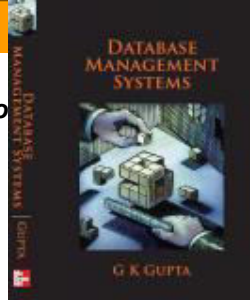
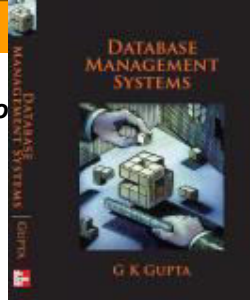
Mc Graw Hill Education

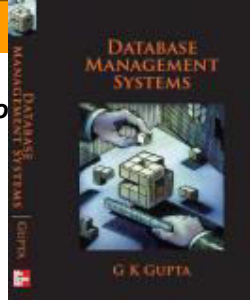


DATABASE MANAGEMENT SYSTEMS

G K GUPTA

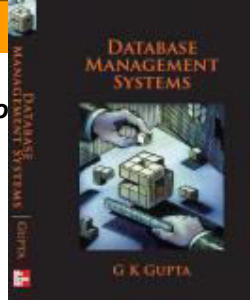


Chapter 5
Query Language SQL
Part 1



Chapter Objectives

- to explain how SQL is used for data definition,
- to describe how SQL is used for retrieving information from a database,
- to explain how SQL is used for inserting, deleting and modifying information in a database,
- to explain the use of SQL in implementing correlated and complex queries including outer joins,
- to discuss embedded SQL and dynamic SQL
- to discuss how data integrity is maintained in a relational database using SQL

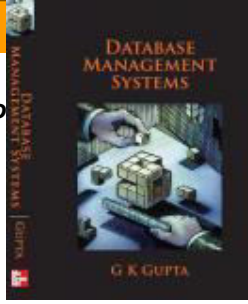


Evolution of SQL

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.

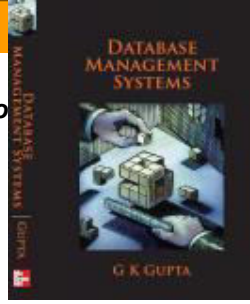
It is now a de facto standard for relational database query languages. The American Standards Association has adopted a standard definition of SQL.



SQL

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

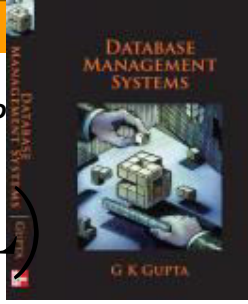
SQL provides facilities for data definition as well as for data manipulation. In addition, the language includes some data control features. We briefly describe them.



Mc Graw Hill Education

Data Definition Language (DDL)

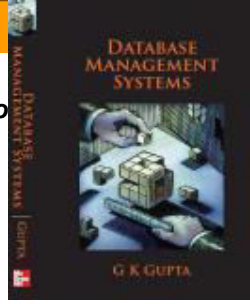
Before a database can be used, it must be created. DDL is used for defining the database to the computer system. It includes facilities for creating a new table or a view, deleting a table or a view, altering or expanding the definition of a table and creating an index on a table, as well as commands 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.



Education

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.

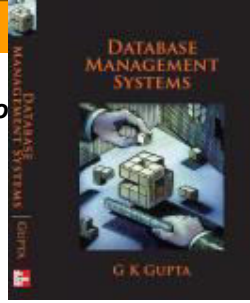


Education

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.

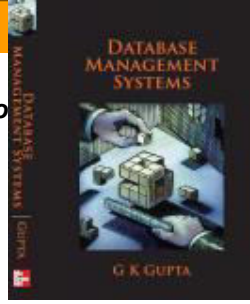
New Features – Each new standard has introduced a variety of new features in the language. For example, SQL:2003 includes XML. Unfortunately we will not be able to cover these features in this book.



Education

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)

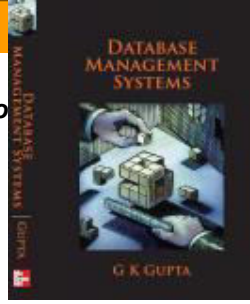
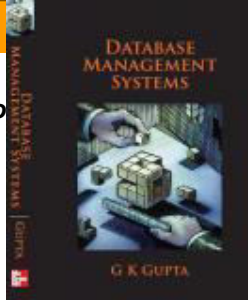


Table Definition

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))

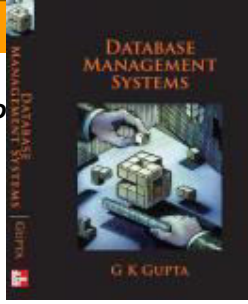
In the above definition, we have specified that the attribute *MatchID* is the primary key of the relation.



SQL Data Types

Basic SQL data types are:

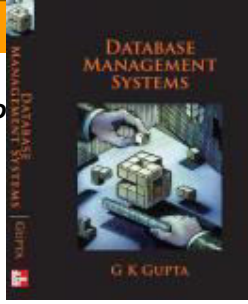
- Boolean
- Character – Char or Varchar
- Exact Numeric – Numeric, decimal, integer, smallint
- Approximate Numeric – float, real, double precision
- Datetime – Date, time
- Interval
- Large objects – Character large objects, binary large objects (BLOBS)



Numeric Data Types

The SQL numeric data types are:

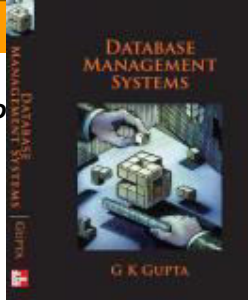
Data Type	Description
SMALLINT	Integer, precision 5
INTEGER	Integer, precision 10
REAL	Mantissa about 7
FLOAT(p)	Mantissa about 16
DOUBLE PRECISION	Same as FLOAT



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.

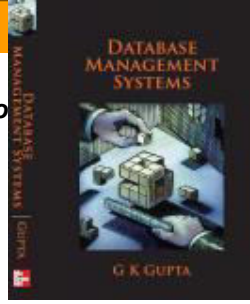


Mc Graw Hill Education

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].	The length is 26.
INTERVAL	Used to represent time intervals like 9 days or 20 hours and 37 minutes.	Can be YEAR-MONTH or DAY-TIME.	

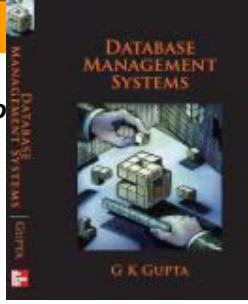


Basic Data Retrieval

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.



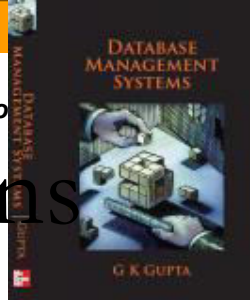
SQL Aliases

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.

```
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

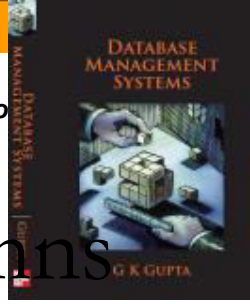
The WHERE clause specifies the condition to be used in selecting rows and is optional, if it is not specified the whole table is selected.

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
```



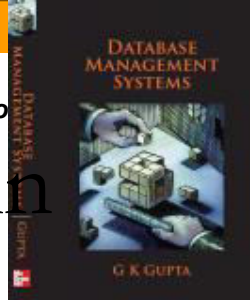
Education

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.

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



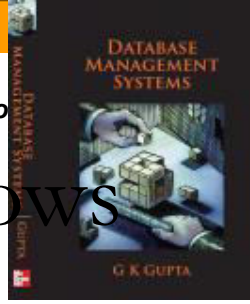
Retrieving All Information about Certain Rows from One Table

(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
```



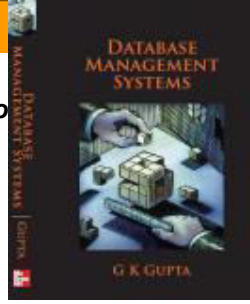
Mc Graw Hill Education

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'
```



More about SELECT

Calculations in SELECT

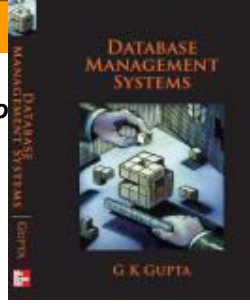
Some calculations are possible in SELECT. For example it is possible to write:

```
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
```

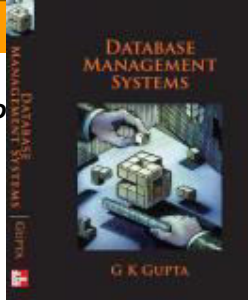


Simple Exercises

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

The condition in this query is slightly more complex.

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

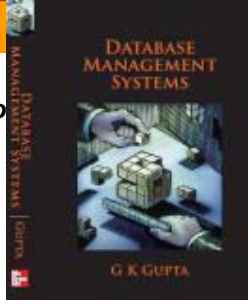


Removing Duplicates

(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*



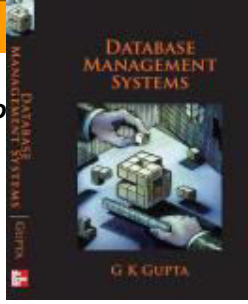
Removing Duplicates

(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'
```



Mc Graw Hill Education

Sorting the Information Retrieved

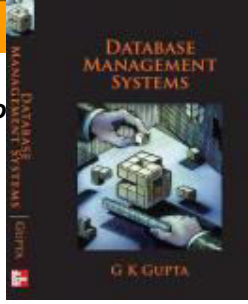
The ORDER BY clause returns results in ascending order (that is the default).

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.

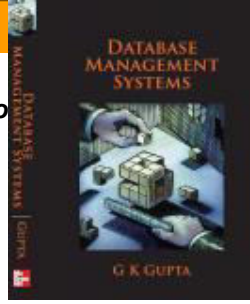


Sorting

(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*



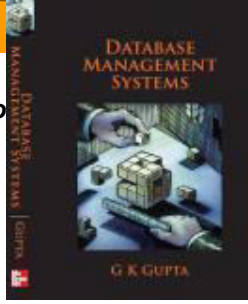
String Matching

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 (%).

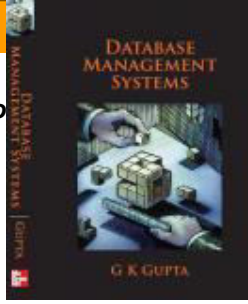
They have special meanings; underscore represents any single character while percent represents any sequence of *n* characters including a sequence of no characters.



String Matching

There are a number of string operations available in SQL. These are:

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

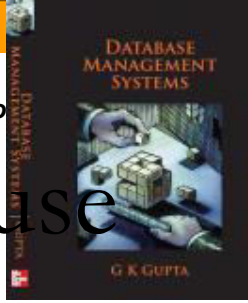


String Matching

(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%'

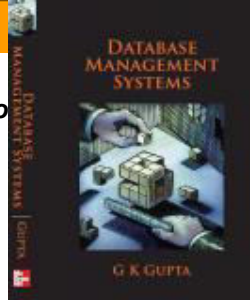


Different Forms of the WHERE Clause

Some of the common forms are:

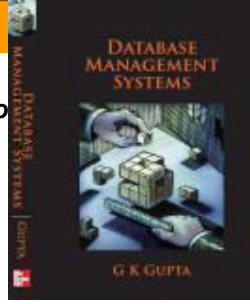
- WHERE C1 AND C2 – each selected row must satisfy both the conditions C1 and C2.
- WHERE C1 OR C2 – each selected row must satisfy either the condition C1 or the condition C2 or both conditions C1 and C2.

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.



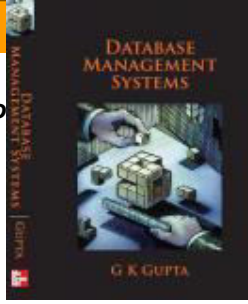
WHERE clause

- 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 (<>).



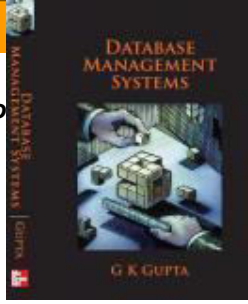
WHERE clause

- 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.



Mc Graw Hill Education

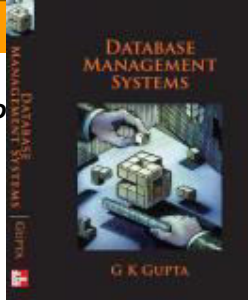
Queries Involving More than One Table – Using Joins and Subqueries

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.

- join
- subqueries

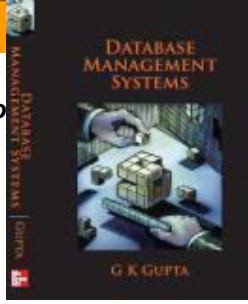


Subqueries

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.

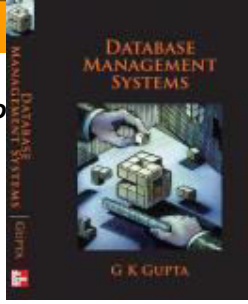


Subqueries

(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')
```



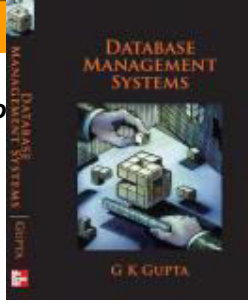
Subqueries

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

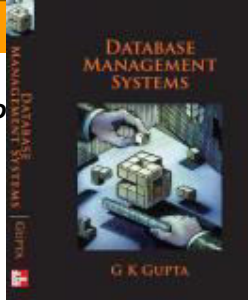


Subqueries

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

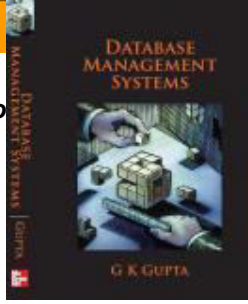
This query must involve the three tables *Match*, *Batting* and *Player* as the match information is in *Match*, the player names are in *Player* and batting information is available only in table *Batting*.

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



Result of Q13

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

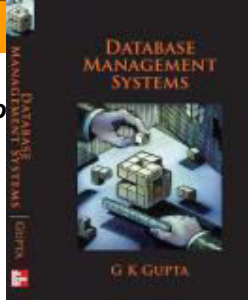


Queries Involving Joins

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'
```

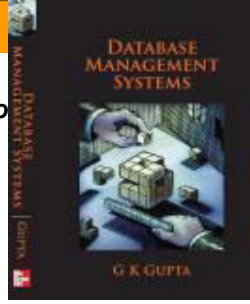


Queries Using Joins

(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
```

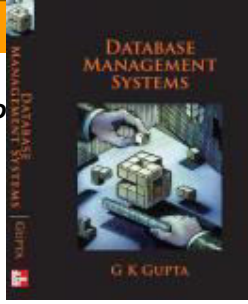


More Subqueries

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

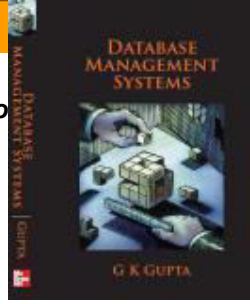
The query needs both the tables *Batting* and *Bowling*. The formulation below uses the subquery to find player IDs of players that bowled in match 2689 and the outer subquery checks they have also batted.

```
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

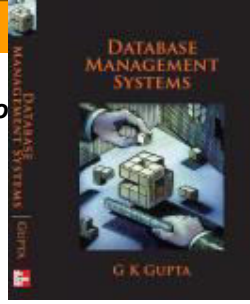


Subqueries

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

Q16 is similar to Q15, again requiring both tables *Batting* and *Bowling*. In this case, the outer query finds players that batted in match 2689 or if they are on the list of players that bowled in the match.

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

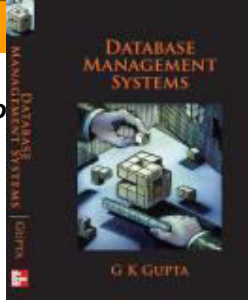


Subqueries

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

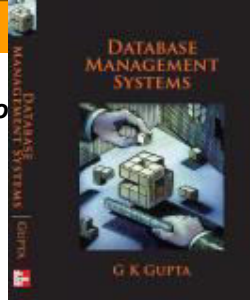
The query Q16 may also be written using the UNION operator as given below.

```
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



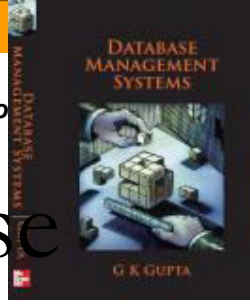
Queries Involving Subqueries

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

The SQL query below shows one way this query may be formulated. The result is shown.

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

PID
89001
25001
99002
95001
27001



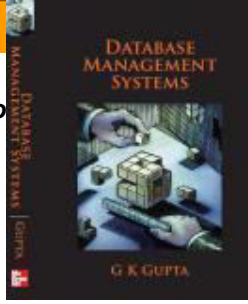
Education

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.

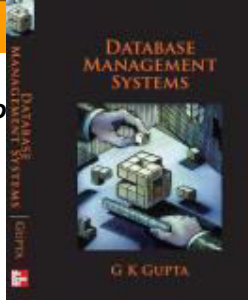


Subqueries

(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

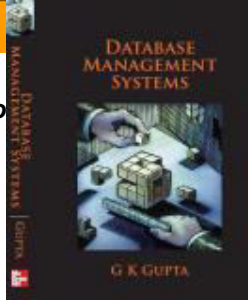


Subqueries

(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 ≥ x and A ≤ y.

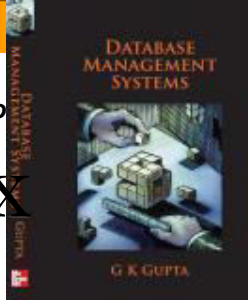


NULLs

(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

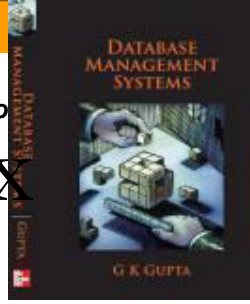


Correlated Subqueries and Complex Queries

The subqueries used so far were needed 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.



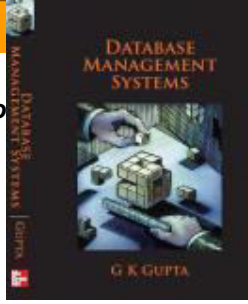
Correlated Subqueries and Complex Queries

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

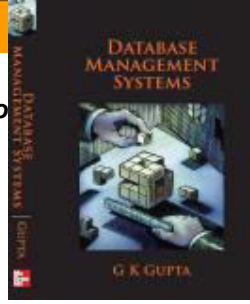
This query needs both *Player* and *Batting* tables because player names are available only in the table *Player* while batting information is given in table *Batting*. In this query we look at each player in the table *Player* and then check, using a subquery, if there is a batting record for him in the table *Batting*.

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

Result of Q22



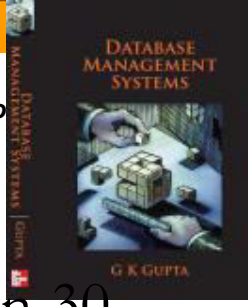
FName	LName
Sachin	Tendulkar
M. S.	Dhoni
Yuvraj	Singh
Adam	Gilchrist
Andrew	Symonds
Brett	Lee
Praveen	Kumar
Ricky	Ponting
Harbhajan	Singh



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'
```

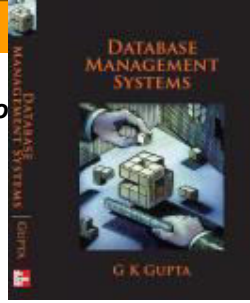


Complex Query

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

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"!

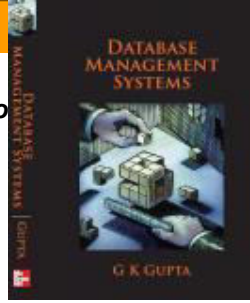
```
SELECT PID AS PlayerID
FROM Batting b1
WHERE NOT EXISTS
                (SELECT *
                FROM Batting b2
                WHERE b1.PID = b2.PID
                AND NRuns < 31)
```



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.



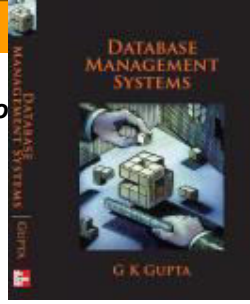
Complex Query

(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 we are asking and this time it becomes quite ugly with a double negative. 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"!

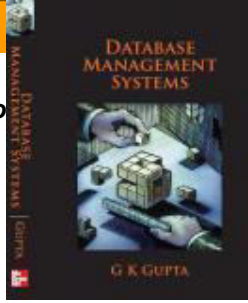
The SQL query that does it is shown on the next slide.



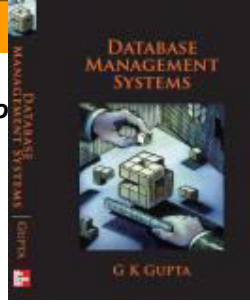
Complex Query Q24

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))
```

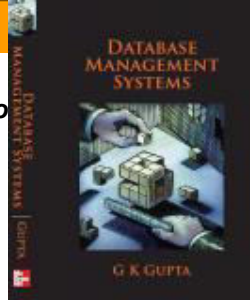


Mc Graw Hill Education

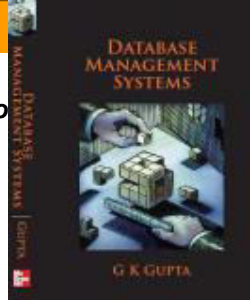


DATABASE MANAGEMENT SYSTEMS

G K GUPTA



Chapter 5
Query Language SQL
Part 2

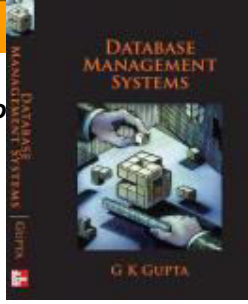


Using Built-in Functions

SQL provides a number of built-in functions. These functions are

- AVG
- COUNT
- MIN
- MAX
- SUM

The AVG function is for computing the average, COUNT counts the occurrences of rows, MIN finds the smallest value, MAX finds the largest value while SUM computes the sum.



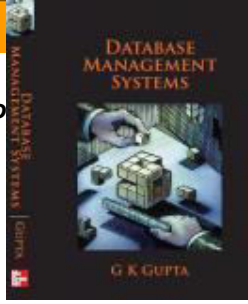
Using Functions

(Q25) Find the number of players that bowled in the ODI match 2689.

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

The result is given below.

NBowlers
4



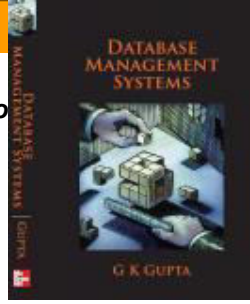
Using Functions

(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'
```

The result of Q26 is given below.

AveRuns_2689
25.2



Using Functions

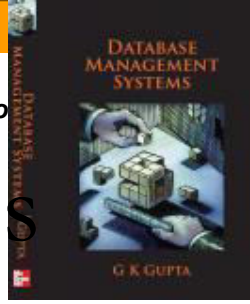
Q(27) Find the youngest player in the database.

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)
```

The result of Q27 is given below. It is not a correlated query.

Youngest_Player
Sharma

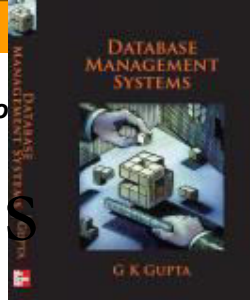


Mc Graw Hill Education

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.



Mc Graw Hill Education

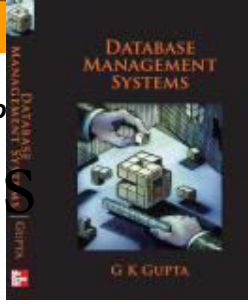
GROUP BY and HAVING Clauses

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.

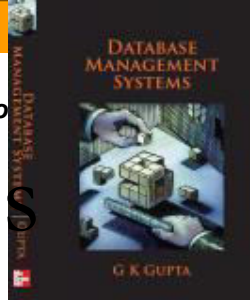
The WHERE clause in the query applies to the table before GROUP BY and HAVING and therefore results in a rows pre-selection.



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



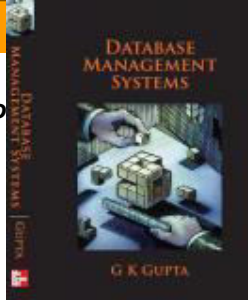
Mc Graw Hill Education

GROUP BY and HAVING Clauses

(Q28) Find the number of players in the database from each country.

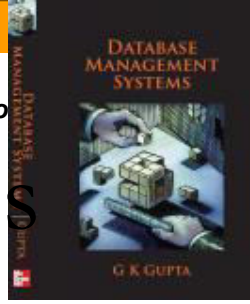
To find the number of players from each country, we need to partition the Player table to create virtual sub-tables for players from each country. Once the table has been so partitioned, we can count them.

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



Result of Q28

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

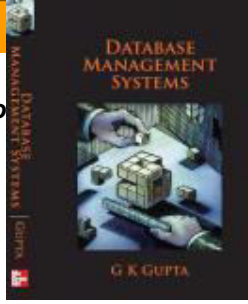
(Q29) Find the batting average of each player in the database. Present the PID of each player.

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
```

The result of Q29 is given on the right.

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



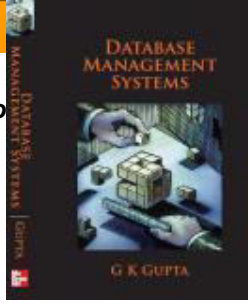
Mc Graw Hill Education

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.

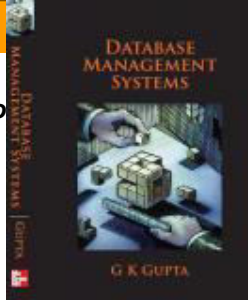
SELECT *FName*, AVE(NRuns) AS AveRuns
FROM *Player*, *Batting*
WHERE PlayerID = PID
AND Country = 'India'
GROUP BY *LName*



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



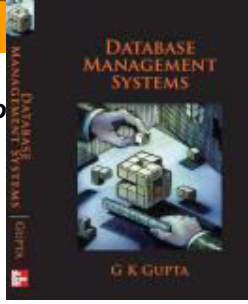
Mc Graw Hill Education

GROUP BY and HAVING

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

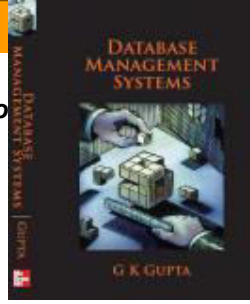
```
SELECT PID AS PlayerID, AVG(NRuns) AS AveRuns
FROM Batting
WHERE MatchID IN
        (SELECT MatchID
        FROM Match
         WHERE Team1 = 'Australia')
GROUP BY PID
```

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

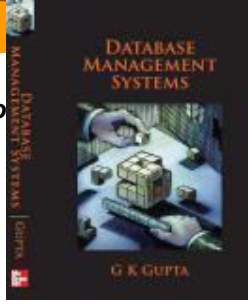
PlayerID	AveRuns
89001	91
23001	38
25001	36
99002	2
95001	1
24001	42
99001	7
27001	7
98002	3



GROUP BY and HAVING

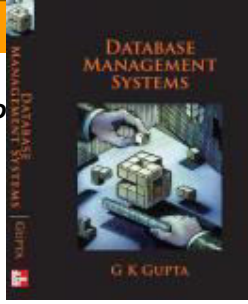
(Q32) Find the ID of players that had a higher average score tha n the average score for all players when they played in Sri Lanka.

```
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



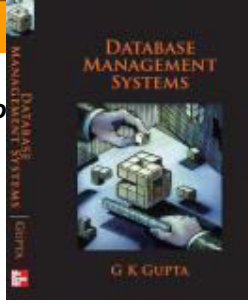
Education

Multiple Aggregate Functions

(Q33) In query Q29, the number of players in the database from each country was found. This query requires that the average number of players from that list be found.

The figure below shows how one might formulate this query. The formulation is however incorrect.

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



Education

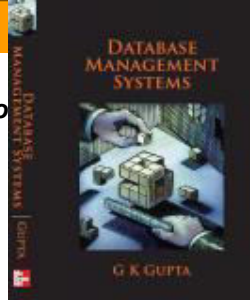
Multiple Aggregate Functions

One approach is to use a subquery in the FROM clause. This subquery then produces the table that we got in Q29. Once we have that table, we may compute the average as shown below.

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

The result is given below.

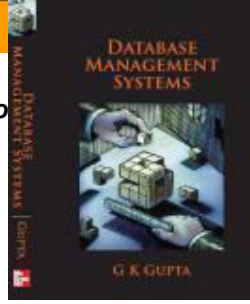
NAvePlayers
3.2



Outer Join

Outer join was discussed in Chapter 4. Now we present an example using SQL to illustrate the concept. Consider the two tables, the first giving best ODI batsmen and another best test batsmen. Both on 1 Jan 2010.

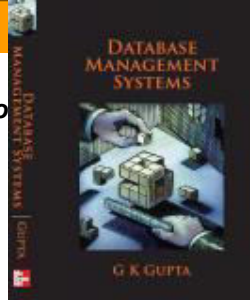
Player	Span	Matches	Innings	Runs	Ave	Strike Rate	100s
SR Tendulkar	1989-2010	440	429	17394	44.71	85.90	45
ST Jayasuriya	1989-2010	444	432	13428	32.43	91.22	28
RT Ponting	1995-2010	330	321	12311	43.19	80.50	28
Inzamam-ul-Haq	1991-2007	378	350	11739	39.52	74.24	10
SC Ganguly	1992-2007	311	300	11363	41.02	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.

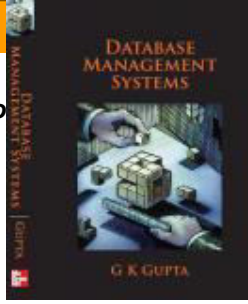
Player	Span	Matches	Innings	TRuns	Ave	T100s
SR Tendulkar	1989-2010	162	265	12970	54.72	43
BC Lara	1990-2006	131	232	11953	52.88	34
RT Ponting	1995-2010	140	236	11550	55.26	38
AR Border	1978-1994	156	265	11174	50.56	27
SR Waugh	1985-2004	168	260	10927	51.06	32
R Dravid	1996-2010	137	237	11256	53.60	28
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

As noted earlier 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.

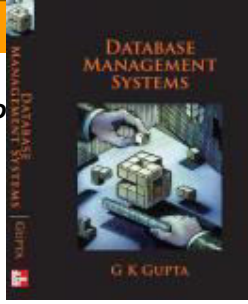
```
SELECT ORuns, 100s, Player, TRuns, T100s
FROM bestODIbatsmen as b1, bestTestbatsmen as b2
WHERE b1.Player = b2.Player
```



Outer Join

The result of the last query is given below. The figure on the next slide explains the concept of outer joins. The figure shows the distribution of join attribute values for two relations *R* and *S*.

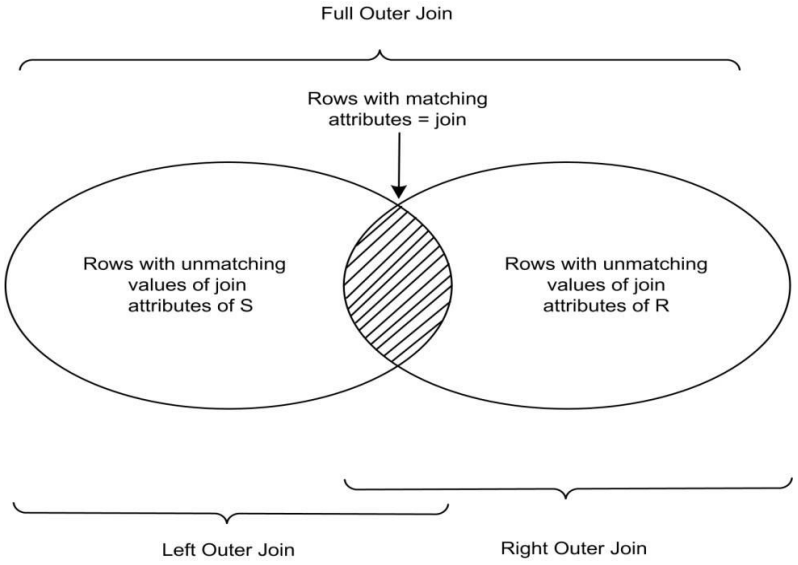
ORuns	100s	Player	TRuns	T100s
17394	45	SR Tendulkar	12970	43
12311	28	RT Ponting	11550	38
10405	19	BC Lara	11953	34
10765	12	R Dravid	11256	28
10409	16	JH Kallis	10479	32

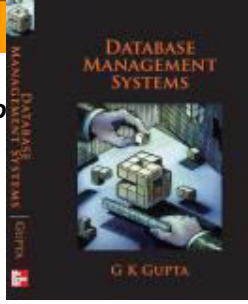


Outer Join

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*

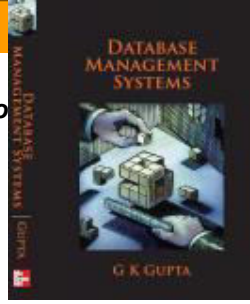




Left Outer Join

(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.

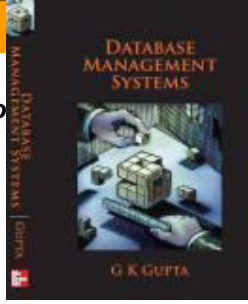
SELECT *ORuns*+*TRuns* as TR, 100s+T100s as TC, SR, Player
From *bestODIbatsmen* AS b1 LEFT OUTER JOIN
bestTestbatsmen AS b2
ON *b1.Player* = *b2.Player*



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

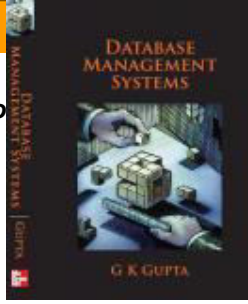


Full Outer Join Query

(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.

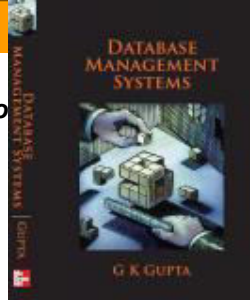
SELECT ORuns, 100s, SR, Player, TRuns, T100s
FROM *bestODIbatsmen* as b1 FULL OUTER JOIN *bestTestbatsmen* as b2
ON b1.Player = b2.Player



Education

Results of Full Outer Join Query

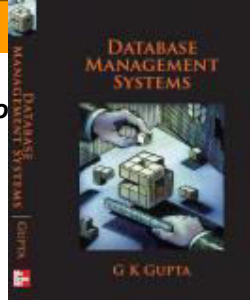
ORuns	100s	SR	Player	TRuns	T100s
17394	45	85.90	SR Tendulkar	12970	43
12311	28	80.50	RT Ponting	11550	38
10405	19	79.51	BC Lara	11953	34
10765	12	71.17	R Dravid	11256	28
10409	16	72.01	JH Kallis	10479	32
13428	28	91.22	ST Jayasuriya	NULL	NULL
11739	10	74.24	Inzamam-ul-Haq	NULL	NULL
11363	22	73.70	SC 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



Outer Join Results

The result shows that there are a total of 15 batsmen that are either in the top ten list of ODI batsmen or in the top ten list of test batsmen or both. Only five of them are in both the lists. These players are Tendulkar, Ponting, Lara, Dravid and Kallis and they are, in my view, the batting superstars. Since three of these five players are still playing, this table might look a bit different by the time you are reading it.

There is very little chance of new players appearing in both the two top ten lists since only a small number of players are close to the bottom of these lists at the time of writing. S. Chanderpaul of the West Indies is the only one who is still playing and has 8576 test runs and 8250 ODI runs.



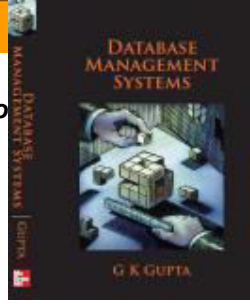
SQL Update Commands

- Adding new data – adding new rows using the INSERT command
- Modifying existing data – modifying column data using the UPDATE command
- Deleting data – removing rows using the DELETE command

(Q36) Insert a new player Michael Clarke with player ID 200311 in the table *Player*.

INSERT INTO Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest) <200311, 'Clarke', 'Michael', NULL, NULL, NULL, NULL>

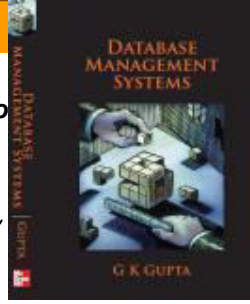
The above insertion procedure is somewhat tedious since it requires all column names to be listed and all corresponding matching values specified.



Update

INSERT can be made a little less tedious if the list of column values that are to be inserted are in the same order as specified in the CREATE TABLE definition and all the column values are present as shown below.

INSERT INTO Player <200311, 'Clarke', 'Michael', NULL, NULL, NULL, NULL>



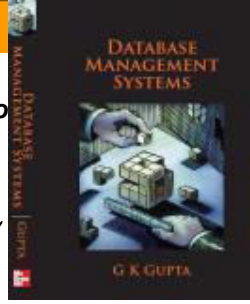
Update – Modifying Existing Data

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

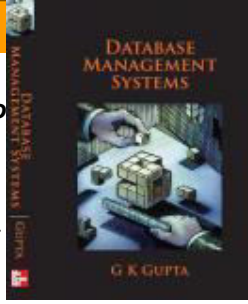
(Q38) Increase the marks of all students in CS100 by 5.

```
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')
```

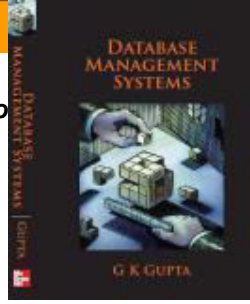
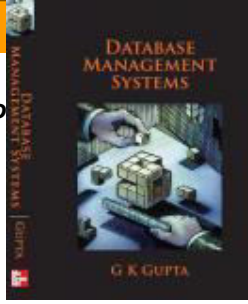


Table Definition

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))

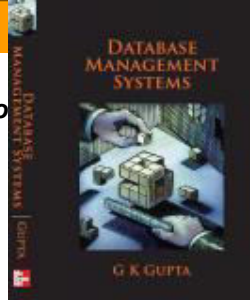
In the above definition, we have specified that the attribute *s-id* may not be NULL because it is the primary key of the relation.



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)



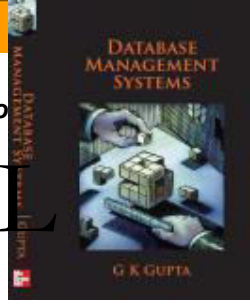
Change Table Definition

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



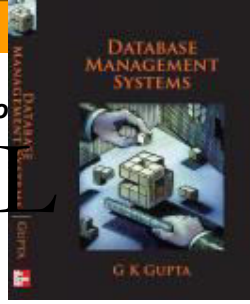
Views – Using Virtual Tables in SQL

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.

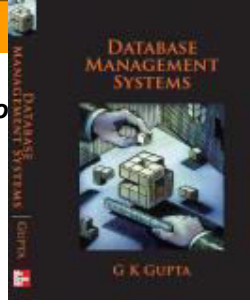


Views – Using Virtual Tables in SQL

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.



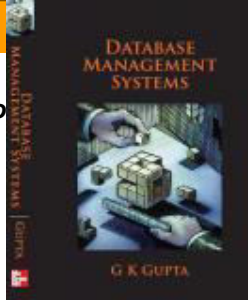
Defining Views

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'
```



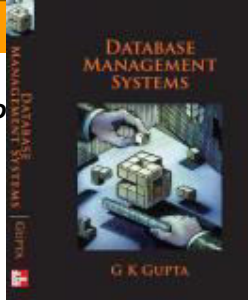
Querying Views

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
```

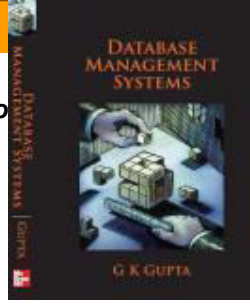


Querying Views

(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
```

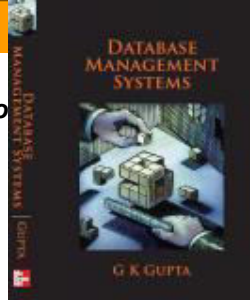


Transforming Queries

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

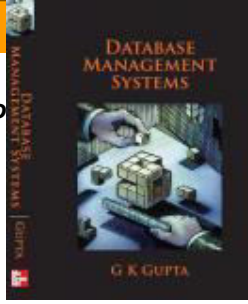


Multiple Aggregations

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)
```

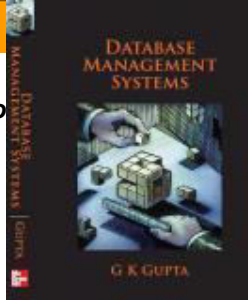


Updating Views

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?

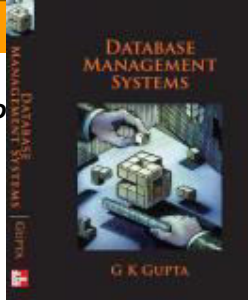
The answer to the first of the above questions is yes. There is no difficulty in using views and base tables in the same query as the query is going to be transformed into a query that uses base tables anyway.



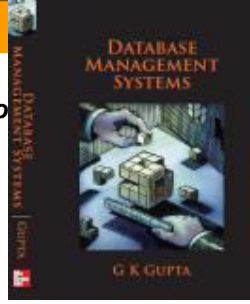
Updating Views

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.

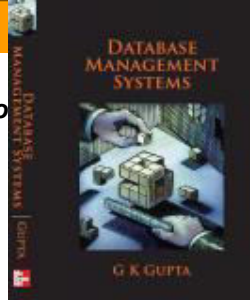


Mc Graw Hill Education

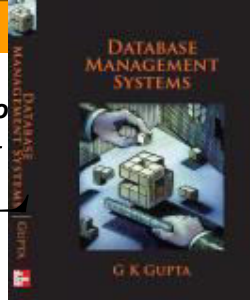


DATABASE MANAGEMENT SYSTEMS

G K GUPTA



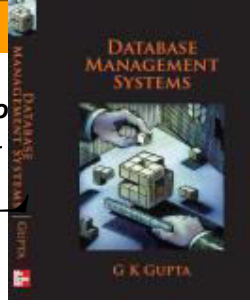
Chapter 5
Query Language SQL
Part 3



Embedded SQL and Dynamic SQL

Embedded SQL programming is useful when a previously written program in a procedural language needs to access a database or because SQL is not providing the facilities, for example, it has no looping facilities like IF..THEN..ELSE.

On the other hand using SQL in a language like C or C++ also creates some problems because the procedural languages are not equipped to deal with collections of multiple rows as a single operand. To overcome this difficulty, a facility is needed for stepping through a collection of rows returned by a SQL query, one row at a time. This is done by introducing the concept of a *cursor*.

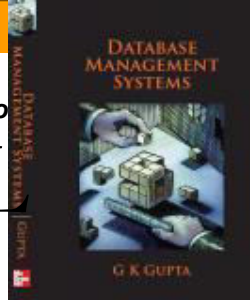


Embedded SQL and Dynamic SQL

It is possible to use SQL from within a program written in almost any of the conventional programming languages.

Since conventional programming languages know nothing about SQL, no database table names may be used in a host language program outside the embedded SQL commands.

An SQL query in a host language can appear anywhere in the host program but it must be identified by beginning it with EXEC SQL command followed by SQL statements or declarations. For example the embedded SQL in a host language may look like the one on the next slide.

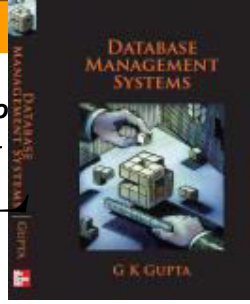


Embedded SQL and Dynamic SQL

```
EXEC SQL
SELECT *
FROM Player
WHERE Country = 'India';
```

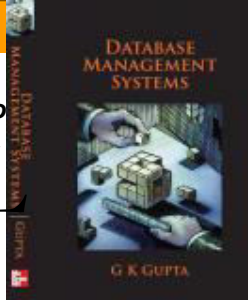
SQL statements in a procedural language are handled by the language pre-processor for that language before the resulting program may be compiled. A pre-processor is therefore required to use SQL in a host language.

SQL variables may be declared within a SQL declaration section by starting the declaration section by EXEC SQL BEGIN DECLARE SECTION and ending it by the command EXEC SQL END DECLARE SECTION.



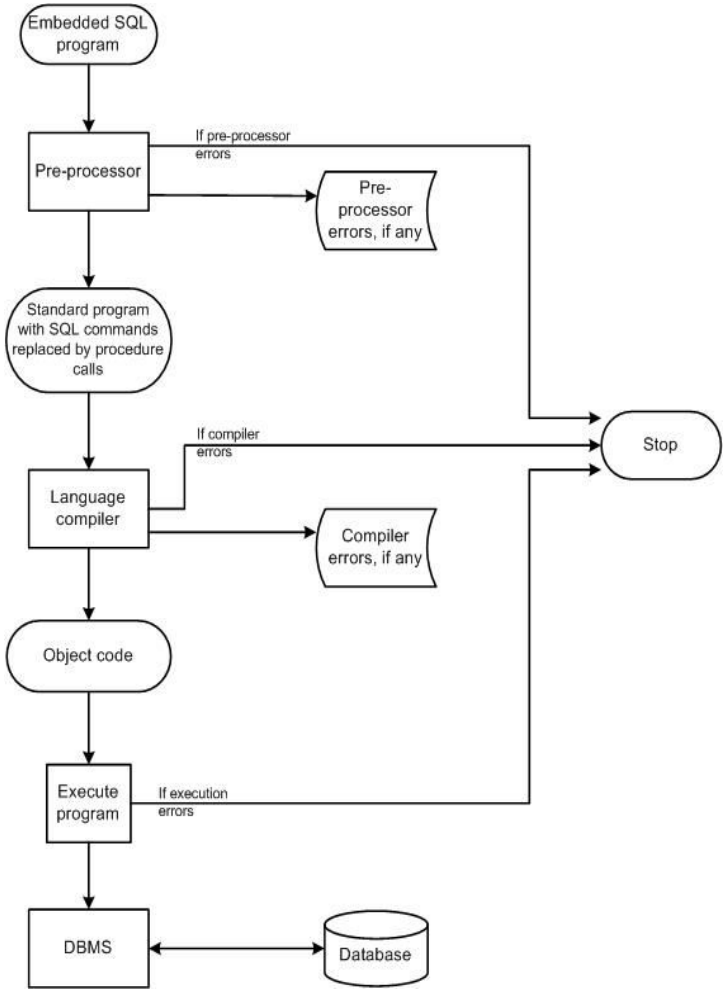
Embedded SQL and Dynamic SQL

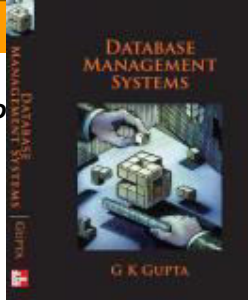
When a program with a SQL query is preprocessed, the SQL query becomes a procedure call of the host language and the program may then be executed. Such a use of SQL is called *embedded SQL*.



Embedded SQL and Dynamic SQL

This process is explained in the
figure that shows the pre-processor phase, followed by the compilation phase and finally the execution phase of the program.



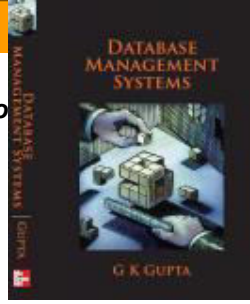


Error Handling

A program in a host programming language knows nothing about the database it uses and thus there is considerable data independence between the program and the database.

To make this interface between the host program and SQL work, the host program needs some communication working storage. A communication area is provided to communicate between the database and the embedded program. This communication area may be accessed in the host program by including the following statement where SQLCA stands for SQL communication area.

```
EXEC SQL INCLUDE SQLCA;
```

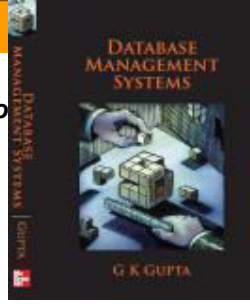


Error Handling

Once SQLCA has been declared, two variables SQLCODE and SQLSTATE may now be used to detect error conditions in an embedded program. All variables used to store database column values that are retrieved must be declared in the host language between the declaration statements as noted earlier.

```
EXEC SQL BEGIN DECLARE SECTION
int PlayerID;
char FName[20], LName[20;
EXEC SQL END DECLARE SECTION
```

The host variables declared here are used in the SQL queries embedded in the host language program. Note that these variables are called shared variables and they are prefixed by a colon (:) when used.

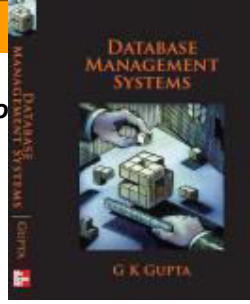


Impedance Mismatch

As noted above, the host language does not have any working storage for SQL results that are tables, usually with more than one row. Some SQL commands like UPDATE, DELETE and INSERT of course do not require any additional storage in the host program as shown below.

Update

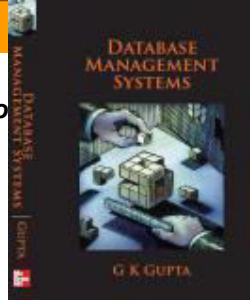
```
EXEC SQL
UPDATE Student
Mark = Mark + 3
Where Student_ID = 876543;
```



SELECT

When a number of rows are selected by an embedded SQL query then the concept of *cursor* must be used. As noted earlier, a cursor is needed since most conventional host programming languages are record-based, while SQL is set-based. This is called *impedance mismatch*.

Therefore we need a binding mechanism that makes it possible to retrieve one record at a time as well as map the attribute values to the host language variables. A cursor (it may be thought of as a pointer) allows us to do that by essentially holding the position of the current tuple of the table that is the result of the SQL query while the host language deals with the table one row at a time. The cursor is not associated with any tables in the database, it is only associated with the table that is the result of an embedded SQL query.



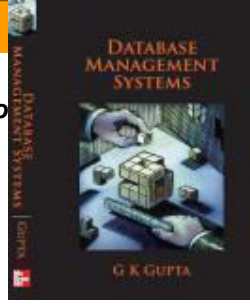
Cursor

Once a cursor is defined, it may be opened to point at just before the first row of the resulting table. There are two other operations available, *fetch* and *move*. Fetch gets the next row of the result while move may be used to move the cursor. Close may be used to close the cursor.

```
EXEC SQL DECLARE cursor1 CURSOR FOR
SELECT PlayerID, FName, LName
FROM Player
WHERE Country = "India";
```

Now we may issue the following commands:

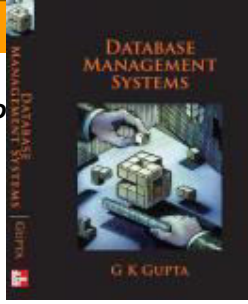
```
OPEN cursor1
FETCH cursor1 INTO :var1, :var2, :var3
MOVE cursor1
CLOSE cursor1
```



Dynamic SQL

Dynamic SQL is a programming technique that enables a program to build SQL statements dynamically at runtime.

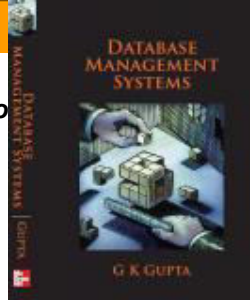
In some cases when using embedded SQL in a host programming language it becomes necessary to use and build the SQL query dynamically at runtime. Building SQL queries dynamically provides the user a flexible environment because the full details of a SQL query may not be known at the time of writing the program. For example, the name of the table that is to be used in the query may not be known prior to running the program.



Dynamic SQL

Dynamic queries are useful where one of the following conditions holds:

- an application program needs to allow users to input or choose query search or sorting criteria at runtime
- an application program queries a database where the data definitions of tables are changing regularly
- an application program queries a database where new tables are being created frequently and these tables are used in the queries

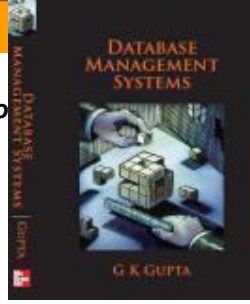


Dynamic SQL

We now describe the difference between static SQL and dynamic SQL.

Static SQL statements do not change from execution to execution. The full static SQL query is known when the query is processed, which provides the following benefits:

- query processing can verify that the SQL query references valid database objects
- query processing can verify that the user has necessary privileges to access the database objects
- performance of static SQL is generally better than that of dynamic SQL

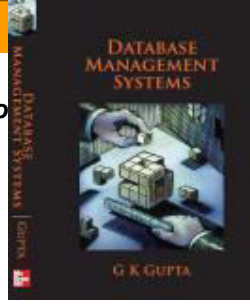


SQL - Data Control

Data control facilities include:

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

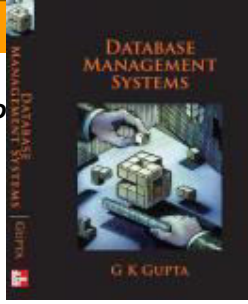
SQL includes support for the transaction concept. A transaction is a sequence of operations that is guaranteed to be atomic for recovery and concurrency purposes. COMMIT and ROLLBACK commands are available for transaction termination.



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.)

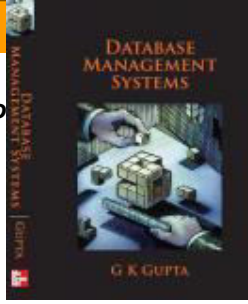
The owner of a base table holds all privileges on that table.

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.

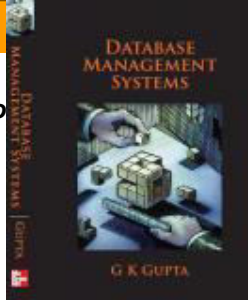


Integrity Control Features

Following integrity constraints are supported:

*NOT NULL - can be specified for any column of the table. Any attempt to introduce a null will be rejected.

*UNIQUE - can be specified for any column or combination of columns of the table. Every such column(s) must also be specified NOT NULL. Any attempt to introduce a row with duplicate values will be rejected.



Integrity Control Features

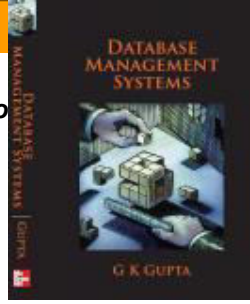
*DOMAIN - can be specified for any column of the table. For example:

CREATE DOMAIN code AS CHAR(5)

CREATE DOMAIN Gender as CHAR
DEFAUL 'F'
CHECK (VALUE IN ('M', 'F'))

Any attempt to introduce any other value will be rejected.

A domain may be dropped.



Integrity Control Features

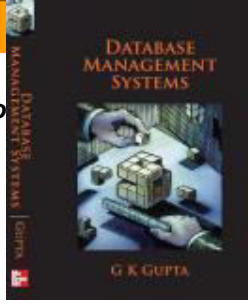
*EXISTENCE – primary key can be specified, for example:

PRIMARY KEY(s-id) is included in Create Table command.

*FOREIGN KEY – foreign key can be specified, for example:

FOREIGN KEY (code) REFERENCES course

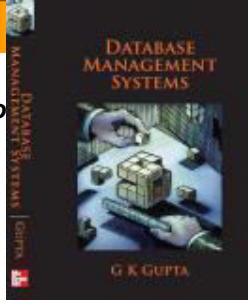
FOREIGN KEY (code) REFERENCES course ON UPDATE SET NULL



Advantages Of SQL

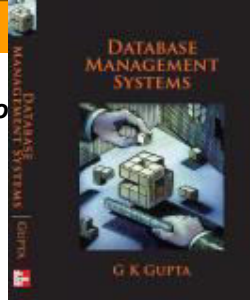
SQL allows data access to be expressed without mentioning or knowing the existence of specific access paths or indexes. Application programs are therefore simpler since they only specify what needs to be done not how it needs to be done.

The DBMS is responsible for selecting the optimal strategy for executing the program.



Advantages Of SQL

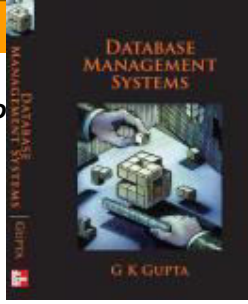
Another advantage of using a very high-level language is that data structures may be changed if it becomes clear that such a change would improve efficiency. Such changes need not affect the application programs.



Advantages Of SQL

In summary

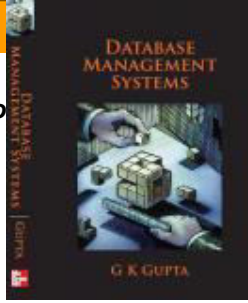
- Simple data structure
- Powerful operators
- Short initial learning period
- Improved data independence
- Integrated data definition and data manipulation
- Double mode of use
- Integrated catalog
- Compilation and optimisation



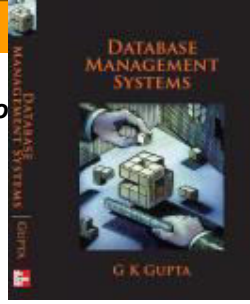
Criticisms OF SQL

SQL is far from perfect!

- It allows duplicate tuples
- It provides very little integrity features
- It shows a lack of orthogonality i.e. distinct concepts are not always clearly separated. For example, arbitrary restrictions are placed on many constructs
- Missing functions

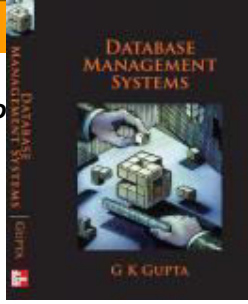


Mc Graw Hill Education

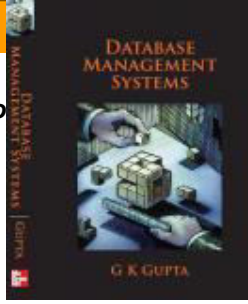


DATABASE MANAGEMENT SYSTEMS

G K GUPTA

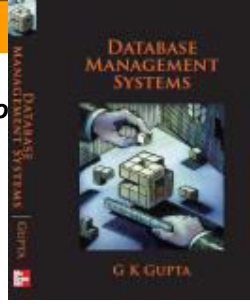


Chapter 6
Normalization



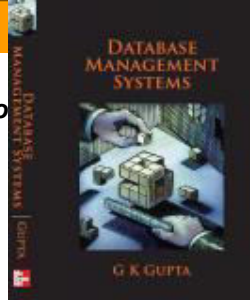
Objectives

- discuss the problems that can occur in a database system due to poor database design
- describe the anomalies that can arise if there is information redundancy
- explain the concept of functional dependencies, closure and minimal cover
- discuss the concept of Normal Forms, in particular the Third Normal Form (3NF) and the Boyce-Codd Normal Form (BCNF) and their purpose



Objectives

- present techniques for normalizing relations
- discuss the desirable properties of decompositions
- describe the concepts of the Fourth Normal Form (4NF) and the Fifth Normal Form (5NF)
- explain the concept of inclusion dependency

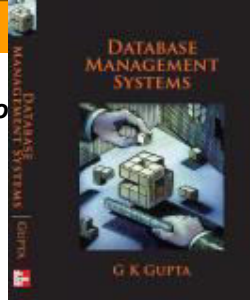


Database design

Mistakes can be made in database modeling especially when the database is large and complex or one may, for some reason, carry out database schema design using techniques other than a modeling technique.

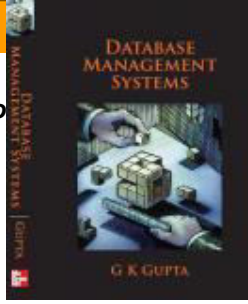
For example, one could collect all the information that an enterprise possesses and use a bottom-up design approach. This approach is likely to lead to a table that suffers from problems like redundant information and update, deletion and insertion anomalies.

Normalization of such large tables will then be essential to avoid (or at least minimize) these problems.



An Example

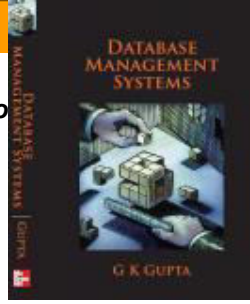
LName	Country	Team1	Team2	MID	Date	Ground	NRuns	NWickets	YBorn	Place
Tendulkar	India	Australia	India	2689	4/3/2008	MCG	91	NULL	1973	Mumbai
Tendulkar	India	Australia	India	2689	4/3/2008	MCG	91	NULL	1973	Mumbai
Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
Sehwag	India	Pakistan	India	2717	26/6/2008	Karachi	119	NULL	1978	Delhi
Gilchrist	Australia	Australia	India	2689	4/3/2008	Gabba	2	NULL	1971	Bellingen
Ponting	Australia	Australia	India	2689	4/3/2008	Gabba	1	NULL	1974	Launces
Lee	Australia	Australia	India	2689	4/3/2008	Gabba	7	1	1976	Wollongg
Jayasuriya	Sri Lanka	Sri Lanka	India	2755	27/8/2008	Colombo	60	0	1969	Matara
ISharma	India	Pakistan	India	2717	26/6/2008	Karachi	NULL	0	1988	Delhi
Dhoni	India	Sri Lanka	India	2755	27/8/2008	Colombo	71	NULL	1981	Ranchi
Kallis	South Africa	South Africa	Australia	2839	9/4/2009	Cape Town	70	0	1975	Cape Town
Yuvraj	India	Pakistan	India	2717	26/6/2008	Karachi	48	0	1978	Delhi
Gambhir	India	Australia	India	2689	4/3/2008	Gabba	15	NULL	1973	Mumbai



An Example

The table (name *ODIplayers*)satisfies the properties of a relation and any such relation is said to be in *first normal form (or 1NF)*.

The main properties of relations concerned here are structural and therefore we note that all attribute values of 1NF relations must be atomic. We acknowledge that relations have other important properties like no ordering of rows, no duplicate rows and unique names of columns but in this chapter we are not particularly concerned with such properties.



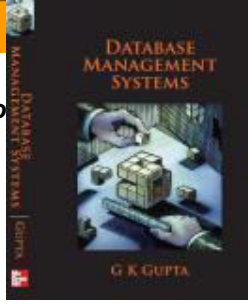
Redundant Information

One of the aims of good database design is that redundancy of information should be eliminated.

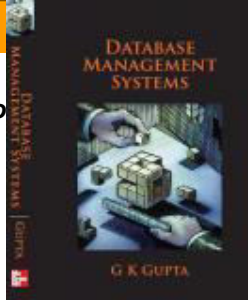
Redundancy exists when same data is stored in several places in the database. In some cases although there is no duplicate data it may be possible to derive certain information from other information in the database.

Some information in the example table is being repeated, for example information about players, like LName, YBorn, Place, are being repeated for Tendulkar and Yuvraj Singh.

Update Anomalies

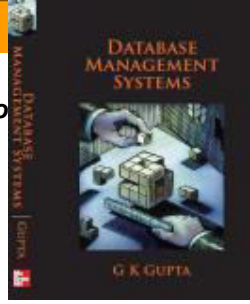


Redundant information makes updates more difficult, for example, changing the name of the Ground sounds strange but such things do happen – for example the name of Bombay was changed to Mumbai. Such a change would require that all rows containing information about that ground be updated.



Insertion Anomalies

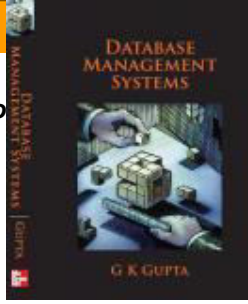
Let the primary key of the above table be *(PID, MID)*. Any new row to be inserted in the table must have a value for the primary key since existential integrity requires that a key may not be totally or partially NULL. However, if one wanted to insert the number and name of a new player in the database, for example a new player who has been selected to tour overseas, it would not be possible until the player plays in a match and we are able to insert values of PID and MID.



Deletion Anomalies

In some instances, useful information may be lost when a row is deleted. For example, if we delete the row corresponding to a player who happens to be the only player selected for a particular ODI, we will lose relevant information about the match (viz. ground name, date, etc).

Similarly deletion of a match from the database may remove all information about a player who was playing in only that ODI. Such problems are called *deletion anomalies*.

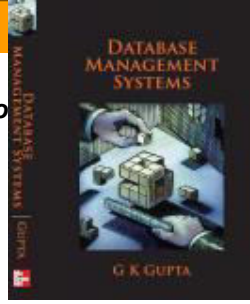


Solution

One solution is to decompose the table into two or more smaller tables. Decomposition may provide further benefits; for example, in a distributed database different tables may be stored at different sites if necessary.

The above table may be decomposed into three tables as shown below so that the semantics of each decomposed table are clearer. It has been noted earlier that the example table appears to include information about players, matches and players' performances.

Therefore the decomposition is based on that understanding of the table and removes most of the undesirable properties listed earlier.



Decomposing the table

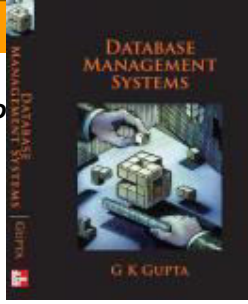
ODIPlayers may be decomposed into the following tables.

Player2(*PID, LName, Country, YBorn, Place*)

ODIMatch(*MID, Team1, Team2, Date, Ground*)

Performance(*PID, MID, NRuns, NWickets*)

Such decomposition is called *normalization* and in some cases decomposition is essential if we wish to overcome undesirable anomalies. The normalization process essentially assists in representing the data required for an application with as little redundancy as possible such that efficient updates of the data in the database are possible and useful data is not lost by mistake when some other data is deleted from the database.



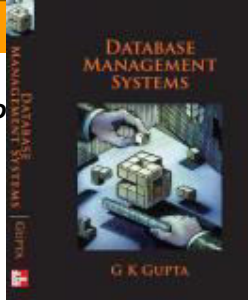
Normalization

There are several stages of the normalization process.

These are called the *first normal form (1NF)*, the *second normal form (2NF)*, the *third normal form (3NF)*, *Boyce-Codd normal form (BCNF)*, the *fourth normal form (4NF)* and the *fifth normal form (5NF)*.

For all practical purposes, 3NF or the BCNF are quite adequate since they remove the anomalies discussed above for most common situations.

It should be clearly understood that there is no obligation to normalize tables to the highest possible level. Performance should be taken into account and this may result in a decision not to normalize, say, beyond second normal form.



Functional Dependency

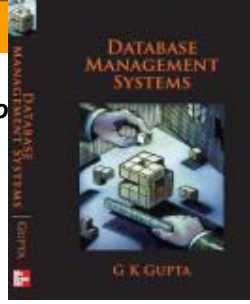
Single-Valued Dependencies

Functional dependency (FD) is an important concept in designing a database. It is a constraint between two sets of attributes of a relation.

The idea of functional dependency is a relatively simple one. Consider a table *R* that has two attributes *A* and *B*. The attribute *B* of the table is *functionally dependent* on attribute *A* if and only if for each value of *A* no more than one value of *B* is associated.

Functional dependency therefore may be considered a kind of integrity constraint.

Functional Dependency



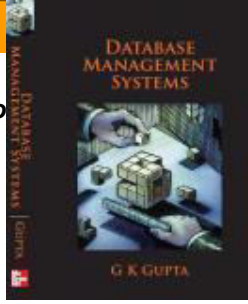
Single-Valued Dependencies

The attributes A and B need not be single columns. They could be any subsets of the columns of the table R (possibly single columns). If B is functionally dependent on A (or A functionally determines B), we may then write

$$R.A \rightarrow R.B$$

The set of attributes $R.A$ are called the left-hand side of the functional dependency (FD) while $R.B$ is called the right-hand side of the FD.

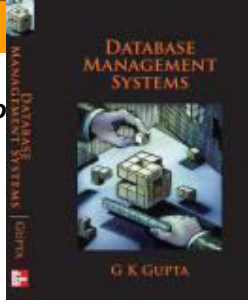
Each FD in a relation reveals something about the semantics of the relation.



Functional Dependency

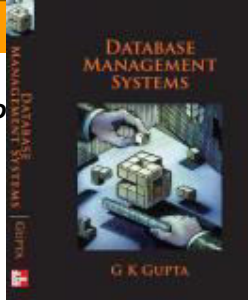
Functional dependencies arise from the nature of the real world that the database models. Often *A* and *B* are facts about an entity where *A* might be some identifier for the entity and *B* some characteristic.

Functional dependencies cannot be automatically determined by studying one or more instances of a database. They can be determined only by a careful study of the real world and a clear understanding of the semantics of the attributes in the table.



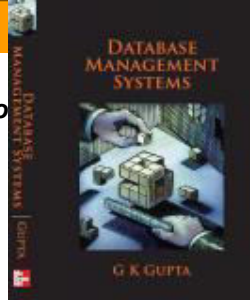
Definition – Full Functional Dependence

Let A and B be distinct collections of columns from a table R *and let* R.*A* → R.*B. B is then fully functionally dependent on A if B is not functionall y dependent on any subset of A.*



Definition – Closure

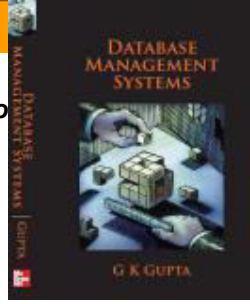
Let a table R have some functional dependencies F specified. The closure of F (usually written as F^+) is the set of all functional dependencies that may be l ogically derived from F..



Single-Valued Normalization
First Normal Form (1NF)

Definition – 1NF

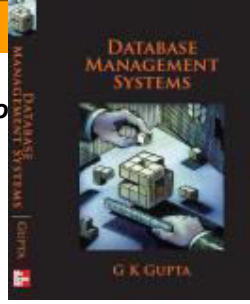
A relation is in 1NF if and only if all underlying domains contain atomic values only.



Single-Valued Normalization
Second Normal Form (2NF)

Definition – 2NF

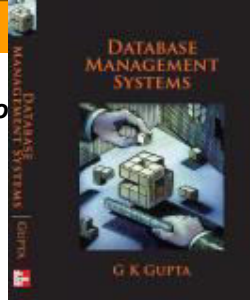
A relation is in 2NF if it is in 1NF and every non-key (also called nonprime) attribute is fully functionally dependent on each candidate key of the relation.



Single-Valued Normalization
Third Normal Form (3NF)

Definition – 3NF

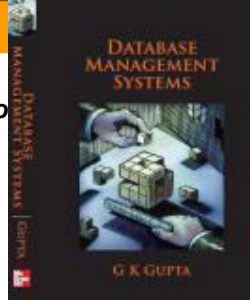
A relation R is in third normal form (3NF) if it is in 2NF and every non-key attribute of R is non-transitively dependent on each candidate key of R.



Single-Valued Normalization
Boyce-Codd Normal Form (BCNF)

Definition – BCNF

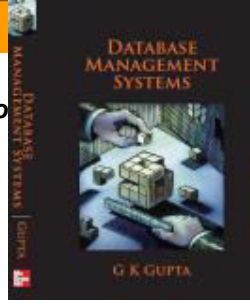
A relation R *is said to be in BCNF whenever* $X \rightarrow A$ *holds in* R*, and* A *is not in* X*, then* X *is a candidate key for* R*.*



Single-Valued Normalization

Summary

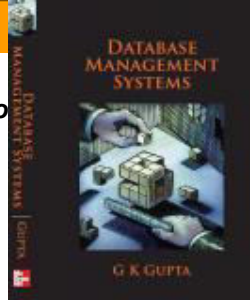
Normal Form	Test	Remedy
First Normal Form (1NF)	A relation is in 1NF if and only if all underlying domains contain atomic values only.	Ensure that all values are atomic
Second Normal Form (2NF)	A relation is in 2NF if it is in 1NF and every non-key (also called nonprime) attribute is fully functionally dependent on each candidate key of the relation.	Decomposition based on non-key attributes that are not fully dependent on the keys
Third Normal Form (3NF)	A relation R is in third normal form (3NF) if it is in 2NF and every non-key attribute of R is non-transitively dependent on each candidate key of R.	Decomposition based on transitive FDs
Boyce-Codd Normal Form (BCNF)	A relation R is said to be in BCNF whenever $X \rightarrow A$ holds in R, and A is not in X, then X is a candidate key for R. This normally happens when a relation has more than one candidate key and the keys are composite.	Decomposition based on FDs within composite keys



Multivalued Dependencies

The normal forms considered so far are based on functional dependencies, dependencies that apply only to single-valued facts. For example, ID → department implies only one department value for each value of ID.

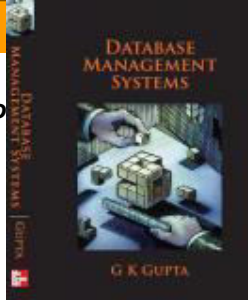
Not all information in a database is single-valued, for example, project in an employee database may be the list of all projects that the employee is currently working on. Although ID determines the list of all projects that an employee is working on, ID → project is not a functional dependency since project is not single-valued.



Multivalued Dependencies

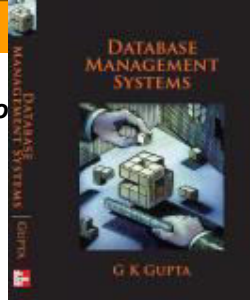
The fourth and fifth normal forms (4NF and 5NF) deal with multivalued dependencies. Before discussing these normal forms we discuss the following example to illustrate the concept of multivalued dependency.

We are now using an example which has little to do with ODI cricket. We have decided to find something different from another popular topic in India, namely Bollywood!



An Example

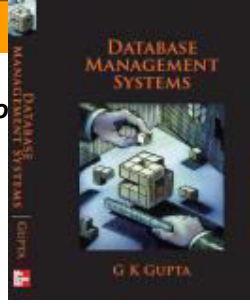
We present a relation *Bstars* that gives information about some Bollywood stars, the cities they have lived in and names of their children. The information is not complete but the number of children and their names hopefully are correct.



An Example

Names	Cities	Children
Amitabh Bachchan	Allahabad	Abhishek
Amitabh Bachchan	Mumbai	Shweta
Madhuri Dixit	Mumbai	Arin
Madhuri Dixit	Los Angeles	Ryan
Shah Rukh Khan	Delhi	Aryan
Shah Rukh Khan	Mumbai	Suzana

The table above includes two multivalued attributes of entity *BStars: Cities and Children. There are no functional dependencies.* Can we be sure about this? Could Children → Name be a functional dependency? It can only be so if we assume that the children's names are unique. We are assuming they are not unique in a large table.

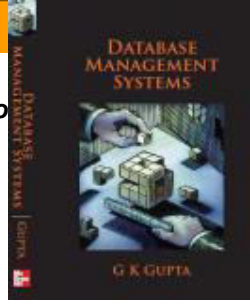


An Example

The attributes Cities and Children are assumed to be independent of each other but the table appears to show that there is some relation between the Cities and Children attributes. How can this be resolved?

We may consider Cities and Children as separate entities and have two relationships (one between Names and Cities and the other between Names and Children). Both the above relationships are one-to-many.

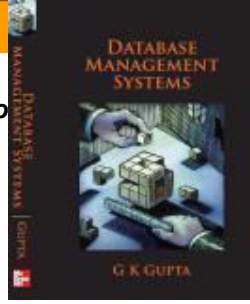
The table is therefore in 3NF, it is even in BCNF, but it still has some disadvantages. How should this information be represented?



Multivalued Dependency (MVD)

Definition

The multivalued dependency X —>> Y is said to hold for a relation R(X, Y, Z) if for a given set of values (set of values if X is more than one attribute) for attribute(s) X, there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.



Multivalued Normalization

Fourth Normal Form (4NF)

Definition

A table R is in 4NF if, whenever a multivalued dependency X —» Y holds in the table R, then either

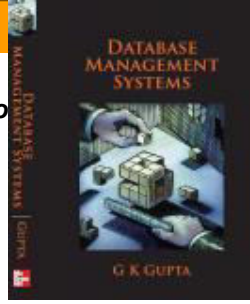
(a) the dependency is trivial, or

(b) X is a candidate key for R.

If a table has more than one multivalue attribute, we decompose it to remove difficulties with the multivalued facts. The table *Bstars* is therefore not in 4NF and needs to be decomposed in the following two tables.

BS1(Names, Cities)

BS2(Names, Children)



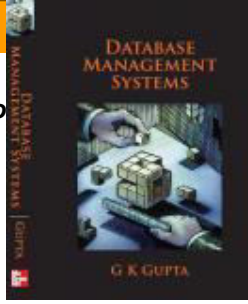
Join Dependencies
Fifth Normal Form (5NF)

Definition – Join Dependency

*Let a relation R have subsets of its attributes A, B, C, ... Then R satisfies the Join Dependency (JD) written as *(A, B, C, ...) if and only if every possible legal value of R is equal to the join of its projections A, B, C, ...*

Consider the following example:

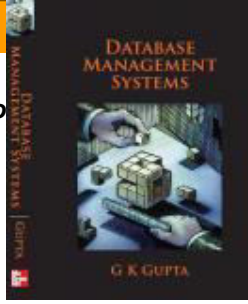
Movies	Heroes	Heroines
Lagaan	Aamir Khan	Gracy Singh
Gajjni	Aamir Khan	Asin Thotumkal
God Tussi Great Ho	Salman Khan	Priyanka Chopra
God Tussi Great Ho	Amitabh Bachchan	Manisha Koirala
Hey Ram	Kamal Hasan	Rani Mukerjee
Paheli	Shah Rukh Khan	Rani Mukerjee



Join Dependency

Note that the table has information on a number of movies; each movie has a hero and a heroine except in one case two heroes and two heroines are given. All three fields are needed to represent the information.

The table does not show MVDs as the attributes Movies, Heroes and Heroines are not independent; they are related to each other and the pairings have significant information in them. If we try to decompose the relation into the relations Movies and Heroes) and (Movies and Heroines) then some information is lost (why??).



5NF

The table can however be decomposed into the following three relations.

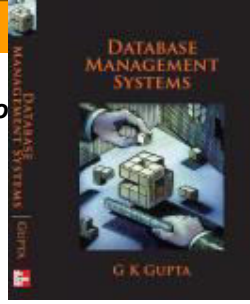
(Movies, Heroes)
(Movies, Heroines)
(Heroes, Heroines)

Definition – Fifth Normal Form (5NF)

A relation R is in 5NF (also called the PJNF) if for all join dependencies at least one of the following conditions holds:

(a) The decomposed tables (A, B, C, ...) are a trivial join-dependency, that is one of the A, B, C, ... is itself the relation R)

(b) Every A, B, C, ... is a candidate key for R.

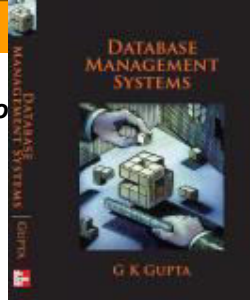


Denormalization

The primary aim of normalization is to overcome anomalies in relations that are not normalized. Normalization however does not consider database performance. Normalization involving decomposing relations can have a performance penalty. Query performance can often be improved by joining relations together if the relations are being joined frequently and the join is expensive to compute. This technique for improving performance is called *denormalization*.

Definition – Denormalization

Denormalization is the intentional duplication of information in the database tables leading to increased data redundancy but improving query processing performance by reducing the number of joins.



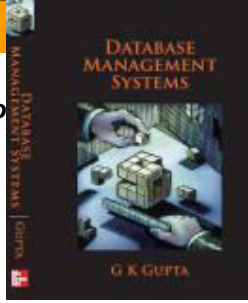
Denormalization

Definition – Denormalization

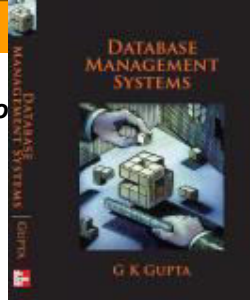
Denormalization is the intentional duplication of information in the database tables leading to increased data redundancy but improving query processing performance by reducing the number of joins.

We may compare normalization and denormalization as follows.

Item	Normalize	Denormalize
Retrieval Performance	High	Higher for some queries
Update Performance	High	Slower
Storage Performance	Efficient	Not so efficient
Level of Normalization	3NF or higher	3NF or lower
Typical applications	OLTP	OLAP
Preserving Integrity	Yes	Not always

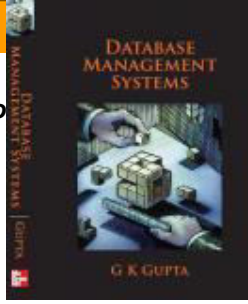


Mc Graw Hill Education

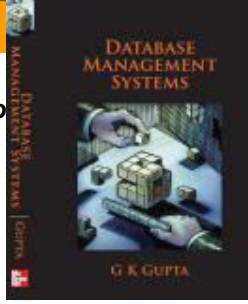


DATABASE MANAGEMENT SYSTEMS

G K GUPTA

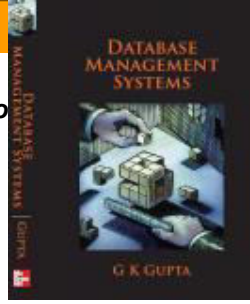


Chapter 9
Transaction Management and Concurrency



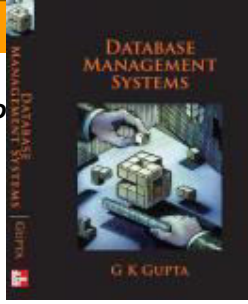
Objectives

- explain the concept of a transaction
- describe the four properties of DBMS transactions
- explain why concurrency is important
- discuss when concurrent execution of two transactions leads to conflict
- give examples of concurrency anomalies
- explain the concept of a schedule
- study the concepts of serializability, final state serializability (FSR), view serializability (VSR) and conflict serializability (CSR)



Objectives (Cont'd)

- discuss how to test for CSR
- describe how serializability can be enforced using locking and timestamping
- discuss the concept of deadlocks, their prevention and resolution
- explain the concept of lock granularity, multiple granularity
- describe use of intention locking in multiple granularity
- describe timestamping concurrency control
- describe the optimistic concurrency control
- briefly look at evaluation of concurrency control techniques

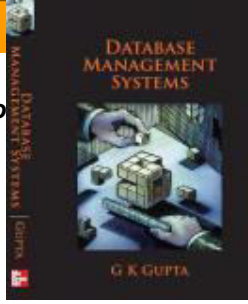


Concurrency

Concurrency control in a database system permits many users, assumed to be interactive, to access a database while preserving the illusion that each user has sole access to the system.

Control is needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained.

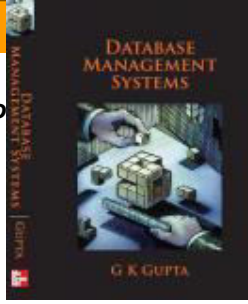
Efficiency is also important since the response time for interactive users ought to be short. The problems in database management systems are related to the concurrency problems in operating systems although there are significant differences between the two.



Concurrency

No problem would arise if all users were accessing the database only to retrieve information and no one was modifying it as in a library catalogue. If one or more of the users are modifying the database, for example, a bank account or airline reservations, an update performed by one user may interfere with an update or retrieval by another.

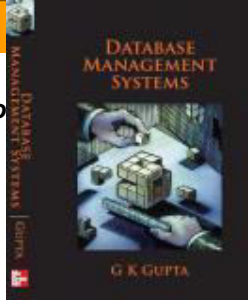
For example, users A and B both may wish to read and update the same record in the database at about the same time.



Concurrency

How to control concurrency or the accessing of the same data by many users? Without concurrency controls, data may be updated or changed improperly or out of sequence, compromising data integrity.

Concurrency concerns apply both to data and to structures. Changes to data should be guaranteed to be made in the order the changes actually occur. Similarly, changes to the definitions of structures, such as tables or indexes, should also occur in the proper order.

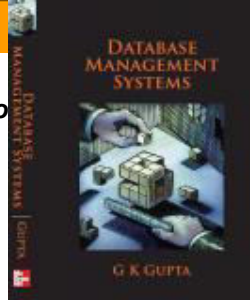


A Transaction

Our discussion of concurrency is transaction based. A transaction *T* is a sequence of actions $[t_1, t_2, \ldots t_n]$ which is defined as follows.

Definition – Transaction

A transaction is defined as a logical unit of work which involves a sequence of steps but which normally will be considered by the user as one action and which preserves consistency of the database.



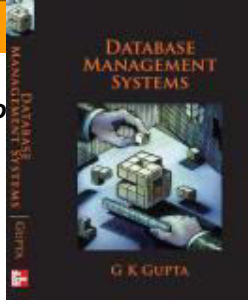
Example of an Transaction

Begin Transaction
Transfer Rs.100 from Account A to Account B
Commit

The number of steps to Carry out the transaction Are shown in the table.

Step	Action
1	Read Account A from Disk
2	If balance is less then Rs. 100, return with an appropriate message
3	Subtract Rs. 100 from balance of Account A
4	Write Account A back to Disk
5	Read Account B from Disk
6	Add Rs 100 to balance of Account B
7	Write Account B back to Disk

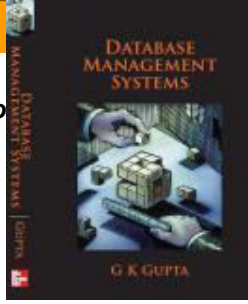
We now discuss the ACID properties of a transaction.



ACID Properties

Atomicity – a transaction usually consists of a number of steps. It is necessary to make sure that either all actions of a transaction are completed without any errors or the transaction is abnormally terminated and has no effect on the database.

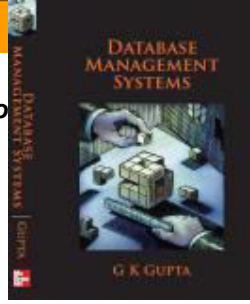
Consistency – A database may become inconsistent during the execution of a transaction, it is assumed that a completed transaction preserves the consistency of the database.



ACID Properties

Isolation – no other transactions should view any partial results of the actions of a transaction since intermediate states may violate consistency. It is essential that other transactions only view results of committed transactions.

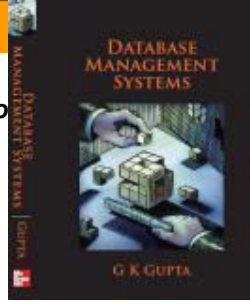
Durability – a transaction that is for some reason interrupted in the middle of its execution must be aborted so that the database is left consistent.



ACID Properties

Summary

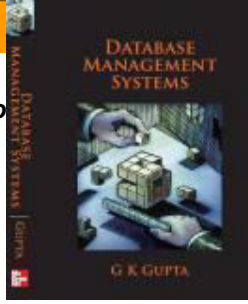
	Property	What is it?
A	Atomicity	All-or-nothing
C	Consistency	Each transaction leaves the database consistent; otherwise rolled back
I	Isolation	Each transaction is executed as if it was the only one running
D	Durability	A committed transaction cannot be aborted and its effects cannot be undone



Concurrency Anomalies

Data or data structures might become compromised in the following ways:

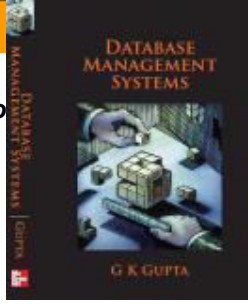
- Lost update (one update overwriting another)
- Inconsistent Retrievals
- Uncommitted Dependency



Lost Update

Consider an example of two transactions *A* and *B* which simultaneously access an airline database wishing to reserve a number of seats on a flight. Let us assume that transaction *A* wishes to reserve five seats while transaction *B* wishes to reserve four seats.

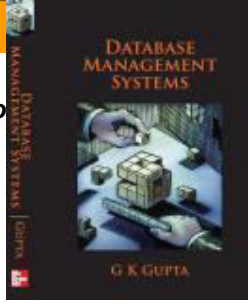
The two users read-in the present number of seats, store it in their local storage, and modify their local copy and write back resulting in the situation in which *A* obtains 5 seats but *B* overwrites the number of seats and obtains 4 seats. The update by *A* is lost.



Inconsistent Retrievals

Consider another example of two transactions *A and B* accessing a salary database simultaneously. Transaction *A* is updating the salaries to give all employees a 3% raise in their salary while transaction *B* is computing the total salary bill.

The two transactions interfere since the total salary bill changes as the transaction *A* updates the employee records. The total salary retrieved by transaction *B* may be a sum of some salaries before the raise and others after the raise. This is not acceptable and is called *inconsistent retrieval*.

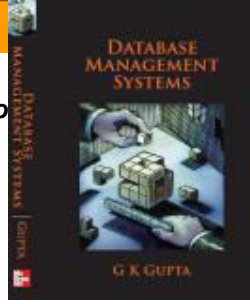


Uncommitted Dependency

Again consider two transactions. Transaction *A* has read the value of *Q* that was *updated* by transaction *B* but was never committed. The result of transaction *A* writing *Q* therefore will lead to an inconsistent state of the database.

Also if transaction A does not write *Q* but only reads it, it would be using a value of *Q* which never really existed! Yet another situation would occur if the rollback happened after *Q* was written by transaction A.

This is called the *uncommitted dependency* anomaly.

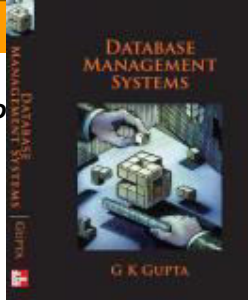


The Concept of Schedule

When two or more transactions are running concurrently, the steps of the transactions would normally be *interleaved*. The interleaved execution of transactions is decided by the database system concurrency control software called the *scheduler* which receives a stream of user requests that arise from the active transactions.

Definition – Schedule

A particular sequencing (usually interleaved) of the actions (including reading and writing) of a set of transactions is called a schedule. The order of actions of every transaction in a schedule must always be the same as they appear in the transaction.



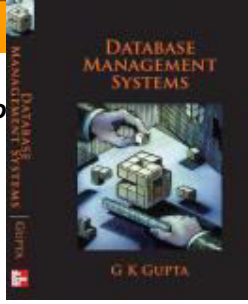
The Concept of Schedule

Since each transaction maintains the integrity of the database, it follows that a serial execution of a number of transactions will also maintain the integrity of the database.

Definition – Serial Schedule

A serial schedule is a schedule in which all the operations of one transaction are completed before another transaction begins (that is, there is no interleaving).

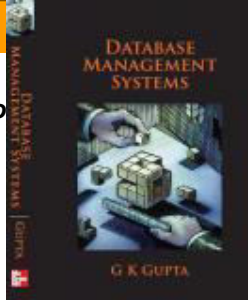
The Concept of Schedule



Since each transaction maintains the integrity of the database, it follows that a serial execution of a number of transactions will also maintain the integrity of the database.

Definition – Serial Schedule

A serial schedule is a schedule in which all the operations of one transaction are completed before another transaction begins (that is, there is no interleaving).

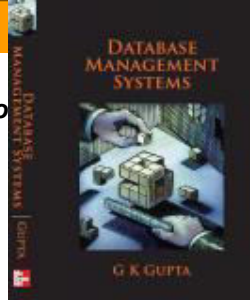


Recoverability

Normally we assume that if a transaction has been aborted or has failed for some reason then the transaction may be rolled back and all its action may be undone leaving the database in a condition as if the transaction never existed. In some situations aborting a transaction so that all its actions are undone may not be possible.

Definition – Recoverability

A schedule is said to be recoverable if it does not allow a transaction A to commit until every other transaction B that wrote values that were read by A has committed.

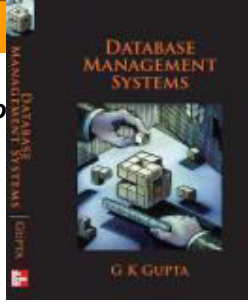


Serializability

Although using only serial schedules to ensure consistency is a possibility, it is not a realistic approach (why?).

Consider an airline reservation system. Three requests arrive: transaction T1 for 3 seats, transaction T2 for 5 seats and transaction T3 for 7 seats. Any of the six serial schedules of the three transactions presented below are correct:

$T_1\ T_2\ T_3$	The seat reservation system may result in different allocations of seats for different serial schedules although each schedule will ensure that no seat is sold twice and no request is denied if there is a free seat available on a desired flight.
$T_1\ T_3\ T_2$	
$T_2\ T_1\ T_3$	
$T_2\ T_3\ T_1$	
$T_3\ T_1\ T_2$	
$T_3\ T_2\ T_1$	

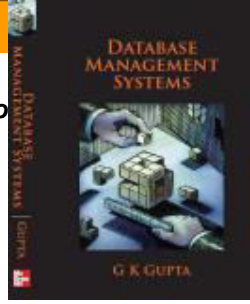


Serializability

We now investigate schedules that are not serial but result in database consistency being maintained. Since all serial schedules are correct, interleaved schedules that are equivalent to them must also be considered correct. We first present a definition.

Definition – Final State Equivalence

Two schedules S1 and S2 are called final state equivalent if both include the same set of operations and result in the same final state for any given initial state.



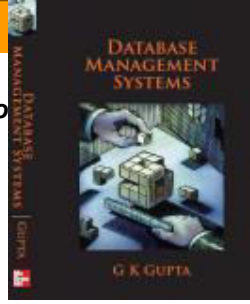
Final State Serializability (FSR)

As noted on the last slide, all schedules that are equivalent to any serial schedule must be considered correct. The concept of FSR is based on that.

Definition – Final State Serializability

A concurrent execution of n transactions T_1, T_2,…T_n (call this execution S) is called final state serializable (FSR) if the execution is final state equivalent to a serial execution of the n transactions.

If a schedule is not serializable, the schedule may be modified so that it is serializable. The modifications are made by the database scheduler.



Notation for Schedules

Transaction 1 = $r_1(A)w_1(A)r_1(C)w_1(C)c_1$

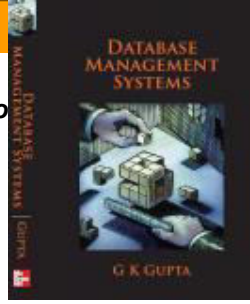
Transaction 2 = $r_2(B)w_2(B)r_2(C)w_2(C)c_2$

Note that we have added symbols c_1 and c_2 to indicate that transactions 1 and 2 commit respectively.

A schedule may then be represented as
$r_1(A)w_1(A)\ r_2(B)w_2(B)\ r_1(C)$
$w_1(C)r_2(C)w_2(C)$

It may also be represented by the table on the RHS.

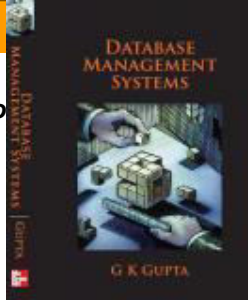
Transaction 1	Time	Transaction 2
Read A	1	–
A := A – 200	2	–
Write A	3	–
–	4	Read B
–	5	B := B – 300
–	6	Write B
Read C	7	–
C := C + 200	8	–
Write C	9	–
–	10	Read C
–	11	C := C + 300
–	12	Write C



Example Schedules

Transaction 1	Time	Transaction 2
–	1	Read B
–	2	B := B – 300
–	3	Write B
Read A	4	–
A := A – 200	5	–
Write A	6	–
Read C	7	–
C := C + 200	8	–
Write C	9	–
–	10	Read C
–	11	C := C + 300
–	12	Write C

Transaction 1	Time	Transaction 2
Read A	1	–
A := A – 200	2	–
Write A	3	–
Read C	4	–
–	5	Read B
–	6	B := B – 300
–	7	Write B
–	8	Read C
–	9	C := C + 300
–	10	Write C
C := C + 200	11	–
Write C	12	–



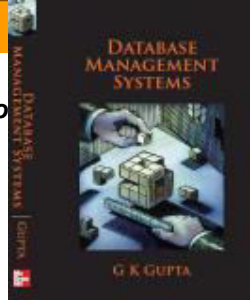
View Serializability (VSR)

Definition – View Equivalence

Two schedules S1 and S2 are called view equivalent if both comprise the same set of operations and result in the same final state for any given initial state and in addition each operation has the same semantics in both schedules.

Definition – View Serializability

A concurrent execution of n transactions T_1, T_2,…T_n (call this execution S) is called view serializable (VSR) if the execution is view equivalent to a serial execution of the n transactions.



Conflict Serializability (CSR)

Definition – Conflict

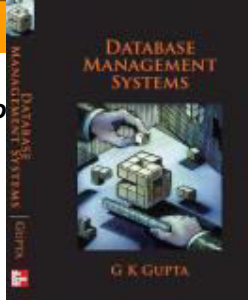
Two data operations belonging to two different transactions in a schedule are in conflict if they access the same data item and at least one of them is a write.

Definition – Conflict Equivalence

Two schedules are called conflict equivalent if they have the same set of operations and the same conflict relations.

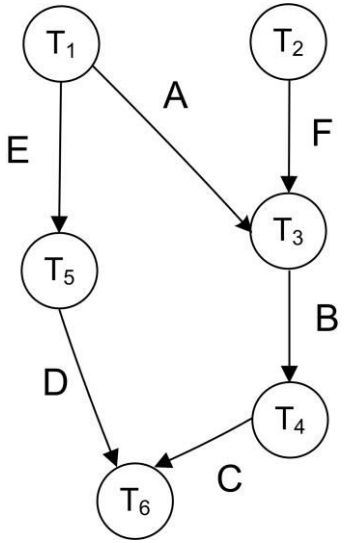
Definition – Conflict Serializability

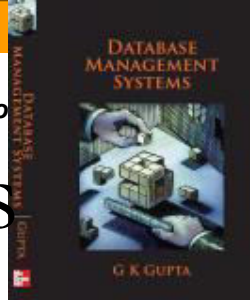
A concurrent execution of n transactions T_1, T_2,…T_n (call this execution S) is called conflict serializable if the execution is conflict equivalent to a serial execution of the n transactions.



Testing for CSR

We now present an algorithm for testing CSR. The algorithm involves constructing a *precedence graph* (also called *conflict graph* or *serialization graph*). The precedence graph is a directed graph in which there is a node for each transaction in the schedule and an edge between T_i *and* T_j exists if any of the conflict operations.



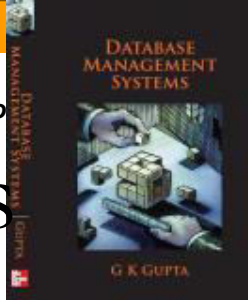


Hierarchy of Serializable Schedules

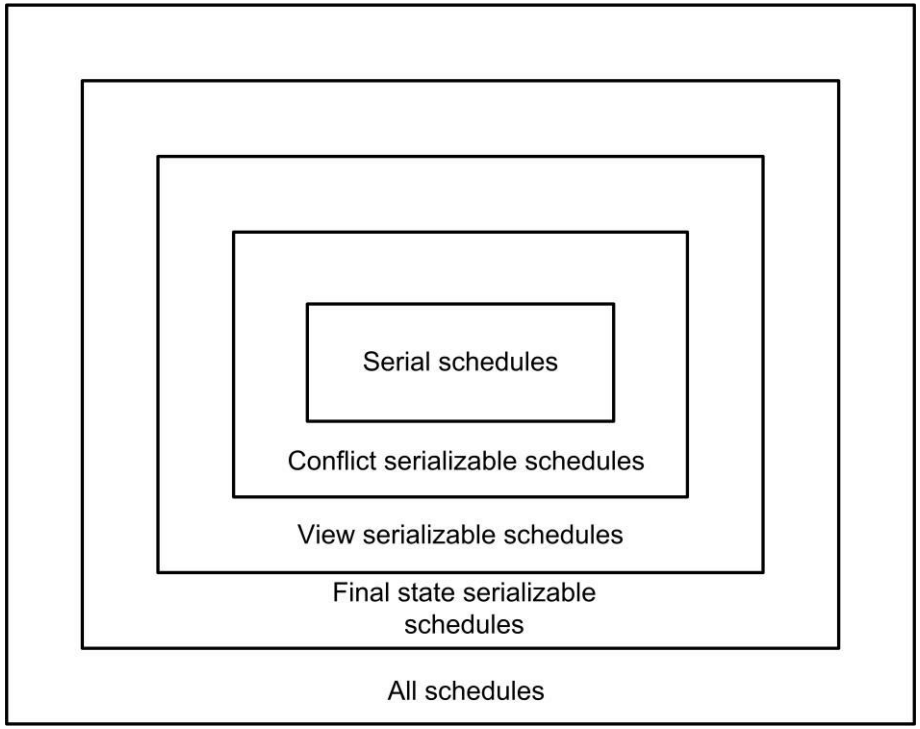
The figure on the next slide shows that serial schedules are the most restrictive serializable schedules since no interleaving of transactions is permitted in serial schedules.

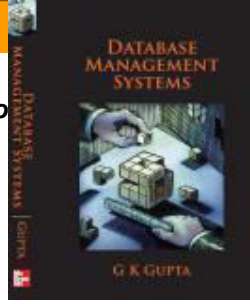
Amongst the serializability requirements we have discussed earlier, CSR is the next most restrictive, VSR comes next and FSR is the weakest requirement of serializability amongst the techniques discussed.

There are a number of other serializability techniques presented in the literature that we have not discussed.



Hierarchy of Serializable Schedules



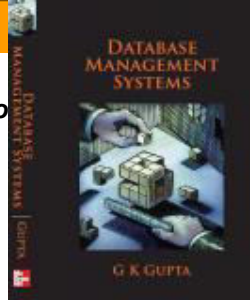


Enforcing Serializability

The easiest way to enforce is to insist that the only schedules that are accepted are those in which data items that are common to the two transactions are seen by the junior transaction only after the older transaction has written them.

The DBMS module that controls which schedules are permitted is called the *scheduler.*

There are three basic techniques used in concurrency control. They are locking, timestamping and optimistic concurrency control. The only schedules that these techniques allow are those that are serializable. We first discuss locking.



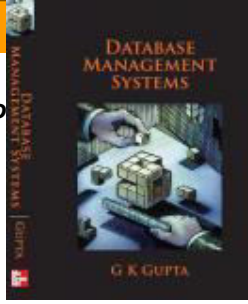
Introduction to Locking

Looks are often used to control concurrent access to data. Locks are intended to prevent destructive interaction between users accessing data. Destructive interaction can be interpreted as any interaction which incorrectly updates data or incorrectly alters underlying data structures.

Each locking protocol consists of:

1. a set of locking types (for example, *shared* lock and *exclusive* lock),
2. a set of rules indicating what locks can be granted concurrently,
3. a set of rules that transactions must follow when acquiring and releasing locks.

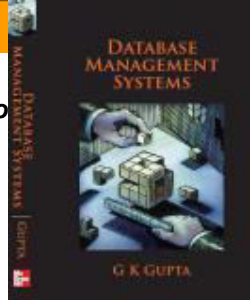
Locking (cont'd)



Locks are used to achieve two important database goals:

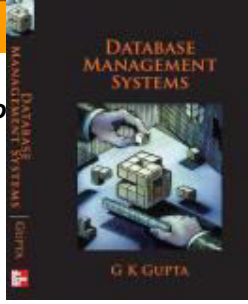
- Consistency assures users that the data they are viewing or changing is not changed until they are through with it.
- Integrity assures that the database's data and structures reflect all changes made to them in the correct sequence.

Thus, to summarize, locks are used to insure data integrity while allowing maximum concurrent access to the data by unlimited users.



Example of Locking

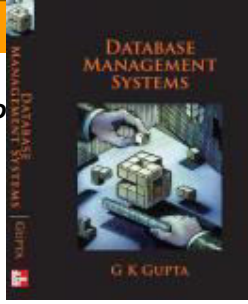
Transaction 1	Time	Transaction 2
XL(A)	1	–
Read(A)	2	–
A := A – 50	3	–
–	4	SL(B)
Write(A)	5	–
–	6	Read(B)
UL(A)	7	–
–	8	UL(B)
–	9	SL(A)
XL(B)	10	–
–	11	Read(A)
Read(B)	12	–
–	13	UL(A)
B := B + 50	14	–
–	15	Display (A + B)
Write (B)	16	–
UL(B)	17	–



What resources can be locked?

A lock can be thought of as assigning a database user "temporary ownership" of a database resource, such as a table or a row of data in a table. Resources include two general types of objects:

- user objects such as tables and rows (structures and data)
- system objects not visible to users, such as shared data structures and database buffers.

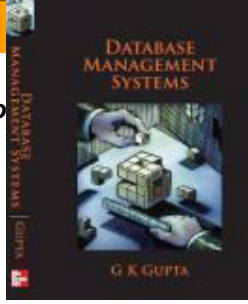


Locking (cont'd)

A lock on a resource is acquired by a user when the user wants to keep another user from using that resource. The lock is released when certain events occur and the first user no longer requires the resource.

Most DBMS automatically lock numerous structures on behalf of users. Users can explicitly lock rows of a table using the SELECT…FOR UPDATE statement or entire tables using the LOCK TABLE statement.

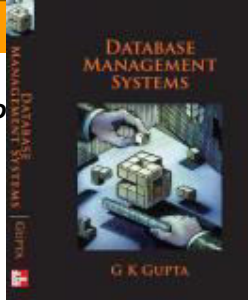
Locking (cont'd)



The following characteristics are true of queries that do not use the FOR UPDATE clause:

A query requires no data locks. A query never waits for any data locks to be released; it can always proceed. A query always returns a read consistent view of the data. Other transactions can access the table being queried, including the particular rows being queried.

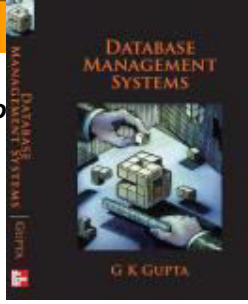
Locking (cont'd)



A transaction executing a query never has to wait for data resources, even when the underlying data is being accessed or altered by other transactions. No data locks are required on rows or tables.

ORACLE establishes a read consistent snapshot of the data at the time the query is executed. The query sees all data that was committed as of the start of the query, plus any updates that are made in the transaction of which the query is a part.

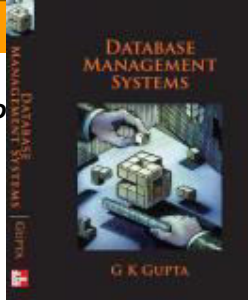
Locking (cont'd)



The following are characteristics of UPDATEs, DELETEs and INSERTs:

A query on behalf of an UPDATE, DELETE or INSERT will always begin with a read consistent view of the data.

The statements acquire exclusive locks (X) on the rows to be modified and row exclusive locks (RX) on the tables containing those rows.

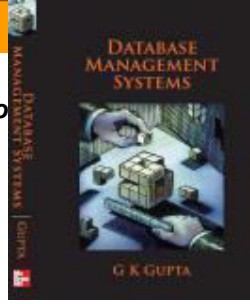


Locking (cont'd)

Other transactions can query the rows to be modified, but cannot alter them until the current transaction has released the row locks.

Other transactions can do any action on other rows in the table, even in the same block as rows affected and even while the locks are held.

The row and table locks are released when the transaction is committed.

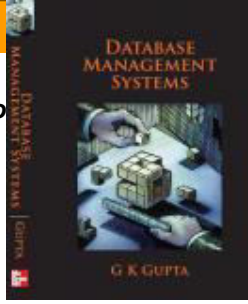


Two-Phase Locking (2PL)

Locking can lead to problems. 2PL is the most commonly used concurrency control technique in which a transaction cannot request a new lock after releasing a lock. The 2PL protocol involves the following two phases:

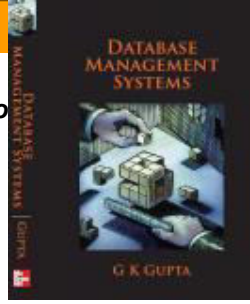
- *Growing Phase* (Locking phase) – locks may be acquired but not released.
- *Shrinking Phase* (Unlocking phase) – locks may be released but not acquired.

The main feature of 2PL is that conflicts are checked at the beginning of the execution and resolved by waiting. This leads to safe interleaving of transactions.



Example of 2PL

Transaction 1	Time	Transaction 2
XL(A)	1	–
Read(A)	2	–
A := A – 50	3	–
–	4	SL(B)
Write(A)	5	–
–	6	Read(B)
XL(B)	7	–
UL(A)	8	–
–	9	SL(A)
–	10	UL(B)
–	11	Read(A)
Read(B)	12	–
–	13	UL(A)
B := B + 50	14	–
–	15	Display (A + B)
Write (B)	16	–
UL(B)	17	–

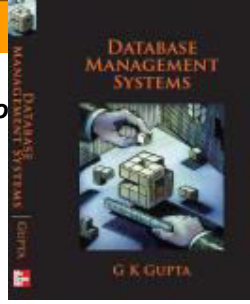


Deadlock

Definition – Deadlock

A deadlock may be defined as a situation in which each transaction in a set of two or more concurrently executing transactions is blocked circularly waiting for another transaction in the set.

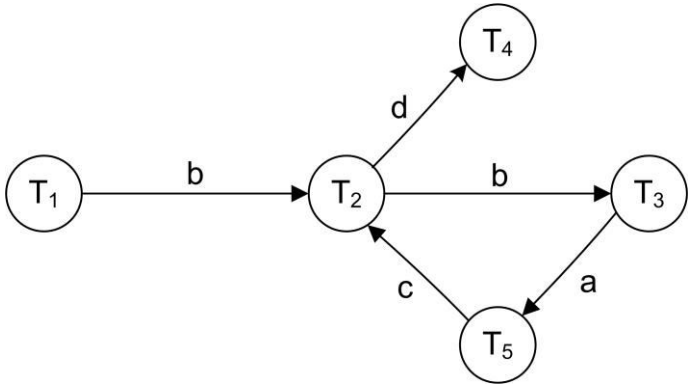
The locks are granted and managed by DBMS software called the lock manager and it maintains a lock table to manage the locks. This table maintains a variety of information including a queue of lock requests that are waiting.

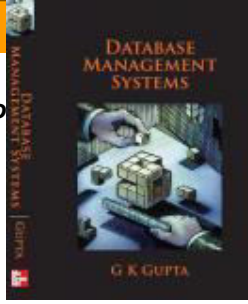


Wait-For Graph (WFG)

When a transaction T_i requests a data item that is currently being held by T_j, then the edge Ti to T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

An example of a WFG. Is there a deadlock?

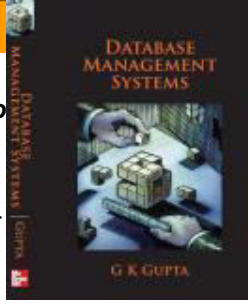




Deadlock Prevention

Deadlocks may be prevented by ensuring that circular waits do not occur. This can be done either by defining an order on who may wait for whom or by eliminating all waiting.

- *Wait-die algorithm*
- *Would-die algorithm*
- *Immediate restart*

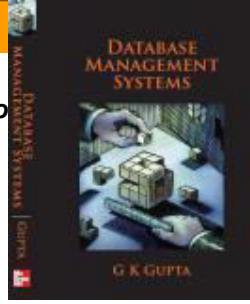


Deadlock Detection and Resolution

One of the transactions in the WPG cycle is selected as a *victim* and is rolled back and restarted. The victim may be selected:

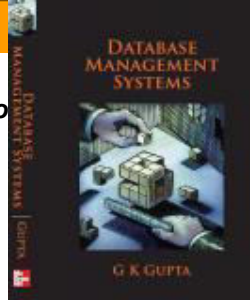
- Randomly: a transaction from the cycle is picked randomly,
- Last blocked: the transaction that blocked most recently,
- Youngest: since this transaction is likely to have done the least amount of work,
- Minimum work: the transaction that has written the least data back to the database,
- Least impact: the transaction that is likely to affect the least number of other transactions.
- Fewest locks: the transaction that has the smallest number of locks.

Mc Graw Hill Education

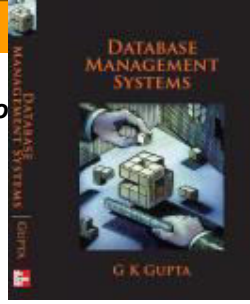


DATABASE MANAGEMENT SYSTEMS

G K GUPTA

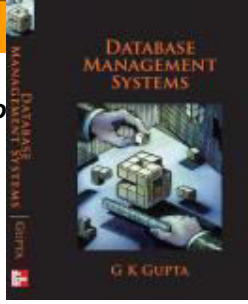


Chapter 2
Entity-Relationship Model



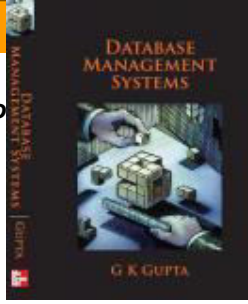
Objectives

- discuss why modeling data is important
- describe the different phases of database modeling
- explain the Entity-Relationship Model for modeling a database
- define entity, entity sets, relationship, relationship sets, different types of relationships, attributes
- describe representing entities, attributes and relationships in entity-relationship diagrams
- explain binary and ternary relationships



Objectives

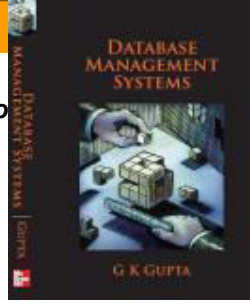
- discuss why modeling data is important
- describe the different phases of database modeling
- explain the Entity-Relationship Model for modeling a database
- define entity, entity sets, relationship, relationship sets, different types of relationships, attributes
- describe representing entities, attributes and relationships in entity-relationship diagrams
- explain binary and ternary relationships



Modelling

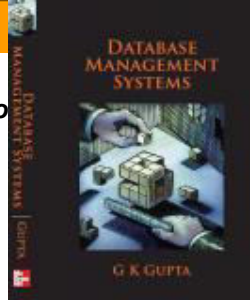
Modelling is not new!

Can we think of modelling in fields other than computing?



Modelling

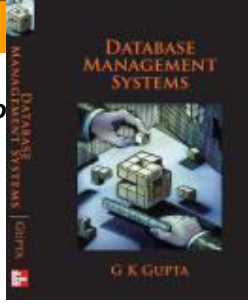
- Architects
- Aeronautical engineers
- Computer architects
- Traffic engineers
- And many more…



Benefits of Modelling

Many reasons for modelling:

- Focussing on essentials
- Ease of communication and understanding
- Product or process improvement
- Exploring alternatives



Types of Models

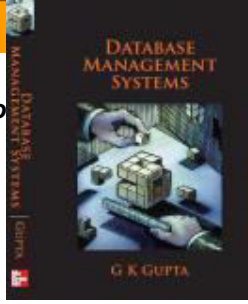
A number of model types:

- Descriptive
- Prescriptive
- Representative

A model does not have to belong to just one class.

What class does database models belong to?

Database Modelling

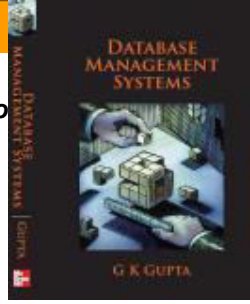


Before building a database system, an overall abstract view of the enterprise data must be developed.

The view can then be translated into a form that is acceptable by t he DBMS.

Often a very complex process.

The process can be divided into two phases.



Two-Step Process

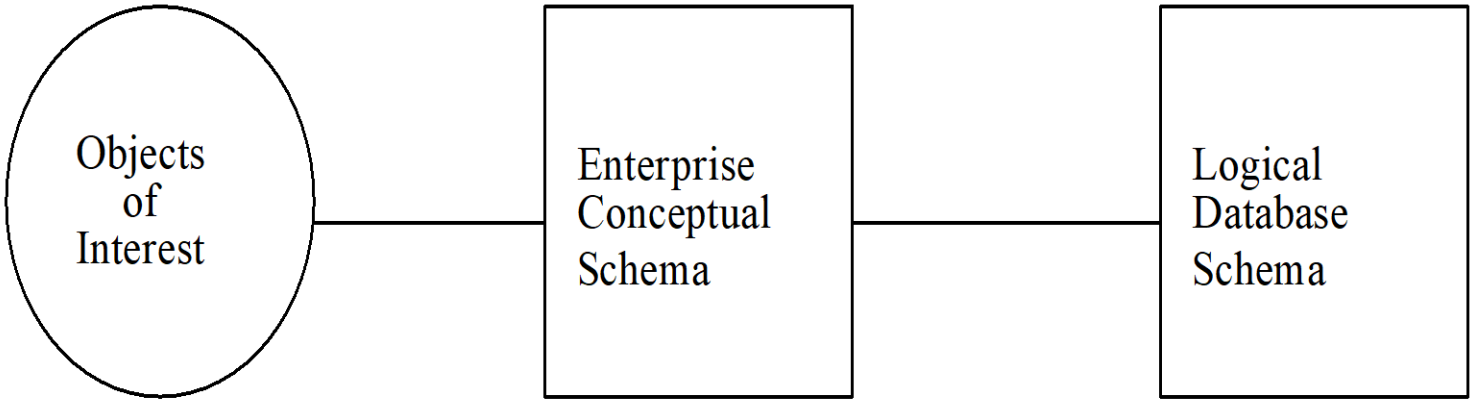
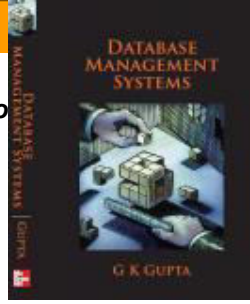


Figure 1



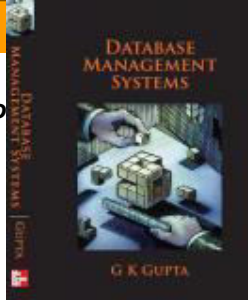
Database Modelling

First phase - an overall view (or model) of the *real world* (also called miniworld or UoD) is built.

The aim is to represent, as accurately as possible, the information structures of the enterprise that are of interest.

Second phase - the model is mapped to a schema appropriate to the DBMS.

No model provides complete knowledge but we accept such imperfections to keep the model simple.



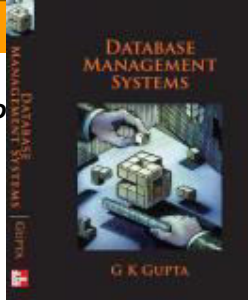
Database Modelling – First Phase

The first phase involves careful analysis of basic assumptions und erlying users' view.

For example:

- What is a player?
- What is a student?
- What is a faculty member?
- Could a person change name during their time?
- Could a person have dual nationality?
- What is a course? What is a degree?

Entity Relationship Model



The ER model views a miniworld as being composed of *entities* t hat have *relationships* between them.

An *entity* may be concrete or it may be abstract but it is an inform ation bearing object that is of interest.

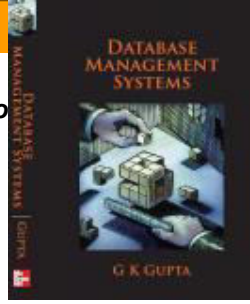
It may be a person, a place, a thing, an object, an event which can be distinctly identified and is of interest.

Many entities are alike since they are instances of the same entity.

Entities are thus classified into different *entity sets* (or entity types).

Each is a set of entity instances or entity occurrences of the sam e type.

Entity Relationship Model

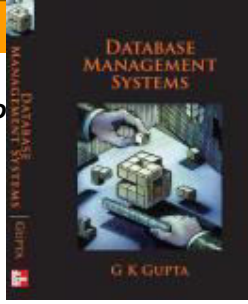


Consider an example.

We want to build a model so that we can print the scorecard of a n ODI.

What are the entities?

What are the relationships?



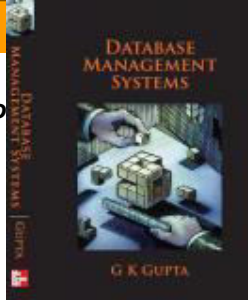
E-R Model

Relationships are associations or connections that exist between ent ities and may be looked at as mappings between two or more entit y sets.

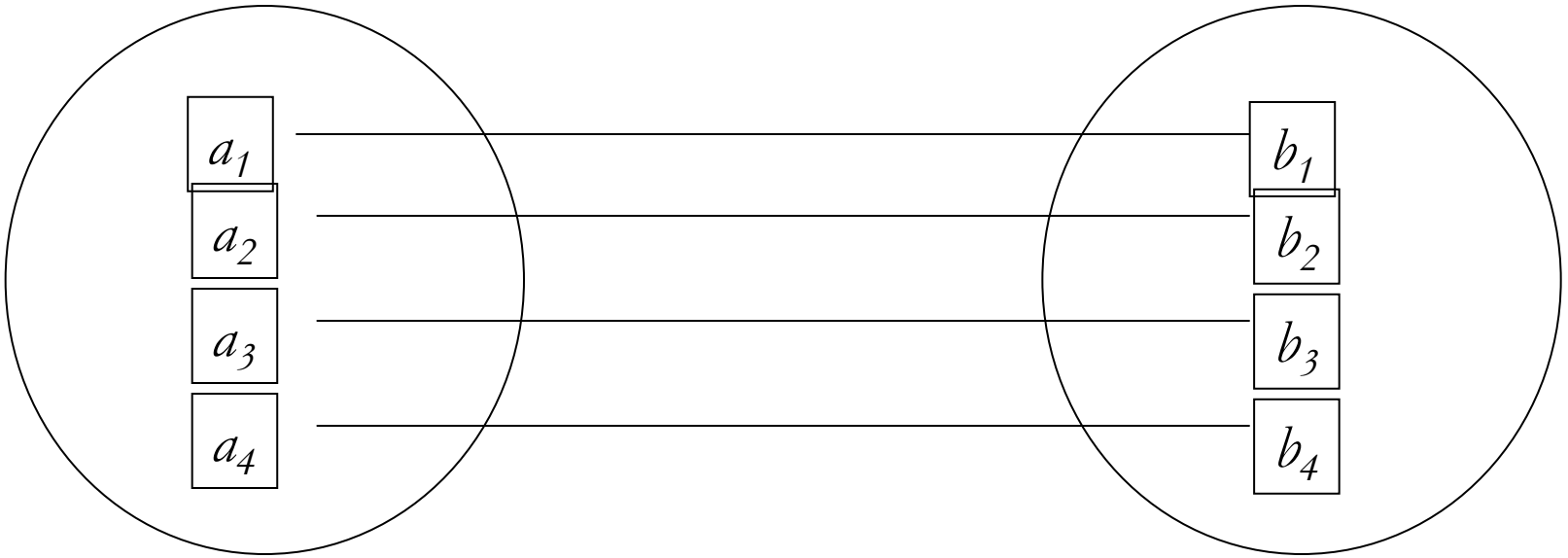
A *relationship* set is a set of relation instances of the same type.

Although we allow relationships among any number of entity sets , the most common cases are binary relationships between two en tity sets.

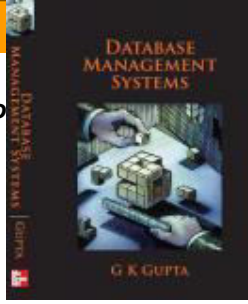
E-R Model



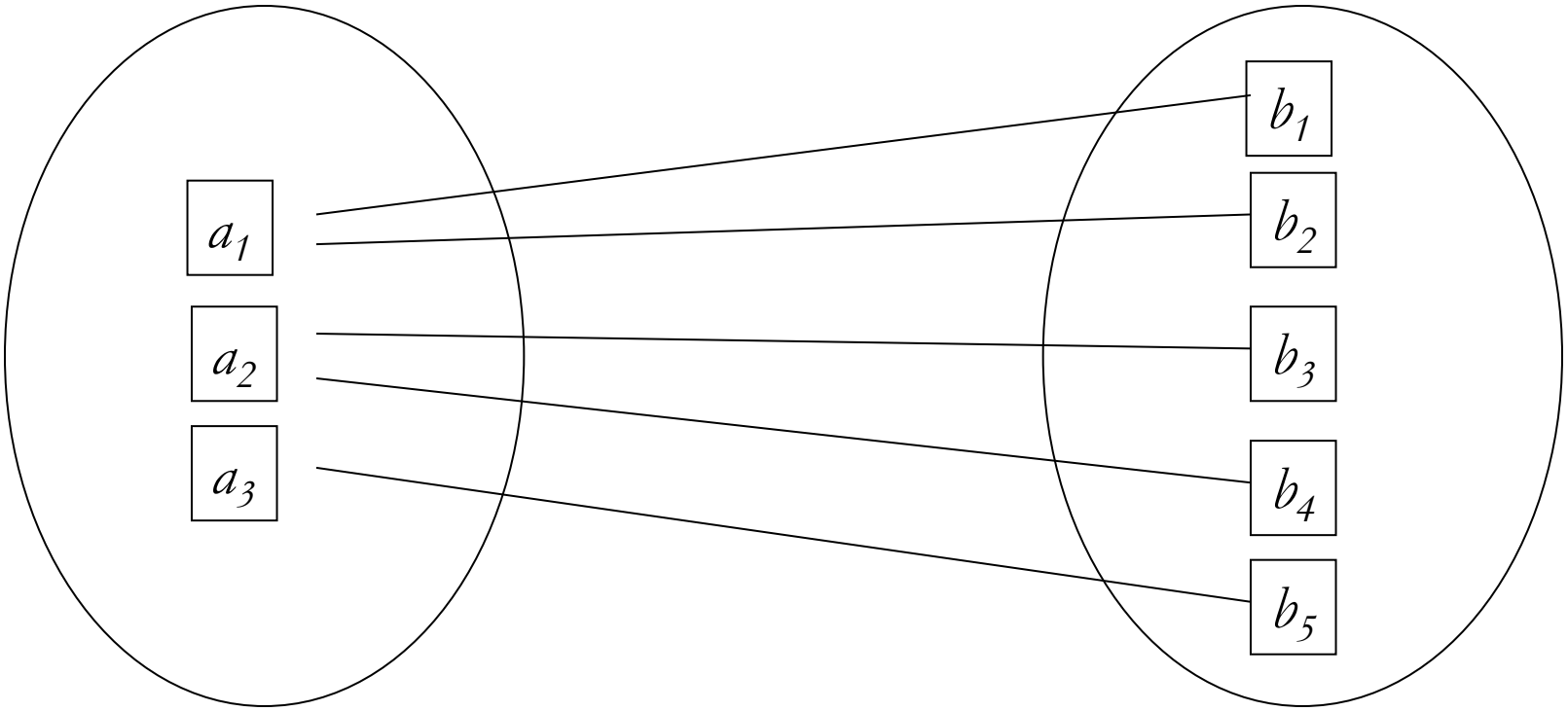
Often the relationships are classified as one-to-one, one-to-many or many-to-many.



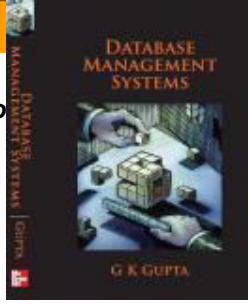
One-to-one relationship



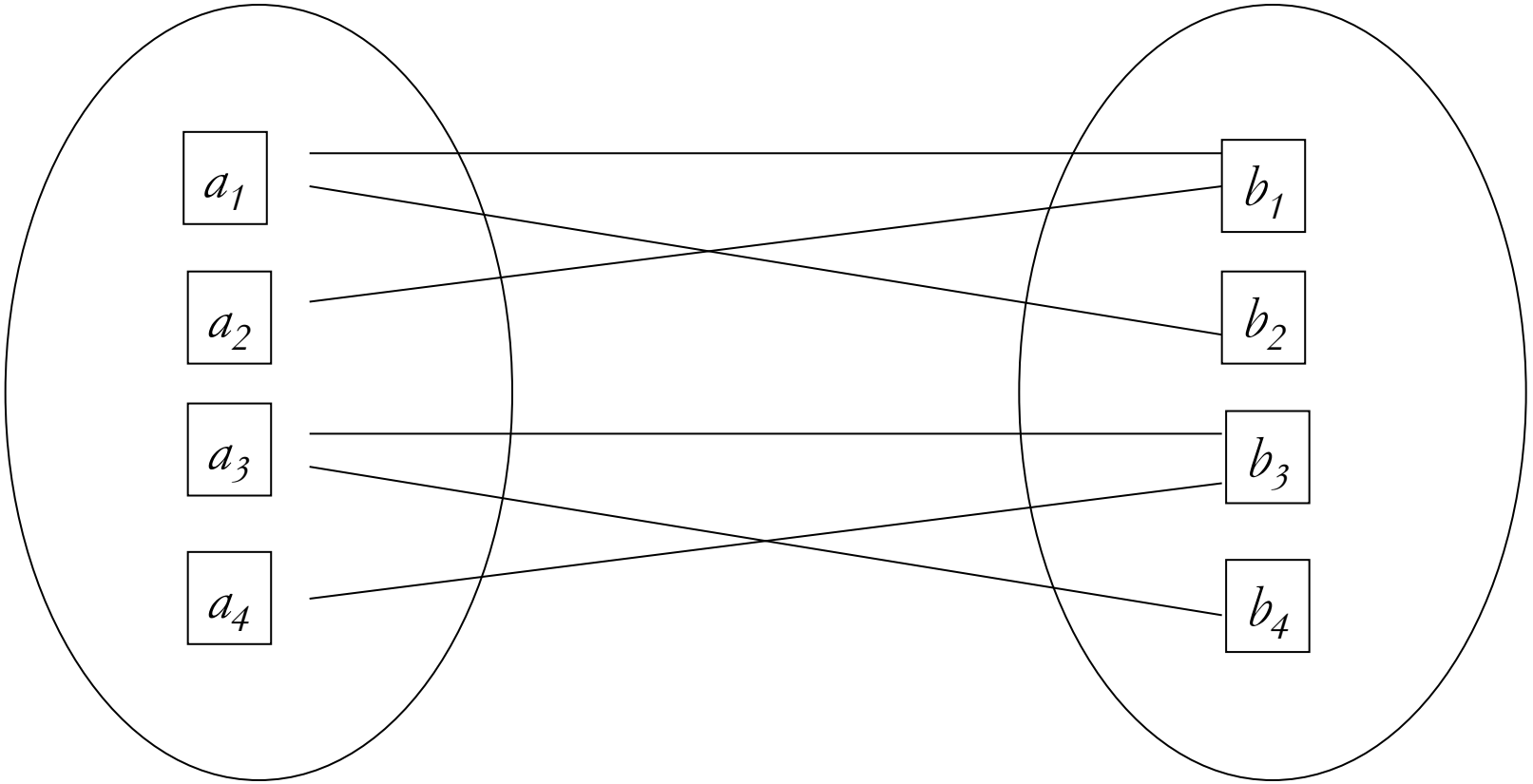
E-R Model



One-to-many relationship

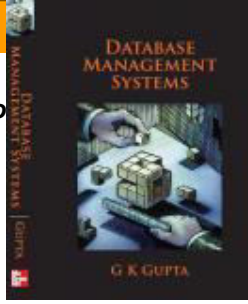


Mc Graw Hill Education



(a) many-to-many relationship (b)

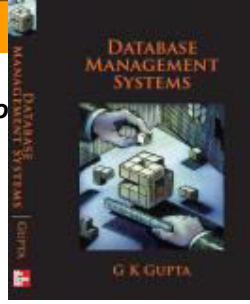
Example



Consider an example of entity sets employee and office.

- 1-1
- 1-n
- m-n

Attributes

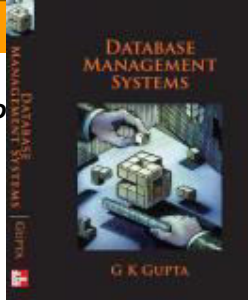


Each entity instance in an entity set has information of interest and is described by its properties or a set of *attributes*. All instances of an entity type must be described by the same properties.

Relationships may also have properties.

For each attribute there are a number of legal or permitted values. T hese set of legal values are sometimes called *value sets* or *domain* of th at attribute.

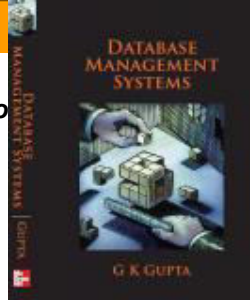
Attributes



Each entity instance in an entity set has information of interest and is described by its properties or a set of *attributes*. All instances of an entity type must be described by the same properties.

Relationships may also have properties.

For each attribute there are a number of legal or permitted values. T hese set of legal values are sometimes called *value sets* or *domain* of th at attribute.



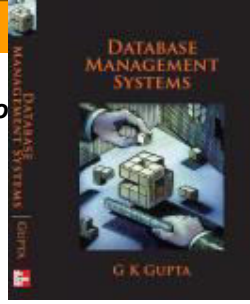
Key

It is essential to be able to identify each entity instance and each r elationship instance.

Therefore the set of attribute values be different for each entity i nstance. Sometimes an artificial attribute may need to be included to simplify entity identification.

A group of attributes (possibly one) used for identifying entities i n an entity set is called an *entity key*.

It is usually required that a key be *minimal* in that no subset of th e key should be key.



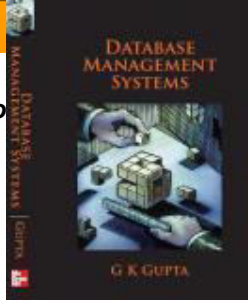
Attributes (cont'd)

When several minimal keys exist (such keys are often called *candida te keys*), one of them is chosen as the *entity primary key*.

Similarly each relationship instance in a relationship set needs to be identified.

This identification is always based on the primary keys of the entity sets involved in the relationship set.

Therefore while entities can exist on their own, relationships cannot since they depend on the entities that they relate.



Weak Entities

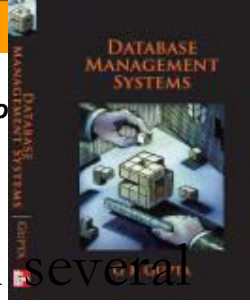
In certain cases, the entities in an entity set cannot be uniquely identified by the values of their own attributes.

Such entities are called *weak entities*.

Entities that have primary keys are called *strong entities*.

Similarly, a relationship may be weak or strong.

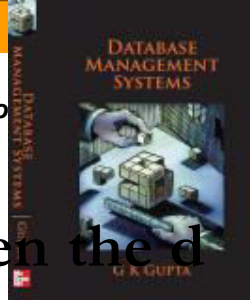
For example, any relationship between the children of employees and the schools they attend would be a weak relationship.



Mc Graw Hill **Education**

The Northern India University (NIU) is a large institution with several campuses scattered across north India. Academically, the university is divided into a number of faculties, such as Faculty of Arts and Faculty of Science. Some of the Faculties operate on a number of campuses. Faculties, in turn, are divided into schools; for example, the School of Physics, the School of Information Technology. The schools are not divided further into departments. Each student in the University is enrolled in a single course of study which involves a fixed core of subjects specific to that course as well as a number of electives taken from other courses. Each course is offered by one particular school. Each school has a number of lecturing staff. A student is awarded a grade in each subject taken.

Design an E-R model for the University



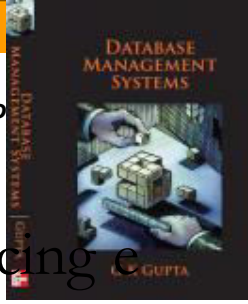
Mc Graw Hill Education

Build an E-R Model and display it as an E-R Diagram given the description of an enterprise given in the last exercise.

The entities need to be defined. Then the relationships are identified. The entities are highlighted below.

Medical Clinic Database – Design a database model for a medical clinic. The clinic has a number of regular **patients** and new patients come to the clinic regularly. Patients make **appointments** to see one of the **doctors**; several doctors are attending the clinic. Patients have **families** and the family relationships are important. A medical **record** of each patient needs to be maintained. Information on prescriptions, insurance, allergies, etc needs to be maintained.

Relationships and attributes may now be identified and an E-R diagram drawn.



Mc Graw Hill **Education**

A horse racing enthusiast wants to build a database of horse racing events and horses. The performance of horses (including dividends, odds before the race) over a number of races at a number of tracks in a variety of track and weather conditions needs to be stored. You also need to store information on incorrect weights, protests (dismissed or upheld), and jockey suspensions. Details of breeding of each horse in terms of its sire and dam.

Information about the race track (length, type, etc) and handicaps needs to be stored. Information about the owners, trainers and jockeys also needs to be stored. The winnings of each horse, owner, trainer over the life-time are sometime needed.

Horses are required to carry weights according to their racing history, age, gender, distance of race and time of year.

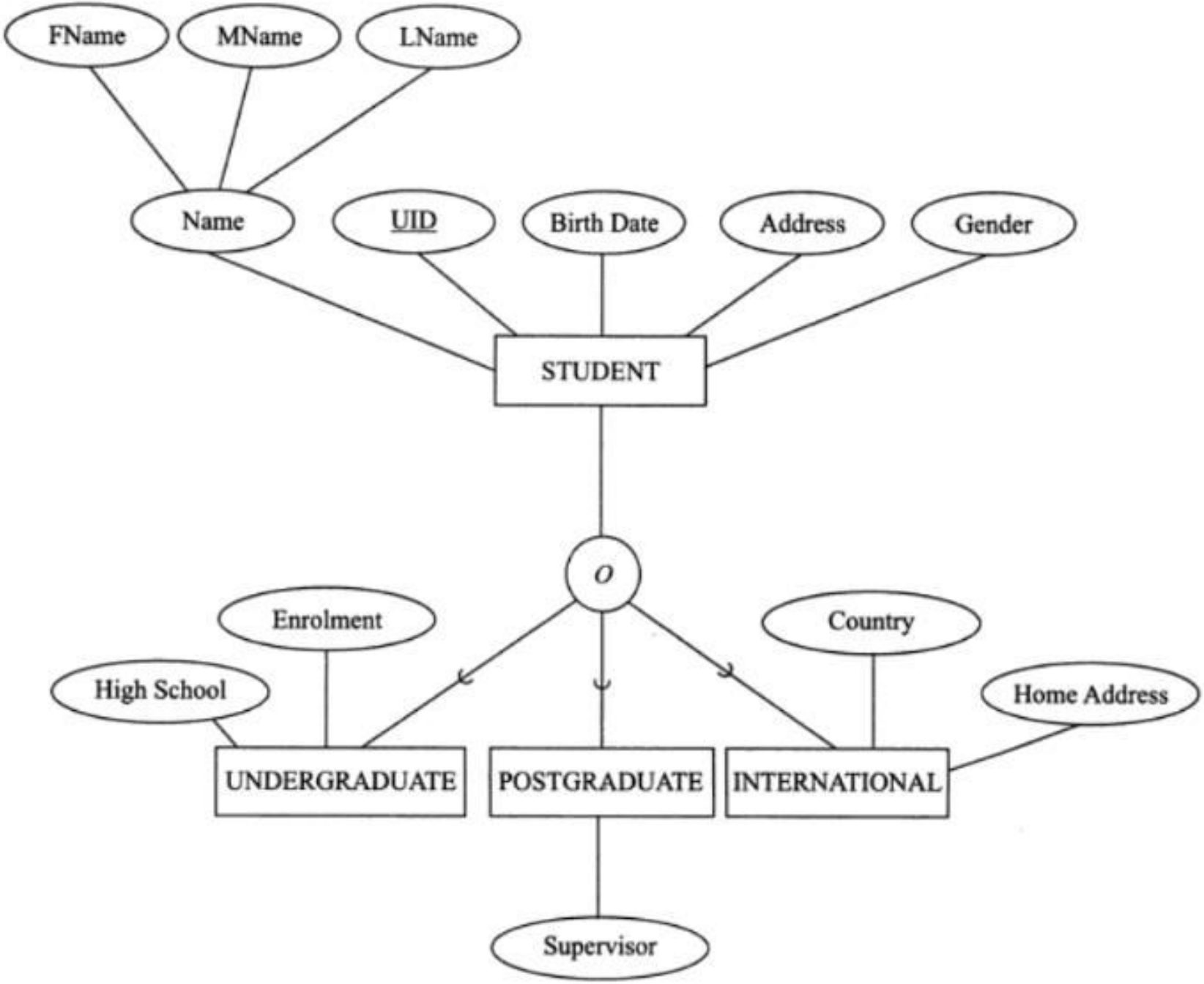
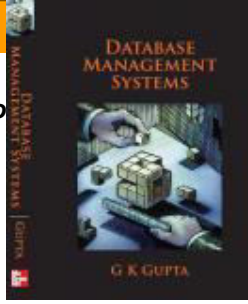


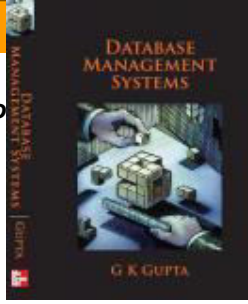
Figure 2.10 EER Diagram showing supertype and subtypes in specialization



ER-Model Guidelines

Although the ER Model is simple, problems can arise.

For example, it is not always easy to decide when a given thing should be considered an entity.



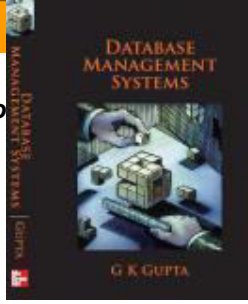
Guidelines

The rule of thumb to use in such situations is to ask the question "is city an object of interest to us?".

If the answer is yes, we must have some more information about city than just its name and then it should be considered an entity.

Otherwise city is an attribute of supplier.

Note that multi-value attributes should be classified as entities.



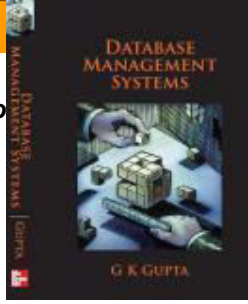
Guidelines

If a property of an entity has several values, the property itself should be made an entity.

This is done to simplify design and implementation decisions to be made later.

Attach attributes to entities they most directly describe.

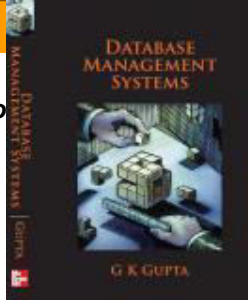
Unlike entities should not be grouped in the same set.



Database Design Process

One possible approach to database design involves the following major steps:

1. Study the description of the application.
2. Identify entity sets that are of interest to the application.
3. Identify relationship sets that are of interest to the application.
4. Determine whether relationships are 1:1, 1:n or M:n.



Design Process

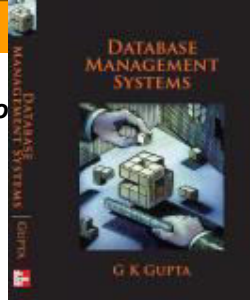
5. Draw an entity-relationship diagram for the application.

6. Identify value sets and attributes for the entity sets and the relatio nship sets.

7. Identify primary keys for entity sets.

8. Check the ERD conforms to the description of the application.

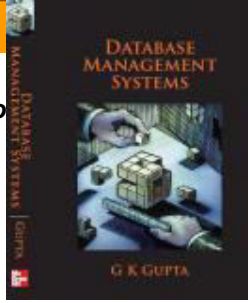
9. Translate the ERD to the DBMS database model.



Strengths of E-R Model

- Simplicity
- Visual Representations
- Effective Communication
- Ease of Mapping to Relational Model
- Focus on the model

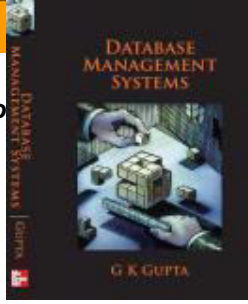
Limitations of the E-R Model



A simple rigorous definition of entity is not possible since there is no absolute distinction between entity types and attributes.

Usually an attribute exists only as related to an entity type but in so me contexts an attribute may be viewed as an entity.

There is no absolute distinction between an entity type and a relatio nship type although a relationship is usually regarded as unable to e xist on its own.

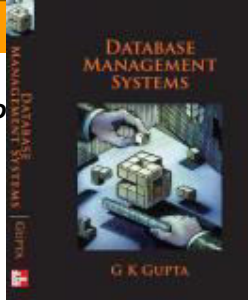


Limitations

For example, an enrolment cannot be expected to exist on its own w ithout the entities students and subjects.

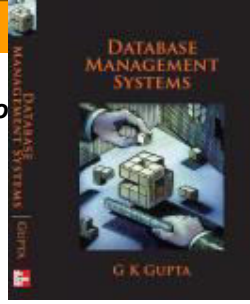
We have assumed that only current attribute values are of interest.

For example, when an employee's salary changes, the last salary value disappears for every. In many applications this situation would not be satisfactory.



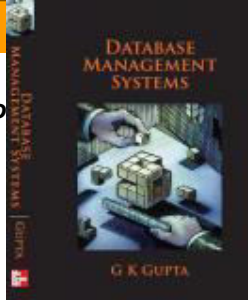
End of Chapter 1

Mc Graw Hill Education

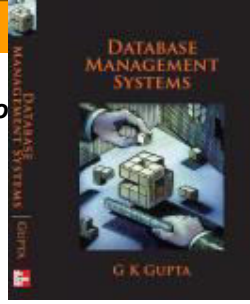


DATABASE MANAGEMENT SYSTEMS

G K GUPTA

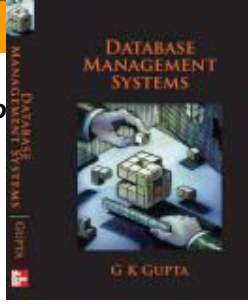


Chapter 3
The Relational Model



Objectives

- to introduce the relational model and its components
- to define the properties of a relation
- to explain how an E-R model is mapped to the relational model
- describe how information is retrieved and modified in the relational model
- explain how data integrity is maintained in the relational model
- discuss the advantages and disadvantages of using the relational model
- discuss the older database models, the network and hierarchical models



Mc Graw Hill Education

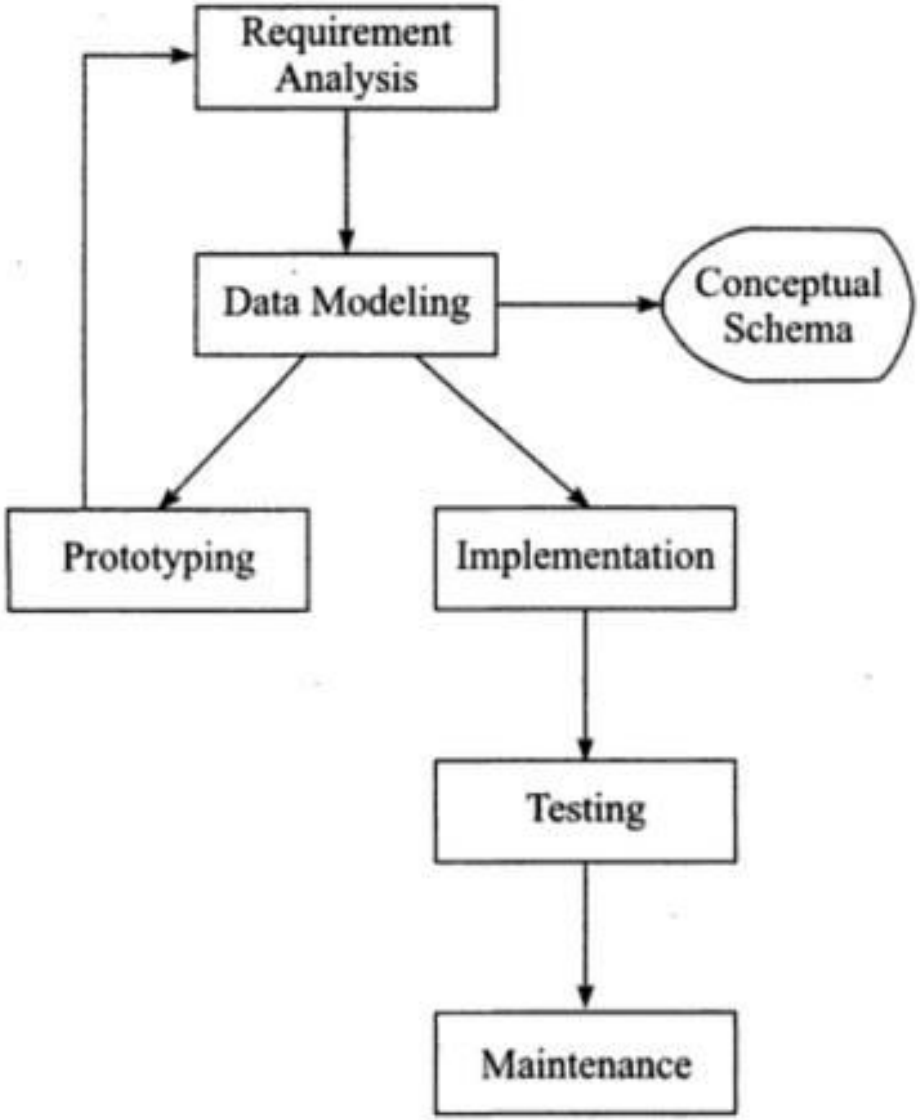
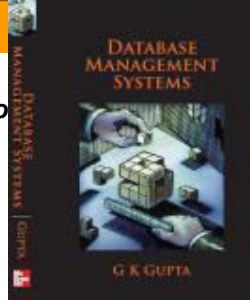


Figure 1.8 Database Design Lifecycle

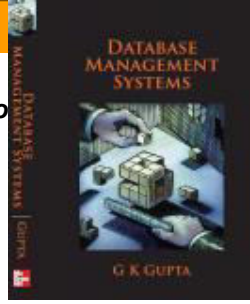


The Relational Model

The model was proposed by E. F. Codd in 1970

and consists of the following three components:

- *Data Structure* - a data structure types for building the database.
- *Data Manipulation* - a collection of operators that may be used to retrieve, derive or modify data stored in the database.
- *Data Integrity* - a collection of rules that implicitly or explicitly define a consistent database state or changes of states.

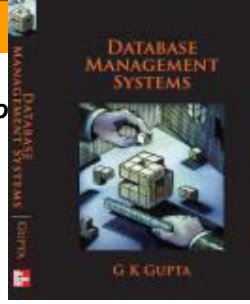


Data Structure

The beauty of the relational model is the simplicity of its structure.

Its fundamental property is that all information about the entities a nd their attributes as well as the relationships is presented to the use r as tables (called *relations*).

The rows of the tables may be considered records and the columns as fields.

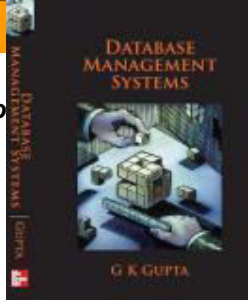


Example of a Relation

attributes (or columns)

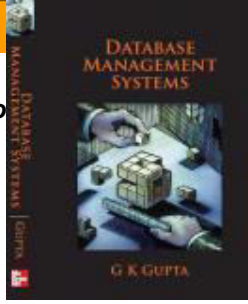
ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

tuples (or rows)



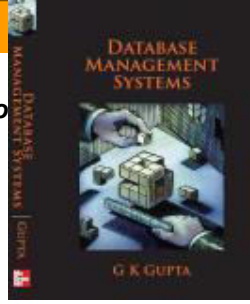
Properties of Relations

- Each relation contains only one record type.
- Each relation has a fixed number of columns that are explicitly and uniquely named.
- No two rows in a relation are the same.
- Each item or element in the relation is atomic.
- Rows have no ordering associated with them.
- Columns have no ordering associated with them (although most commercially available systems do).



Terminology

- *Relation* - essentially a table
- *Tuple* - a row in the table
- *Attribute* - a column in the relation
- *Degree of a relation* - number of attributes in the relation. A relation with degree *N* is *N-ary* relation.
- *Cardinality of a relation* - number of tuples
- *Domain* - a set of values that an attribute is permitted to take
- *Primary key* - an attribute (or a set of attributes) that uniquely identifies each tuple and satisfies minimality



Some Definitions

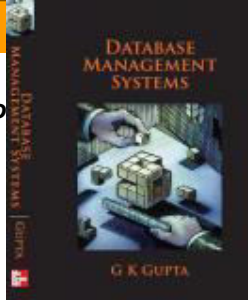
Definition – Candidate Key

An attribute (or a set of attributes) is called a candidate key of the relation if it satisfies the following properties:

(a) the attribute or the set of attributes uniquely identifies each tuple in the relation (called the uniqueness property), and

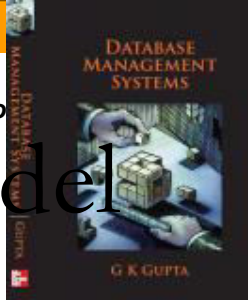
(b) if the key is a set of attributes then no subset of these attributes has property (called the minimality property).

Some Definitions



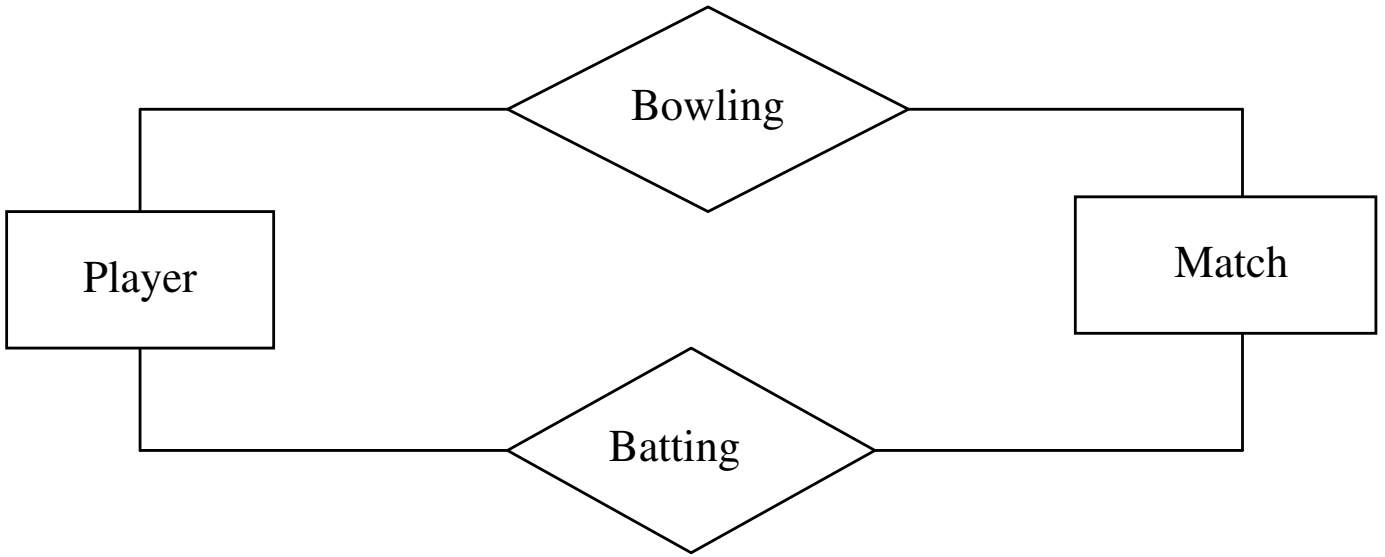
Definition – Primary Key

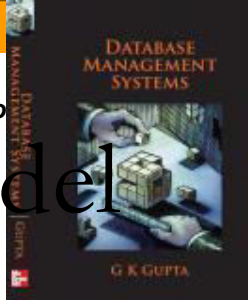
One (and only one) of the candidate keys is arbitrarily chosen as the primary key of the table. Primary key therefore has the properties of uniqueness and minimality.



Mapping ER Model to Relational Model

Consider the following ODI cricket database





Mapping ER Model to Relational Model

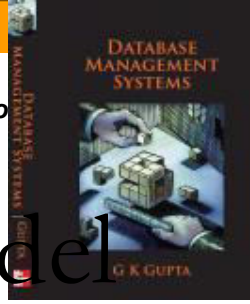
The attributes for the entities *Match* and *Player* are:

Match (MatchID, Team1, Team2, Ground, Date, Winner)
Player (PlayerID, LName, Fname, Country, Yborn, Bplace, Ftest)

The attributes for the entities *Batting* and *Bowling* are:

Batting(MatchID, PID, Order, Hout, FOW, Nruns, Mts, Nballs, Fours, Sixes)

Bowling(MatchID, PID, Novers, Maidens, Nruns, NWickets)



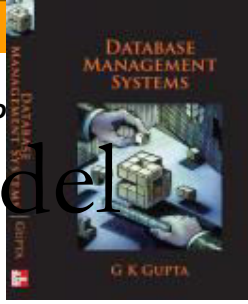
Mapping ER Model to Relational Model

Step 1 – Mapping Strong Entities

Each strong entity in an E-R model is represented by a table in the relational model.

Each attribute of the entity becomes a column of the table.

For example, the entities *Player* and *Match* may be directly transformed into two tables of the same names.



Mapping ER Model to Relational Model

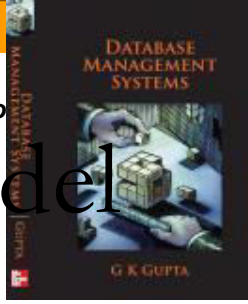
Step 2 – Mapping Weak Entities

Each weak entity may be mapped to a relation that includes all simple attributes of the entity as well as the primary key of the dominant entity.

The primary key of the relation representing a weak entity will be the combination of the primary key of the dominant entity and the partial key of the weak entity.

Step 3 – Mapping Relationships

There can be three types of relationship; 1:1, 1:m and m:n. Let us deal with each of them.

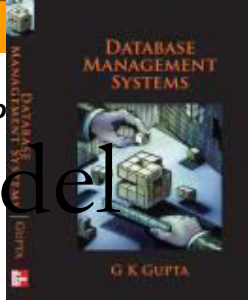


Mapping ER Model to Relational Model

Step 3b – Mapping one-to-many relationships

In a 1:m relationship, one entity instance of the first entity may be related to a number of instances of the second entity but each instance of the second entity is related to only one instance of the first entity.

It is straightforward to represent the relationship by putting the primary key of the second entity in the first entity. This is called *key propagation*.

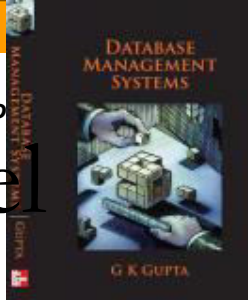


Mapping ER Model to Relational Model

Step 3c – Mapping many-to-many relationships

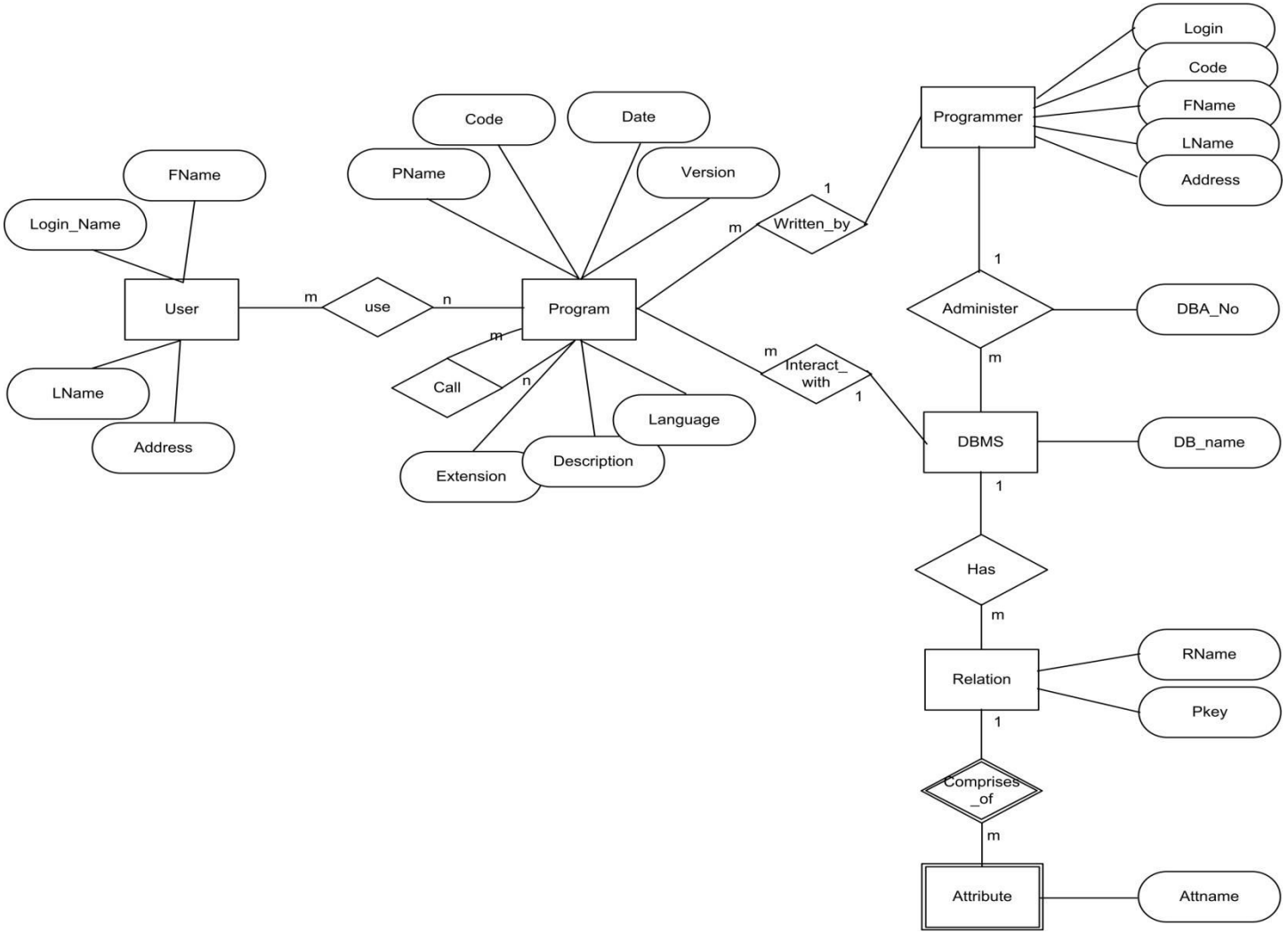
Key propagation is not possible for m:n relationships because it is not possible to propagate primary keys of instances of one of the related entity to an instance of the related other entity since there may be more than one such instance of the other entity.

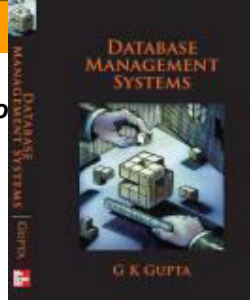
Many-to-many relationships therefore must be represented by a separate table.



Education

Mapping a more complex ER Model





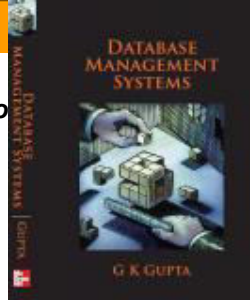
Data Manipulation

Querying a relational database is manipulating the database.

For example, in a college database, one may wish to know

names of all students enrolled in AA , or
names of all subjects taught by Abdul Salam.

Such queries may be answered by using a data manipulation language that is available for relational databases.



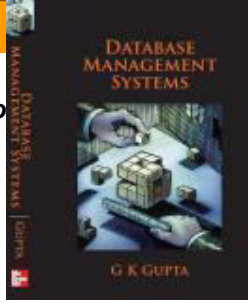
Data Integrity

The aim of data integrity is to specify rules that implicitly or explicitly define a consistent database state or changes of state.

This is done to stop the user from doing things that generally do not make sense.

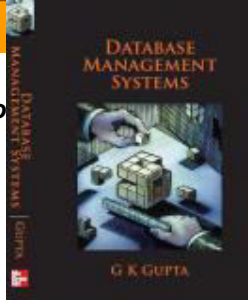
The integrity constraints are necessary to avoid situations like the following;

Some data has been inserted in the database but it cannot be identified (that is, it is not clear which object or entity the data is about).



Integrity

- A student is enrolled in a course but no data about him is available in the relation that has information about students.
- During query processing, a student number is compared with a course code (this should never be required).
- A student quits the university and is removed from the student relation but is still enrolled in a course.

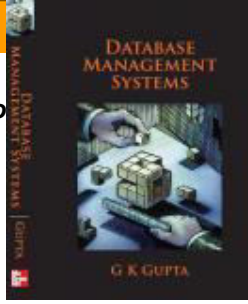


Integrity

Integrity constraints on a database may be divided into two types:

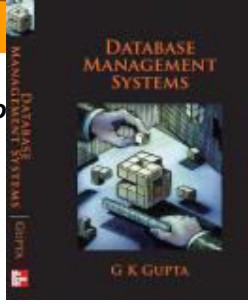
(1) *Static Integrity Constraints* - these are constraints that define valid states of the data. These constraints include designations of primary keys etc.

(2) *Dynamic Integrity Constraints* - these are constraints that define side-effects of various kinds of transactions (e.g. insertions and deletions).



Static Integrity

- Primary Keys
- Domains
- Foreign Keys and Referential Integrity
- Nulls

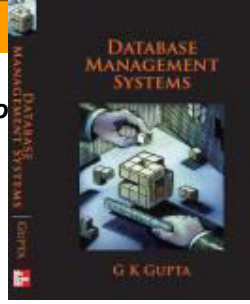


Primary Keys

The primary key of a relation is a permanently selected unique identifier.
It is an attribute or a set of attributes with the property that no two rows of the table have the same values of the identifier.

Before defining the concept of primary keys precisely, let us define *candidate key*.

An attribute or a set of attributes *K* of the relation *R* is a candidate key of *R* if and only if it satisfies the following:



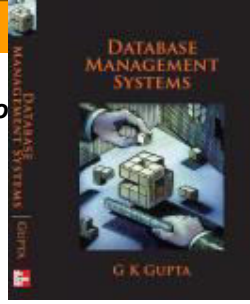
Keys

- *Uniqueness* – The attribute or a set of attributes uniquely identifies each row in a relation.
i.e. no two tuples of R will ever have the same values of *K*.

- *Minimality* – If the key is a set of attributes then no subset of these attributes has the uniqueness property.

From the set of candidate keys, one is chosen as the primary key. Any tuple in a database may be identified by specifying relation name and the value of tuple's primary key.

For a tuple to exist in a relation, it must be identifiable and therefore it must have primary key.

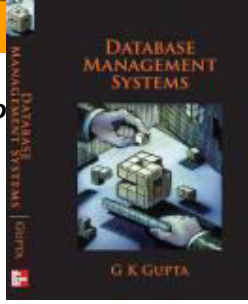


Keys

The relational data model therefore imposes the following two integrity constraints:

- no component of a primary key value can be null
- attempts to change the value of a primary key must be carefully controlled.

The first constraint is necessary because if we want to store information about some entity, then we must be able to identify it, otherwise difficulties are likely to arise.



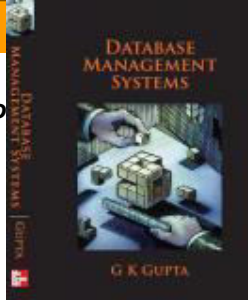
Integrity

For example, if a relation CLASS (s+id, lecturer, cno) has (s_id, lecturer) as the primary key then allowing tuples like the following leads to ambiguity:

3123	NULL	IT100
NULL	Smith	IT100

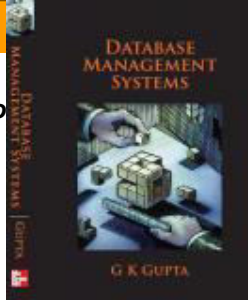
Since the primary key is the tuple identifier, changing it needs very careful controls.

Codd has suggested three possible approaches:



Integrity

- Only a select group of users be authorised to change primary key values.
- Updates on primary key values be banned. If it was necessary to change a primary key, the tuple would first be deleted and then a new tuple with new primary key value but same other attributes' values be inserted.
- A different command for updating primary keys be made available.

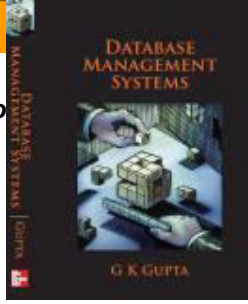


Domains

Although some commercial database systems do not provide facilities for specifying domains, domains could be specified as below:

create	DOMAIN	Name1	CHAR(10)
create	DOMAIN	S-id	INTEGER
create	DOMAIN	Name2	CHAR(10)

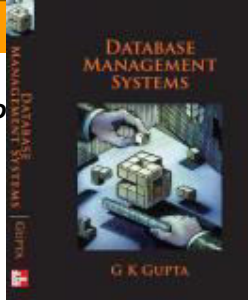
Note that Name1 and Name2 are both character strings of length 10 but they now belong to different (semantic) domains.



Domains

It is important to denote different domains to

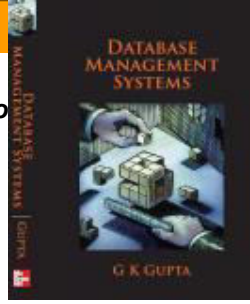
- constrain unions, intersections, differences, and equi-joins of relations.
- let the system check if two occurrences of the same database value denote the same real world object.



Foreign Keys

As discussed earlier, the primary key of a relation is a unique identifier for that relation.

Often it is necessary to refer to the object that is described by one relation in another relation. It is then necessary to use the same unique identifier. Any column(s) in the other relation containing those values of the identifier are called the *foreign key*.

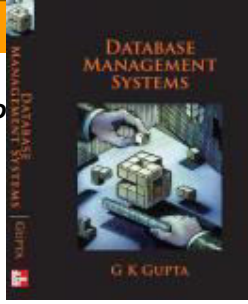


Foreign Keys

To be more precise, a column or column combination *FK* of a table *R2* is a foreign key if

- Each nonnull value of *FK* is identical to the value PK in some relation *R1*.
- Each value of *FK* is either wholly null or wholly nonnull.

A foreign key is a reference to the row containing the matching primary key value. In the following relation the supervisor number is a foreign key (why?): *employee (empno, empname, supervisor-no, dept)*

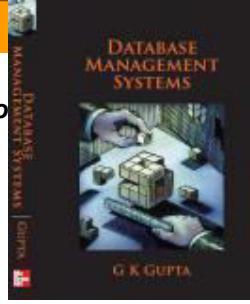


Foreign Keys

A given foreign key and the corresponding primary key must be defined on the same underlying domain.

The foreign key may or may not be a component of the primary key of its containing relation.

When a primary key is modified and that primary key value is referred in other relations of the database, each of these references must either be modified accordingly or be replaced by NULL values. Only then we can maintain referential integrity.



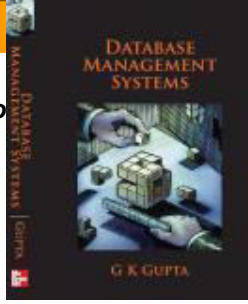
Foreign Keys

What are the foreign keys in the following relations?

Nation(name, area, population, year1, GDP, year2)
Neighbour(name1, name2)
Olympics(event_name, year_started, year_dropped)
Gold(event_name, year, winner, nationality)

The following constraint is called *referential* integrity constraint:

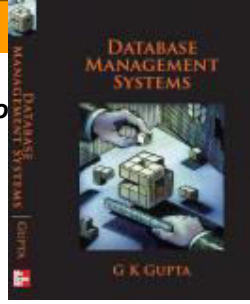
If a foreign key F in relation R matches the primary key P of relation S than every value of F must either be equal to a value of P or be wholly null.



Foreign Keys

The justification for referential integrity constraint is simple.

If there is a foreign key in a relation then its value must match with one of the primary key values to which it refers. That is, if an object or entity is being referred to, the constraint ensures the referred object or entity exists in the database.

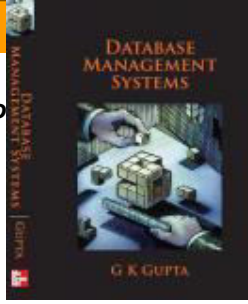


Nulls and Missing Information

The issue of Null values in a database is extremely important since information is often incomplete in the real world. A large number of issues arise. For example, if two NULL values are compared, should they be considered equal? How should NULL values be treated when aggregate operators are used?

When information is missing from a database, the DBA must consider:

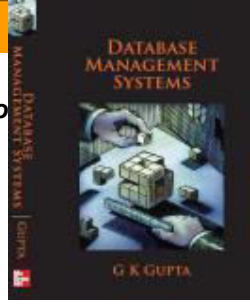
(1)What kind of information is missing?
(2)What is the main reason for it being missing?



Nulls and Missing Information

The missing information may be *missing-but-applicable* or it may be *missing-and-inapplicable*. The other types of missing information is possible e.g. *missing - not defined* or *missing - does not exist*.

It has been suggested that different type of *marks* be used to represent different types of missing information, for example, *A-mark* and *I-mark*.

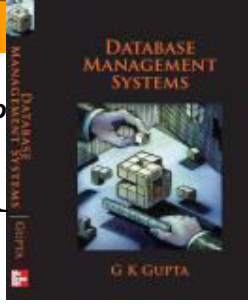


Nulls

Codd suggests use of three-value logic to solve some of the difficulties that arise because of missing information. For example, let us pose the following queries:

find all students from Germany.
find all students from outside Germany.

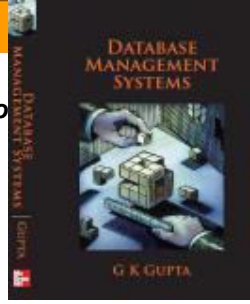
We may assume that the two queries will include all the students but they may not if some information is missing.



Advantages of the relational model

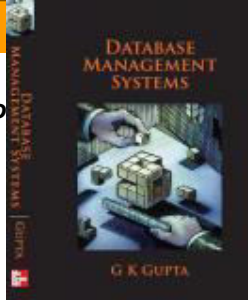
Codd in his 1970 paper indicated that there were four major objectives in designing the relational data model.

- *Data Independence* - to provide a sharp and clear boundary between the logical and physical aspects of database management.

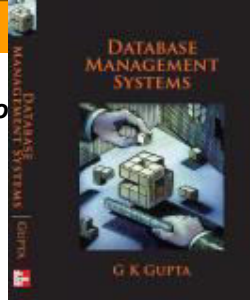


Advantages

- *Simplicity* - to provide a simpler structure than were being used at that time. A simple structure is easy to communicate to users and programmers and a wide variety of users can interact with a simple model.
- *Set-processing* - to provide facilities for manipulating a set of records at a time so that programmers are not operating on the database record by record.
- *Sound Theoretical Background* - to provide a theoretical background for database management field.

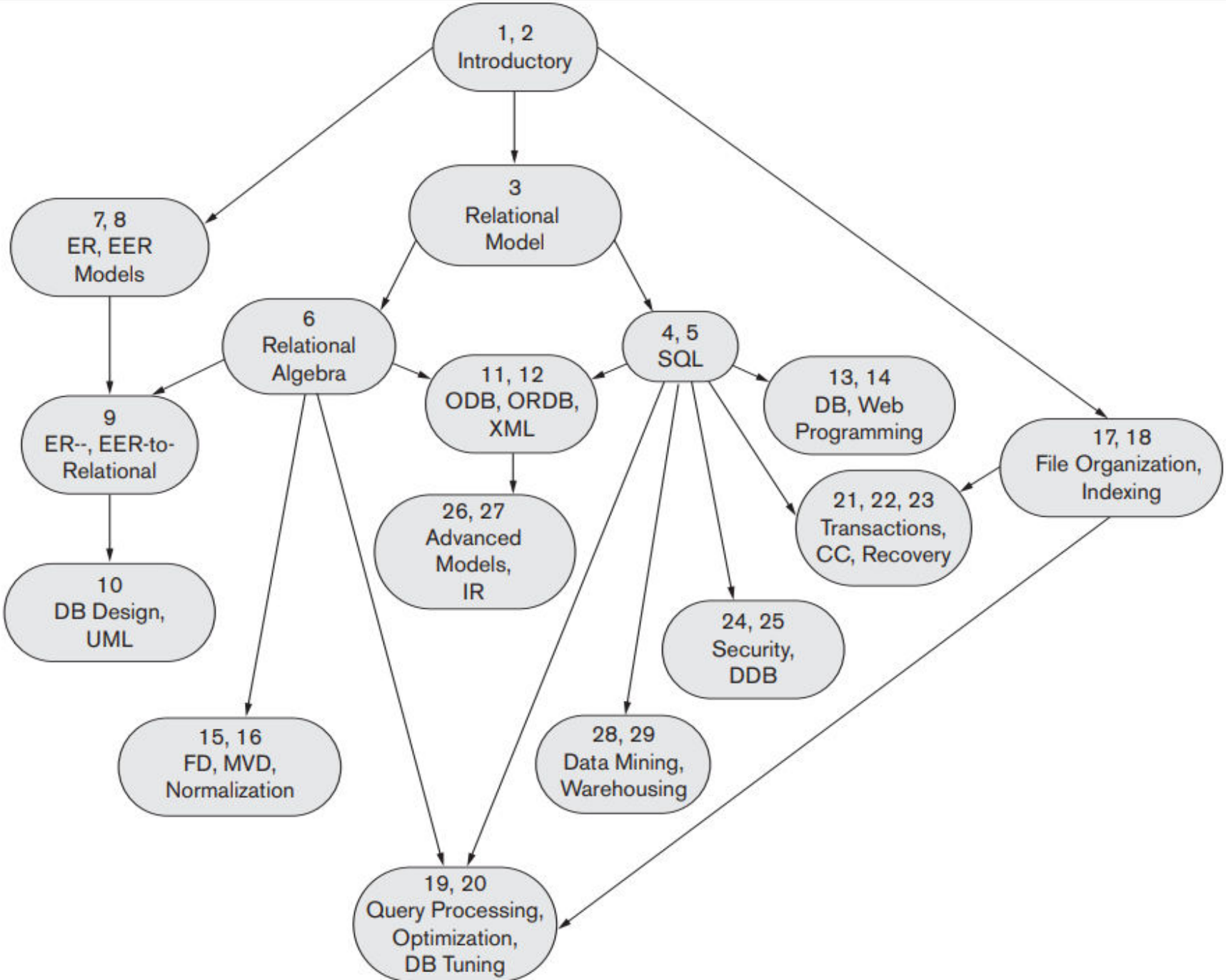


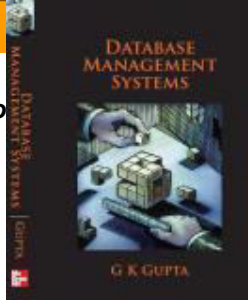
Mc Graw Hill Education



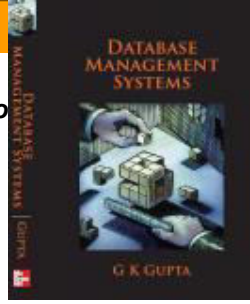
DATABASE MANAGEMENT SYSTEMS

G K GUPTA


1, 2
Introductory
7, 8
ER, EER
Models
3
Relational
Model
6
Relational
Algebra
4, 5
SQL
11, 12
ODB, ORDB,
XML
13, 14
DB, Web
Programming
17, 18
File Organization,
Indexing
9
ER--, EER-to-
Relational
21, 22, 23
Transactions,
CC, Recovery
26, 27
Advanced
Models,
IR
10
DB Design,
UML
24, 25
Security,
DDB
15, 16
FD, MVD,
Normalization
28, 29
Data Mining,
Warehousing
19, 20
Query Processing,
Optimization,
DB Tuning



Relational Algebra



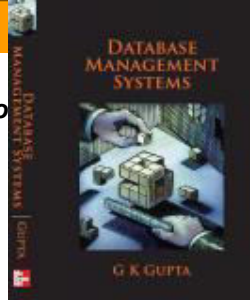
Relational Algebra

Relational algebra is a procedural language that uses certain primitive operators that take tables as inputs and produce tables as outputs.

The language provides a step-by-step procedure for computing the result.

Each relational algebra operator is either unary or binary that is it performs data manipulation on one or two tables.

Relational algebra does not include a looping construct.



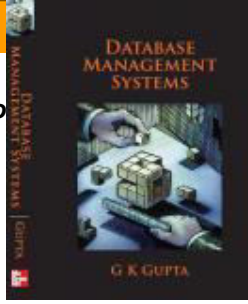
Relational Algebra

The relational algebra expressions are similar to algebraic expressions (e.g. 2*a+b*c) except that we are now dealing with relations and not numbers.

The result of applying one or more relational operators to a set of (possibly one) relations is a new relation.

Basic Algebra Commands

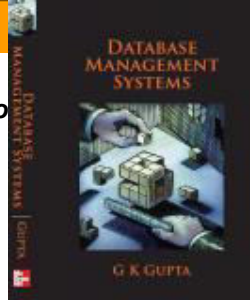
SELECT	PROJECT
UNION	PRODUCT
JOIN	DIVIDE



Relational Algebra

There are a number of other operations:

- Outer Join
- Intersection
- Difference
- Division



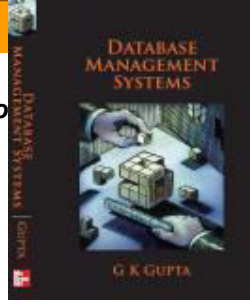
Mc Graw Hill Education

Unary Relational Operator – Selection

The operation of selecting certain rows from one table is called *selection* or *restriction*.
It is an unary operator (i.e. it operates on a single relation).

A restriction of a relation R such that an attribute A has value a is $R[A=a]$.

Similarly $R[A>a]$ is a restriction on R such that attribute A has values greater than a.

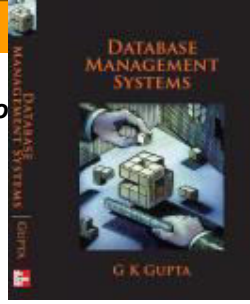


Mc Graw Hill Education

Unary Relational Operator – Selection

A selection condition (for example, attribute C not being equal to c_2) results in selecting three rows of the table.

A	B	C	D	E
a_1	b_1	c_2	d_1	e_1
a_2	b_2	c_1	d_1	e_1
a_3	b_1	c_2	d_2	e_2
a_4	b_1	c_1	d_3	e_3
a_5	b_1	c_3	d_1	e_4



Unary Relational Operator – Selection

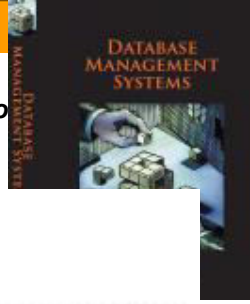
More formally, selection of a table *R* is a subset table (call it *S*) that includes all those rows in *R* that meet a condition specified by the user.

The number of columns (degree) in *S* is the same as that of *R*.

The number of rows (cardinality) of *S* is equal to the number of rows in R that meet the specified condition.

The result of applying selection to a table *R* such that the column *A* has value *a* will be denoted by *R[A=a]*.

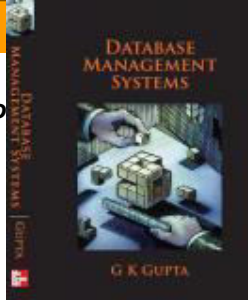
A more mathematical way of writing selection is to use the notation $\sigma_{A=a}[R]$.



Mc Graw Hill **Education**

Table 4.1 Result of a selection operator

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	NA
27002	Sharma	Ishant	India	1988	Delhi	2007



Examples

Example 1 – find all the Indian players that are in the database.

We do so by selecting rows in the table *Player* where the value of the attribute *Country* is India.

Player[*Country* = *'India'*]

or

$\sigma_{\text{Country = 'India'}}$[Player]

Mc Graw Hill Education

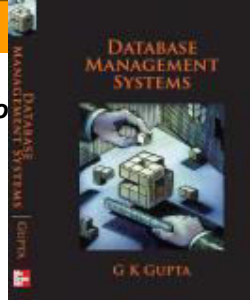
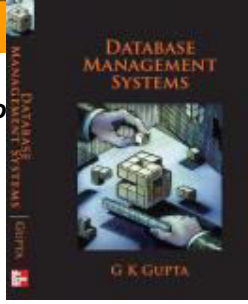


Table 4.2 Result of applying a selection operator to *Match*

MatchID	*Team1*	*Team2*	*Ground*	*Date*	*Winner*
2688	Australia	India	Sydney	2/3/2008	Team2
2689	Australia	India	Brisbane	4/3/2008	Team2



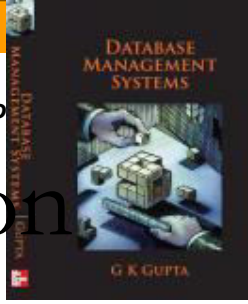
Example

Example 2 – Find what matches have been played in Australia.

We select those rows from the table *Match* that have column *Team1* value equal to 'Australia' assuming that *Team1* is the home team and these matches are not being played at a neutral country.

We could write it as below.

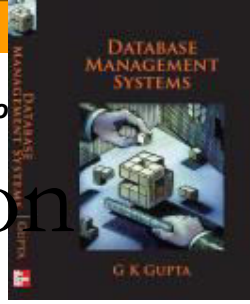
Match[*Team1* = *'Australia'*] *or* $\sigma_{\text{Team1 = 'Australia'}}$[Match]



Another Unary Operator – Projection

Projection is the operation of selecting certain columns from a table *R* to form a new relation *S*. Like restriction, projection is a unary operator.

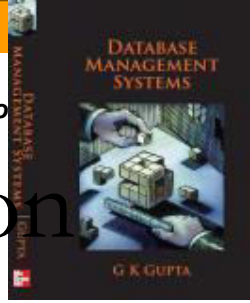
R[*A*] is a projection of *R* on *A*, *A* being a list of attributes (not necessarily distinct). Duplicate tuples, if any, are removed.



Another Unary Operator – Projection

A projection of the table on attributes C and E results in selecting the two highlighted columns of the table. Note that the first and second rows in the table with two columns are duplicates.

B	C	D	E
b_1	c_1	d_1	e_1
b_2	c_1	d_1	e_1
b_1	c_2	d_2	e_2
b_1	c_1	d_3	e_3
b_1	c_3	d_1	e_4



Another Unary Operator – Projection

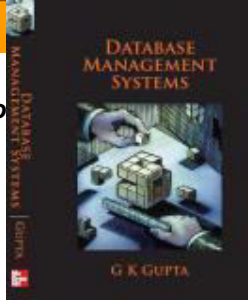
A projection of *R* on columns *a, b, c* is denoted by *R*[*a,b,c*]. Some authors use $\pi_{a,b,c}[R]$ to denote the projection.

More generally, $\pi_s(R)$ is a projection of table *R* where *s* is a list of columns, possibly only one column.

Example – Find the names of all players in the database.

Apply the projection operator to the table *Player* on attributes *FName* and *LName*.

Player[*Fname, LName*] *or* $\pi_{\text{Fname, LName}}$[Player]

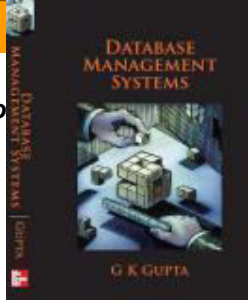


Examples of Projection

What do the following simple projections do?

Player[PlayerID, LName] *or* $\pi_{\text{PlayerID, LName}}$[Player]

Batting[PID] *or* π_{PID}[Batting]



Mc Graw Hill Education

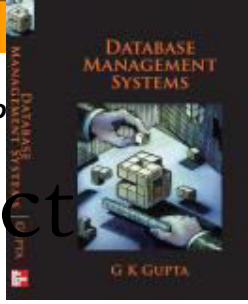
Binary Operation – Cartesian Product

The *Cartesian product* (also known as the *cross product*) of any two tables R and S (of degree m and n respectively) is a table $R \times S$ of degree $m + n$.

This product table has all the columns that are present in both the tables R and S and the rows in $R \times S$ are all possible combinations of each row from R combined with each row from S.

The number of rows in $R \times S$ therefore is pq if p is the number of rows in S and q in R.

The number of columns of $R \times S$ is $m + n$ if m and n are the number of columns in R and S respectively.



Education

Binary Operation – Cartesian Product

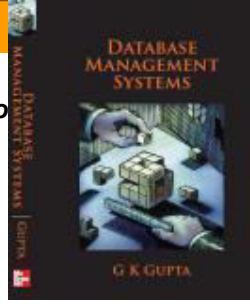
Player(PlayerID, LName, FName, Country, YBorn, BPlace, FTest)

Batting(MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes)

The product will have the following attributes:

Product(PlayerID, LName, FName, Country, YBorn, BPlace, FTest, MatchID, PID, Order, HOut, FOW, NRuns, Mts, NBalls, Fours, Sixes)

Is this product useful?

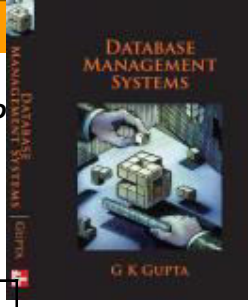


Binary Operation – Join

A Cartesian product followed by a restriction operator is a very important operator called the *join*.

PlayerID	LName	FName
89001	Tendulkar	Sachin
24001	Symonds	Andrew
99001	Lee	Brett
25001	Dhoni	M. S.

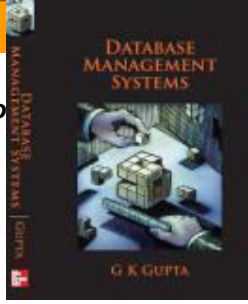
MID	PID	Order	NRuns
2689	89001	2	91
2689	99001	8	7
2689	24001	5	42
2755	25001	5	71



Education

Cartesian Product and the Join

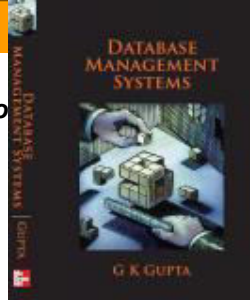
PlayerID	LName	FName	MID	PID	Order	NRuns
89001	Tendulkar	Sachin	2689	89001	2	91
24001	Symonds	Andrew	2689	89001	2	91
99001	Lee	Brett	2689	89001	2	91
25001	Dhoni	M. S.	2689	89001	2	91
89001	Tendulkar	Sachin	2689	99001	8	7
24001	Symonds	Andrew	2689	99001	8	7
99001	Lee	Brett	2689	99001	8	7
25001	Dhoni	M. S.	2689	99001	8	7
89001	Tendulkar	Sachin	2689	24001	5	42
24001	Symonds	Andrew	2689	24001	5	42
99001	Lee	Brett	2689	24001	5	42
25001	Dhoni	M. S.	2689	24001	5	42
89001	Tendulkar	Sachin	2755	25001	5	71
24001	Symonds	Andrew	2755	25001	5	71
99001	Lee	Brett	2755	25001	5	71
25001	Dhoni	M. S.	2755	25001	5	71



Mc Graw Hill Education

Equi-Join (has two equal columns)

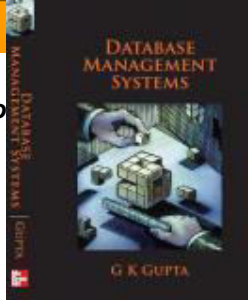
PlayerID	LName	FName	MID	PID	Order	NRuns
89001	Tendulkar	Sachin	2689	89001	2	91
99001	Lee	Brett	2689	99001	8	7
24001	Symonds	Andrew	2689	24001	5	42
25001	Dhoni	M. S.	2755	25001	5	71



Natural Join

Natural join derived from the equi-join by removing one of the two equal columns is given below.

PlayerID	LName	FName	MID	Order	NRuns
89001	Tendulkar	Sachin	2689	2	91
99001	Lee	Brett	2689	8	7
24001	Symonds	Andrew	2689	5	42
25001	Dhoni	M. S.	2755	5	71

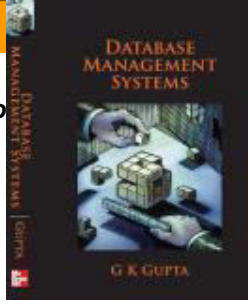


Join

The join of two relations is a restriction of their Cartesian product such that a specified condition is met.

The join is normally defined on one attribute from one relation in the join and one attribute from the second relation in the join such that the attributes are from the same domain and are therefore related.

The specified condition (called the *join predicate*) is a condition on the two attributes.

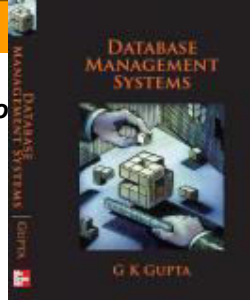


Join

The most commonly needed join is the join in which the specified condition is equality of the two attributes. This join is called an *equi-join*.

Since equi-join by definition has two equal attributes one of these is removed by applying projection.

The resulting relation is called the *natural join*.



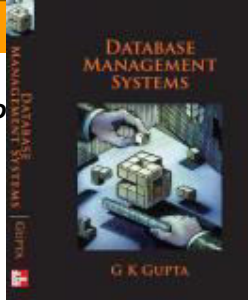
Natural Join

The natural join is obtained by applying a restriction and a projection to the Cartesian product $R \; X \; S$:

(a) For each attribute a that is common to both relations R and S, we select tuples that satisfy the condition $R.a = S.a$.

(b) For each attribute a that is common to both relations R and S, we project out the column $S.a$.

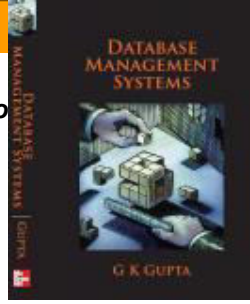
More efficient methods are used in practice.



Join

Is a natural join possible between two relations when the relations have more than one attribute in common?

The formal definition of natural join does in fact include that possibility.



Mc Graw Hill Education

Table 4.6 The relation *Player2*

PlayerID	*LName*	*FName*
89001	Tendulkar	Sachin
24001	Symonds	Andrew
99001	Lee	Brett
25001	Dhoni	MS

Table 4.7 The relation *Batting2*

MID	*PID*	*Order*	*NRuns*
2689	89001	2	91
2689	99001	8	7
2689	24001	5	42
2755	25001	5	71

Table 4.8 The relation *Bowling2*

MatchID	*PID*	*NWickets*
2689	99001	1
2755	91001	0
2689	24001	1

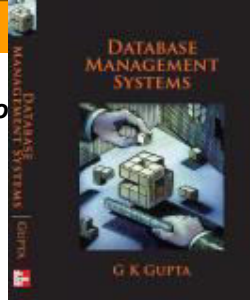
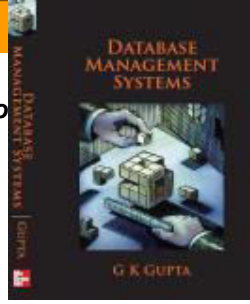


Table 4.12 Cartesian product of *Batting2* and *Bowling2*

MID	*PID*	*Order*	*NRuns*	*MatchID*	*BPID*	*NWickets*
2689	89001	2	91	2689	99001	1
2689	**99001**	**8**	**7**	**2689**	**99001**	**1**
2689	24001	5	42	2689	99001	1
2755	25001	5	71	2689	99001	1
2689	89001	2	91	2755	91001	0
2689	99001	8	7	2755	91001	0
2689	24001	5	42	2755	91001	0
2755	25001	5	71	2755	91001	0
2689	89001	2	91	2689	24001	1
2689	99001	8	7	2689	24001	1
2689	**24001**	**5**	**42**	**2689**	**24001**	**1**
2755	25001	5	71	2689	24001	1



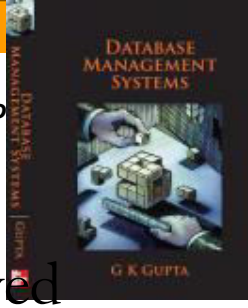
Mc Graw Hill Education

Table 4.13 Equi-join of the tables *Bowling2* and *Batting2* on two common attributes

MID	*PID*	*Order*	*NRuns*	*MatchID*	*BPID*	*NWickets*
2689	99001	8	7	2689	99001	1
2689	24001	5	42	2689	24001	1

Table 4.14 Natural join of the tables *Bowling2* and *Batting2* on two common attributes

MID	*PID*	*Order*	*NRuns*	*NWickets*
2689	99001	8	7	1
2689	24001	5	42	1

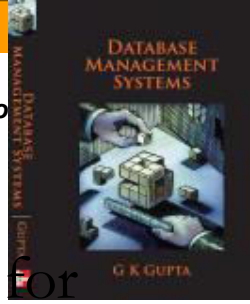


Education

year in *student* indicates the time that the student has been enrolled in the *degree*. *time_limit* provides maximum time allowed for each of the degrees.

student_id	student_name	address	degree	year
8700074	John Smith	9, Davis Hall	B.Sc.	6
8900020	Arun Kumar	90, Second Hall	M.Sc.	2
8801234	Peter Chew	88, Long Hall	B.A.	3
8612345	Chris Wantanabe	11, Main Street	Ph.D.	4
8465432	Bill Wilson	22, Second Hall	B.Com	6

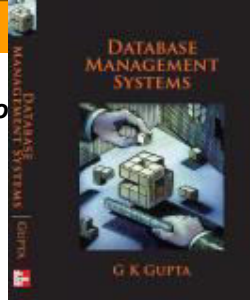
degree	years
B.A.	6
M.Sc.	6
B.Com	6
LL.B.	8
M.Sc.	3
Ph.D.	7



Mc Graw Hill **Education**

Find the list of students who have been enrolled in a degree for the maximum time allowed. Join *student* and *time_limit* and obtain the following:

student_id	student_name	address	degree	year
8700074	John Smith	9, Davis Hall	B.Sc.	6
8465432	Bill Wilson	22, Second Hall	B.Com	6



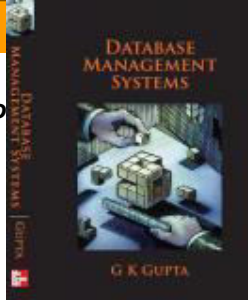
Theta-Join

To illustrate theta-join, we modify the relation *student* to introduce a new relation *scholarship which provides* information on scholarships available.

student(*student*_id, *student*_name, *address*, age)

scholarship(*name*, age_*limit*)

Find the list of students and the scholarships that they are eligible for. We carry out a theta-join of the two relations on the attributes *age* and *age_limit* such that age ≤ *age*_limit.



Union

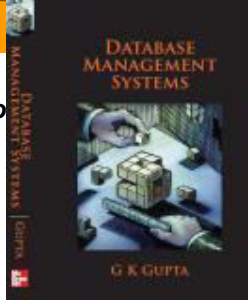
The usual set union of two relations *R* and *S* which are union-compatible.

Two relations are called union-compatible if they are of the same degree *m* and for each i (i=1, 2, …, m), a_i and b_i have compatible domains

$R(a_1, a_2, ..., a_m)$

$S(b_1, b_2, ..., b_m)$

The degree of *R* U *S* is the same as that of *R* and *S* and the cardinality is a+b if a and b are cardinalities of *R* and *S* respectively and there were no duplicate tuples.

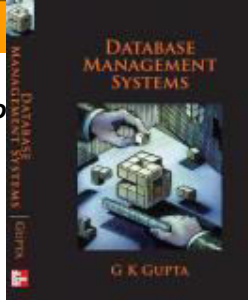


Intersection

Usual set intersection of two relations that are union-compatible.

The intersection provides a relation consisting of all tuples that are common to both relations *R* and *S*.

The degree of R∩S is the same as that of *R* and *S* and the cardinality is equal to the number of tuples common in *R* and *S*.

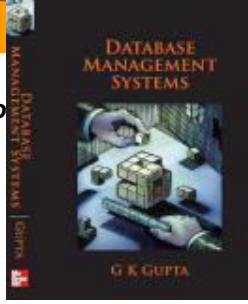


Difference

The difference operator when applied to two relations *R* and *S* (written as *R* - *S*) results in a relation that consists of all tuples in the first relation that are not also in the second relation.

If R∩S is null then *R* - *S* = *R*, otherwise R - *S* = R - *R* ∩ S.

The degree of *R* - *S* is same as that of *R* and *S* and its cardinality is equal to the cardinality of *R* minus the cardinality of R ∩ *S*.

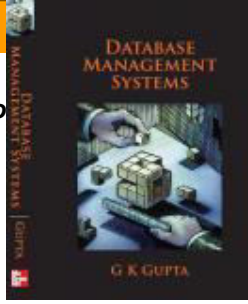


Outer Join

A join gives data from only those tuples from the two tables where the joining attribute(s) contain matching data in both tables.

It is not unusual to require that a query give a listing of both matched and non-matched values, along with other data values from those rows where possible.

This is called an *outer join*.



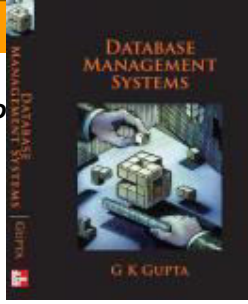
Outer Join

Suppose we wish to produce reports showing for each student the courses that the student is enrolled in.

If we join *student* and *enrolment*, the result will not report on those students that are not enrolled in any course this year.

It may be important to report that Student 18CO01 had no enrolment.

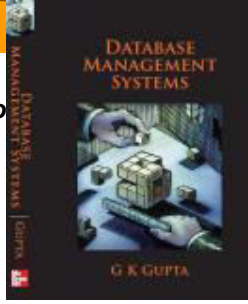
Outer join keeps the unmatched rows when a join is made.



Outer Join

The outer join, where unmatched rows in the leftmost table are preserved, seems to be the most useful of the various forms of outer join.

The need for preserving unmatched rows in the right-hand table, or in both tables, seems far less prevalent.



Examples

(A1)Find the names of all students.

We apply projection on the attribute *student_name* and obtain the list of names of all students.

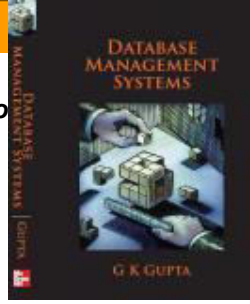
In relational algebra notation, the query may be written as

student[student_name]

(A2)Find the student id's of all students enrolled in CS100.

This query involves a restriction and a projection to be applied to the relation *enrolment*.

enrolment[code = 'CS100'][student_id]



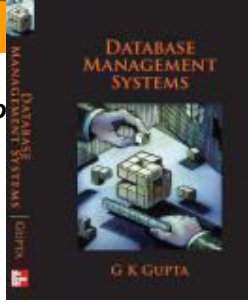
Examples (cont.)

(A3) Find the names of all students enrolled in CS100.

To answer the query, we carry out a natural join of relations *student* and *enrolment* and then apply restriction *code* = 'CS100'.

This is followed by a projection on attribute *student_name*.

(Student ⋈ Enrollment) [code= 'CS100'] [student_name]

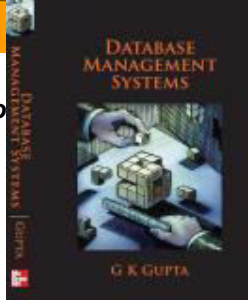


Examples (cont.)

(A4)Find the names of all courses in which John Smith is enrolled in.

One approach to formulating this query is to:

- join *student* and *enrolment*
- apply restriction *student_name* = 'John Smith'
- project on attribute *course_name*.



Examples (cont.)

(A5)Find the student id's of students not enrolled in any course.

The above query may be formulated as follows:

student[student_id] - enrolment[student_id]

(A6)Find the course code's of courses that do have some enrolment.

This only involves a projection on relation *enrolment*.

enrolment[code]