# Structured Query Language

# Select Statements Revisited

- select clause ---------- attribute selection part
- from clause ---------- relation selection part
- where clause ---------- join, selection conditions part
- group by clause ---------- partition part
- having clause ---------- partition filtering part
- order by clause ---------- ordering rows part

+

Aggregates, set operations, subqueries

# Banking Example

- branch (branch-name, branch-city, assets)
- customer (customer-name, customer-street, customer-only)
- account (account-number, branch-name, balance)
- loan (loan-number, branch-name, amount)
- depositor (customer-name, account-number)
- borrower (customer-name, loan-number)

# Branch Table

| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

# Customer Table

| customer-name | customer-street | customer-only |
|---|---|---|
| Joe | Joe_street | Y |
| Alan | Mary_street | Y |
| Jason | Jason_street | N |
| Mary | Mary_street | N |
| Mike | Mary_street | Y |
| Keith | Keith_street | N |

# Account Table

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

# Loan Table

| loan-number | branch-name | Amount |
|---|---|---|
| 1 | B | $100 |
| 2 | E | $27 |
| 3 | F | $543 |
| 4 | A | $129 |
| 5 | A | $26 |
| 6 | B | $67 |

# Depositor Table

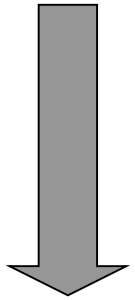
| customer-name | account-number |
|---|---|
| Joe | 1 |
| Joe | 2 |
| Mary | 2 |
| Keith | 4 |
| Mike | 5 |
| Keith | 6 |
| Joe | 3 |

# Borrower Table

| customer-name | loan-number |
|---|---|
| Joe | 1 |
| Jason | 2 |
| Joe | 3 |
| Keith | 4 |
| Mary | 5 |
| Joe | 6 |

# Subquery predicates revisited ---- IN…Example 1

"*Find the account numbers opened at branches of the bank in Riverside*"



```
SELECT account-number
FROM account
WHERE branch_name IN (
        SELECT  branch-name
        FROM    branch
        WHERE   branch-city='Riverside')
```

# Example 1 - Result

Account Table

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

Branch Table

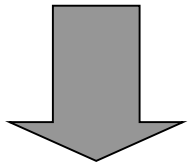
| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

```
SELECT account-number
FROM account
WHERE branch_name IN
        (SELECT branch-name
          FROM branch
          WHERE branch-city='Riverside')
```

| account-number |
|---|
| 2 |
| 3 |
| 5 |

# Subquery predicates revisited ---- IN…Example 2

"*Find the account numbers opened at A and B branches of the bank*"



**SELECT account-number**

**FROM account**

**WHERE branch_name IN ('A', 'B')**

# Example 2 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

**SELECT account-number**

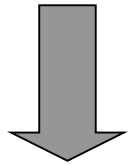
**FROM account**

**WHERE branch_name IN (‘A’, ‘B’)**

| account-number |
|---|
| 1 |
| 2 |
| 3 |
| 5 |
| 6 |

# Subquery predicates revisited ---- IN…Example 3

"*Find the account numbers opened at branches of the bank in Riverside*"



**SELECT account-number**

**FROM account S**

**WHERE 'Riverside' IN (**

**SELECT branch-city**

**FROM branch T**

**WHERE S.branch-name=T.branch-name)**

***correlated subquery***

# Example 3 - Result

Account Table

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

Branch Table

| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

```
SELECT account-number
FROM account S
WHERE 'Riverside' IN (
        SELECT branch-city
        FROM branch T
        WHERE S.branch-name =
              T.branch-name)
```

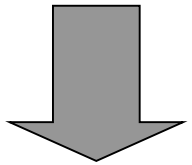
| account-number |
|---|
| 2 |
| 3 |
| 5 |

# Subquery predicates revisited ---- EXISTS

- The *EXISTS* predicate is TRUE if and only if the Subquery returns a non-empty set.
- The *NOT EXISTS* predicate is TRUE if and only if the Subquery returns an empty set.
- The *NOT EXISTS* can be used to implement the MINUS operator from relational algebra.

# Subquery predicates revisited ---- EXISTS…Example 1

"*Select all the account balances where the account has been opened in a branch in Riverside* "



```
SELECT account-balance
FROM account S
WHERE EXISTS (  SELECT *
                FROM branch T
                WHERE T.branch-city = 'Riverside'
                and T.branch_name = S.branch-name)
```

# Example 1 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

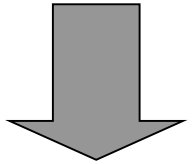
| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

```
SELECT account-balance
FROM account S
WHERE EXISTS (SELECT *
              FROM branch T
              WHERE T.branch-city =
              'Riverside' and
               T.branch_name =
               S.branch-name)
```

| account-number |
|---|
| $50 |
| $30 |
| $500 |

# Subquery predicates ---- EXISTS…Example 2

*"Select all the account balances where the account has not been opened in a Riverside branch "*



```
SELECT account-balance
FROM account S
WHERE NOT EXISTS (SELECT *
                  FROM branch T
                  WHERE T.branch-city='Riverside'
                  and T.branch_name=S.branch-name)
```

# Example 2 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

```
SELECT account-balance
FROM account S
WHERE NOT EXISTS (SELECT *
                  FROM branch T
                  WHERE T.branch-city = 'Riverside'
                     AND
                  T.branch_name = S.branch-name)
```

| account-number |
|---|
| $100 |
| $120 |
| $324 |

# Subquery predicates
# Quantified Comparison Predicate

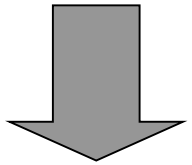
- A *quantified predicate* compares a simple value of an expression with the result of a *Subquery*.

- Given a comparison operation θ, representing some operator in the set {<, ≤, =, <>, >, ≥}, the equivalent predicates '*expr θ SOME (Subquery)*' *and 'expr θ ANY (Subquery)*' are TRUE if and only if, for at least one element *s* returned by the *Subquery*, it is true that '*expr θ s*' is TRUE.

- the predicate '*expr θ ALL (Subquery)*' is TRUE if and only if '*expr θ s*' is TRUE for every one of the elements *s* of the *Subquery*;

# Subquery predicates
# Quantified Comparison Predicate Example 1

"*Select account numbers of the accounts with the minimum balance*"



```
SELECT account-balance
FROM account
WHERE balance <= ALL (  SELECT balance
                        FROM account)
```

# Example 1 - Result

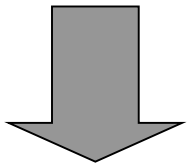
| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

**SELECT account-balance**
**FROM account**
**WHERE balance <= ALL (SELECT balance**
**FROM account)**

| account-number |
|---|
| 3 |

# Subquery predicates
# Quantified Comparison Predicate Example 2a

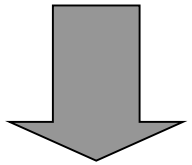
"*Select account balances for accounts located in Riverside*"



```
SELECT account-balance
FROM account
WHERE branch-name = ANY (SELECT T.branch-name
                         FROM branch T
                         WHERE T.branch-city='Riverside')
```

# Subquery predicates
# Quantified Comparison Predicate Example 2b

"*Select account balances for accounts located in Riverside*"



```
SELECT account-balance
FROM account
WHERE branch-name = SOME (SELECT T.branch-name
                          FROM branch T
                          WHERE T.branch-city='Riverside')
```

# Example 2 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

```
SELECT account-balance
FROM account
WHERE branch-name = SOME (SELECT T.branch-name
                          FROM branch T
                          WHERE T.branch-city ='Riverside')
```

| account-number |
|---|
| $50 |
| $30 |
| $500 |

# Set Functions in SQL…revisited

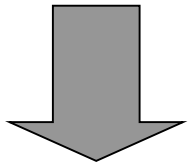
- SQL provides five built-in aggregate functions that operate on sets of column values in tables:

- *COUNT( ), MAX( ), MIN( ), SUM( ), AVG( )*.

- With the exception of *COUNT( )*, these set functions must operate on sets that consist of simple values-that is, sets of numbers or sets of character strings, rather than sets of rows with multiple values.
- NULL values are not counted.

# Set Functions in SQL … Example 1

"*Select the total amount of balance of the account in branches located in Riverside*"



```
SELECT sum(balance) AS total_amount
FROM account S, branch T
WHERE T.branch-city='Riverside'
        and T.branch_name= S.branch_name
```

# Example 1 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

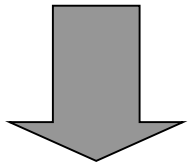
| branch-name | branch-city | assets |
|---|---|---|
| A | Riverside | $10,000 |
| B | LA | $20,000 |
| C | Long Beach | $15,000 |
| D | Irvine | $12,000 |
| E | Pomona | $7,000 |
| F | San Jose | $18,000 |

```
SELECT sum(balance) AS total_amount
FROM account S, branch T
WHERE T.branch-city='Riverside'
        and T.branch-name= S.branch-name
```

| total_amount |
|---|
| $580 |

# Set Functions in SQL
# Example 2

"*Select the total number of opened accounts*"



**SELECT count(account-number) FROM account**

***OR***

**SELECT count(*) FROM account**

# Example 2 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

**SELECT count(account-number) FROM account**

***OR***

**SELECT count(*) FROM account**

| count(*) |
|---|
| 6 |

# Groups of Rows in SQL (1)

- SQL allows Select statements to provide a kind of natural ‘report’ function, grouping the rows on the basis of commonality of values and performing set functions on the rows grouped:
- *SELECT branch_name, SUM(balance) FROM account GROUP BY branch_name*.
- The GROUP BY clause of the Select statement will result in the set of rows being generated as if the following loop-controlled query were being performed:

```
FOR EACH DISTINCT VALUE v OF branch_name IN account
        SELECT branch_name, SUM(balance) FROM account
        WHERE branch_name=v
END FOR
```

# Groups of Rows in SQL (2)

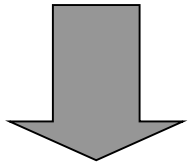
- A set of functions occurring in the SELECT list aggregates for the set of rows in each group and thus creates a single value for each group.
- It is important that all of the attributes named in the select list have a single atomic value, for each group of common GROUP BY values:

SELECT account-number, branch-name, SUM(balance)

GROUP BY account-number ---------- INVALID

# Groups of Rows in SQL –Example 1

"*Find the total amount of money owed by each depositor*"



**SELECT c.customer-name, SUM(balance)**

**FROM account S, customer C, depositor D**

**WHERE S.account-number = D.account-number and**

**C.customer-name = D.customer-name**

**GROUP BY customer-name**

# Example 1 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

| customer-nam | customer-name | account-number |
|---|---|---|
| Joe | Joe_street | 1 |
| Alan | Mary_street | 2 |
| Jason | Jason_street | 4 |
| Mary | Mary_street | 5 |
| Mike | Mary_street | 6 |
| Keith | Keith_street | 3 |
| | Joe | 1 |
| | Joe | 2 |
| | Mary | 2 |
| | Keith | 4 |
| | Mike | 5 |
| | Keith | 6 |
| | Joe | 3 |

**SELECT c.customer-name, SUM(balance)**
**FROM account S, customer C, depositor D**
**WHERE S.account-number = D.account-number**
**and**
**C.customer-name = D.customer-name**
**GROUP BY customer-name**

| customer-name | sum(balance) |
|---|---|
| Joe | $180 |
| Mary | $50 |
| Keith | $444 |
| Mike | $500 |

# Filter grouping

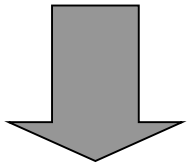
- To eliminate rows from the result of a select statement where a *GROUP BY* clause appears we use the *HAVING* clause, which is evaluated after the *GROUP BY*.
- For example, the query:

  *SELECT account-branch, SUM(balance) FROM account GROUP BY account-branch HAVING SUM(balance)>1000*.

  will print the account branches and total balances for every branch where the total account balance exceeds 1000.
- The *HAVING* clause can only apply tests to values that are single-valued for groups in the SELECT statement.
- The *HAVING* clause can have a nested subquery, just like the *WHERE* clause

# Filter Grouping –Example 1

"*Find the total amount of money owed by each depositor, for each depositor that own at least 2 accounts*"



**SELECT C.customer-name, SUM(balance)**

**FROM account S, customer C, depositor D**

**WHERE S.account-number = D.account-number and**

**C.customer-name = D.customer-name**

**GROUP BY customer-name**

**HAVING COUNT(*) > 1**

# Example 1 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

| customer-name | customer-street | customer-only |
|---|---|---|
| Joe | Joe_street | Y |
| Alan | Mary_street | Y |
| Jason | Jason_street | N |
| Mary | Mary_street | N |
| Mike | Mary_street | Y |
| Keith | Keith_street | N |

| customer-name | account-number |
|---|---|
| Joe | 1 |
| Joe | 2 |
| Mary | 2 |
| Keith | 4 |
| Mike | 5 |
| Keith | 6 |
| Joe | 3 |

**SELECT C.customer-name, SUM(balance)**
**FROM account S, customer C, depositor D**
**WHERE S.account-number = D.account-number and**
**C.customer-name = D.customer-name**
**GROUP BY customer-name**
**HAVING COUNT(*) > 1**

| customer-name | sum(balance) |
|---|---|
| Joe | $180 |
| Keith | $444 |

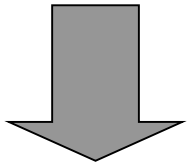
# Order Results

- We use the *ORDER BY* clause when we want the output to be presented in a particular order.

- We provide the list of attributes to order on.

# Order –Example 1

"*Find the total amount of money owed by each depositor, for each depositor that own at least 2 accounts, present the results in descending order of total balance*"



```
SELECT C.customer-name, SUM(balance) AS sbalance
From account S, customer C, depositor D
WHERE S.account-number = D.account-number and
            C.customer-name = D.customer-name
GROUP BY customer-name
HAVING COUNT(*) > 1
ORDER BY Desc sbalance
```

# Example 1 - Result

| account_number | branch-name | balance |
|---|---|---|
| 1 | B | $100 |
| 2 | A | $50 |
| 3 | A | $30 |
| 4 | F | $120 |
| 5 | A | $500 |
| 6 | B | $324 |

| customer-name | customer-street | customer-only |
|---|---|---|
| Joe | Joe_street | Y |
| Alan | Mary_street | Y |
| Jason | Jason_street | N |
| Mary | Mary_street | N |
| Mike | Mary_street | Y |
| Keith | Keith_street | N |

| customer-name | account-number |
|---|---|
| Joe | 1 |
| Joe | 2 |
| Mary | 2 |
| Keith | 4 |
| Mike | 5 |
| Keith | 6 |
| Joe | 3 |

```
SELECT C.customer-name, SUM(balance) AS sbalance
From account S, customer C, depositor D
WHERE S.account-number = D.account-number and
        C.customer-name = D.customer-name
GROUP BY customer-name
HAVING COUNT(*) > 1
ORDER BY Desc sbalance
```

| customer-name | sbalance |
|---|---|
| Keith | $444 |
| Joe | $180 |

# Null Values

- We use *null* when the column value is either *unknown* or *inapplicable*.
- A comparison with at least one null value always returns *unknown*.
- SQL also provides a special comparison operator *IS NULL* to test whether a column value is *null*.
- To incorporate nulls in the definition of duplicates we define that two rows are duplicates if corresponding rows are equal or both contain *null*.

# Outer Joins

- Let R and S be two tables. The outer join preserves the rows of R and S that have no matching rows according to the join condition and outputs them with nulls at the non-applicable columns.

- There exist three different variants: *left outer join*, *right outer join* and *full outer join*.

# Conceptual order in query evaluation

- First the relational products of the tables in the *FROM* clause are evaluated.
- From this, rows not satisfying the *WHERE* clause are eliminated.
- The remaining rows are grouped in accordance with the *GROUP BY* clause.
- Groups not satisfying the *HAVING* clause are then eliminated.
- The expressions in the *SELECT* list are evaluated.
- If the keyword *DISTINCT* is present, duplicate rows are now eliminated.
- Evaluate *UNION*, *INTERSECT* and *EXCEPT* for Subqueries up to this point.
- Finally, the set of all selected rows is sorted if the *ORDER BY* is present.