A Commercial Query Language – SQL

Hi! In this lecture I would like to discuss with you the database language**, the Structured Query Language.**

In this section we discuss a query language that is now being used in most commercial relational DBMS. Although a large number of query languages and their variants exist just as there are a large number of programming languages available, the most popular query language is SQL. SQL has now become a de facto standard for relational database query languages. We discuss SQL in some detail.

SQL is a non-procedural language that originated at IBM during the building of the now famous experimental relational DBMS called System R. Originally the user interface to System R was called SEQUEL which was later modified and its name changed to SEQUEL2. These languages have undergone significant changes over time and the current language has been named SQL (Structured Query Language) although it is still often pronounced as if it was spelled SEQUEL. Recently, the American Standards Association has adopted a standard definition of SQL (Date, 1987).

A user of a DBMS using SQL may use the query language interactively or through a host language like C or COBOL. We will discuss only the interactive use of SQL although the SQL standard only defines SQL use through a host language.

We will continue to use the database that we have been using to illustrate the various features of the query language SQL in this section. As a reminder, we again present the relational schemes of the three relations *student*, *subject* and *enrolment*.

students(student_id, student_name, address)
enrolment(student_id, subject_id)
subject(subject_id, subject_name, department)

SQL consists of facilities for data definition as well as for data manipulation. In addition, the language includes some data control features. The data definition facilities include commands for creating a new relation or a view, altering or expanding a relation (entering a new column) and creating an index on a relation, as well as commands for dropping a relation, dropping a view and dropping an index.

The data manipulation commands include commands for retrieving information from one or more relations and updating information in relations including inserting and deleting of tuples.

We first study the data manipulation features of the language.

Data Manipulation - Data Retrieval Features

The basic structure of the SQL data retrieval command is as follows

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

The SELECT clause specifies what attributes we wish to retrieve and is therefore equivalent to specifying a projection. (There can often be confusion between the SELECT clause in SQL and the relational operator called Selection. The selection operator selects a number of tuples from a relation while the SELECT clause in SQL is similar to the projection operator since it specifies the attributes that need to be retrieved). The FROM clause specifies the relations that are needed to answer the query. The WHERE clause specifies the condition to be used in selecting tuples and is therefore like the predicate in the selection operator. The WHERE clause is optional, if it is not specified the whole relation is selected. A number of other optional clauses may follow the WHERE clause. These may be used to group data and to specify group conditions and to order data if necessary. These will be discussed later.

The result of each query is a relation and may therefore be used like any other relation. There is however one major difference between a base relation and a relation that is retrieved from a query. The relation retrieved from a query may have duplicates while the tuples in a base relation are unique.

We present a number of examples to illustrate different forms of the SQL SELECT command.

Simple Queries

We will first discuss some simple queries that involve only one relation in the database. The simplest of such queries includes only a SELECT clause and a FROM clause.

- (Q1) Find the student id's and names of all students.

  SELECT *student_num, student_name*
  FROM *student*

  When the WHERE clause is omitted, the SELECT attributes FROM relations construct is equivalent to specifying a projection on the relation(s) specified in the FROM clause. The above query therefore results in applying the projection operator to the relation *student* such that only attributes *student_num* and *student_name* are selected.

- (Q2) Find the names of subjects offered by the Department of Computer Science.

  SELECT *subject_name*
  FROM *subject*
  WHERE *department* = 'Comp. Science'

  The above query involves algebraic operators selection and projection and therefore the WHERE clause is required.

  One way of looking at the processing of the above query is to think of a pointer going down the table *subject* and at each tuple asking the question posed in the

WHERE clause. If the answer is yes, the tuple being pointed at is selected, otherwise rejected.

- (Q3) Find all students that are enrolled in something.

  SELECT DISTINCT (*student_id*)
  FROM *enrolment*

  The above query is a projection of the relation *enrolment*. Only the *student_id* attribute has been selected and the duplicate values are to be removed.

- (Q4) Find student id's of all students enrolled in CP302.

  SELECT *student_id*
  FROM *enrolment*
  WHERE *student_id* = 'CP302'

  This query involves a selection as well as a projection.

<u>Queries Involving More than One Relations</u>

- (Q5) Find subject id's of all subjects being taken by Fred Smith.

  SELECT *subject_id*
  FROM *enrolment*
  WHERE *student_id* IN
  (SELECT *student_id*
  FROM *student*
  WHERE *student_name* = 'Fred Smith')

  The above query must involve two relations since the subject information is in relation *subject* while the student names are in relation *student*. The above formulation of the query has a sub-query (or a *nested query*) associated with it.

The sub-query feature is very useful and we will use this feature again shortly. Queries may be nested to any level.

It may be appropriate to briefly discuss how such a query is processed. The subquery is processed first resulting in a new relation (in the present case a single attribute relation). This resulting relation is then used in the WHERE clause of the outside query which becomes true if the *student_id* value is in the relation returned by the subquery. The IN part of the WHERE clause tests for set membership. The WHERE clause may also use construct NOT IN instead of IN whenever appropriate.

Another format for the IN predicate is illustrated by the following query.

- (Q6) List subject names offered by the Department of Mathematics or the Department of Computer Science.

SELECT *subject_name*
FROM *subject*
WHERE *department* IN {'Mathematics', 'Comp. Science'}

In the above query, subject name is retrieved when the department name attribute value is either Mathematics or Comp. Science. Any number of values may be included in the list that follows IN.

Constants like 'Mathematics' or 'Comp. Science' are often called a *literal tuple*. A literal tuple of course could have more than one value in it. For example it could be

WHERE {CP302, Database, Comp. Science} IN *subject*.

Obviously the set of attributes in the list before IN must match the list of attributes in the list that follows IN for the set membership test carried out by IN to be legal (the list in the example above being a relation).

The above query is equivalent to the following

SELECT *subject_name*
FROM *subject*
WHERE *department* = 'Mathematics'
or *department* = 'Comp. Science'

SQL allows some queries to be formulated in a number of ways. We will see further instances where a query can be formulated in two or more different ways.

- (Q7) Find the subject names of all the subjects that Fred Smith is enrolled in.

SELECT *subject_name*
FROM *subject*
WHERE *subject_id* IN
(SELECT *subject_id*
FROM *enrolment*
WHERE *student_id* IN
(SELECT *student_id*
FROM *student*
WHERE *student_name* = 'Fred Smith'))

This query of course must involve all the three relations since the subject names are in relation *subject* and the student names are in *student* and students enrollments are available in *enrolment*. The above query results in the last subquery being processed first resulting in a relation with only one tuple that is the student number of Fred Smith. This result is then used to find all enrollments of Fred Smith. The enrollments are then used to find subject names.

The two queries Q5 and Q7 that involve subqueries may be reformulated by using the join operator.

SELECT *subject_name*
FROM *student, enrolment*
WHERE *enrolment.student_id* = *student.student_id*
AND *student_name* = 'Fred Smith'.

SELECT *subject_name*
FROM *enrolment, student, subject*
WHERE *student.student_id* = *enrolment.student_id*
AND *enrolment.subject_id* = *subject.subject_id*
AND *student.name* = 'Fred Smith'.

The above two query formulations imply the use of the join operator. When the FROM clause specifies more than one relation name, the query processor forms a cartesian product of those relations and applies the join predicate specified in the WHERE clause. In practice, the algorithm used is not that simple. Fast algorithms are used for computing the joins.

We should also note that in the queries above, we qualified some of the attributes by specifying the relation name with the attribute name (e.g. *enrolment.student_id*). This qualification is needed only when there is a chance of an ambiguity arising. In queries later, we will also need to give alias to some relation names when more than one instance of the same relation is being used in the query.

- (Q8) Display a sorted list of subject names that are offered by the department of Computer Science.

SELECT *subject_name*
FROM *subject*
WHERE *department* = 'Comp. Science'
ORDER BY *subject_name*

The list returned by the above query would include duplicate names if more than one subject had the same name (this is possible since the subjects are identified by their *subject_id* rather than their name). DISTINCT could be used to remove duplicates if required.

The ORDER BY clause as used above would return results in ascending order (that is the default). Descending order may be specified by using

ORDER BY *subject_name* DESC

The order ascending may be specified explicitly. When the ORDER BY clause is not present, 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. For example, if the ORDER BY clause in a query was ORDER BY *department, subject_name*, then the result will be ordered by *department* and the tuples with the same *department* will be ordered by *subject_name*. The ordering may be ascending or descending and may be different for each of the fields on which the result is being sorted.

- (Q9) Find student id's of students enrolled in CP302 and CP304.

SELECT *student_id*
FROM *enrolment*
WHERE *subject_id* = 'CP302'
AND *student_id* IN
(SELECT *student_id*
FROM *enrolment*
WHERE *subject_id* = 'CP304')

The above query could also be formulated as an intersection of the two queries above. Therefore the clause "AND *student_id* IN" could be replaced by "INTERSECT".

- (Q10) Find student id's of students enrolled in CP302 or CP304 or both.

  SELECT *student_id*
  FROM *enrolment*
  WHERE *subject_id* = 'CP302'
  OR *subject_id* = 'CP304'

  The above query may also be written using the UNION operator as follows.

  SELECT *student_id*
  FROM *enrolment*
  WHERE *subject_id* = 'CP302'

  UNION

  SELECT *student_id*
  FROM *enrolment*
  WHERE *subject_id* = 'CP304'

  It is of course also possible to replace the UNION operator by "OR *student_id* IN" thereby making the second query a sub-query of the first query. It should be noted that the above query would result in presenting duplicates for students that are doing both subjects.

- (Q11) Find student id's of students enrolled in CP302 but not in CP304.

  SELECT *student_id*
  FROM *enrolment*
  WHERE *subject_id* = 'CP302'
  AND *student_id* NOT IN
  (SELECT *student_id*
  FROM *enrolment*
  WHERE *subject_id* = 'CP304')

This query also can be formulated in another way. Rather than using a sub-query, we may take the difference of the results of the two queries. SQL provides an operator "MINUS" to take the difference.

So far we have only used the equality comparison in the WHERE clause. Other relationships are available. These include greater than (>), less than (<), less than equal ($\leq$), greater than equal ($\geq$) and not equal (<>).

The WHERE clause has several different forms. Some of the common forms are:

1. WHERE C1 AND C2
2. WHERE C1 OR C2
3. WHERE NOT C1 AND C2
4. WHERE A operator ANY
5. WHERE A operator ALL
6. WHERE A BETWEEN x AND y
7. WHERE A LIKE x
8. WHERE A IS NULL
9. WHERE EXISTS
10. WHERE NOT EXISTS

Some of these constructs are used in the next few queries. To illustrate these features we will extend the relation *enrolment* to include information about date of each enrolment and marks obtained by the student in the subject. In addition, we will extend the relation *student* to include the *student_id* of the student's tutor (each student is assigned to a tutor who acts as the student's mentor). Each tutor is also a student. Therefore relational schema for *enrolment* and *student* are now:

enrolment(student_id, subject_id, date, mark)
student(student_id, student_name, address, tutor_id)

- (Q12) Find student id's of those students who got a mark better than any of the marks obtained by student 881234 in Mathematics subjects.

SELECT *student_id*
FROM *enrolment*
WHERE *student_id* <> 881234
AND *mark* > ANY
(SELECT *mark*
FROM *enrolment*
WHERE *student_id* = 881234
AND *subject_id* IN
(SELECT *subject_id*
FROM *subject*
WHERE *department* = 'Mathematics'))

Note that if the outermost part of the query did not include the test for *student_id* not being 881234, than we will also retrieve the id 881234.

When ANY is used in the WHERE clause, the condition is true if the value in the outer loop satisfies the condition with at least one value in the set returned by the subquery. When ALL is used, the value in the outer query must satisfy the specified condition with all the tuples that are returned by the subquery.

- (Q13) Find subject id's of those subjects in which student 881234 is enrolled in which he got a mark better than his marks in all the subjects in Mathematics.

SELECT *subject_id*
FROM *enrolment*
WHERE *student_id* = 881234
AND *mark* > ALL
(SELECT *mark*
FROM *enrolment*
WHERE *student_id* = 881234
AND *subject_id* IN
(SELECT *subject_id*

```
FROM subject
WHERE department = 'Mathematics'))
```

- (Q14) Find those subjects in which John Smith got a better mark than all marks obtained by student Mark Jones.

```
SELECT subject_id
FROM enrolment, student
WHERE student.student_id = enrolment.student_id
AND student_name = 'John Smith'
AND mark > ALL
(SELECT mark
FROM enrolment
WHERE student_id IN
(SELECT student_id
FROM student
WHERE student_name = 'Mark Jones'))
```

This query uses a join in the outer query and then uses sub-queries. The two sub-queries may be replaced by one by using a join.

- (Q15) Find student id's of students who have failed CP302 but obtained more than 40%.

```
SELECT student_id
FROM enrolment
WHERE subject_id = 'CP302'
AND mark BETWEEN 41 AND 49
```

Note that we are looking for marks between 41 and 49 and not between 40 and 50. This is because (A BETWEEN x AND y) has been defined to mean $A \geq x$ and $A \leq y$.

- (Q16) Find the student id's of those students that have no mark for CP302

  SELECT *student_id*
  FROM *enrolment*
  WHERE *subject_id* = 'CP302'
  AND *mark* IS NULL

  Note that mark IS NULL is very different than mark being zero. Mark will be NULL only if it has been defined to be so.

- (Q17) Find the names of all subjects whose subject id starts with CP.

  SELECT *subject_name*
  FROM *subject*
  WHERE *subject_id* LIKE 'CP%'

  The expression that follows LIKE must be a character string. The characters underscore (_) and percent (%) have special meaning. Underscore represents any single character while percent represents any sequence of n characters including a sequence of no characters. Escape characters may be used if the string itself contains the characters underscore or percent.

- (Q18) Find the names of students doing CP302.

  SELECT *student_name*
  FROM *student*
  WHERE EXISTS
  (SELECT *
  FROM *enrolment*
  WHERE *student.student_id* = *enrolment.student_id*
  AND *subject_id* = 'CP302');

The WHERE EXISTS clause returns true if the subquery following the EXISTS returns a non-null relation. Similarly a WHERE NOT EXISTS clause returns true only if the subquery following the NOT EXISTS returns a null relation.

Again, the above query may be formulated in other ways. One of these formulations uses the join. Another uses a subquery. Using the join, we obtain the following:

SELECT *student_name*
FROM *enrolment, student*
WHERE *student.student_id* = *enrolment.student_id*
AND *subject_id* = 'CP302');

- (Q19) Find the student id's of students that have passed all the subjects that they were enrolled in.

SELECT *student_id*
FROM *enrolment* e1
WHERE NOT EXISTS
(SELECT *
FROM *enrolment* e2
WHERE *e1.student_id* = *e2.student_id*
AND *mark* < 50))

The formulation of the above query is somewhat complex. One way to understand the above query is to rephrase the query as "find student_id's of all students that have no subject in which they are enrolled but have not passed"!

Note that the condition in the WHERE clause will be true only if the sub-query returns a null result. The sub-query will return a null result only if the student has no enrolment in which his/her mark is less than 50.

We also wish to note that the above query formulation has a weakness that an alert reader would have already identified. The above query would retrieve *student_id* of a student for each of the enrolments that the student has. Therefore if a student has five subjects and he has passed all five, his id will be retrieved five time in the result. A better formulation of the query would require the outer part of the subquery to SELECT *student_id* FROM *student* and then make appropriate changes to the sub-query.

- (Q20) Find the student names of all students that are enrolled in all the subjects offered by the Department of Computer Science.

SELECT *student_name*
FROM *student*
WHERE NOT EXISTS
(SELECT *
FROM *subject*
WHERE *department* = 'Comp. Science'
AND NOT EXISTS
(SELECT *
FROM *enrolment*
WHERE *student.student_id* = *enrolment.student_id*
AND *subject.subject_id* = *enrolment.subject_id*))

It is worthwhile to discuss how the above query is processed. The query consists of three components: the outermost part, the middle sub-query and the innermost (or the last) sub-query. Conceptually, the outermost part of the query starts by looking at each tuple in the relation *student* and for the tuple under consideration evaluates whether the NOT EXISTS clause is true. The NOT EXISTS clause will be true only if nothing is returned by the middle subquery. The middle subquery will return a null relation only if for each of the subject offered by the Department of Computer Science, the NOT EXISTS clause is false. Of course, the NOT EXISTS clause will be false only if the innermost subquery returns a non-null result. That is, there is no tuple in *enrolment* for an enrolment in the subject that the middle sub-query is considering for the student of interest.

We look at the query in another way because we suspect the above explanation may not be sufficient. SQL does not have a universal quantifier (*forall* or ∀) but does have an existential quantifier (*there exists* or ∃). As considered before, the universal quantifier may be expressed in terms of the existential quantifier since $\forall x(P(x))$ is equivalent to $Not(\exists x(Not P(x))$. The query formulation above implements a universal quantifier using the existential quantifier. Effectively, for each student, the query finds out if there exists a subject offered by the Department of Computer Science in which the student is not enrolled. If such a subject exists, that student is not selected for inclusion in the result.

Using Built-in Functions

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

AVG
COUNT
MIN
MAX
SUM

As the names imply, 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.

We consider some queries that use these functions.

- (Q21) Find the number of students enrolled in CP302.

SELECT COUNT(*)
FROM *enrolment*
WHERE *subject_id* = 'CP302'

The above query uses the function COUNT to count the number of tuples that satisfy the condition specified.

- (Q22) Find the average mark in CP302.

SELECT AVG(mark)
FROM *enrolment*
WHERE *subject_id* = 'CP302'

This query uses the function AVG to compute the average of the values of attribute *mark* for all tuples that satisfy the condition specified.

Further queries using the built-in functions are presented after we have considered a mechanism for grouping a number of tuples having the same value of one or more specified attribute(s).

Using the GROUP BY and HAVING Clauses

In many queries, one would like to consider groups of records that have some common characteristics (e.g. employees of the same company) and compare the groups in some way. The comparisons are usually based on using an aggregate function (e.g. max, min, avg). Such queries may be formulated by using GROUP BY and HAVING clauses.

- (Q23) Find the number of students enrolled in each of the subjects

SELECT *subject_id*, COUNT(*)
FROM *enrolment*
GROUP BY *subject_id*

The GROUP clause groups the tuples by the specified attribute (*subject_id*) and then counts the number in each group and displays it.

- (Q24) Find the average enrolment in the subjects in which students are enrolled.

```
SELECT AVG(COUNT(*))
FROM enrolment
GROUP BY subject_id
```

This query uses two built-in functions. The relation *enrolment* is divided into groups with same *subject_id* values, the number of tuples in each group is counted and the average of these numbers is taken.

- (Q25) Find the subject(s) with the highest average mark.

```
SELECT subject_id
FROM enrolment
GROUP BY subject_id
HAVING AVG( mark) =
(SELECT MAX(AVG( mark))
FROM enrolment
GROUP BY subject_id);
```

The above query uses the GROUP BY and HAVING clauses. The HAVING clause is used to apply conditions to groups similar to the way WHERE clause is used for tuples. We may look at the GROUP BY clause as dividing the given relation in several virtual relations and then using the HAVING clause selecting those virtual relations that satisfy the condition specified in the clause. The condition must, of course, use a built-in function because any condition on a group must include an aggregation. If a GROUP BY query does not use an aggregation then GROUP BY clause was not needed in the query.

- (Q26) Find the department with the second largest enrolment.

```
SELECT department
FROM enrolment e1, subject s1
WHERE e1. subject_id = s1. subject_id
GROUP BY department
HAVING COUNT(*) =
(SELECT MAX(COUNT(*))
FROM enrolment e2, subject s2
WHERE e2. subject_id = s2. subject_id
GROUP BY department
HAVING COUNT(*) <>
(SELECT MAX(COUNT(*))
FROM enrolment e3, subject s3
WHERE e3. subject_id = s3. subject_id
GROUP BY department));
```

Finding the second largest or second smallest is more complex since it requires that we find the largest or the smallest and then remove it from consideration so that the largest or smallest of the remaining may be found. That is what is done in the query above.

- (Q27) Find the subject with the most number of students.

```
SELECT subject_id
FROM enrolment
GROUP BY subject_id
HAVING COUNT(*) =
(SELECT MAX(COUNT(*))
FROM enrolment
GROUP BY subject_id);
```

- (Q28) Find the name of the tutor of John Smith

```
SELECT s2.student_name
FROM student s1, student s2
WHERE s1.tutor_id = s2.student_id
AND s1.student_name = 'John Smith'
```

In this query we have joined the relation *student* to itself such that the join attributes are *tutor_id* from the first relation and *student_id* from the second relation. We now retrieve the *student_name* of the tutor.

Update Commands

The update commands include commands for updating as well as for inserting and deleting tuples. We now illustrate the UPDATE, INSERT and DELETE commands.

- (Q29) Delete course CP302 from the relation *subject*.

```
DELETE subject
WHERE subject_id = 'CP302'
```

The format of the DELETE command is very similar to that of the SELECT command. DELETE may therefore be used to either delete one tuple or a group of tuples that satisfy a given condition.

- (Q30) Delete all enrolments of John Smith

```
DELETE enrolment
WHERE student_id =
(SELECT student_id
FROM student
WHERE student_name = 'John Smith')
```

The above deletes those tuples from *enrolment* whose *student_id* is that of John Smith.

- (Q31) Increase CP302 marks by 5%.

  UPDATE *enrolment*
  SET *marks* = *marks* * 1.05 WHERE *subject_id* = 'CP302'

  Again, UPDATE may also be used to either update one tuple or to update a group of tuples that satisfy a given condition. Note that the above query may lead to some marks becoming above 100. If the maximum mark is assumed to be 100, one may wish to update as follows

  UPDATE *enrolment*
  SET *marks* = 100
  WHERE *subject_id* = 'CP302'
  AND *marks* > 95;

  UPDATE *enrolment*
  SET *marks* = *marks* * 1.05 WHERE *subject_id* = 'CP302'
  AND *marks* ≤ 95;

- (Q32) Insert a new student John Smith with student number 99 and subject CP302.

  INSERT INTO *student(student_id, student_name, address, tutor)*: <99, 'John Smith', NULL, NULL>

  INSERT INTO *subject(subject_id, subject_name, department)*: <'CP302', 'Database', NULL>

  Note that the above insertion procedure is somewhat tedius. It can be made a little less tedius if the list of attribute values that are to be inserted are in the same order as specified in the relation definition and all the attribute values are present. We may then write:

  INSERT INTO *student* <99, 'John Smith', NULL, NULL>

Most database systems provide other facilities for loading a database. Also note that often, when one wishes to insert a tuple, one may not have all the values of the attributes. These attributes, unless they are part of the primary key of the relation, may be given NULL values.

Data Definition

We briefly discuss some of the data definition facilities available in the language. To create the relations *student, enrolment* and *subject*, we need to use the following commands:

CREATE TABLE *student*
(*student_id* INTEGER NOT NULL,
student_name CHAR(15),
address CHAR(25))

In the above definition of the relation, we have specified that the attribute *student_id* may not be NULL. This is because *student_id* is the primary key of the relation and it would make no sense to have a tuple in the relation with a NULL primary key.

There are a number of other commands available for dropping a table, for altering a table, for creating an index, dropping an index and for creating and dropping a view. We discuss the concept of view and how to create and drop them.

VIEWS

We have noted earlier that the result of any SQL query is itself a relation. In normal query sessions, a query is executed and the relation is materialized immediately. In other situations it may be convenient to store the query definition as a definition of a relation. Such a relation is often called a *view*. A view is therefore a virtual relation, it has no real existence since it is defined as a query on the existing base relations. The user would see a view like a base table and all SELECT-FROM query commands may query views just like they may query the base relations.

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 that is visible through the 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 relations that exist since a view may be based on one or more base relations. If the structure of the base relations is changed (e.g. a column added or a relations split in two), the view definition may need changing but the users view will not be affected.

As an example of view definition we define CP302 students as

CREATE VIEW DOING_CP302 (*student_id, student_name, Address*)
AS SELECT *student_id, student_name, Address*
FROM *enrolment, student*
WHERE *student.student_id = enrolment.student_id*
AND *subject_id* = 'CP302';

Another example is the following view definition that provides information about all the tutors.

CREATE VIEW *tutors* (*name, id, address*)
AS SELECT (*student_name, student_id, address*)
FROM *student*
WHERE *student_id* IN
(SELECT *tutor_id*
FROM *student*)

Note that we have given attribute names for the view *tutors* that are different than the attribute names in the base relations.

As noted earlier, the views may be used in retrieving information as if they were base relations. When a query uses a view instead of a base relation, the DBMS retrieves the view definition from meta-data and uses it to compose a new query that would use only the base relations. This query is then processed.

Advantages of SQL

There are several advantages of using a very high-level language like SQL. Firstly, the language 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.

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.

Review Question

1. Explain and List down various command in DML.
2. Give the command for creating table.
3. What are views? Give the command for creating view.
4. List down the advantages of SQL.

References

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000
2. Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.
3. http://www.cs.ucf.edu/courses/cop4710/spr2004

Summary

The most popular query language is SQL. SQL has now become a de facto standard for relational database query languages. We discuss SQL in some detail. SQL is a non-procedural language that originated at IBM during the building of the now famous experimental relational DBMS called System R. Originally the user interface to System R was called SEQUEL which was later modified and its name changed to SEQUEL2. These languages have undergone significant changes over time and the current language has been named SQL (Structured Query Language) although it is still often pronounced as if it was spelled SEQUEL. Recently, the American Standards Association has adopted a standard definition of SQL (Date, 1987). A user of a DBMS using SQL may use the query language interactively or through a host language like C or COBOL. We will discuss only the interactive use of SQL although the SQL standard only defines SQL use through a host language.