SQL relation is defined using the **create table** command:

  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)



©Silberschatz, Korth and Sudarshan

Integrity Constraints in Create Table

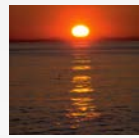
- **not null**
- **primary key** $(A_1, ..., A_n)$

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

.

create table *branch*
(*branch_name* char(15)**,**
branch_city char(30),
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



©Silberschatz, Korth and Sudarshan

Drop and Alter Table Constructs

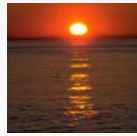
- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation:

  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

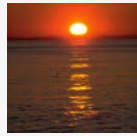
- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

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

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

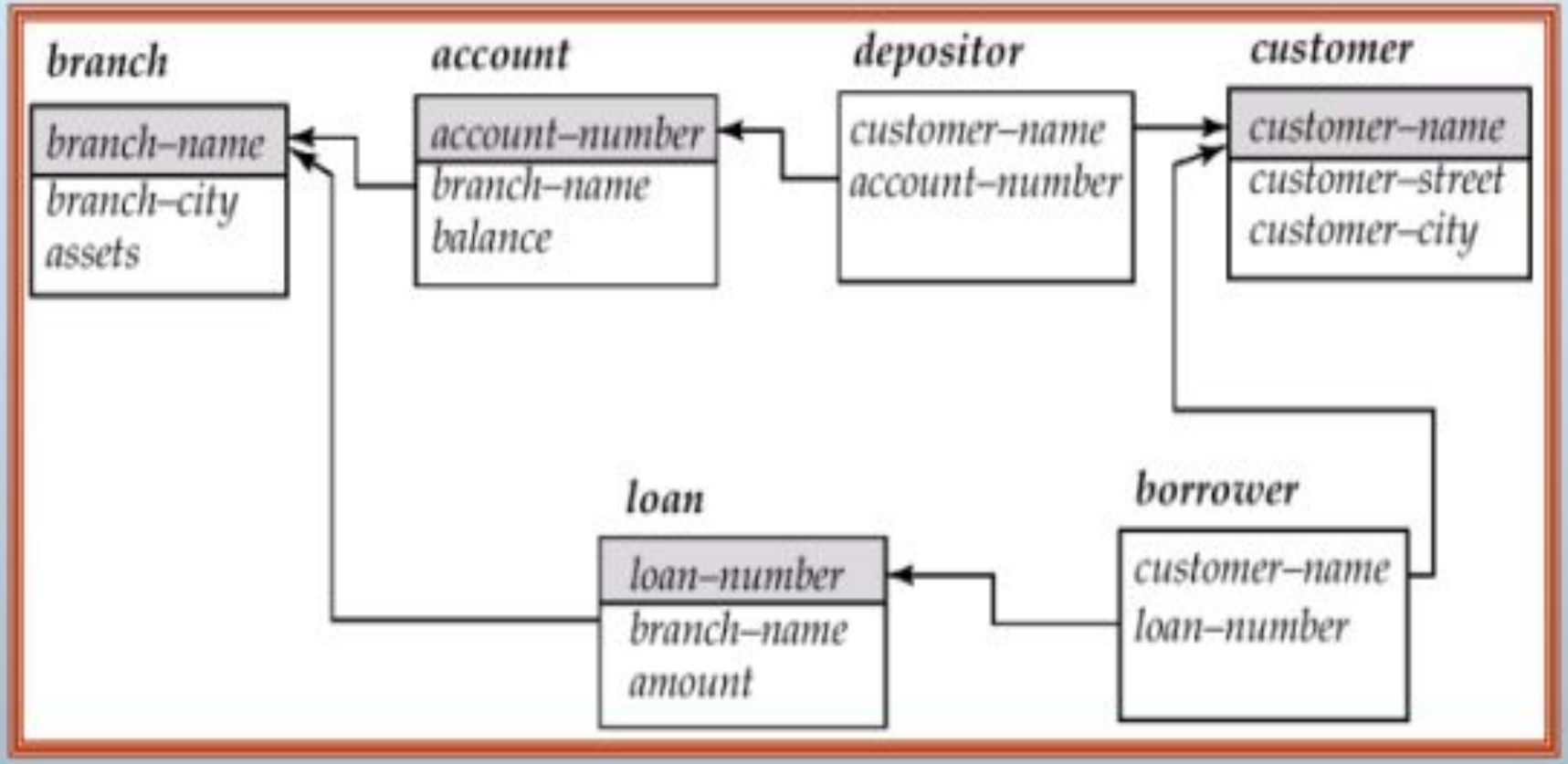
$$\prod_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.



©Silberschatz, Korth and Sudarshan

Schema Used in Examples



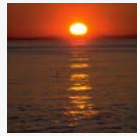
 ©Silberschatz, Korth and Sudarshan

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* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.



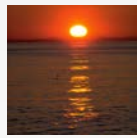
The select Clause (Cont.)

- 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*



©Silberschatz, Korth and Sudarshan

The select Clause (Cont.)

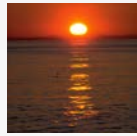
- An asterisk in the select clause denotes "all attributes"

  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 that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100.

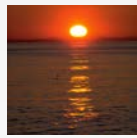


©Silberschatz, Korth and Sudarshan

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.

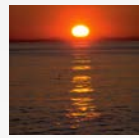


©Silberschatz, Korth and Sudarshan

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, ≥ \$90,000 and ≤ \$100,000)

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



©Silberschatz, Korth and Sudarshan

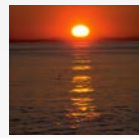
The from Clause

- 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'
```



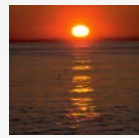
The Rename Operation

- 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*



 ©Silberschatz, Korth and Sudarshan

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.

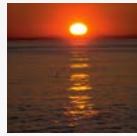
select *customer_name, T.loan_number, S.amount*
from *borrower* **as** *T, loan* **as** *S*
where *T.loan_number = S.loan_number*

- 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'

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

borrower **as** *T* ≡ *borrower T*

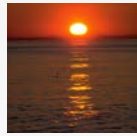


String Operations

- 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.

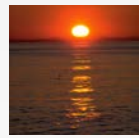


Ordering the Display of Tuples

- 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*

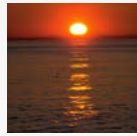
- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *customer_name* **desc**



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



 ©Silberschatz, Korth and Sudarshan

Set Operations

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

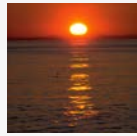
  (select *customer_name* **from** *depositor*)
  union
  (select *customer_name* **from** *borrower)*

- 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)*

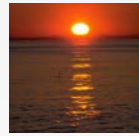


©Silberschatz, Korth and Sudarshan

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



©Silberschatz, Korth and Sudarshan

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

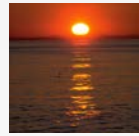
  select avg *(balance)*
  from *account*
  where *branch_name* = 'Perryridge'

- 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*



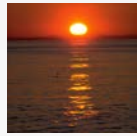
©Silberschatz, Korth and Sudarshan

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



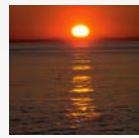
©Silberschatz, Korth and Sudarshan

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

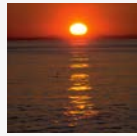


Null Values

- 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.
  - 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 + *null* returns null
- However, aggregate functions simply ignore nulls
  - More on next slide



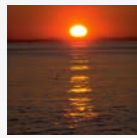
©Silberschatz, Korth and Sudarshan

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.



©Silberschatz, Korth and Sudarshan

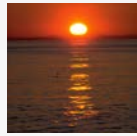
Set Comparison

- 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')
```

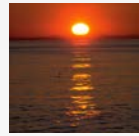


©Silberschatz, Korth and Sudarshan

Example Query

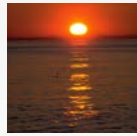
- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
    from branch
    where assets > all
        (select assets
        from branch
        where branch_city = 'Brooklyn')
```



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.



 ©Silberschatz, Korth and Sudarshan

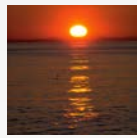
Example Query

- 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 )
```



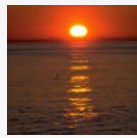
 ©Silberschatz, Korth and Sudarshan

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.

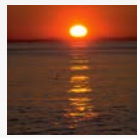


Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- 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**.

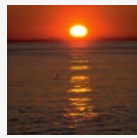


View Definition

- 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.



©Silberschatz, Korth and Sudarshan

Example Queries

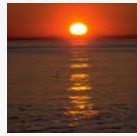
- A view consisting of branches and their customers

```
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

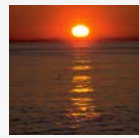
```
Create view test as

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.



©Silberschatz, Korth and Sudarshan

View Expansion

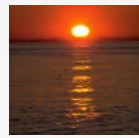
- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  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



©Silberschatz, Korth and Sudarshan

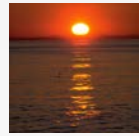
Modification of the Database – Deletion

- 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')

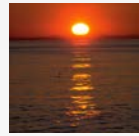


Example Query

- 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)



©Silberschatz, Korth and Sudarshan

Modification of the Database – Insertion

- Add a new tuple to *account*

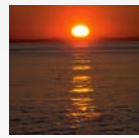
  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  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*)

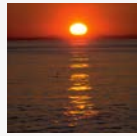


Modification of the Database – Insertion

- 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 (otherwise queries like
  insert into *table*1 **select * from** *table*1
  would cause problems)

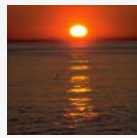


Modification of the Database – Updates

- 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)

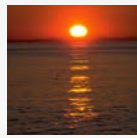


©Silberschatz, Korth and Sudarshan

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 the *loan* relation, hiding the *amount* attribute

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

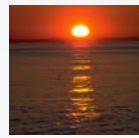
- Add a new tuple to *branch_loan*

  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



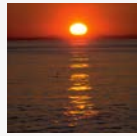
©Silberschatz, Korth and Sudarshan

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



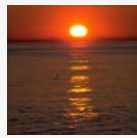
©Silberschatz, Korth and Sudarshan

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
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$



Joined Relations – Datasets for Examples

- Relation *loan*
- Relation *borrower*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

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

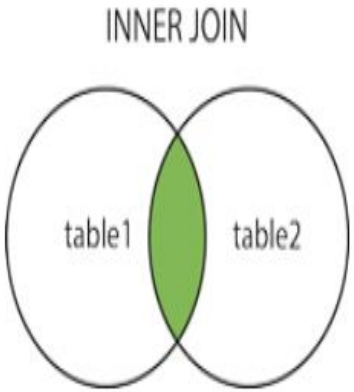
©Silberschatz, Korth and Sudarshan

SQL INNER JOIN Keyword

The `INNER JOIN` keyword selects records that have matching values in both tables.

INNER JOIN Syntax

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```



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*

©Silberschatz, Korth and Sudarshan

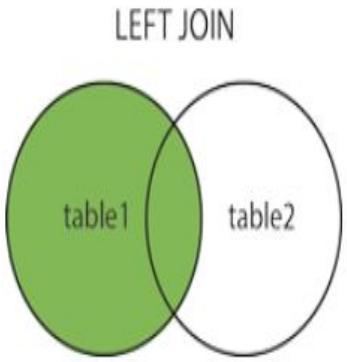
SQL LEFT JOIN Keyword

The `LEFT JOIN` keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Note: In some databases LEFT JOIN is called LEFT OUTER JOIN.



Joined Relations – Examples

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

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *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

©Silberschatz, Korth and Sudarshan

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**

©Silberschatz, Korth and Sudarshan

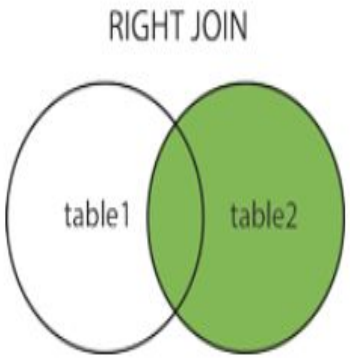
SQL RIGHT JOIN Keyword

The `RIGHT JOIN` keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

RIGHT JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Note: In some databases `RIGHT JOIN` is called `RIGHT OUTER JOIN`.



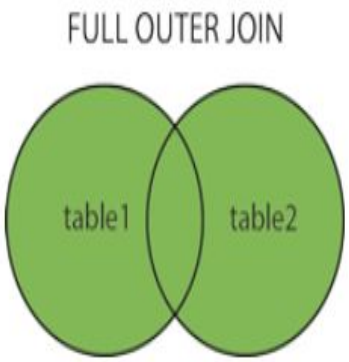
SQL FULL OUTER JOIN Keyword

The `FULL OUTER JOIN` keyword returns all records when there is a match in left (table1) or right (table2) table records.

Tip: `FULL OUTER JOIN` and `FULL JOIN` are the same.

FULL OUTER JOIN Syntax

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

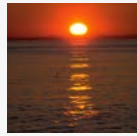


Difference between Natural Join and Inner Join

SN	Natural Join	Inner Join
1.	It joins the tables based on the same column names and their data types.	It joins the tables based on the column name specified in the ON clause explicitly.
2.	It always returns unique columns in the result set.	It returns all the attributes of both tables along with duplicate columns that match the ON clause condition.
3.	If we have not specified any condition in this join, it returns the records based on the common columns.	It returns only those rows that exist in both tables.
4.	The syntax of natural join is given below: SELECT [column_names | *] FROM table_name1 NATURAL JOIN table_name2;	The syntax of inner join is given below: SELECT [column_names | *] FROM table_name1 INNER JOIN table_name2 ON table_name1.column_name = table_name2.column_name;

DCL

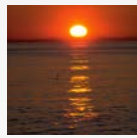
- DCL is the abstract of Data Control Language.
- Data Control Language includes commands such as **GRANT**, and is concerned with rights, permissions, and other controls of the database system.
- DCL is used to grant/revoke permissions on databases and their contents.



©Silberschatz, Korth and Sudarshan

GRANT

- It provides the user's access privileges to the database.
- The MySQL database offers both the administrator and user a great extent of the control options.
- The administration side of the process includes the possibility for the administrators to control certain user privileges over the MySQL server by restricting their access to an entire database or usage limiting permissions for a specific table.
- It creates an entry in the security system that allows a user in the current database to work with data in the current database or execute specific statements.



Syntax :

Statement permissions:

GRANT { ALL | statement [,...n] }

TO security_account [,...n]

Normally, a database administrator first uses CREATE USER to create an account, then GRANT to define its privileges and characteristics.

For example:

```
01. CREATE USER vatsa@'localhost' IDENTIFIED BY 'mypass';
02. GRANT ALL ON MY_TABLE TO vatsa@'localhost';
03. GRANT SELECT ON Users TO vatsa@'localhost';
```

REVOKE

The REVOKE statement enables system administrators and to revoke (back permission) the privileges from MySQL accounts.

Syntax:

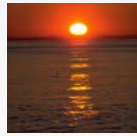
REVOKE
priv_type [(column_list)]
[, priv_type [(column_list)]] ...
ON [object_type] priv_level
FROM user [, user] ...
REVOKE ALL PRIVILEGES, GRANT OPTION
FROM user [, user] ...

For example:

```
01. REVOKE INSERT ON *.* FROM 'vatsa'@'localhost';
```

Transaction commands

- Commit
  - A COMMIT means that the changes made in the current transaction are made permanent and become visible to other sessions.
- Rollback
  - A ROLLBACK statement, on the other hand, **cancels all modifications made by the current transaction**.



©Silberschatz, Korth and Sudarshan

Revision

- Create - database,table,view
- Drop- database,table,view
- Alter table
  - Modify
  - Add
  - Drop
  - Rename
  - Rename table
  - Aliasing
  - Tuple variable
  - IN and not IN
  - Select (distinct,*,attributes,all)
  - From
  - Where
  - Like
  - Groupby
- Orderby clause –asc/desc
- Having
- Union/intesect./except
- Null value
- Join
- Subquries
- Aggigation(avg,sum,min,max,count)
- Natural join
- Inner join
- Left join
- Right join
- Full outer join
- DCL (grant /revock) concept
- Transaction command(commit/rollback) concept

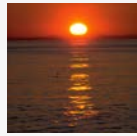


Figure 3.1: Database Schema

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

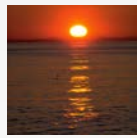
customer (*customer_name*, *customer_street*, *customer_city*)

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

borrower (*customer_name*, *loan_number*)

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

depositor (*customer_name*, *account_number*)



 ©Silberschatz, Korth and Sudarshan

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

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

loan

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

borrower

©Silberschatz, Korth and Sudarshan

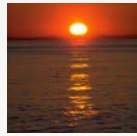
Figure 3.4: The *loan* and *borrower* relations

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower



©Silberschatz, Korth and Sudarshan