Chapter 4

Structured Query Language SQL

Evolution of SQL

SQL is a non-procedural language that originated at IBM. Originally called SEQUEL, later modified and name changed to SEQUEL2.

The current language has been named SQL (Structured Query Language), some still pronounce as if it was spelled SEQUEL.

It is now a de facto standard for relational database query languages. The American Standards Association has adopted a standard definition of SQL.

SQL

A user of a DBMS using SQL may use the query language interactively or through a host language like C.

SQL provides facilities for data definition as well as for data manipulation. In addition, the language includes some data control features. We briefly describe them.

Data Definition Language (DDL)

- Before a database can be used, it must be created.
- DDL is used for defining the database to the computer system.
- It includes facilities for
  - creating a new table or a view, deleting a table or a view,
  - altering or expanding the definition of a table and
  - creating an index on a table, as well as commands for
  - dropping an index.
- DDL also allows a number of integrity constraints to be defined either at the time of creating a table or later.

Data Manipulation Language (DML)

The DML facilities include commands for retrieving information from one or more tables and updating information in tables including inserting and deleting rows.

The commands SELECT, UPDATE, INSERT and DELETE are available for these operations.

Data Control Language (DCL)

The DCL facilities include database security control including privileges and revoke privileges.

DCL is responsible for managing which users have access to which tables and what data.

New Features – Each new standard has introduced a variety of new features in the language. For example, SQL:2003 includes XML.

Basic DDL and DML Commands

- SELECT - to retrieve data
- UPDATE - to modify values
- INSERT - to add new rows to a table
- DELETE - to delete rows from table
- CREATE - to create a table
- ALTER - to modify tables
- DROP - to delete tables
- GRANT - to give system privileges
- REVOKE - to delete privileges

Example Database

Consider the following database schema with four tables that we will use in illustrating SQL.

Match(MatchID, Team1, Team2, Ground, Date, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets)

Table Definition

To create the relation *Match* we need to use the following command:

CREATE TABLE *Match*
(*MatchID* INTEGER,
Team1 CHAR(15),
Team2 CHAR(15),
Ground CHAR(20),
Date CHAR(10),
Result CHAR(10),
PRIMARY KEY (MatchID))

SQL Data Types

Basic SQL data types are:

- Boolean
- Character – Char or Varchar
- Exact Numeric – Numeric, decimal, integer, smallint
- Approximate Numeric – float, real, double precision
- Datetime – Date, time
- Interval
- Large objects – Character large objects, binary large objects (BLOBS)

Numeric Data Types

The SQL numeric data types are:

Data Type	**Description**
SMALLINT	Integer, precision 5
INTEGER	Integer, precision 10
REAL	Mantissa about 7
FLOAT(p)	Mantissa about 16
DOUBLE PRECISION	Same as FLOAT

Character Strings and Binary Data Types

The SQL character strings and binary data types are:

Data Type	Abbreviation	Description	Comments
CHARACTER(n)	CHAR(n)	Fixed length character string of length n (max n = 255 bytes). Padded on right.	CHAR(5) may be 'India' but not 'Kerala'
CHARACTER VARYING(n)	VARCHAR(n)	Variable length character string up to length n (max n = 2000).	VARCHAR(10) may be 'India' or 'Kerala'
CHRACTER LARGE OBJECT(n)	CLOB(n)	Variable length character string (usually large)	May be a book. n specified in Kbytes, Mbytes or GBytes.
BINARY VARYING(n)	VARBINARY(n)	Variable length character string up to length n	n can be between 1 and 8000.
BINARY LARGE OBJECT(n)	BLOB(n)	Variable length binary string (usually large)	May be a photo. n specified in Kbits, Mbits or GBits.

Date and Time Data Types

The SQL date and time data types are:

Data Type	Description	Example	Comments
Date	YEAR, MONTH and DAY in the format YYYY-MM-DD. Less than comparison may be used.	2010-05-25 or 25-May-2010	Year varies from 0001 to 9999.
Time	HOUR, MINUTE and SECOND in the format HH:MM:SS. Less than comparison may be used.	22:14:37 (fraction of a second may also be represented)	Hour varies from 00 to 23.
TIMESTAMP(n)	Used to specify time of an event in the format YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND	The format is YYYY-MM-DD-HH:MM-SS[.s].	
INTERVAL	Used to represent time intervals like 9 days or 20 hours and 37 minutes.	Can be YEAR-MONTH or DAY-TIME.	

Changing Table Definition

To change the relation *student* we need to use the following command:

ALTER TABLE *student*
(ADD COLUMN marks INTEGER)

In the above definition, we have specified that the attribute *s-id* may not be NULL because it is the primary key of the relation.

DROP TABLE student

Integrity

A database is a very valuable resource.

Its consistency must be maintained at all times.

Valuable and personal information must be protected from unauthorized retrieval.

(1) *recovery* maintains consistency in spite of an accidental failure or damage to the database.

(2) *concurrency* maintains consistency when transactions are being carried out concurrently.

What is Integrity?

Definition – Integrity

Integrity deals with ensuring that the transactions that the users are running are correct and therefore ensures that every database insertion, deletion and update is accurate, valid and consistent according to the constraints specified.

Integrity

Integrity may be *syntactic integrity* or *semantic integrity*.

Syntactic integrity is usually specified when creating a table and may include rules about valid values, for example specifying the domain for each column, the primary and foreign keys, and some alternate keys. Such syntactic constraints are independent of the database that is being created.

Syntactic Integrity

- Primary Key –specifies the primary key for the table to satisfy the *existential* integrity. The primary key is an attribute or set of attributes that possess the *uniqueness* property, and if the key is a set of attributes then also satisfy the *minimality* property.
- Foreign Key –the foreign key in a table *R* is a set of columns (possibly one) whose values are required to match those of the primary key of some table *S*. *R* and *S* are not necessarily distinct. This is *referential* integrity.

Syntactic Integrity

- NOT NULL – a value in a database may be null for several reasons. The most obvious reason is that the value is not known at the present time. If an attribute is defined to be NOT NULL then it cannot accept a NULL value.
- Default – this allows specification of a default value for a column. Without a DEFAULT clause, the default value is NULL.

Syntactic Integrity

- Unique – This constraint ensures that a column or a group of columns in a table have a distinct value. When some columns are defined to be unique, it may be considered as a specification of an alternate key although UNIQUE allows NULL values.
- Domain – this allows a column domain to be defined. These domains include INTEGER, SMALLINT, DECIMAL, VARCHAR or DATE.

Semantic Integrity

Semantic integrity involves compliance of the database with constraints which are derived from users' understanding of the database and what is and what is not allowed in the database.

The *maintenance* or *enforcement* of semantic integrity involves preventing data which represents a disallowed state from being inserted in the database.

A disallowed state is determined by rules about the actions that may be performed when a event like update, delete or insertion takes place.

Semantic Integrity

Semantic integrity constraints may be classified as:

- *State constraints* – time independent constraints for a particular database that are independent of the update at hand and are usually specified by a single predicate that specifies the state of the current database. Usually checked when a single database state is available.
- *Transitional constraints* – these constraints also apply to the update at hand. Scope varies from being applicable to a single attribute in a single relation to involving several attributes in several relations. Checking of each such constraint requires two consecutive database states, the state just before a transaction and the state just after the transaction.

Semantic Integrity

- *Single variable, aggregate free*

  These are single relational constraints involving single variables and are therefore the simplest to implement.

  Eg. “Each full-time employee must be paid at least Rs 10,000 a month”.

Semantic Integrity

- *Multivariable, aggregate free with only one tuple variable being updated.*

  These could be domain constraints or intra-relation dependencies.

  Eg, “every employee’s salary shall be greater than any recorded salary value for that employee in the salary history”.

Semantic Integrity

- *Multivariable multirelational constraints without aggregates.*

  These constraints involve relationships between relations (for example, referential integrity or inclusion dependencies).

  Eg “every employee must earn less than his manager”. assuming managers’ information is in another table.

Semantic Integrity

- *Data aggregate assertions.*

  These are often the most expensive because of the need to process the aggregates.

  Eg. “total salaries cannot exceed the departmental budget”.

Column Constraints and Table Constraints

Some constraints may be specified within a column definition of a table by specifying the constraint after the column definition.

They are called *column constraints*.

These may be specified for the primary key, the foreign key, the NOT NULL and the CHECK constraint if these constraints involve only one attribute.

Column Constraints and Table Constraints

- If more than one attribute is involved the table constraint must be used.
- A single attribute constraint may also be used as a table constraint but it is then likely to be checked more frequently as it will be checked every time an insert, deletion or update is made to the table.
- A column constraint will not be checked if values in other columns are being updated.

PK as a Column Constraint

A column constraint is usually used when the PK is a single attribute.

```
CREATE TABLE Match
    (MatchID INTEGER PRIMARY KEY,
    Team1 CHAR(15),
    Team2 CHAR(15),
    Ground CHAR(20),
    Date CHAR(10),
    Result CHAR(10));
```

PK as a Table Constraint

A table constraint is usually used when the PK is more than a single attribute.

CREATE TABLE *Bowling*
(*MID* INTEGER,
PID INTEGER,
NOvers INTEGER,
Maidens INTEGER,
NRuns INTEGER,
NWickets INTEGER,
PRIMARY KEY (*MID, PID*));

FK as a Column Constraint

A column constraint is usually used when the FK is a single attribute.

CREATE TABLE *Employee*
(*EmpID* NUMBER(6) PRIMARY KEY,
Name CHAR(20),
Dept CHAR(10), REFERENCES *Department* (*DeptID*),
Address CHAR (50)
Position CHAR (20));

FK as a Table Constraint

A table constraint is usually required when the FK is more than a single attribute.

CREATE TABLE *Bowling*
(*MatchID* INTEGER,
PID INTEGER,
NOvers INTEGER,
Maidens INTEGER,
NRuns INTEGER,
NWickets INTEGER,
PRIMARY KEY (*MID, PID*)
FOREIGN KEY (*MatchID*) REFERENCES *Match,*
FOREIGN KEY (*PID*) REFERENCES *Player*);

NULL as a Column Constraint

```
CREATE TABLE Match
        (MatchID INTEGER PRIMARY KEY,
        Team1 CHAR(15) NOT NULL,
        Team2 CHAR(15) NOT NULL,
        Ground CHAR(20),
        Date CHAR(10),
        Result CHAR(10));
```

DEFAULT as a Column Constraint

CREATE TABLE *Match*
(*MatchID* INTEGER PRIMARY KEY
Team1 CHAR(15) DEFAULT 'India',
Team2 CHAR(15),
Ground CHAR(20),
Date CHAR(10),
Result CHAR(10));

UNIQUE as a Column Constraint

A column constraint is usually used when UNIQUE is a single attribute.

CREATE TABLE *Employee*
(*EmpID* NUMBER(6) PRIMARY KEY,
Name CHAR(20),
DeptID CHAR(10),
Telephone INTEGER UNIQUE,
Address CHAR (50),
Position CHAR (20);

UNIQUE as a Table Constraint

A table constraint is usually required when UNIQUE is more than a single attribute.

```
CREATE TABLE Player
   (PlayerID INTEGER PRIMARY KEY,
   LName CHAR(15),
   FName CHAR(15),
   Country CHAR(20),
   YBorn INTEGER,
   BPlace CHAR(20)
   FTest INTEGER,
   UNIQUE (LName, FName));
```

Domain Constraint

CREATE DOMAIN *Name1* CHAR(15),

CREATE DOMAIN *PlayerID1* INTEGER,

CREATE DOMAIN *Name2* CHAR(15);

CHECK Constraint

	Possible conditions in the CHECK clause
1	attribute A > value v
2	attribute A between value v1 and value v2
3	attribute A IN (list of values)
4	Attribute A IN subquery
5	attribute A condition C1 OR condition C2
6	attribute A condition C1 AND condition C2

CHECK as a Column Constraint

CREATE TABLE *Player*
(*PlayerID* INTEGER PRIMARY KEY,
LName CHAR(15),
FName CHAR(15),
Country CHAR(20),
YBorn INTEGER CHECK (YBorn > 1950),
BPlace CHAR(20)
FTest INTEGER));

CHECK as a Table Constraint

A table constraint is used when the CHECK constraint has more than a single attribute.

```
CREATE TABLE Player
        (PlayerID INTEGER PRIMARY KEY,
        LName CHAR(15) NOT NULL,
        FName CHAR(1) NOT NULL,
        Country CHAR(20),
        YBorn INTEGER,
        BPlace CHAR(20)
        FTest INTEGER,
        CHECK (FTest > YBorn + 15));
```

Insertion

- To insert data into a relation, we either specify a tuple to be inserted or write query whose result is a set of tuples to be inserted.
- Obviously, the attribute values for inserted tuples must be members of the corresponding attribute's domain.
- Similarly, tuples inserted must have the correct number of attributes.

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*)

Insertion

insert into *Player* **values**
(1, 'Gupta' , *'R'*,
'India', 1978, 'Mumbai' , '1990');

(*PlayerID* INTEGER PRIMARY KEY,
LName CHAR(15) NOT NULL,
FName CHAR(1) NOT NULL,
Country CHAR(20),
YBorn INTEGER,
BPlace CHAR(20)
FTest INTEGER,
CHECK (*FTest* > *YBorn* + 15

Basic Data Retrieval

The basic structure of the data retrieval command is as follows

```
SELECT something_of_interest
FROM table(s)
WHERE condition_holds
```

The SELECT clause specifies what information is to be retrieved and the FROM clause specifies the table(s) that are needed to answer the query.

SQL Aliases

A table or a column may be given another name in a SQL query by using an alias. This can be helpful if, for example, we want to change the columns' names in the output from the names used in the base tables. Two examples of alias are given below.

```
SELECT column_name(s)
FROM table_name AS alias_name
```

```
SELECT column_name AS alias_name
FROM table_name
```

Simple Examples – Selecting Columns and Rows

The WHERE clause specifies the condition to be used in selecting rows and is optional, if it is not specified the whole table is selected.

Retrieving the whole table using *

(Q1) Print the Player table.

This query may be formulated as follows. It's result is table *Player*.

```
SELECT *
FROM Player
```

Using Projection to Retrieve Some Columns of All the Rows from One Table

(Q2) Find the IDs and first and last names of all players.

This query may be formulated as below. The result is given on the nest slide.

```
SELECT PlayerID, FName, LName
FROM Player
```

Result of Q2

PlayerID	FName	LName
89001	Sachin	Tendulkar
90001	Brian	Lara
95001	Ricky	Ponting
96001	Rahul	Dravid
96002	Herschelle	Gibbs
92001	Shane	Warne
95002	Shaun	Pollock
99003	Michael	Vaughan
92003	Inzamam	Ul-Huq
94004	Stephen	Fleming
93002	Heath	Streak
90002	Anil	Kumble
93003	Gary	Kirsten
95003	Jacques	Kallis
94002	Chaminda	Vaas
92002	Muthiah	Muralitharan
97004	Daniel	Vettori
25001	M. S.	Dhoni
23001	Yuvraj	Singh
96003	Saurav	Ganguly
99002	Adam	Gilchrist
24001	Andrew	Symonds
99001	Brett	Lee
91001	Sanath	Jayasuriya
21001	Virender	Sehwag
98001	Shahid	Afridi
98002	Harbhajan	Singh
27001	Praveen	Kumar
27002	Ishant	Sharma

Retrieving All Information about Certain Rows from One Table

(Q3) Find all the information from table *Player* about players from India.

```
SELECT *
FROM Player
WHERE Country = 'India'
```

(Q4) Find all the information from table *Player* about players from India who were born after 1980.

```
SELECT *
FROM Player
WHERE Country = 'India'
AND YBORN > 1980
```

Result of Q3

PlayerID	LName	FName	Country	YBorn	BPlace	FTest
89001	Tendulkar	Sachin	India	1973	Mumbai	1989
96001	Dravid	Rahul	India	1973	Indore	1996
90002	Kumble	Anil	India	1970	Bangalore	1990
25001	Dhoni	M. S.	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
96003	Ganguly	Saurav	India	1972	Calcutta	1996
21001	Sehwag	Virender	India	1978	Delhi	2001
98002	Singh	Harbhajan	India	1980	Jalandhar	1998
27001	Kumar	Praveen	India	1986	Meerut	NULL
27002	Sharma	Ishant	India	1988	Delhi	2007

Result of Q4

PlayerID	LName	FName	Country	YBorn	BPlace	FTest
25001	Dhoni	M. S.	India	1981	Ranchi	2005
23001	Singh	Yuvraj	India	1981	Chandigarh	2003
27001	Kumar	Praveen	India	1986	Meerut	NULL
27002	Sharma	Ishant	India	1988	Delhi	2007

Retrieving Certain Columns of Certain Rows from One Table

(Q5) Find details of matches that have been played in Australia.

This query may be formulated in SQL as follows. A WHERE clause specifies the condition that Team1 is Australia.

```
SELECT Team1, Team2, Ground, Date
FROM Match
WHERE Team1 = 'Australia'
```

Result of Q5

Team1	Team2	Ground	Date
Australia	India	Melbourne	10/2/2008
Australia	India	Sydney	2/3/2008
Australia	India	Brisbane	4/3/2008

More about SELECT

Calculations in SELECT

Some calculations are possible in SELECT. For example it is possible to write:

```
SELECT 500*credit

SELECT salary+bonus
```

New Names for retrieved information

New names may be given in the SELECT command. For example:

```
SELECT 500*credit AS Fees

SELECT salary+bonus AS Total
```

Simple Exercises

(Q6) List matches played in which India or Australia was Team1.

The condition in this query is slightly more complex. We show below how this condition may be formulated in SQL.

SELECT *Team1, Team2, Ground, Date*
FROM *Match*
WHERE *Team1* IN {'Australia', 'India'}

Constants like 'Australia' or 'India' are often called a *literal tuple*. A literal tuple could have a number of values. For example it could be

WHERE {2675, Australia, India, Melbourne, 10/2/2008, Team1} IN Match

Result of Q6

Team1	Team2	Ground	Date
India	England	Delhi	28/3/2006
India	Australia	Mohali	29/10/2006
India	West Indies	Nagpur	21/1/2007
India	West Indies	Vadodara	31/1/2007
India	Sri Lanka	Rajkot	11/2/2007
India	Australia	Mumbai	17/10/2007
India	Pakistan	Guwahati	5/11/2007
Australia	India	Melbourne	10/2/2008
India	Sri Lanka	Adelaide	19/2/2008
Australia	India	Sydney	2/3/2008
Australia	India	Brisbane	4/3/2008

Removing Duplicates

(Q7) Find IDs of all players that have bowled in an ODI match in the database.

This query requires removal of duplicates as shown below.

SELECT DISTINCT (*PID*)
FROM *Bowling*

The result of this query is given.

PID
99001
24001
23001
94002
92002
91001
23001
98002

Removing Duplicates

(Q8) Find names of teams and grounds where India has played an ODI match outside India.

This query shows use of DISTINCT when the query retrieves more than one column. The query may be formulated in SQL as follows.

SELECT DISTINCT (*Team1*, *Ground*)
FROM *Match*
WHERE Team2 = 'India'

Result of Q8

Team1	**Ground**
Pakistan	Peshawar
Pakistan	Rawalpindi
West Indies	Kingston
Sri Lanka	Colombo
South Africa	Cape Town
England	Southampton
Australia	Melbourne
Australia	Sydney
Australia	Brisbane
Pakistan	Karachi

Sorting the Information Retrieved

The ORDER BY clause returns results in ascending order (that is the default). Descending order may be specified by using

```
ORDER BY name DESC
```

Without the ORDER BY clause, the order of the result depends on the DBMS implementation.

The ORDER BY clause may be used to order the result by more than one attribute.

Sorting

(Q10) Display a sorted list of ground names where Australia has played as Team1.

The query may be formulated as follows:

```
SELECT Ground
FROM Match
WHERE Team1 = 'Australia'
ORDER BY Ground
```

Ground
Adelaide
Brisbane
Melbourne
Sydney

The result is shown on the right.

String Matching

SQL provides string pattern matching facilities using the LIKE command.

LIKE is a powerful string matching operator that uses two special characters (called *wildcards*).

These characters are underscore (_) and percent (%).

They have special meanings; underscore represents any single character while percent represents any sequence of *n* characters including a sequence of no characters.

Operation	Meaning
UPPER (string)	Converts the string into uppercase
LOWER (string)	Converts the string into lowercase
INITCAP (string)	Converts the initial letter to uppercase
LENGTH (string)	Finds the string length
SUBSTR(string, n, m)	Get the substring starting at position n

String Matching

(Q11) Find the names of all players whose last name starts with Sing.

The query may be formulated as follows:

```
SELECT FName, LName
FROM Player
WHERE LName LIKE 'Sing%'
```

The result of this query is:

FName	LName
Yuvraj	Singh
Harbhajan	Singh

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.

  Suppose a tuple occurs m times in r and n times in s, then, it occurs:
  - $m + n$ times in r **union all** s
  - $\min(m,n)$ times in r **intersect all** s
  - $\max(0, m - n)$ times in r **except all** s

Set Operations

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

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

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

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

- Find all customers who have an account but no loan.

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

Aggregate Functions

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

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

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch

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

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

```
select count (*)
        from customer
```

- Find the number of depositors in the bank.

```
select count (distinct customer_name)
        from depositor
```

Aggregate Functions – Group By

- Find the number of depositors for each branch.

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

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

Aggregate Functions – Having Clause

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

```
select branch_name, avg (balance)
        from account
        group by branch_name
        having avg (balance) > 1200
```

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

Using Aggregate Functions

Find the number of players that bowled in the ODI match 2689.

```
SELECT COUNT(*) AS NBowlers
FROM Bowling
WHERE MatchID = '2689'
```

NBowlers
4

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

Using Aggregate Functions

(Q26) Find the average batting score of all the players that batted in the ODI match 2689.

```
SELECT AVG(NRuns) AS AveRuns_2689
FROM Batting
WHERE MatchID = '2689'
```

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

AveRuns_2689
25.2

Using Functions

Find the youngest player in the database.

The player's age is not an attribute in the *Player* table. The only attribute related to age is the Player's year of birth (*YBorn*). To find the youngest player we will find the player who was born last according to *YBorn*.

```
SELECT Lname AS Youngest_Player
FROM Player
WHERE YBorn =
        (SELECT MAX(YBorn)
        FROM Player)
```

Player(PlayerID, LName, FName, Country, YBorn, BPlace, Ftest)

Youngest_Player
Sharma

GROUP BY and HAVING Clauses

So far we have considered relatively simple queries that use aggregation functions like AVG, MAX, MIN and SUM.

These functions are much more useful when used with GROUP BY and HAVING clauses which divide a table into virtual tables and apply qualifications to those virtual tables.

GROUP BY and HAVING clauses allow us to consider groups of records together that have some common characteristics (for example, players from the same country) and compare the groups in some way or extract information by aggregating data from each group.

The group comparisons also are usually based on using an aggregate function.

GROUP BY and HAVING Clauses

The general form of a GROUP BY query is as follows.

```
SELECT something_of_interest
FROM table(s)
WHERE condition_holds
GROUP BY column_list
HAVING group_condition
```

The SELECT clause must include column names that are in the GROUP BY column list (other column names are not allowed in SELECT) and aggregate functions applied to those columns.

The WHERE clause in the query applies to the table before GROUP BY and HAVING and therefore results in a rows pre-selection.

GROUP BY and HAVING Clauses

The procedure in executing a GROUP BY and HAVING clause works as follows:

- All the rows that satisfy the WHERE clause (if given) are selected
- All the rows are now grouped virtually according to the GROUP BY criterion
- The HAVING clause is applied to each virtual group
- Groups satisfying the HAVING clause are selected
- The aggregation function(s) are now applied
- Values for the columns in the SELECT clause and aggregations are now retrieved

GROUP BY and HAVING Clauses

Find the number of players in the database from each country.

To find the number of players from each country, we need to partition the Player table to create virtual sub-tables for players from each country. Once the table has been so partitioned, we can count them.

```
SELECT Country, COUNT(*) AS NPlayers
FROM Player
GROUP BY Country
```

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

Result

Country	NPlayers
South Africa	4
Australia	5
England	1
Pakistan	2
New Zealand	2
Zimbabwe	1
India	10
Sri Lanka	3
West Indies	1

GROUP BY and HAVING Clauses

(Q29) Find the batting average of each player in the database. Present the PID of each player.

To find the batting average of each player, we need to partition the *Batting* table to create virtual sub-tables for each player. Once the table has been so partitioned, we can find the average of each player.

```
SELECT PID, AVE(NRuns) AS Ave
FROM Batting
GROUP BY PID
```

PID	Ave
23001	38
24001	42
25001	36
27001	7
89001	91
91001	60
92002	1
94002	17
95001	1
98002	3
99001	7
99002	2

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

GROUP BY and HAVING

(Q30) Find the batting average of each Indian player in the database. Present the firstname of each player.

To find the batting average of each player, we need to join the tables *Player* and *Batting* and then partition the joined table to create virtual sub-tables for each player. Once the table has been so partitioned, we can find the average of each player.

```
SELECT FName, AVE(NRuns) AS AveRuns
FROM Player, Batting
WHERE PlayerID = PID
AND Country = 'India'
GROUP BY fName
```

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

Result of Q30

We have used the first names since the last names are not unique in our database.

Fname	AveRuns
Yuvraj	38
M.S.	36
Praveen	7
Sachin	91
Harbhajan	3

GROUP BY and HAVING

(Q31) Find the average score for each player when playing in Australia.

```
SELECT PID AS PlayerID, AVG(NRuns) AS AveRuns
FROM Batting
WHERE MatchID IN
        (SELECT MatchID
        FROM Match
        WHERE Team1 = 'Australia')
GROUP BY PID
```

This query first finds all rows in Batting that are for matches played in Australia by using the subquery. Then the rows are grouped by PID and the average score for each player is computed.

Result of Q31

PlayerID	AveRuns
89001	91
23001	38
25001	36
99002	2
95001	1
24001	42
99001	7
27001	7
98002	3

GROUP BY and HAVING

(Q32) Find the ID of players that had a higher average score than the average score for all players when they played in Sri Lanka.

```
SELECT PID AS PlayerID, AVG(NRuns) AS AveRuns
FROM Batting
GROUP BY PID
HAVING AVG(NRuns) >
        (SELECT AVG(NRuns)
        FROM Batting
        WHERE MatchID IN
                (SELECT MatchID
                FROM Match
                WHERE Team1= 'Sri Lanka')
        GROUP BY PID)
```

Result of Q32

PlayerID	AveRuns
25001	53.5
91001	60.0
89001	91.0
24001	42.0

Multiple Aggregate Functions

```
SELECT AVG(NPlayers) AS NAvePlayers
FROM
        (SELECT Country, COUNT(*) AS NPlayers
        FROM Player
        GROUP BY Country) NP
```

NAvePlayers
3.2

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

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example: *5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*,
    (*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)
  from *loan*

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

Different Forms of the WHERE Clause

So far we have used WHERE class for testing equality conditions. Some of the common forms are:

- WHERE C1 AND C2 – each selected row must satisfy both the conditions C1 and C2.
- WHERE C1 OR C2 – each selected row must satisfy either the condition C1 or the condition C2 or both conditions C1 and C2.

AND and OR operators may be combined for example in a condition like A AND B OR C.

In such compound conditions the AND operation is done first. Parentheses could be used to ensure that the intent is clear.

WHERE clause

- WHERE NOT C1 AND C2 – each selected row must satisfy condition C2 but must not satisfy the condition C1.
- WHERE A IN – each selected row must have the value of A in the list that follows IN. NOT IN may be used to select all that are not in the list that follows NOT IN.
- WHERE A operator ANY – each selected row must satisfy the condition for any of the list that follows ANY.

Any of the following operators may be used in such a clause: greater than (>), less than (<), less than equal (≤), greater than equal (≥) and not equal (<>).

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 )
```

WHERE clause

- WHERE A operator ALL – each selected row now must satisfy the condition for each of the list that follows ALL. Any of the following operators may be used in such a clause: greater than (>), less than (<), less than equal (≤), greater than equal (≥) and not equal (<>).
- WHERE A BETWEEN x AND y – each selected row must have value of A between x and y including values x and y. NOT BETWEEN may be used to find rows that are outside the range (x, y).
- WHERE A LIKE x – each selected row must have a value of A that satisfies the string matching condition specified in x. The expression that follows LIKE must be a character string and enclosed in apostrophes, for example 'Delhi'.

WHERE clause

- WHERE A IS NULL – each selected row must satisfy the condition that A is NULL, that is, a value for A has not been specified
- WHERE EXISTS – each selected row be such that the subquery following EXISTS does return something, that is, the subquery does not return a NULL result.
- WHERE NOT EXISTS – each selected row be such that the subquery following NOT EXISTS does not return anything, that is, the subquery returns a NULL result.

Queries Involving More than One Table – Using Joins and Subqueries

Many relational database queries require more than one table because each table in the database is about a single entity.

For example, player information including player name is in table *Player* while batting and bowling performances are provided in tables *Batting* and *Bowling*.

Two possible approaches in using more than one table. Either use a join or use subqueries

When using a join, most queries below only consider an inner equi-join.

Subqueries

A subquery is a query that is part of another query. Subqueries may continue to any number of levels. The subqueries in examples are used in the WHERE clause but subqueries may also be used in the FROM clause

(Q12) Find Match IDs of all matches in which Tendulkar batted.

We use a subquery to obtain the PID of Tendulkar and use that to obtain Match IDs of matches that he has played in.

```
SELECT MatchID
    FROM Batting
    WHERE PID IN
        (SELECT PlayerID
        FROM Player
        WHERE Lname = 'Tendulkar' )
```

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

Subqueries

The query may be formulated as below:

```
SELECT MatchID
FROM Batting
WHERE PID IN
        (SELECT PlayerID
        FROM Player
        WHERE Lname = 'Tendulkar' )
```

The result of the above query is given below. Due to very limited data in the database only one ODI is found.

MatchID
2689

Subqueries

(Q13) Find the match information of all matches in which Dhoni has batted.

This query must involve the three tables *Match*, *Batting* and *Player* as the match information is in *Match*, the player names are in *Player* and batting information is available only in table *Batting*.

```
SELECT Team1, Team2, Ground, Date
    FROM Match
    WHERE MatchID IN
        (SELECT MatchID
        FROM Batting
        WHERE PID IN
                (SELECT PlayerID
                FROM Player
                WHERE LName = 'Dhoni'))
```

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

Result of Q13

MatchId	Team1	Team2	Ground	Date
2689	Australia	India	Brisbane	4/3/2008
2755	Sri Lanka	India	Colombo	27/8/2008

Queries Involving Joins

The two queries, Q12 and Q13, involving subqueries may be reformulated using the join operator as given below.

```
SELECT MatchID
FROM Batting, Player
WHERE PID = PlayerID
AND LName = 'Tendulkar'
```

```
SELECT Team1, Team2, Ground, Date
FROM Match, Batting, Player
WHERE MatchID = MID
 AND PlayerID = PID
 AND LName = 'Dhoni'
```

Queries Using Joins

(Q14) Find the player ID of players who have made a century in each of the ODI matches 2755 and 2689.

The table *Batting* has a separate row for each innings. We can therefore either use a subquery or use a join. We use a join of table *Batting* with itself.

```
SELECT PID
FROM Batting b1, Batting b2
WHERE b1.PID = b2.PID
AND b1.MatchID = 2755
AND b2.MatchID = 2689
AND b1.NRuns > 99
AND b2.NRuns > 99
```

More Subqueries

(Q15) Find IDs of players that have both bowled and batted in the ODI match 2689.

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)
Bowling(MatchID, PID, NOvers, Maidens, NRuns, NWickets

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
AND PID IN
    (SELECT PID
    FROM Bowling
    WHERE MatchID = '2689')
```

Result of Q15

PID
99001
24001
23001
98002

Subqueries

(Q16) Find IDs of players that have either bowled or batted (or did both) in the ODI match 2689.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
OR PID IN
    (SELECT PID
    FROM Bowling
    WHERE MatchID = '2689')
```

Subqueries

(Q16) Find IDs of players that have either bowled or batted (or did both) in the ODI match 2689.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
UNION
SELECT PID
FROM Bowling
WHERE MatchID = '2689'
```

Result of Q16

PID
89001
98002
23001
25001
99002
95001
24001
99001
27001

Queries Involving Subqueries

(Q17) Find IDs of players that have batted in match 2689 but have not bowled.

```
SELECT PID
FROM Batting
WHERE MatchID = '2689'
AND PID NOT IN
    (SELECT PID
    FROM Bowling
    WHERE MatchID = '2689')
```

PID
89001
25001
99002
95001
27001

Using Subquery in the FROM Clause

(Q18) Find the match IDs of matches in which Sachin Tendulkar has played.

```
SELECT MatchID
FROM Batting, (SELECT PlayerID
               FROM Player
               WHERE Lname = 'Tendulkar' ) ST
WHERE PID = ST.PlayerID
```

This subquery uses a subquery in the FROM clause which returns a table as its result and is therefore treated like a virtual table. In this particular case, the table returned has only one element which is the PlayerID of Sachin Tendulkar.

Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)

Subqueries

(Q20) Find IDs and scores of players who scored less than 75 but more than 50 in Colombo.

```
SELECT PID, NRuns
FROM Batting
WHERE NRuns BETWEEN 51 AND 74
AND MatchID IN
        (SELECT MatchID
        FROM Match
        WHERE Ground = 'Colombo')
```

Note that we are looking for scores between 51 and 74 and not between 50 and 75. This is because (A BETWEEN x AND y) has been defined to mean $A \geq x$ and $A \leq y$.

NULLs

(Q21) Find the player IDs of those players whose date of first test match (FTest) is not given in the database.

It should be noted that the column value being NULL is very different than it being zero. The value is NULL only if it has been defined to be NULL.

SELECT *PlayerID*
FROM *Player*
WHERE *FTest* IS NULL

Set Comparison (Some)

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

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

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

Branch(branch_name, branch_city, assets)

Definition of Some Clause

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

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**

However, (≠ **some**) ≢ **not in**

Example Query

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

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

Definition of all Clause

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

(5 < **all** | 0 / 5 / 6 |) = false

(5 < **all** | 6 / 10 |) = true

(5 = **all** | 4 / 5 |) = false

(5 ≠ **all** | 4 / 6 |) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≡ **in**

Subqueries

(Q19) Find match IDs in which player 27001 bats and makes more runs than he made at every match he played at Brisbane.

```
SELECT MatchID
FROM Batting
WHERE PID = '27001'
AND NRuns > ALL
        (SELECT NRuns
        FROM Batting
        WHERE PID = '27001'
        AND MatchID IN
                (SELECT MatchID
                FROM Match
                WHERE Ground = 'Brisbane'))
```

MatchID
2755

Match(MatchID, Team1, Team2, Ground, Dat, Winner)
Batting(MatchID, PID, Order, HOut, FOW, NRun, Mts, NBalls, Fours, Sixes)

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-1980 00:00:00	800		20
499	ALLEN	SALESMAN	7698	20-FEB-1981 00:00:00	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-1981 00:00:00	1250	500	30
7566	JONES	MANAGER	7839	02-APR-1981 00:00:00	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-1981 00:00:00	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-1981 00:00:00	2850		30
7782	CLARK	MANAGER	7839	09-JUN-1981 00:00:00	2450		10
7788	SCOTT	ANALYST	7566	19-APR-1987 00:00:00	3000		20
7839	KING	PRESIDENT		17-NOV-1981 00:00:00	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-1981 00:00:00	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-1987 00:00:00	1100		20
7900	JAMES	CLERK	7698	03-DEC-1981 00:00:00	950		30
7902	FORD	ANALYST	7566	03-DEC-1981 00:00:00	3000		20
7934	MILLER	CLERK	7782	23-JAN-1982 00:00:00	1300		10

Any Operator

ANY=SOME

SELECT empno, sal FROM emp

WHERE sal > ANY (2000, 3000, 4000);

EMPNO	SAL
7566	2975
7698	2850
7782	2450
7788	3000
7839	5000
7902	3000

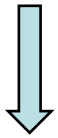
Any Operator

```
SELECT empno, sal FROM emp
        WHERE sal > 2000 OR sal > 3000 OR
        sal > 4000;
```

EMPNO	SAL
7566	2975
7698	2850
7782	2450
7788	3000
7839	5000
7902	3000

Any Operator

- SELECT e1.empno, e1.sal FROM emp e1
  WHERE e1.sal > ANY (SELECT e2.sal
  FROM emp e2 WHERE
  e2.deptno = 10);



SELECT e1.empno, e1.sal FROM emp e1
WHERE EXISTS (SELECT e2.sal
FROM emp e2 WHERE e2.deptno =
10 AND e1.sal > e2.sal);

ALL Operator

SELECT empno, sal FROM emp

WHERE sal > ALL (2000, 3000, 4000);

EMPNO SAL

7839 5000

Transformed to equivalent statement without ALL.

SELECT empno, sal FROM emp

WHERE sal > 2000 AND sal > 3000 AND sal > 4000;

EMPNO SAL

7839 5000

ALL Operator

```
SELECT e1.empno, e1.sal FROM emp e1 WHERE
        e1.sal > ALL (SELECT e2.sal FROM emp e2
        WHERE e2.deptno = 20);
```

EMPNO	SAL
7839	5000

Transformed to equivalent statement using ANY.

```
SELECT e1.empno, e1.sal FROM emp e1
   WHERE NOT (e1.sal <= ANY (SELECT e2.sal
            FROM emp e2 WHERE e2.deptno = 20));
```

EMPNO	SAL
7839	5000

Correlated Subqueries and Complex Queries

Some subqueries used need to be evaluated only once. Such subqueries are called *non-correlated subqueries*.

There is another type of subqueries that are more complex and are called *correlated subqueries*.

In these subqueries the value retrieved by the subquery depends on a variable (or variables) that the subquery receives from the outer query.

A correlated subquery thus cannot be evaluated once and for all since the outer query and the subquery are related.

It must be evaluated repeatedly; once for each value of the variable received from the outer query.

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** $r \Leftrightarrow r = \emptyset$

CUSTOMER

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

ORDERS

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20

Find all of the records from the *customers* table where there is at least one record in the *orders* table with the same *customer_id*.

```
SELECT * FROM customers WHERE
        EXISTS (SELECT * FROM orders
        WHERE customers.customer_id =
        orders.customer_id);
```

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL

```
SELECT * FROM customers WHERE
    NOT EXISTS (SELECT * FROM orders
        WHERE customers.customer_id =
        orders.customer_id);
```

customer_id	last_name	first_name	favorite_website
6000	Ferguson	Samantha	bigactivities.com
9000	Johnson	Derek	techonthenet.com

Correlated Subqueries and Complex Queries

(Q22) Find the names of players who batted in match 2689.

we look at each player in the table *Player* and then check, using a subquery, if there is a batting record for him in the table *Batting*.

```
SELECT FName, LName
FROM Player
WHERE EXISTS
    (SELECT *
    FROM Batting
    WHERE PlayerID = PID
    AND MatchID = '2689')
```

Result of Q22

FName	LName
Sachin	Tendulkar
M. S.	Dhoni
Yuvraj	Singh
Adam	Gilchrist
Andrew	Symonds
Brett	Lee
Praveen	Kumar
Ricky	Ponting
Harbhajan	Singh

Reformulation of Q22

Query Q22 may be formulated in other ways. One of these formulations uses a join. Another uses a different subquery. Using the join, we obtain the following SQL query.

```
SELECT FName, LName
FROM Player, Batting
WHERE PlayerID  = PID
AND MatchID = '2689'
```

Complex Query

(Q23) Find the player IDs of players that have scored more than 30 in every ODI match that they have batted in.

SQL provides no direct way to check that for a given player all his batting or bowling performances satisfy a condition like all scores being above 30 but we can obtain the same information by reformulating the query which makes the query somewhat complex because the query now becomes "find the players that have no score of less than 31 in any innings that they have batted in"!

```
SELECT PID AS PlayerID
    FROM Batting b1
    WHERE NOT EXISTS
            (SELECT *
            FROM Batting b2
            WHERE b1.PID = b2.PID
            AND NRuns < 31)
```

Difference between IN and Exists

IN -- The inner query is executed first and the list of values obtained as its result is used by the outer query.The inner query is executed for only once.

EXISTS -- The first row from the outer query is selected ,then the inner query is executed and , the outer query output uses this result for checking.This process of inner query execution repeats as many no.of times as there are outer query rows. That is, if there are ten rows that can result from outer query, the inner query is executed that many no.of times.

Difference between IN and Exists

- In operator test for the particular value in the subquery and exist operator is a boolean operator. so it works more efficient and faster than In operator

- In operator scan all the values inside the IN block where as EXIST quit after 1st occurrence.

Complex Query

Note that the subquery in Q23 cannot be executed only once since it is looking for rows in batting that satisfy the condition NRuns < 31 for each player in the table *Batting* whose PID has been supplied to the subquery by the outside query. The subquery therefore must be evaluated repeatedly; once for each value of the variable PID received by the subquery. The query is therefore using a correlated subquery.

Note that the condition in the WHERE clause will be true only if the subquery returns a null result and the subquery will return a null results only if the player has no score which is less than 31.

Complex Query

(Q24) Find the names of all players that have batted in all the ODI matches in Melbourne that are in the database.

This is a complex query that uses a correlated subquery because of the nature of SQL as it does not include the universal quantifier (forall) that is available in relational calculus.

Therefore we need to reformulate the question.

The reformulated question is "Find the names of players for whom there is no ODI match played in Melbourne that they have not batted in"!

Complex Query Q24

This nested query consists of three components: the outermost query, the middle subquery and the innermost (or the last) subquery. Try to understand what each subquery does.

```
SELECT FName, LName
FROM Player
WHERE NOT EXISTS
        (SELECT *
        FROM Match
        WHERE Ground = 'Melbourne'
        AND NOT EXISTS
                (SELECT *
                FROM Batting
                WHERE PID = PlayerID
                AND (Batting.MatchID = Match.MatchID))
```

Update – Modifying Existing Data

The UPDATE command is used to carry out modifications. The basic form of UPDATE command is as follows:

```
UPDATE Table
SET Newvalues
[WHERE condition]
```

(Q37) Increase the mark of student 20086532 in CS100 by 5.

```
UPDATE Enrolment
SET Mark = Mark + 5
WHERE ID = 20086532
AND Code = "CS100"
```

Update – Modifying Existing Data

(Q38) Increase the marks of all students in CS100 by 5.

```
UPDATE Enrolment
SET Mark = Mark + 5
WHERE Code = "CS100"
```

Deleting Data

(Q39) Delete match 2689 from the table *Match*.

```
DELETE Match
WHERE MatchID = '2689'
```

Update – Modifying Existing Data

(Q40) Delete all bowling records of Brian Lara.

We use a subquery to find the player ID of Brian Lara and then delete all rows for that player ID from the table *Bowling*.

DELETE *Bowling*
WHERE *PID* =
    (SELECT *PlayerID*
    FROM *Player*
    WHERE *LName* = 'Lara')

Table Definition

To create the relation *student* we need to use the following command:

CREATE TABLE *student*
(*s-id* INTEGER NOT NULL,
s-name CHAR(15),
address CHAR(25))

In the above definition, we have specified that the attribute *s-id* may not be NULL because it is the primary key of the relation.

Change Table Definition

To change the relation *student* we need to use the following command:

ALTER TABLE *student*
(ADD COLUMN marks INTEGER)

In the above definition, we have specified that the attribute *s-id* may not be NULL because it is the primary key of the relation.

DROP TABLE student

Outer Join

Consider the two tables, the first giving best ODI batsmen and another best test batsmen. Both on 1 Jan 2010.

Player	Span	Matches	Innings	Runs	Ave	Strike Rate	100s
SR Tendulkar	1989-2010	440	429	17394	44.71	85.90	45
ST Jayasuriya	1989-2010	444	432	13428	32.43	91.22	28
RT Ponting	1995-2010	330	321	12311	43.19	80.50	28
Inzamam-ul-Haq	1991-2007	378	350	11739	39.52	74.24	10
SC Ganguly	1992-2007	311	300	11363	41.02	73.70	22
R Dravid	1996-2010	339	313	10765	39.43	71.17	12
BC Lara	1990-2007	299	289	10405	40.48	79.51	19
JH Kallis	1996-2010	295	281	10409	45.25	72.01	16
AC Gilchrist	1996-2008	287	279	9619	35.89	96.94	16
Mohammed Yousuf	1998-2009	276	261	9495	42.96	75.30	15

Outer Join

The best Test batsmen in the world at the beginning of 2010 are given in the table below.

Player	Span	Matches	Innings	TRuns	Ave	T100s
SR Tendulkar	1989-2010	162	265	12970	54.72	43
BC Lara	1990-2006	131	232	11953	52.88	34
RT Ponting	1995-2010	140	236	11550	55.26	38
AR Border	1978-1994	156	265	11174	50.56	27
SR Waugh	1985-2004	168	260	10927	51.06	32
R Dravid	1996-2010	137	237	11256	53.60	28
JH Kallis	1995-2010	133	225	10479	54.57	32
SM Gavaskar	1971-1987	125	214	10122	51.12	34
GA Gooch	1975-1995	118	215	8900	42.58	20
Javed Miandad	1976-1993	124	189	8832	52.57	23

Outer Join

The natural join command finds matching rows from the two tables that are being joined and rejects rows that do not match.

The SQL query to find the total number of runs scored and the number of centuries by each player in ODI matches and Test matches is given below.

```
SELECT ORuns, 100s, Player, TRuns, T100s
FROM bestODIbatsmen as b1, bestTestbatsmen as b2
WHERE b1.Player = b2.Player
```

Outer Join

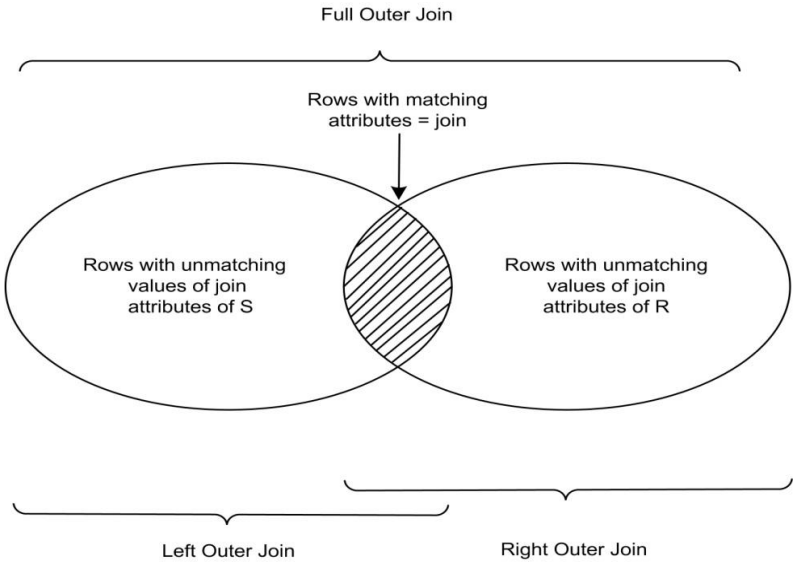
The result of the last query is given below.

ORuns	100s	Player	TRuns	T100s
17394	45	SR Tendulkar	12970	43
12311	28	RT Ponting	11550	38
10405	19	BC Lara	11953	34
10765	12	R Dravid	11256	28
10409	16	JH Kallis	10479	32

Outer Join

There are three groups of rows from the two relations *R* and *S*:

- Rows from both relations that have matching join attribute values
- Rows from relation *S* that did not match rows from *R*
- Rows from relation *R* that did not match rows from *S*



Left Outer Join

(Q34) Find the left outer join of the two tables given before and find the total number of runs scored, total number of centuries, and the strike rate in the ODIs for each player.

SELECT *ORuns*+*TRuns* as TR, 100s+T100s as TC, SR, Player
From *bestODIbatsmen* AS b1 LEFT OUTER JOIN
bestTestbatsmen AS b2
ON *b1.Player* = *b2.Player*

Result of Query 34

The result shows additional information about players.

TR	TC	SR	Player
30364	88	85.90	SR Tendulkar
23861	66	80.50	RT Ponting
22358	53	79.51	BC Lara
22021	40	71.17	R Dravid
20888	48	72.01	JH Kallis
NULL	NULL	91.22	ST Jayasuriya
NULL	NULL	74.24	Inzamam-ul-Haq
NULL	NULL	73.70	SC Ganguly
NULL	NULL	96.94	AC Gilchrist
NULL	NULL	75.30	Mohammed Yusuf

Full Outer Join Query

(Q35) Find the full outer join of the two tables and find the number of runs scored in ODIs and tests, total number of centuries in ODIs and tests, and the strike rate in the ODIs for each player.

The query may be formulated similar to the other outer joins.

SELECT ORuns, 100s, SR, Player, TRuns, T100s
FROM *bestODIbatsmen* as b1 FULL OUTER JOIN *bestTestbatsmen* as b2
ON b1.Player = b2.Player

Results of Full Outer Join Query

ORuns	100s	SR	Player	TRuns	T100s
17394	45	85.90	SR Tendulkar	12970	43
12311	28	80.50	RT Ponting	11550	38
10405	19	79.51	BC Lara	11953	34
10765	12	71.17	R Dravid	11256	28
10409	16	72.01	JH Kallis	10479	32
13428	28	91.22	ST Jayasuriya	NULL	NULL
11739	10	74.24	Inzamam-ul-Haq	NULL	NULL
11363	22	73.70	SC Ganguly	NULL	NULL
9619	16	96.94	AC Gilchrist	NULL	NULL
9495	15	75.30	Mohammed Yousuf	NULL	NULL
NULL	NULL	NULL	AR Border	11174	27
NULL	NULL	NULL	SR Waugh	10927	32
NULL	NULL	NULL	SM Gavaskar	10122	34
NULL	NULL	NULL	GA Gooch	8900	20
NULL	NULL	NULL	Javed Miandad	8832	23

Views – Using Virtual Tables in SQL

The result of any SQL query is itself a table.

In normal query sessions, a query is executed and a table is materialized, displayed and, once the query is completed, discarded. In some situations it may be convenient to store the query definition as a definition of a table that could be used in other queries. Such a table is called a *view* of the database.

The real tables in the database are called *base tables*.

Since a view is a virtual table, it is automatically updated when the base tables are updated. The user may in fact also be able to update the view if desired, and update the base tables used by the view, but, as we shall see later, not all views can be updated.

Views – Using Virtual Tables in SQL

The facility to define views is useful in many ways.

It is useful in controlling access to a database. Users may be permitted to see and manipulate only that data which is visible through some views.

It also provides logical independence in that the user dealing with the database through a view does not need to be aware of the tables that exist since a view may be based on one or more base tables.

If the structure of the base tables is changed (e.g. a column added or a table split in two), the view definition may need changing but the user's view will not be affected.

Defining Views

Views may be defined in a very simple way in SQL. As an example of view definition we define *Batsmen* as follows.

```
CREATE VIEW Batsmen (PID, FName, LName, Country, MID, Score)
AS SELECT PlayerID, FName, LName, Country, MatchID, NRuns
FROM Player, Batting
WHERE PlayerID = PID
```

Another example is given below.

```
CREATE VIEW Bowling2689 (PID, FName, LName, Country, NOvers,
NWickets)
AS SELECT PlayerID, FName, LName, Country, NOvers, NWickets
FROM Player, Bowling
WHERE PlayerID = PID
AND MatchID = '2689'
```

Querying Views

Views may be used in retrieving information as if they were base tables although views do not exist physically. When a query uses a view instead of a base table, the DBMS retrieves the view definition from meta-data and uses it to compose a new query that would use only the base tables. This query is then processed.

(Q41) Find the names of all players who have scored centuries.

This query may be formulated in SQL using a view as below. In this case the view *Batsmen* defined earlier is used.

```
SELECT FName, LName
FROM Batsmen
WHERE Score ≥ 100
```

Querying Views

(Q42) Find the names of bowlers that have taken five wickets or more in the ODI match 2689.

We show below how this query may be formulated in SQL using a view. In this case the view *Bowling2689* defined earlier is being used.

SELECT FName, LName
FROM *Bowling2689*
WHERE NWickets ≥ 5

Transforming Queries

The queries on last two slides using views are transformed before executing them as follows:

SELECT *FName*, *LName*
FROM *Player, Batting*
WHERE *PlayerID* = *PID*;
WHERE *NRuns* ≥ 100

SELECT *FName*, *LName*
FROM *Player, Bowling*
WHERE *PlayerID* = *PID*
AND *MatchID* = '2689'
AND *NWickets* ≥ 5

Multiple Aggregations

Finally, this query illustrates the use of multiple aggregations using views.

(Q43) Find the country that has the maximum number of players in the database.

```
CREATE VIEW NPlayers (Country, Count (*) as C)
AS SELECT Country, COUNT (*)
FROM Player
GROUP BY Country

SELECT Country, COUNT(*)
FROM Player
GROUP BY Country
 HAVING COUNT(*) =
            (SELECT MAX(C)
            FROM NPlayers)
```

Updating Views

There are a number of interesting questions that arise regarding views since they are virtual tables. For example:

- Can queries use views as well as base tables?
- Can views be updated?
- Can rows be inserted or deleted in a view in spite of views being virtual tables?

The answer to the first of the above questions is yes. There is no difficulty in using views and base tables in the same query as the query is going to be transformed into a query that uses base tables anyway.

Updating Views

The answer to the second and third question is "maybe"! Essentially if the view definition is such that a row in a view can directly identify one or more rows in the base tables then the views may be updated and rows inserted in the view. It means the view must have a primary key of one or more base tables. The following points are worth noting:

- A view to be modified must not contain any aggregate functions although a subquery used in the view may.
- The view to be modified must not use the DISTINCT.
- The view must not include calculated columns.
- A view may be deleted in a similar fashion to deleting a table.

SQL - Data Control

Data control facilities include:

(a) recovery and concurrency
(b) security, and
(c) integrity

SQL includes support for the transaction concept. A transaction is a sequence of operations that is guaranteed to be atomic for recovery and concurrency purposes. COMMIT and ROLLBACK commands are available for transaction termination.

SQL - Data Control

Security is provided via the VIEW mechanism and the GRANT operation.

To perform a given operation on a given object, the user must hold the necessary privilege for that operation and object. The privileges are SELECT, UPDATE, DELTE and INSERT.

Data Control (cont.)

The owner of a base table holds all privileges on that table.

The owner of an object can grant privileges on that object to other users by using GRANT command.

GRANT INSERT, DELETE ON *student* to STU10

GRANT SELECT ON *enrolment* to FAC12

A person who has been granted some privileges may pass them on to some others. Privileges may be revoked. Revokes cascade.

SQL Triggers

- Objective: to monitor a database and take initiate action when a condition occurs
- Triggers are expressed in a syntax similar to assertions and include the following:
  - Event
    - Such as an insert, deleted, or update operation
  - Condition
  - Action
    - To be taken when the condition is satisfied

SQL Triggers: An Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
   SALARY, SUPERVISOR_SSN ON EMPLOYEE
   FOR EACH ROW
       WHEN
       (NEW.SALARY> (SELECT SALARY FROM EMPLOYEE
                     WHERE SSN=NEW.SUPERVISOR_SSN))
       INFORM_SUPERVISOR
  (NEW.SUPERVISOR_SSN,NEW.SSN);
```

(a) R1:
```
CREATE TRIGGER Total_sal1
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.Dno IS NOT NULL )
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + NEW.Salary
    WHERE Dno = NEW.Dno;
```

R2:
```
CREATE TRIGGER Total_sal2
AFTER UPDATE OF Salary ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.Dno IS NOT NULL )
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + NEW.Salary – OLD.Salary
    WHERE Dno = NEW.Dno;
```

R3:
```
CREATE TRIGGER Total_sal3
AFTER UPDATE OF Dno ON EMPLOYEE
FOR EACH ROW
    BEGIN
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + NEW.Salary
    WHERE Dno = NEW.Dno;
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal – OLD.Salary
    WHERE Dno = OLD.Dno;
    END;
```

```
R4:  CREATE TRIGGER Total_sal4
     AFTER DELETE ON EMPLOYEE
     FOR EACH ROW
     WHEN ( OLD.Dno IS NOT NULL)
          UPDATE DEPARTMENT
          SET Total_sal = Total_sal – OLD.Salary
          WHERE Dno = OLD.Dno;
```

```
(b)  R5:  CREATE TRIGGER Inform_supervisor1
          BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn
               ON EMPLOYEE
          FOR EACH ROW
          WHEN ( NEW.Salary > ( SELECT Salary FROM EMPLOYEE
                                WHERE Ssn = NEW.Supervisor_ssn ) )
               inform_supervisor(NEW.Supervisor_ssn, NEW.Ssn );
```