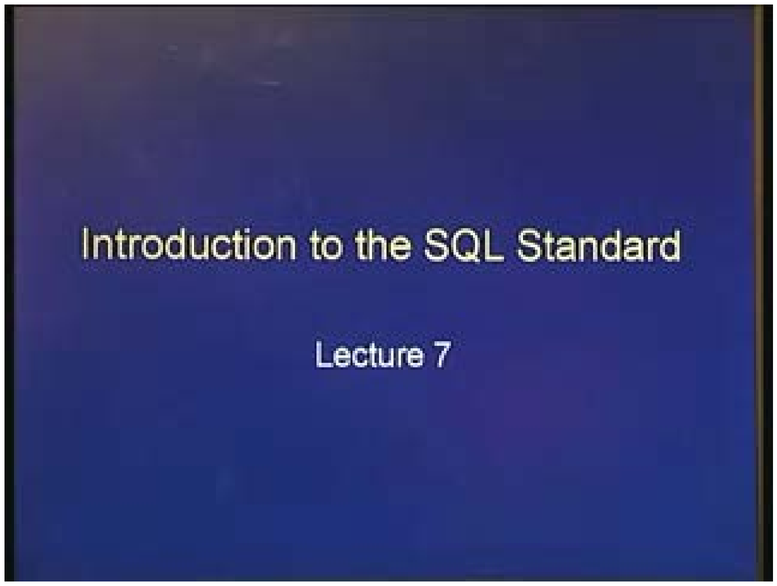
Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 6

Structured Query Language II

Hello and greetings. In our ongoing explorations of databases, we had started out exploring the default or the standard for database querying namely the structured query language or SQL.

(Refer Slide Time: 00:01:36)

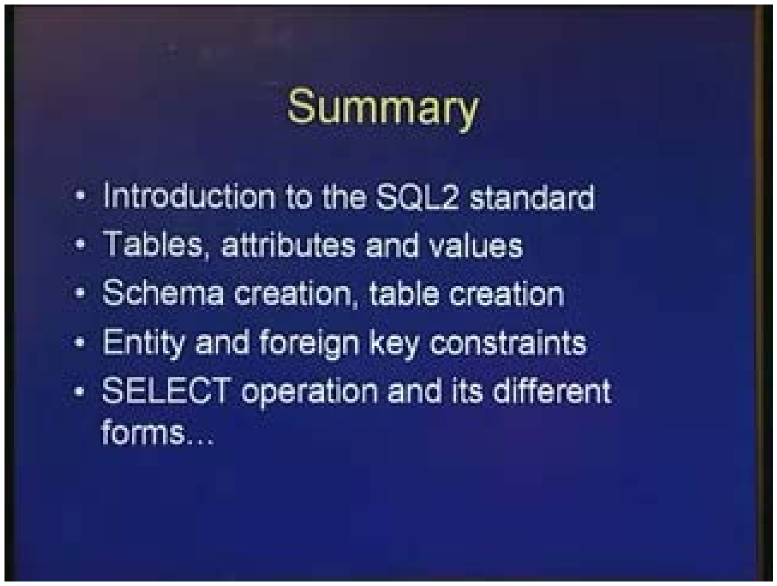


Introduction to the SQL Standard

Lecture 7

So let us continue with SQL in this session today and look at some of the more advanced aspects and what kinds of queries that we can express within simple SQL statements. So before we begin let us summarize what we have studied until now in terms of SQL. Now we looked into an introduction to the SQL standard and what are the building blocks of SQL namely tables, attributes and values which or tables, rows and columns and we also saw how to create schema in SQL and what is meant by a schema and how do we create tables and what kinds of column descriptions can we give when we create tables and what kinds of constraints can we specify when creating tables and so on.

(Refer Slide Time: 00:01:53)



Summary

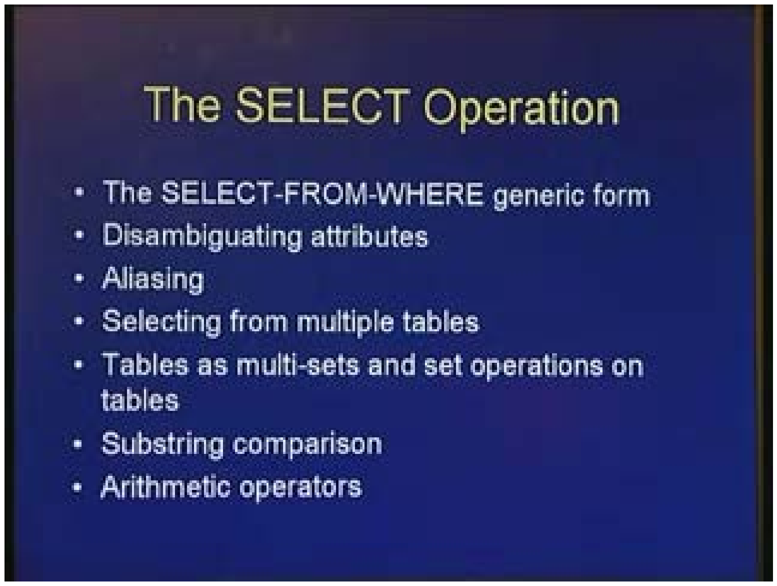
- Introduction to the SQL2 standard
- Tables, attributes and values
- Schema creation, table creation
- Entity and foreign key constraints
- SELECT operation and its different forms…

We also saw how to specify entity constraints specifying what is a primary key and what are the secondary keys in a relation or a table. We also saw how to specify constraints like something should not be null or something should have a default value and we also saw how to specify foreign key constraints so that referential integrity is maintained and we also started by looking into the different forms of the select operation which is perhaps the most frequently used SQL operation for retrieving tuples from the database.

So the most generic form of select operation is the select from where clause where you say select attribute list and say from table list where a given condition holds true. We also saw how we can disambiguate attribute names especially when we would be using multiple tables and two or more tables have attributes with the same name. One simple way of disambiguating attributes names is to prepened the name of the table before the attribute name. So we saw something like instead of saying name, we say employee dot name or department dot name and so on.

So, on the other hand we can also use aliasing for disambiguating names mainly because when table names are also the same, when we saw the example of a join between employee and employee tables.

(Refer Slide Time: 00:03:01)



The SELECT Operation

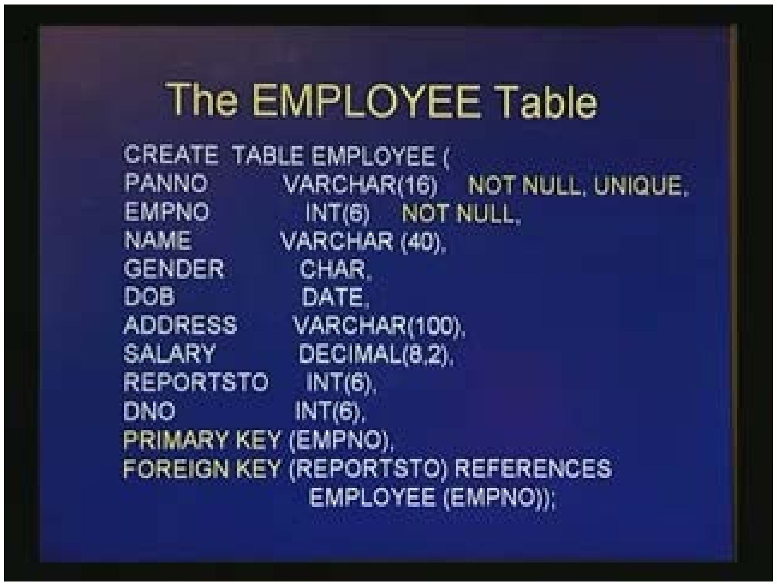
- The SELECT-FROM-WHERE generic form
- Disambiguating attributes
- Aliasing
- Selecting from multiple tables
- Tables as multi-sets and set operations on tables
- Substring comparison
- Arithmetic operators

So we basically use the notion of aliasing where you say use the term or use the table employee and call it as some other name like say e or boss or whatever. Then we also saw how we can select from multiple tables, the Cartesian products of multiple tables and what happens when we omit the where clause and so on. And we also saw the different kinds of set operations on tables. By default in SQL and very unlike relational algebra, in SQL tables are treated as multi-sets or bags that means it can tolerate or it is valid to have different tuples of the same having the same data in a SQL table which is not valid in or which is not valid by default in relational algebra.

So we can make a table into a set by using the keyword called distinct. So when we say select distinct some query, a query is returned with all duplicates removed from the query. We also saw how to perform set operations on tables and sets by default, these operations by default assume that table are sets like union, intersection and except for set difference. If we have to specify that the tables are not sets but instead they are bags, we have to specify, we have to use the qualifier called all. So we say union all whatever, so we say employee union all department and so on or employee union all managers and so on.

And we also saw how to compare substrings using the percentage symbol or the underscore symbol which matches a single character and we also saw some arithmetic operators like addition, subtraction, multiplication and so on which need not just be used within the where clause but they can also be used right after the select clause itself.

(Refer Slide Time: 00:06:21)



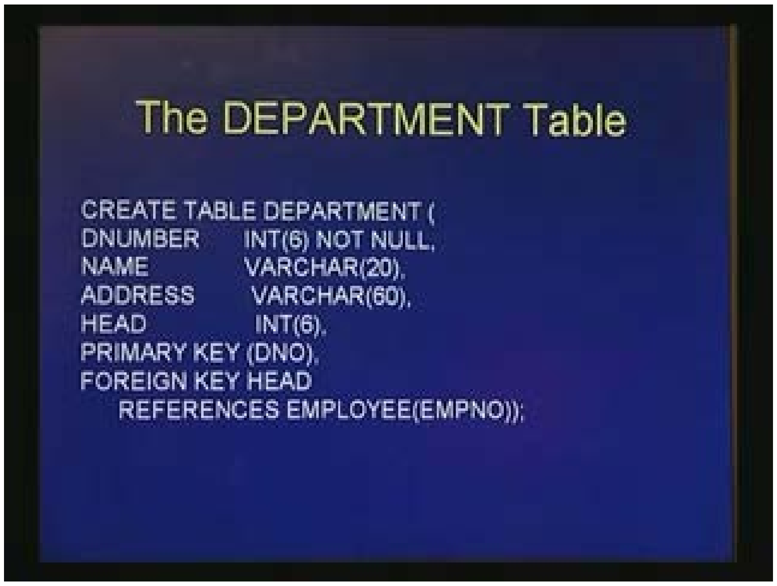
The EMPLOYEE Table

```
CREATE  TABLE EMPLOYEE (
PANNO          VARCHAR(16)     NOT NULL, UNIQUE,
EMPNO            INT(6)     NOT NULL,
NAME           VARCHAR (40),
GENDER           CHAR,
DOB              DATE,
ADDRESS        VARCHAR(100),
SALARY          DECIMAL(8,2),
REPORTSTO        INT(6),
DNO             INT(6),
PRIMARY KEY (EMPNO),
FOREIGN KEY (REPORTSTO) REFERENCES
                  EMPLOYEE (EMPNO));
```

So let us move on further today and in today's talk, I will be using two example tables in order to illustrate several features of SQL. So let us look into these tables once again. The first table that we are going to be considering is the employee table. So the employee table contains employee number which is the second field shown in the slide here as the primary key. It also has a secondary key called pan number of the employee which obviously has to be unique and non null and employee is given a name, gender, date of birth, address, salary and reports to which is a foreign key.

That is reports to contains another employee number which refers to the manager to which this particular employee reports to. If reports to is null then it means that this employee is the head of the company or that is he doesn’t report to any other employee. Then there is a department number or the dnumber which shows where the employee works in. The second table that we are going to be considering today for our examples is the department table.

(Refer Slide Time: 00:07:39)



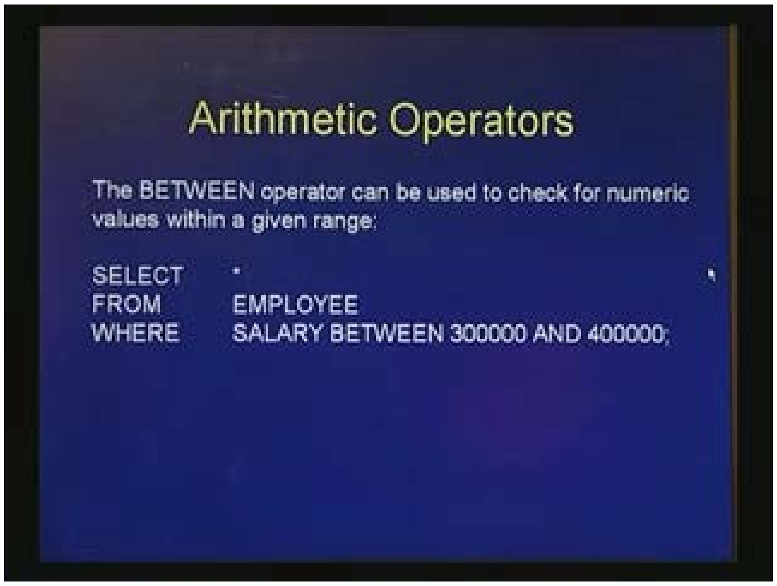
The DEPARTMENT Table

```
CREATE TABLE DEPARTMENT (
DNUMBER      INT(6) NOT NULL,
NAME         VARCHAR(20),
ADDRESS      VARCHAR(60),
HEAD         INT(6),
PRIMARY KEY (DNO),
FOREIGN KEY HEAD
    REFERENCES EMPLOYEE(EMPNO));
```

The department table as shown in the slide here is indexed by or uses as primary key the field called department number or dnumber which is the same as the dnumber used in the employee relation that means the same domain. Department also has the name, address and head which is again a foreign key which points to the employee number of the employee who heads a particular department. So here are the specific declarations which says primary key is dnumber and head is a foreign key which references employee number in the employee table.

So we had started out with looking at arithmetic operator, so let us continue from that point on. And we had seen how you can use plus, minus, star and slashes to perform arithmetic operations namely addition, subtraction, multiplication and division. Similarly you can use the operator called or you can use the keyword called between to check for a range that is to check whether a given parameter lies within a given range.

(Refer Slide Time: 00:08:17)



Arithmetic Operators

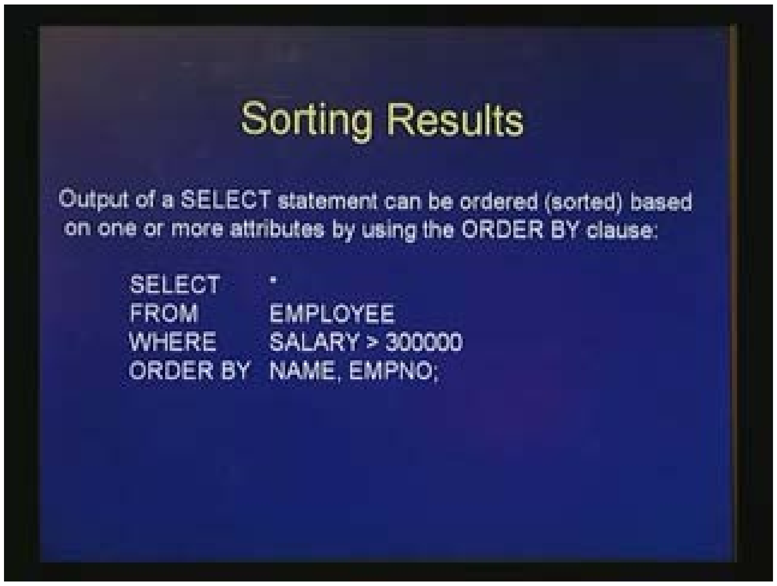
The BETWEEN operator can be used to check for numeric values within a given range:

```
SELECT    *
FROM      EMPLOYEE
WHERE     SALARY BETWEEN 300000 AND 400000;
```

The slide here shows an example which is of the form select star from employee where salary between 3 lakh and 4 lakh. Therefore this query returns all employee records select star in this case note that the star here is not the multiplication operator but star here refers to the entire tuple in this when star occurs by itself. Therefore this query selects the entire set of tuples, the entire set of columns for all tuples where the salary field or the salary attribute is between 3 lakhs and 4 lakhs. It is also possible to sort the output of a given query using one or more parameters.

The sorting is achieved by a new construct called order by, this construct is shown in this slide here. This slide shows another small query which says select star from employee where salary is greater than 3 lakhs and order by name and employee number. Therefore this query returns the same result as the previous query that is the set of all tuples, the complete tuples for all employees well not exactly the same of course for all employees whose salary is greater than 3 lakh but not necessarily less than 4 lakhs and the output here is ordered first by name and next by employee number. So that means that it is first ordered, the records are first ordered by name and wherever there is a tie that is two or more employees having the same name, it then orders those tuples using employee number as the ordering attribute.

(Refer Slide Time: 00:09:41)

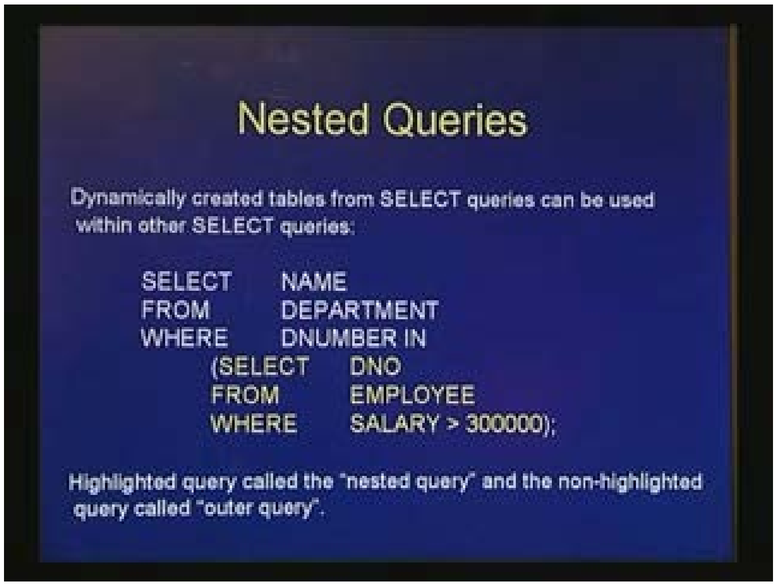


Sorting Results

Output of a SELECT statement can be ordered (sorted) based on one or more attributes by using the ORDER BY clause:

```
SELECT     *
FROM       EMPLOYEE
WHERE      SALARY > 300000
ORDER BY   NAME, EMPNO;
```

(Refer Slide Time: 00:10:39)



Nested Queries

Dynamically created tables from SELECT queries can be used within other SELECT queries:

```
SELECT     NAME
FROM       DEPARTMENT
WHERE      DNUMBER IN
        (SELECT    DNO
         FROM      EMPLOYEE
         WHERE     SALARY > 300000);
```

Highlighted query called the "nested query" and the non-highlighted query called "outer query".

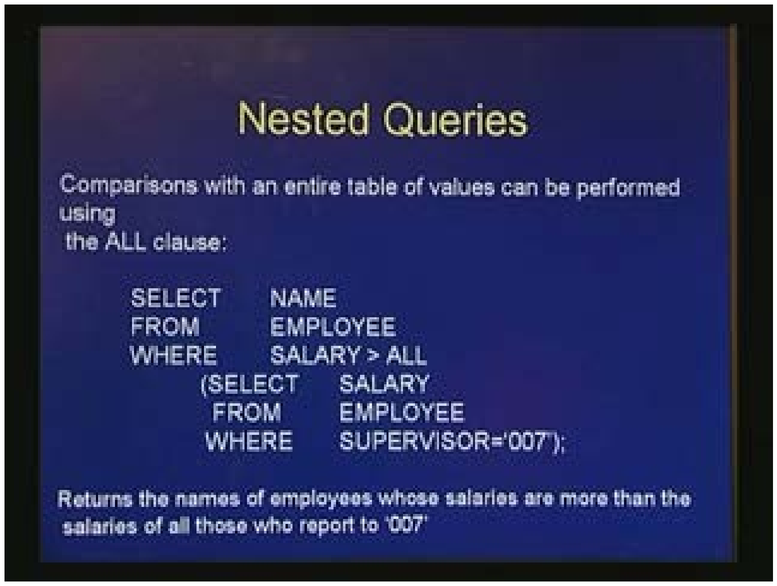
We now come to the next aspect of SQL querying perhaps what gives it compositionality so to say that is by which you can compose queries, bigger queries from small queries this is what is called as the nested query. Note that just like in relational algebra, the output of an SQL statement is a table especially the select statement. The output of a select statement is a table and the input is also a table. It can be multiple tables which are treated as a Cartesian product in which case. Therefore the output of a particular query can be used as part of another query to perform further searches.

So the slide here shows such an example where a query has two different parts in it. For the sake of convenience they are shown in two different colors white and yellow. There is an outer queries so called outer query which says select name from department that is select the names of departments where dnumber in, so in is a new keyword that we are also introducing here which essentially stands for set membership, that is where the dnumber or the department number belongs to the set of all tuples that are returned by the query which is shown in yellow. That is where dnumber in the query called select dnumber from employee where salary is greater than 3 lakhs.

So let us analyze this query. What does this query do? Firstly take a look at the inner query. The inner query says select department numbers of all employees whose salary is greater than 3 lakhs and the outer query is saying select the names of those departments which are returned by the inner query. So essentially this query is asking the database show me all departments which pays greater than 3 lakhs as a salary. So the highlighted query, the yellow part of the query here is called the nested query and the non-highlighted query is called the outer query in this case.

So let us continue with nested queries and note that a nested query contains two different queries and the inner query or the nested query in this case returns a complete relation or returns a complete table. For all practical purposes we can consider the table as sets or bags comprising of different tuples and the in clause that we used in the previous slide performed exactly that, that is did exactly that. That is it considered the table that is returned by the inner query as set and essentially performed a test for set membership.

(Refer Slide Time: 00:13:11)



Nested Queries

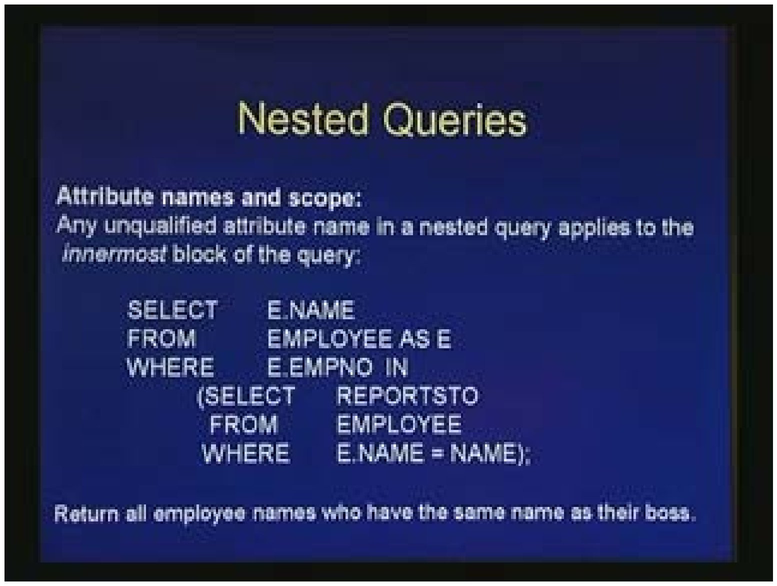
Comparisons with an entire table of values can be performed using
the ALL clause:

```
SELECT    NAME
FROM      EMPLOYEE
WHERE     SALARY > ALL
      (SELECT    SALARY
       FROM      EMPLOYEE
       WHERE     SUPERVISOR='007');
```

Returns the names of employees whose salaries are more than the salaries of all those who report to '007'

That is whether the dnumbers specified in the outer query belongs to the set that is run in the inner query. Similarly one can do a check for a particular attribute against all elements in a set using the keyword called all. This is shown in the slide here. The outer query says select name from employee where salary is greater than and then the inner query begins this is shown in the slide here. So the inner query here says select salary from employee where supervisor equal to 007. So let us try to analyze the query step by step, first take the nested query or the inner query here. It says select salary from employee that is give me the salaries of all employees where it should actually be reports to there is a small bug in this slide where reports to equal to 007 that is give me the salaries of all employees who report to 007. So we just want to know what is this employee number 007 or agent 007 pays all of his subordinates. And then the outer query says select name from employee that is give me the names of all employees where salary greater than all of this inner query. That is whose salary is greater than all of the salary that is paid by 007 that is it is greater than the maximum salary that is paid by 007. So it returns the names of all employees whose salaries are more than the salaries of all those who report to 007.

(Refer Slide Time: 00:15:45)



Nested Queries

Attribute names and scope:
Any unqualified attribute name in a nested query applies to the *innermost* block of the query:

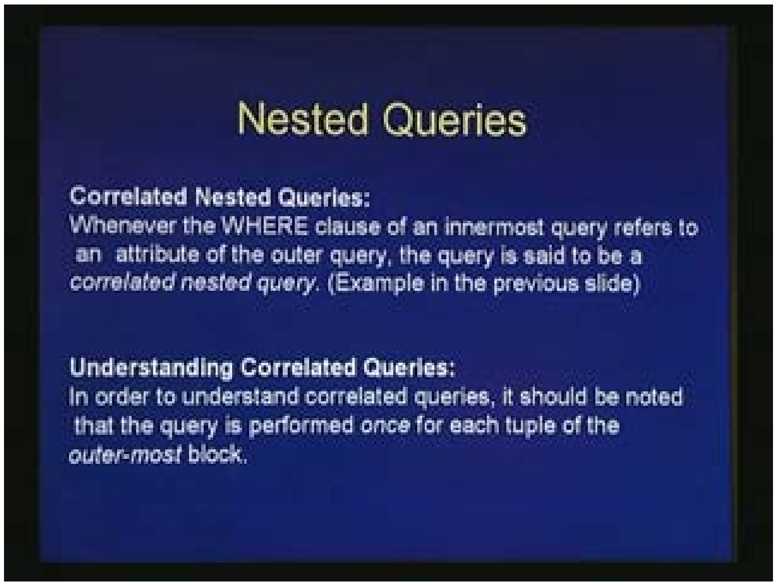
```
SELECT    E.NAME
FROM      EMPLOYEE AS E
WHERE     E.EMPNO  IN
        (SELECT    REPORTSTO
         FROM      EMPLOYEE
         WHERE     E.NAME = NAME);
```

Return all employee names who have the same name as their boss.

Now what happens to disambiguation of attribute names when we are considering nested queries? So this slide here shows such an example. Firstly, the rule which says that any unqualified attribute name in a nested query applies to the inner most block of the query. So what does it mean? Have a look at the query here. There are again two levels to this query, the outer query says select E. name and then I say from employee as E that is I am aliasing employee table as E where E.employee number in and the inner query begins which says selects reports to from employee where E.name. Note that I am using the alias called E inside the inner query as well, while the alias is actually defined in the outer query and then I say E.name equal to name.

So what is this query do actually? So if you notice closely, the outer query as well as the inner query works on the same table called employee. The outer query calls itself as E and the inner query does not change the name of the table, it is still called employee. Therefore in the inner query when I say select reports to that is give me the set of all employee ids of the bosses of all employees where the name of the employee is the same as the name of the boss that is E.name here would be the boss here because I am looking for E.employee number to occur within this set of all employee numbers who are bosses. So the query essentially returns all employee names who have the same name as their boss.

(Refer Slide Time: 00:17:49)



Nested Queries

Correlated Nested Queries:
Whenever the WHERE clause of an innermost query refers to an attribute of the outer query, the query is said to be a *correlated nested query*. (Example in the previous slide)

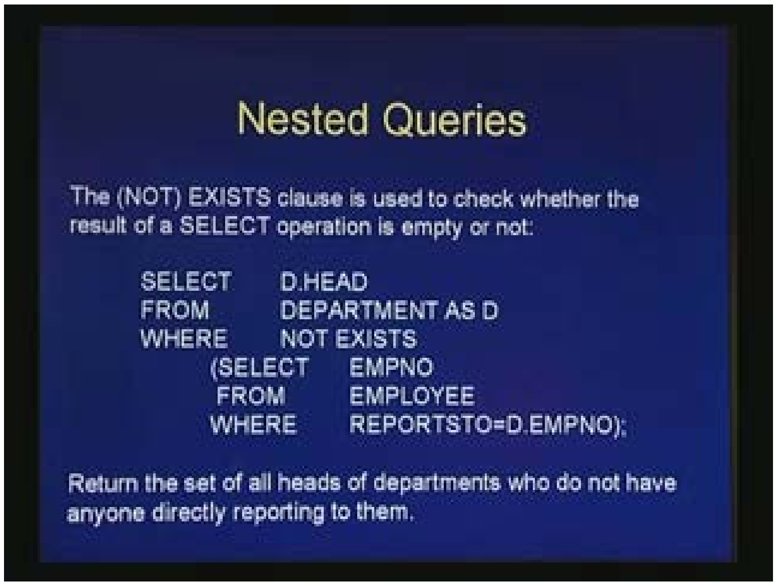
Understanding Correlated Queries:
In order to understand correlated queries, it should be noted that the query is performed *once* for each tuple of the *outer-most* block.

So let us look at few more definitions pertaining to nested queries. The kind of query which we just saw now that is where a particular alias is defined in an outer query and then used inside an inner query, such kinds of queries are called correlated nested queries. That is there is a greater correlation other than the fact that an inner query or a nested query occurs within a condition, there is a greater correlation between these two queries. Now how do we understand or how do we analyze the behavior of a nested query. Now to understand how a nested query begins, it should be noted that or it is sufficient if you just note that every select query is performed or every select condition is performed exactly once on each tuple that is specified in the table that, for the table specified in the query.

Therefore let us have a look at the previous slide once again. In the previous slide for the time being consider just the outer query. Assume that there is just one select statement which is the outer query. So this statement says select some attribute that is E.name from the set of all tuples in employee where some condition that is E.employee number in whatever. So let us not worry about what the condition is let us just assume that there is some condition called C.

Now this condition that is in this case the nested query and the set membership function that is E.employee number in that nested query, this whole condition is checked once for every tuple that forms the employee record that is that forms the employee table. So for every employee, this particular condition is checked that is for every employee we are checking whether his employee number occurs in the set of all employee ids of people who have the same name as one of the subordinates. So in that way as long as we remember this fact that select query or the select condition is checked once for each tuple, its easy to understand nested queries in some kind of recursive fashion that is you have to understand this for each level in a given nested query.

(Refer Slide Time: 00:20:25)



Nested Queries

The (NOT) EXISTS clause is used to check whether the result of a SELECT operation is empty or not:

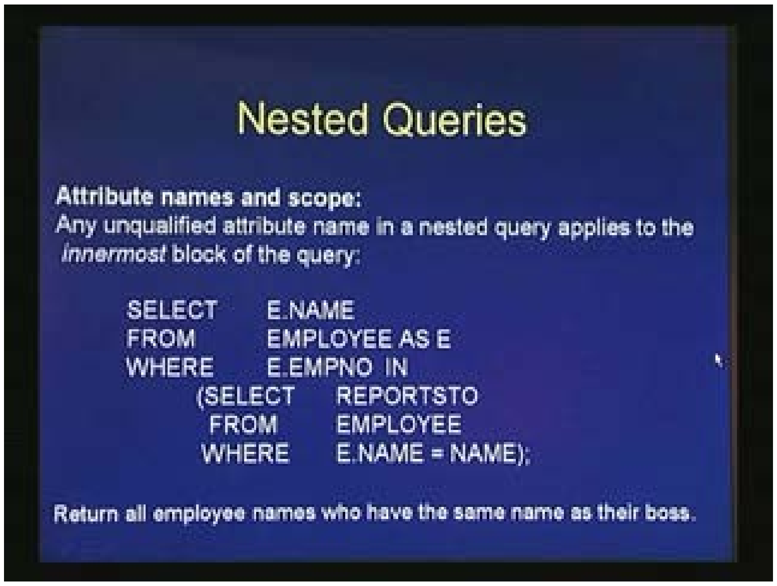
```
SELECT      D.HEAD
FROM        DEPARTMENT AS D
WHERE       NOT EXISTS
        (SELECT     EMPNO
         FROM       EMPLOYEE
        WHERE       REPORTSTO=D.EMPNO);
```

Return the set of all heads of departments who do not have anyone directly reporting to them.

So continuing further with next nested queries we can use a term or the keyword called exists or of course not exists to check whether the output of a given nested query is empty or not. So if the output is not empty then exists returns true that is there exists some results from this query and if the output is empty then exist returns false or not exists returns true. So there is an example given in this slide regarding this which just says select D.head where from department as D of course that is alias department as D where not exists where the following condition does not exist.

What is the following condition which is the inner query, select employee number from employee where reportsto equal to D.employee number. So that means essentially I am looking at the head of all departments that is I am looking at D.head, so the employee numbers of the heads of all departments where this does not exist. What is this? Select employee number from employee where reportsto equal to, this should actually be a, there is again another bug here, this should actually be employee.employee number that is or D.head rather. So where where reportsto equal to this head that means give me the set of all heads of departments who do not have anything, who do not have anyone working under them because this condition should not exist is what we are checking for.

(Refer Slide Time: 00:22:15)



Nested Queries

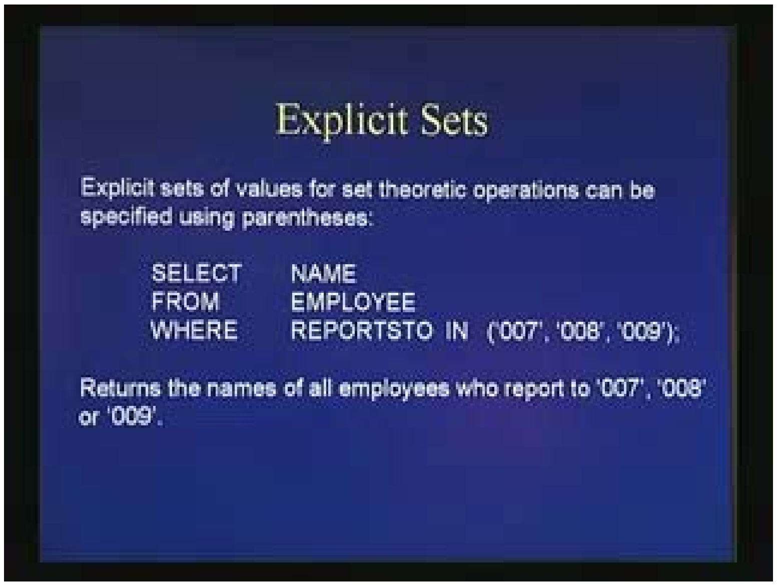
Attribute names and scope:
Any unqualified attribute name in a nested query applies to the *innermost* block of the query:

```
SELECT     E.NAME
FROM       EMPLOYEE AS E
WHERE      E.EMPNO  IN
         (SELECT     REPORTSTO
          FROM       EMPLOYEE
          WHERE      E.NAME = NAME);
```

Return all employee names who have the same name as their boss.

We can also specify explicit sets. Until now we have been looking at implicitly defined sets, we have looked at set membership in the form of in condition, we have looked at set comparison in the form of all condition and we have also checked for empty sets using the exist condition.

(Refer Slide Time: 00:22:33)



Explicit Sets

Explicit sets of values for set theoretic operations can be specified using parentheses:

```
SELECT     NAME
FROM       EMPLOYEE
WHERE      REPORTSTO  IN   ('007', '008', '009');
```

Returns the names of all employees who report to '007', '008' or '009'.

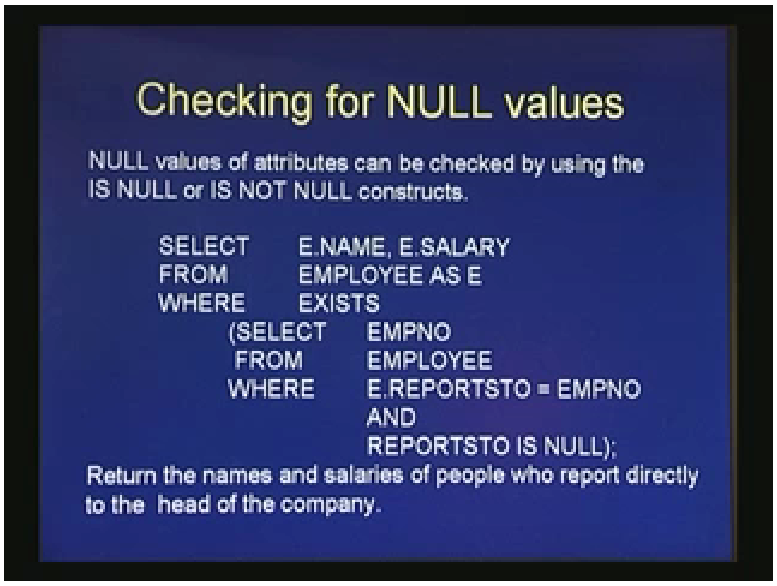
But all of these sets where actually specified in the form of a query that is the nested query.

We can actually specify sets in a more explicit fashion using just parentheses that is we just parenthesize and enumerate all elements of the set explicitly. The slide here shows one such example which says select name from employee where reportsto in 007, 008, 009. Therefore what it says is that give me the names of all employees who report to either 007, 008 or 009 or that is whether the reportsto field or the reportsto attribute holds one of these values in this.

one can check for null values in an SQL table using an SQL command called null or a SQL keyword called null capital n u l l and you can also check for whether something is null or is not null. So remember what constitutes a null value for a particular attribute. a null value is a value which is not applicable or an attribute which has no value or no semantic value associated with for a particular tuple. This is different from saying that the value the attribute is zero or unknown.

So the example here shows a query which says select E.name E.salary from employee has E, so that is select the names and salaries of employees where something exists. What is that exists or that is where the following set is not null. What is the set which should not be null? The set which says select employee number from employee where E.reports to equal to employee number and reportsto is null. So let us analyze the interior query or the inner query again.

(Refer Slide Time: 00:23:18)


Checking for NULL values

NULL values of attributes can be checked by using the IS NULL or IS NOT NULL constructs.

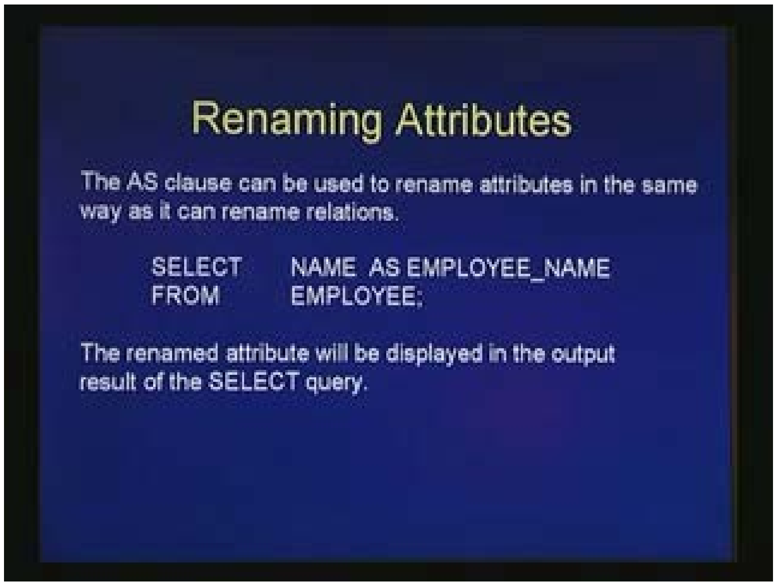
```
SELECT      E.NAME, E.SALARY
FROM        EMPLOYEE AS E
WHERE       EXISTS
        (SELECT     EMPNO
         FROM       EMPLOYEE
        WHERE       E.REPORTSTO = EMPNO
                    AND
                    REPORTSTO IS NULL);
```

Return the names and salaries of people who report directly to the head of the company.

It says select or give me the set of all employee numbers from the employee table where E.reportsto that is the employee from the outer query is a subordinate of me that is the employee in the inner query and reportsto is null that is I don't report to anybody else. So essentially the query returns the names and salaries of all people who report directly to the head of the company.

That is this the name and salary of all people where the following condition exists. The following condition here is the set of all their bosses who also happens to be the head of the company that is who doesn't have any other supervisor himself or other self. So note that here the use of the condition is null that is whether reportsto is null. This is different from saying reportsto is 000 suppose 000 is the valid employee number, this is different from null that it means that this employee has a boss. But in this case this is looking for the fact where this employee does not have a boss or does not have anyone to report to.

(Refer Slide Time: 00:26:04)



Renaming Attributes

The AS clause can be used to rename attributes in the same way as it can rename relations.

```
SELECT      NAME  AS EMPLOYEE_NAME
FROM        EMPLOYEE;
```

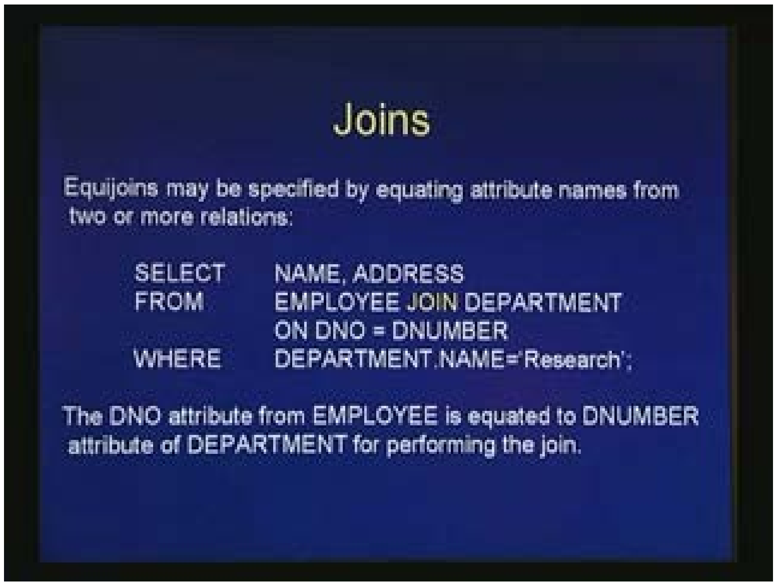
The renamed attribute will be displayed in the output result of the SELECT query.

Just like we have been using aliasing for renaming table names, the as key word or the as clause can be used to rename attribute names as well. So the slide here shows such an example. It says select name as employee underscore name from employee. So what does it do? It simply selects the set of all employee attributes from this table, however while returning it when the table is returned the name of this attribute is changed from name to employee underscore name.

Now this in turn may probably be as part of a larger query were this would matter or even if it is just printed out on the output, the name of the attribute would have changed from name to employee name or employee underscore name. Note that when a select query is run over multiple tables that is the from clause contains multiple tables select name, salary from employee, department. By default it assumes that we are having a Cartesian product between the tables that are specified. Therefore if we have to compute a join between employee and department for example we have to identify, we have to explicitly identify which is the attribute and which has to be compared.

Therefore in a simple select operation, let us say we have to join, we have to compute a join between employee and department we would say something like select star from employee, department where employee.dnumber dno.

(Refer Slide Time: 00:26:58)



Joins

Equijoins may be specified by equating attribute names from two or more relations:

```
SELECT      NAME, ADDRESS
FROM        EMPLOYEE JOIN DEPARTMENT
            ON DNO = DNUMBER
WHERE       DEPARTMENT.NAME='Research';
```

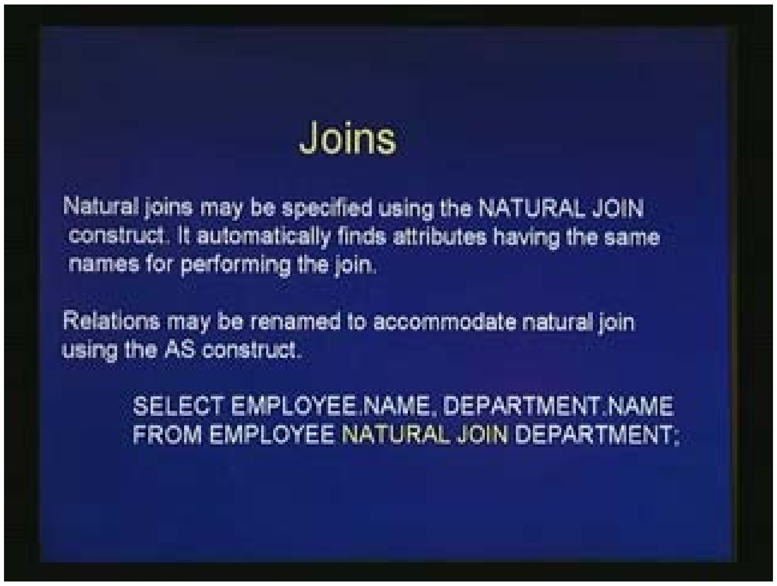
The DNO attribute from EMPLOYEE is equated to DNUMBER attribute of DEPARTMENT for performing the join.

Remember what the employee record look like. The employee record had one of the fields as the dnumber or the department number of the employee where he is working in, so where employee.dnumber equal to department.dnumber. So we have to explicitly equate these two attributes to perform a join. Otherwise it is constitute as a Cartesian join between employee and department.

On the other hand we can specify a join that is rather than a Cartesian product using the keyword called join, the slide here shows such an example. The slide shows a query which says select name, address from employee join department. Now when we just say join without any further qualifiers, there is still not enough information to identify which attributes to use for join condition. What is the theta or what is the join condition here? Therefore we specify that condition explicitly. So we say as shown in the slide here select name, address from employee join department on dno equal to dnumber.

Note that dno is an attribute of employee and dnumber is an attribute of department. So this is specifying an equijoin condition that is it is equating dnumber dno to dnumber and of course there is a where condition which says department.name equal to research. Therefore it is computing the join between employee and department and selecting only those tuples where department name is called research and then printing the set of all names and addresses of employees who work in this department.

(Refer Slide Time: 00:29:53)



Joins

Natural joins may be specified using the NATURAL JOIN construct. It automatically finds attributes having the same names for performing the join.

Relations may be renamed to accommodate natural join using the AS construct.

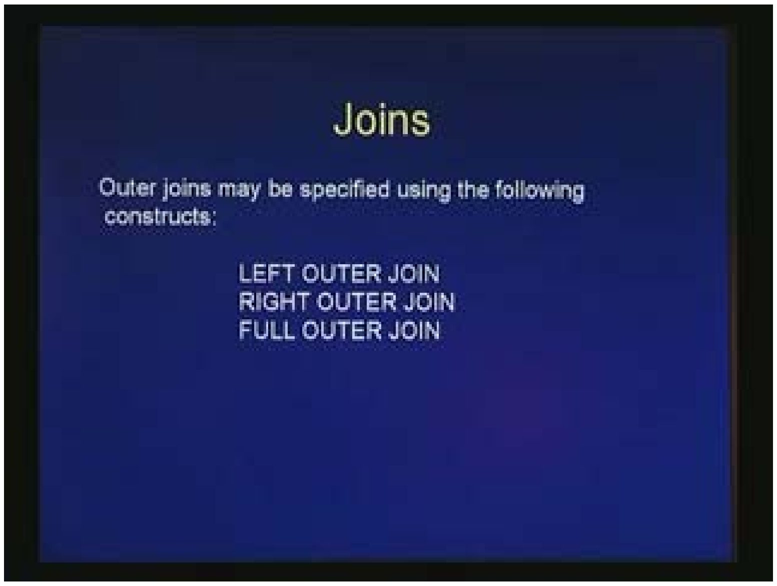
```
SELECT EMPLOYEE.NAME, DEPARTMENT.NAME
FROM EMPLOYEE NATURAL JOIN DEPARTMENT;
```

It is also possible to specify natural join directly without having to specify the equality condition. Note that in a natural join if we have two tables, at least one of the attribute names should be the same or should be common between the two tables. So natural join which in relational algebra was depicted by the star operator just performs in equijoin between two relations where the set of all, where it has some subsets of attributes having the same name. So the same condition also holds true in SQL that is if you are performing a natural join, you have to have at least one attribute name which is the same.

You can always change the name of a table and its attributes names using the as clause which we have already seen. So a natural join may be specified using the natural join construct. It automatically finds attributes having the same names for performing the join and of course relations can be renamed, relations as well as attribute names can be renamed in order to accommodate natural join. So there is a specific example here which says select employee.name, department.name from employee natural join department.

In our specific example though, the only attribute name which was common between these two was the name clause itself. Therefore for our particular example schemas that we have taken, this query may not make sense but the syntax of the query is illustrated. The main idea behind the example is to illustrate the syntax of the query which just says select employee.name, department.name from a natural join between employee and department. That is to identify all attributes which have common names between employee and department, we compute a natural join and then project the set of required set of attributes from them.

(Refer Slide Time: 00:32:04)



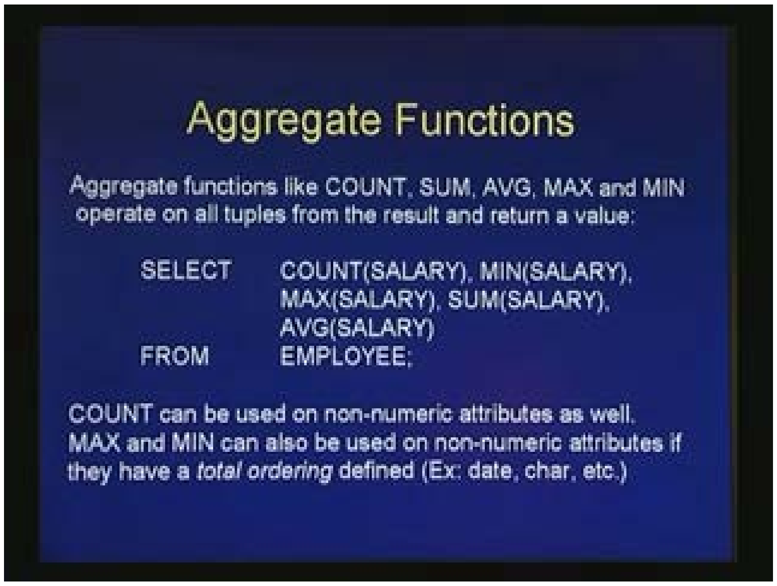
Joins

Outer joins may be specified using the following constructs:

LEFT OUTER JOIN
RIGHT OUTER JOIN
FULL OUTER JOIN

Similarly other kinds of joins in which we saw in the session on relational algebra can also be specified using the appropriate keywords in SQL. Just like you have natural joins, you can also specify left outer join, right outer join and full outer join as part of any SQL statement. We next come to the notion of aggregate functions in SQL. Until now we have been working on generating a set of tuples or the set of all tuples that match a particular criteria.

(Refer Slide Time: 00:32:26)



Aggregate Functions

Aggregate functions like COUNT, SUM, AVG, MAX and MIN operate on all tuples from the result and return a value:

```
SELECT     COUNT(SALARY), MIN(SALARY),
           MAX(SALARY), SUM(SALARY),
           AVG(SALARY)
FROM       EMPLOYEE;
```

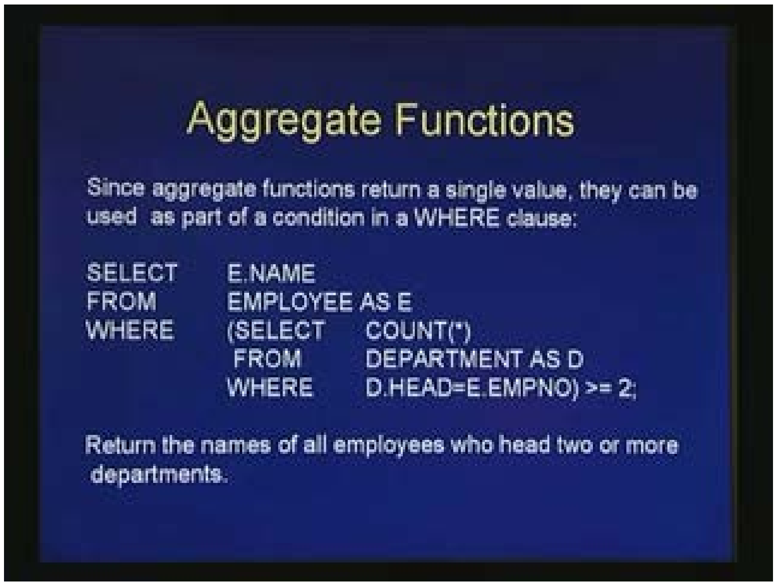
COUNT can be used on non-numeric attributes as well. MAX and MIN can also be used on non-numeric attributes if they have a *total ordering* defined (Ex: date, char, etc.)

Sometimes we may need aggregate properties of a query rather than the set of all query results. So there are number of aggregate function in SQL, some of them are shown here.

There for example count which counts the number of tuples in the query result, sum which computes the sum of the set of all values in the query result in which case the query result should return single numeric attribute. Average which computes the average of all values in the query result. Again the query should return numeric attributes, max and min which computes the maximum and minimum values of all the query results and of course max and min will work if the query results are numeric or they are ordinal which basically means that they have some kinds of total ordering that is specified among them.

For example date, date has a total ordering you can always compare two or more dates. The query can result the maximum and minimum among dates and of course char, you can always compare characters using their ASCII equivalent. So the maximum and minimum can be returned. Here is a small example query shown in the slide which just says select count of salary min of salary max of salary sum of salary and average of salary from employee which is obvious. It just counts the number of salary elements, as you can see I could have as well set count star, it does not make any difference because it just counts the number of tuples in the relation and min of salary, min of the salary field max of the salary, salary attributes, sum and average of all the salaries.

(Refer Slide Time: 00:34:40)



Aggregate Functions

Since aggregate functions return a single value, they can be used as part of a condition in a WHERE clause:

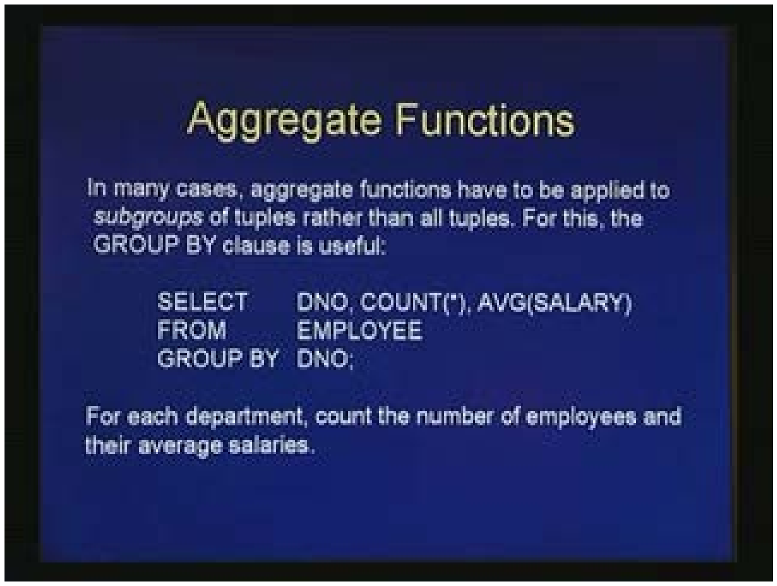
```
SELECT    E.NAME
FROM      EMPLOYEE AS E
WHERE     (SELECT    COUNT(*)
           FROM      DEPARTMENT AS D
          WHERE      D.HEAD=E.EMPNO) >= 2;
```

Return the names of all employees who head two or more departments.

Since aggregate functions return a single value and not a table, they can actually be used as a part of a logical operator within a where clause. Until now when we have been talking about nested queries, we have been treating the nested query as the set and we have only been applying set theoretic operators like in and all and exist and not exist and so on. But once we use aggregation function and reduce our query result to a single value, I might as well use it as part of a logical operator.

The example in this slide shows such a case which says the outer query says select E. name from employee as E where the inner query says select count star that is select the count of all, the set of all tuples that are returned by this query and what is the query? From department as D where department. head equal to E.employee number. That is it is selecting the set of all employee numbers who heads departments and counting the set of all such queries that is it is count, it is for each employee that is search in the outer query it is first searching the set of all departments that the employee heads and return in their count. And this count is compared with two here that is greater than or equal to two. Therefore the semantics of this query is return the names, note that we are finally retuning E.name. Return the names of all employees who head two or more departments.

(Refer Slide Time: 00:36:32)



Aggregate Functions

In many cases, aggregate functions have to be applied to *subgroups* of tuples rather than all tuples. For this, the GROUP BY clause is useful:

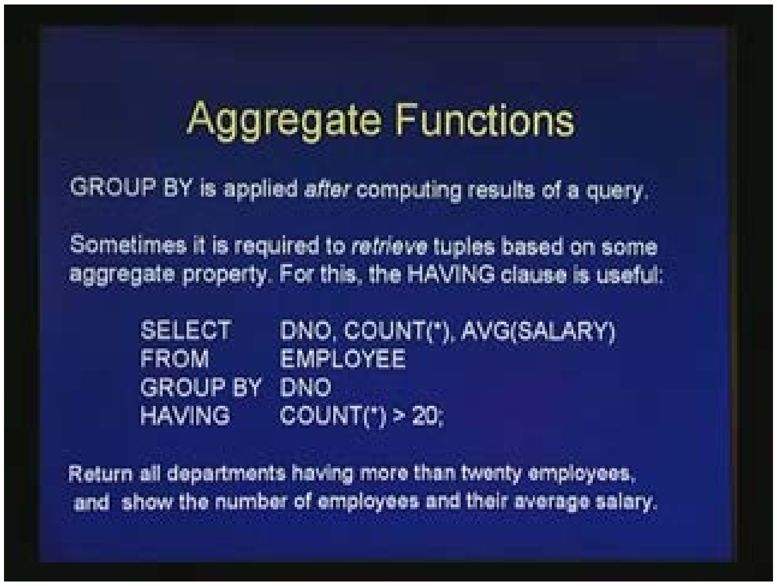
```
SELECT     DNO, COUNT(*), AVG(SALARY)
FROM       EMPLOYEE
GROUP BY   DNO;
```

For each department, count the number of employees and their average salaries.

In some cases when we are talking about aggregate functions, it is not really desirable to apply the aggregate function to the set of all query results, set of all tuples that have been returned by the query. It may be more desirable to apply this, the query or the aggregation functions two different sub groups of the query results. Now you can specify such sub groups using what is called as the group by clause. The group by clause is shown in this query here in this slide.

The slide shows a small example, select dnumber dno, count star, average salary from employee group by dnumber. What it does is for each department that is it first checks the or it first scans the set of all tuples in the employee table and then groups the set of tuples according to dnumber or dno. So for each department that are contained in the employee table count the number of people working in it, that is because we are counting the set of all employee records which have this dno. So count the number of people working in this department and also count the average salary of the people who are working in this department.

(Refer Slide Time: 00:38:04)



Aggregate Functions

GROUP BY is applied *after* computing results of a query.

Sometimes it is required to *retrieve* tuples based on some aggregate property. For this, the HAVING clause is useful:

```
SELECT      DNO, COUNT(*), AVG(SALARY)
FROM        EMPLOYEE
GROUP BY    DNO
HAVING      COUNT(*) > 20;
```

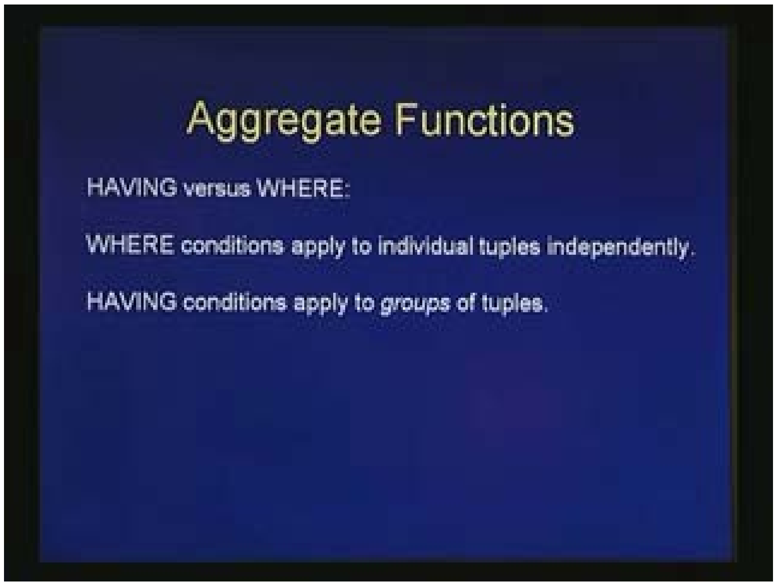
Return all departments having more than twenty employees, and show the number of employees and their average salary.

But note that the group by function is computed after computing the results of a query. That is the query first scans the set of all employee records as in the previous example and then groups those records or groups those tuples based on dno and then performs the select operation. However if you want to select certain tuples based on some aggregate property, not an individual property note the important difference here.

Suppose they want to select a set of tuples from a table based on some aggregate property, we can use different keyword called having clause. The slide here shows such an example, it says select dno that is select the department number count and average salary from employee group by dnumber that is it's the same query as earlier that is for each department return me the count and number of people working in it and average of salary having count greater than 20. That is there is a further constraint here. For all departments having more than 20 people working, show me the department number, the number of people that are working there and their average salaries.

Note the difference between having and where clause. You might you might be wondering we could have just used the where clause, select from where, where actually specifies a particular condition and having is also specifying a particular condition. Now what is the difference between having and where?

(Refer Slide Time: 00:39:49)



Aggregate Functions

HAVING versus WHERE:

WHERE conditions apply to individual tuples independently.

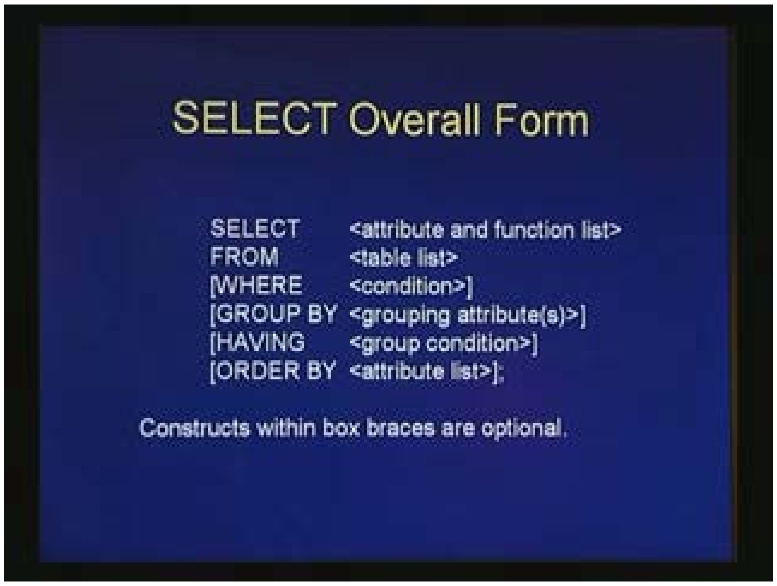
HAVING conditions apply to *groups* of tuples.

Having versus where: Where conditions apply to individual tuples independently that is the where condition where I say something like salary greater than 3 lakhs is applied to each tuple independently. However the having condition applies to groups of tuples, its an aggregate property that is having count greater than 20 that is the number of tuples is greater than 20 or the average of salary is greater than something else and so on. So having is specifies a condition that applies to a group of tuples, whereas where specifies conditions that apply only to individual tuples.

So let us look back at select now. How does select look like after going through all this different variants of select? Now once I look back looking at all this different variants, we see that the select condition has the following constructs. Select attribute and or function list I can also give a function remember, I can always say select 10 times 10 star salary. That is what happens if my salary increases by 10 times and so on.

So select attribute list or function list from table list and the rest are all optional, where condition is an option. if you don't specify where then every tuple is checked. Group by grouping attributes. How should the tuples or how should the output be grouped? Having group condition that is some kind of aggregate properties that we can check. Then finally order by which is the sorting condition for the output.

(Refer Slide Time: 00:40:30)

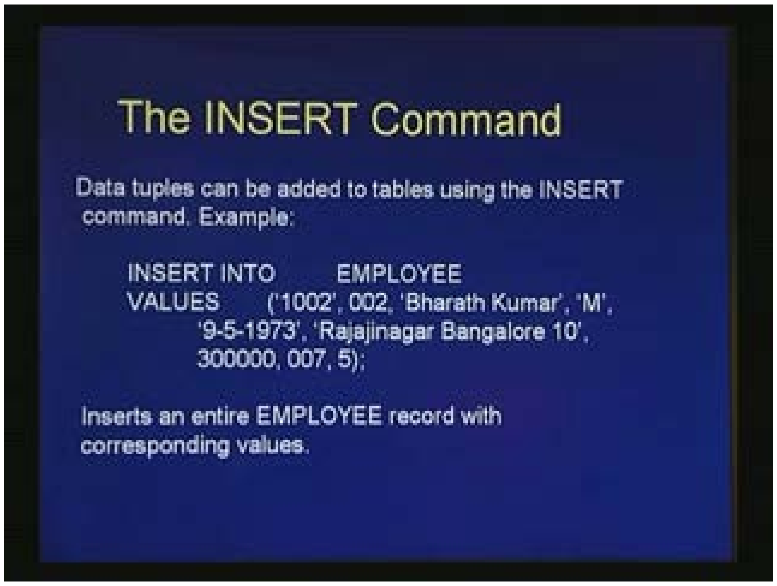


SELECT Overall Form

```
SELECT       <attribute and function list>
FROM         <table list>
[WHERE       <condition>]
[GROUP BY    <grouping attribute(s)>]
[HAVING      <group condition>]
[ORDER BY    <attribute list>];
```

Constructs within box braces are optional.

(Refer Slide Time: 00:41:44)



The INSERT Command

Data tuples can be added to tables using the INSERT command. Example:

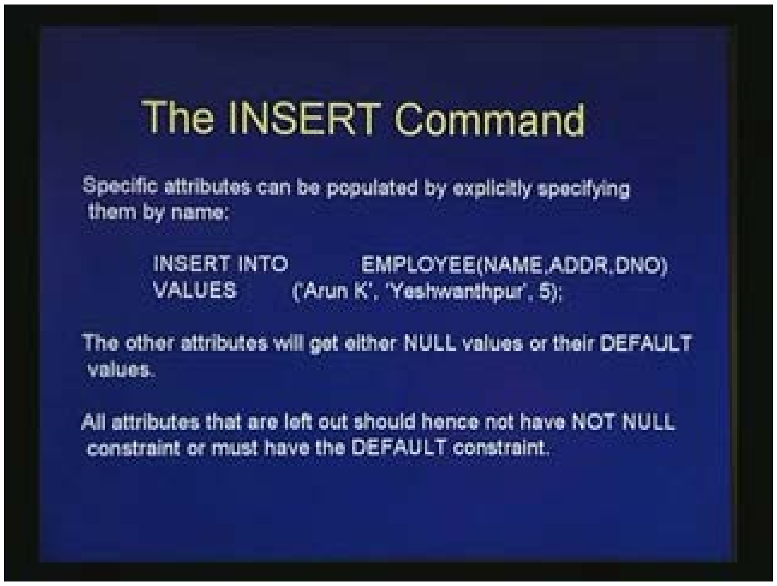
```
INSERT INTO      EMPLOYEE
VALUES      ('1002', 002, 'Bharath Kumar', 'M',
        '9-5-1973', 'Rajajinagar Bangalore 10',
        300000, 007, 5);
```

Inserts an entire EMPLOYEE record with corresponding values.

We now move into other operations within SQL, select obviously is the most widely used operation and hence the most detailed in terms of some its syntax but we still need operations for inserting or adding data into tables and modifying tables, deleting data from tables and so on. By modifying tables here I mean modifying data in tables not modifying the structure of tables which can be done using the alter table command. So insertion of data into tables can be performed using the insert command. The insert command is shown in the slide. Just like we have select from, we have insert into.

So we say insert into employee values and I give the entire record for the employee within parenthesis. So I am inserting a complete record where the first 1002 stands for the pan number, the second 002 stands for employee id, the third field Bharath Kumar stands for the name then M stands for the gender, 9-5-1973 stands for date of birth and so on. So we can specify the entire tuple in line into the query itself and corresponding fields and the corresponding attributes are matched. So inserts an entire employee record with corresponding values.

(Refer Slide Time: 00:43:16)



The INSERT Command

Specific attributes can be populated by explicitly specifying them by name:

```
INSERT INTO        EMPLOYEE(NAME,ADDR,DNO)
VALUES        ('Arun K', 'Yeshwanthpur', 5);
```

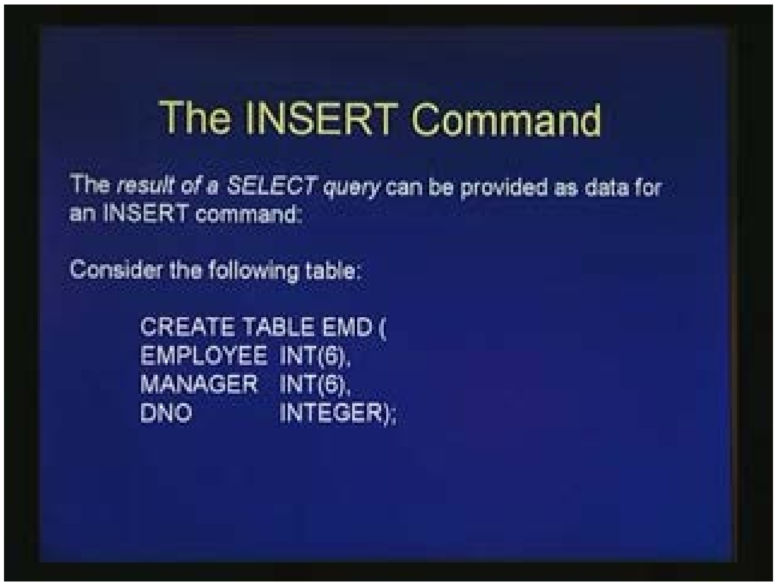
The other attributes will get either NULL values or their DEFAULT values.

All attributes that are left out should hence not have NOT NULL constraint or must have the DEFAULT constraint.

On the other hand I can insert or I can specify only partial set of attributes within the within the tuple. So this slide shows such an example which says select into employee or rather insert into employee name, address and dnumber values Arun K Ysehwanthpur and 5. Therefore it returns or it inserts only those fields or only those attributes called name, address and department number or dno. What happens to the other attributes? The other attributes will either get a null value or the default value if a default is specified.

Therefore you can see that there is an implicit constraint in this select operation. That is I cannot leave out, when I am inserting a tuple I cannot leave out any attribute name which contains a not null constraint and does not have a default value. If it does not have a default value and it is not allowed to be null then I have to specify a value during insertion or else insertion is going to fail.

(Refer Slide Time: 00:44:26)



The INSERT Command

The *result of a SELECT query* can be provided as data for an INSERT command:

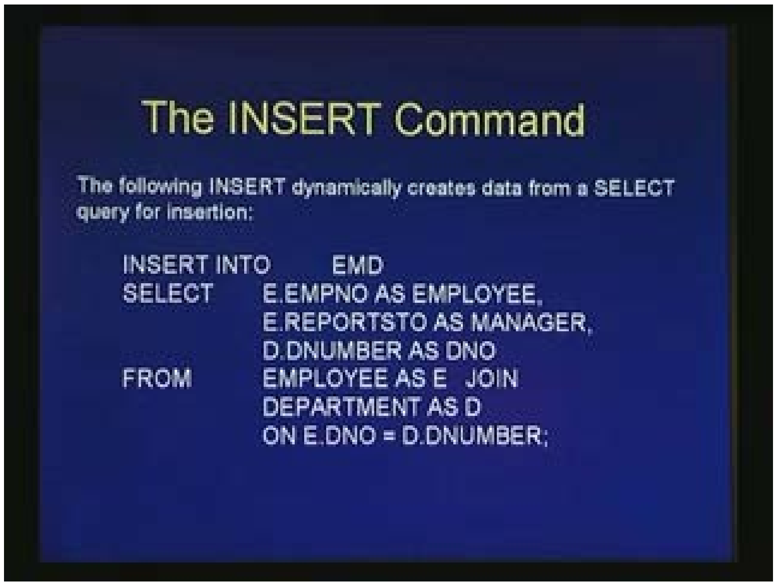
Consider the following table:

```
CREATE TABLE EMD (
EMPLOYEE  INT(6),
MANAGER   INT(6),
DNO       INTEGER);
```

An insert or insertion of more than one tuples to a table can also be performed using a select operation. Note that select actually returns a table. Now if I return and table is nothing but a set of tuples. Now if a select operation returns a set of tuples and these tuples are in a format that a ready to be inserted into another table I can directly specify the select command within the insert command. This is shown in this example here. First I just create a table called EMD which contains just three attributes employee, manager and dno that is the department number. Then I give an insert command from the employee and department table using the following syntax.

I just say insert into EMD that is insert into the new table name and what should I insert? The output of the following select operation then I just give the select command that is select E.employee number as a employee, E.reports to as manager and D. dnumber as dno and then from employee as E join department and so on. So basically I am getting a set of employee number, manager number and department number which is what is going into the new table.

(Refer Slide Time: 00:45:08)

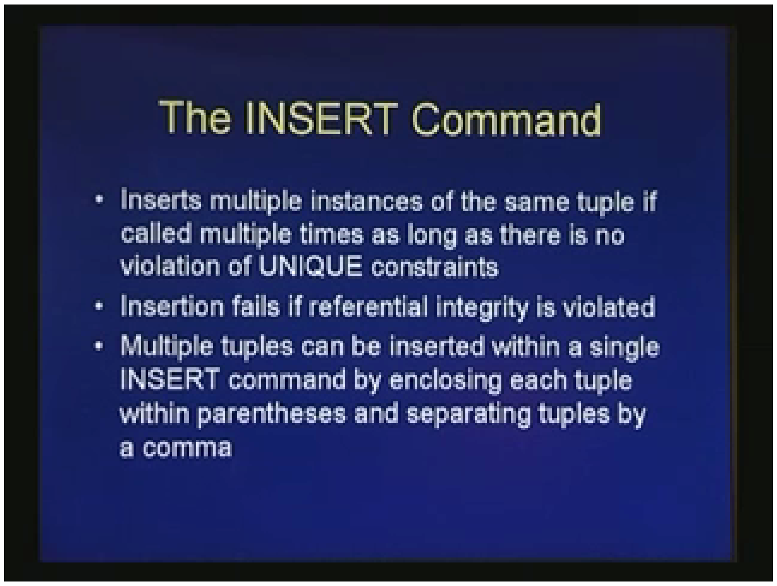


The INSERT Command

The following INSERT dynamically creates data from a SELECT query for insertion:

```
INSERT INTO       EMD
SELECT      E.EMPNO AS EMPLOYEE,
            E.REPORTSTO AS MANAGER,
            D.DNUMBER AS DNO
FROM        EMPLOYEE AS E   JOIN
            DEPARTMENT AS D
            ON E.DNO = D.DNUMBER;
```

(Refer Slide Time: 00:45:49)



The INSERT Command

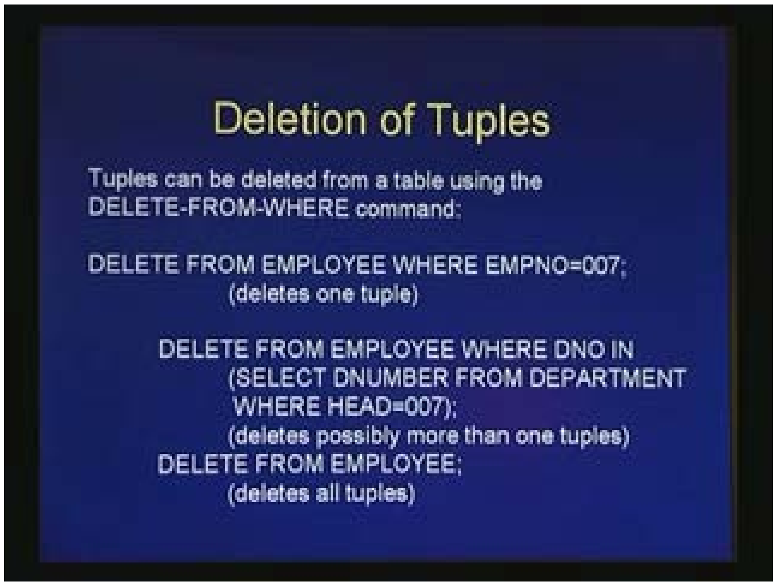
- Inserts multiple instances of the same tuple if called multiple times as long as there is no violation of UNIQUE constraints
- Insertion fails if referential integrity is violated
- Multiple tuples can be inserted within a single INSERT command by enclosing each tuple within parentheses and separating tuples by a comma

So what are the properties of the insert command? Let us summarize the insert command once again. So insert tuples that are specified as part of the command and all attributes which are not null and do not have a default value have to be specified as part of the insert statement. And what happens if I give the same insert statement twice? That is the same insert statement with the same set of data elements and it is actually performed twice. It doesn’t, insert does not do any checks that is remember that tables are treated as multi-sets rather than sets. Therefore insert just goes and inserts the tuple again. That is second occurrence of the same tuple. This is done as long as the second insertion does not violate any unique constructs.

That is if I give a set of data values in the first tuple and give the same set of data values in the second and one of the attributes has to be unique then the condition, then the constraint fails and in turn insert also fails. So as long as the unique construct is not violated, insert will just insert multiple tuples into the table. Insert also fails if referential integrity is violated, this if your DBMS supports it of course that is if I try to insert an employee number as manager where which refers to a manager entity which does not exist, referential integrity fails.

Multiple tuples can be inserted within a single insert command, first of course by using the select statement or by just giving multiple tuples one after the other separated by commas and each tuple is enclosed within parenthesis.

(Refer Slide Time: 00:47:42)



Deletion of Tuples

Tuples can be deleted from a table using the DELETE-FROM-WHERE command:

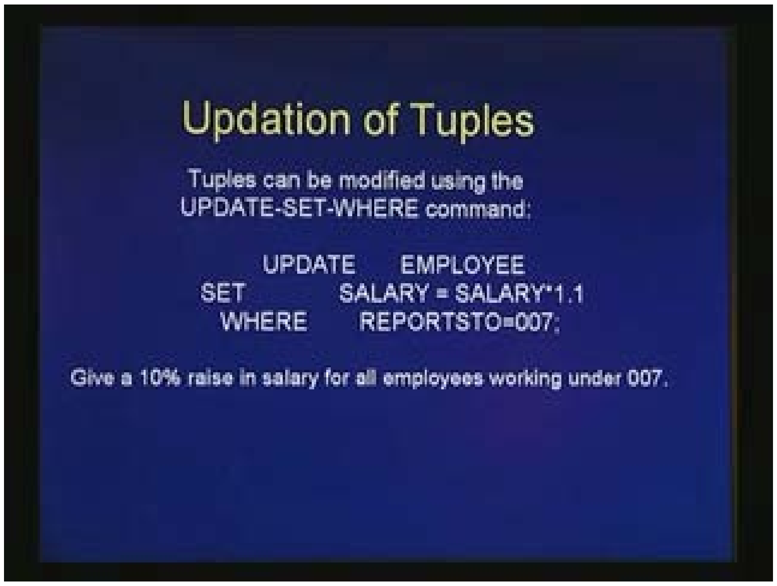
```
DELETE FROM EMPLOYEE WHERE EMPNO=007;
            (deletes one tuple)

      DELETE FROM EMPLOYEE WHERE DNO IN
            (SELECT DNUMBER FROM DEPARTMENT
             WHERE HEAD=007);
            (deletes possibly more than one tuples)
      DELETE FROM EMPLOYEE;
            (deletes all tuples)
```

Deletion of tuples: How do we delete tuples from a table? Deletion of tuples is very similar to the select statement and has the same structure which is of the form delete from where. It says delete from employee where some condition that is employee number equal to 007, the first one which just deletes one tuple. Note that employee number is the primary key therefore it is unique and therefore it just deletes one tuple.

On the other hand the second tuple says delete from employee where department number in the set of all departments that are headed by 007, so it can delete possibly more than one tuple in this statement. On the other hand if I remove the condition and I just say delete from employee, it deletes all tuples. Note that deleting all tuples is different from dropping the table. Here deleting all tuples corresponds to truncating the table that is the table exists but it has no tuples in it. Whereas, if I drop the table the table itself does not exist in the database.

(Refer Slide Time: 00:48:54)



Updation of Tuples

Tuples can be modified using the UPDATE-SET-WHERE command:

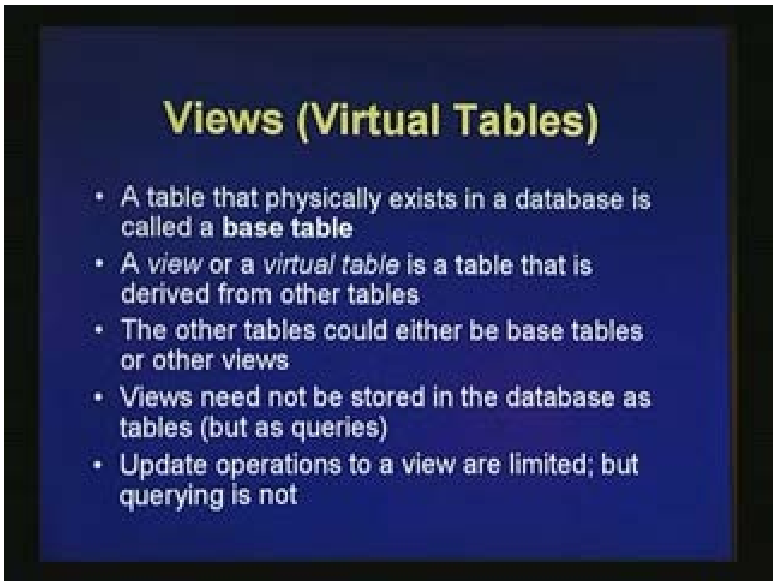
```
        UPDATE      EMPLOYEE
    SET           SALARY = SALARY*1.1
      WHERE       REPORTSTO=007;
```

Give a 10% raise in salary for all employees working under 007.

Updation of tuples: How do we modify tuples? You can update tuples by using the update operation and the update operation also has a very simple syntax which is of the form update set where that is update employee as shown in this example here. So update employee set whatever updation I need to make that is set salary equal to salary times 1.1 where reportsto equal to 007.

Therefore what I am doing here is that for all people working under 007, I am updating their salary by or I am increasing their salary by 10 %, I am giving them a 10 % rise by this update statement here.

(Refer Slide Time: 00:49:37)



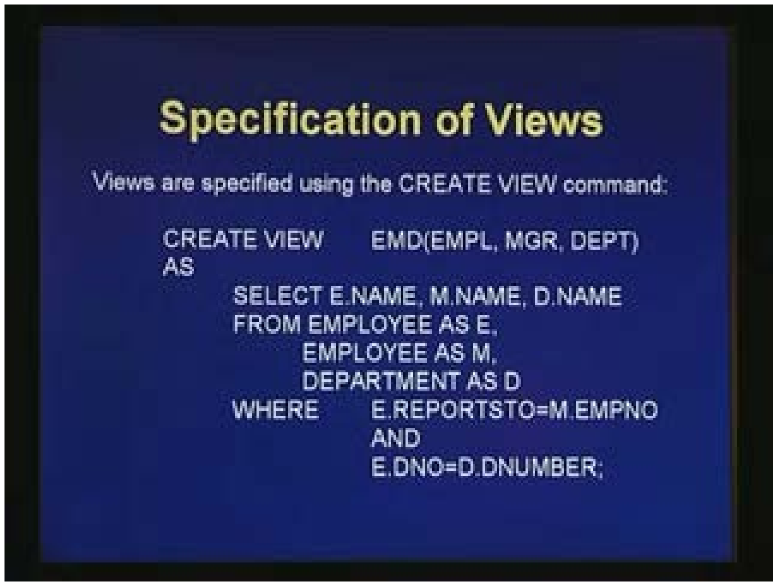
Views (Virtual Tables)

- A table that physically exists in a database is called a **base table**
- A *view* or a *virtual table* is a table that is derived from other tables
- The other tables could either be base tables or other views
- Views need not be stored in the database as tables (but as queries)
- Update operations to a view are limited; but querying is not

We now come to the last leg of this session where we take up the notion of views or virtual tables. This is again an important concept when designing large databases. View as you can typically or intuitively understand is one particular view of the database that is one particular projection of the database which is suitable for certain kinds of work. That is let us say a HR manager needs to know only the employee that is the HR related details of an employee. He doesn't need to know the technical details of an employee or the project related details of an employee.

Therefore we have basically created a view of the employee record for the HR manager which is different from the view created for let us say the project manager. So table or a view is also called a virtual table, this is a table that is derived from other tables. In contrast a table that exists in the database is called a base table. So a view can be derived from either other base tables or other views. Views are need not be stored, it's not that they are not stored but views need not be stored in the database. That is the data contained in the views need not be stored in the database. But they are typically stored as queries and not as tables and update operations for views are limited, this is as the result of storing them as queries but querying is not, you can query but updation is limited in terms of views.

(Refer Slide Time: 00:51:24)



Specification of Views

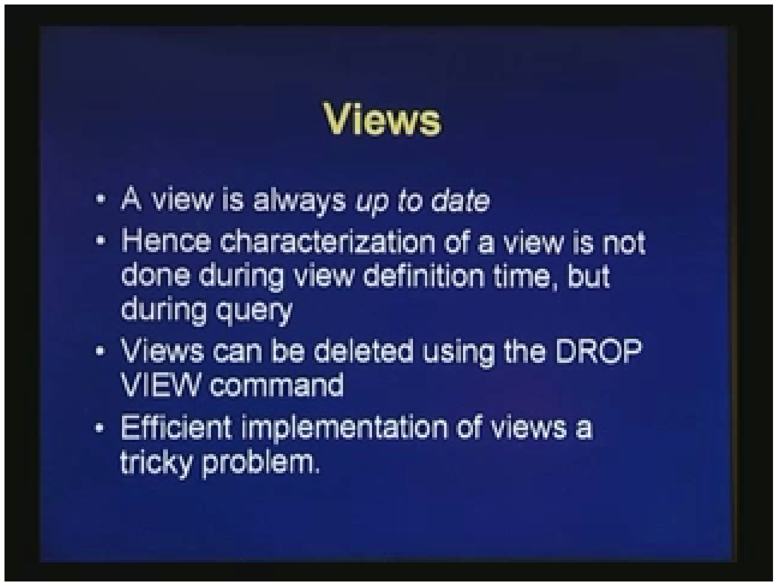
Views are specified using the CREATE VIEW command:

```
CREATE VIEW      EMD(EMPL, MGR, DEPT)
AS
        SELECT E.NAME, M.NAME, D.NAME
        FROM EMPLOYEE AS E,
                EMPLOYEE AS M,
                DEPARTMENT AS D
        WHERE       E.REPORTSTO=M.EMPNO
                    AND
                    E.DNO=D.DNUMBER;
```

So views can be created in SQL using the create view command. This is again pretty simple construct which is shown in the slide here which says create view and EMD which is the name of the view and there are three fields EMPL, MGR and department. So this is the structure of the view that is the name and the attribute list and this view is created as this query that is whenever I need to compute this view, I just need to run this or execute this query. This query is going to return a table which will populate the data that is required by this view.

What are some of the properties of views? Because a view is stored as a query, a view is always up to date. I don't need to modify a view, when I modify some data elements in my table because there is no data element that is stored in a view. Hence characterization of a view is not done during view definition time that is when I define a view I just leave it like this. When I define a view like this, it is just left like that. It is not characterized, it is not computed.

(Refer Slide Time: 00:52:06)



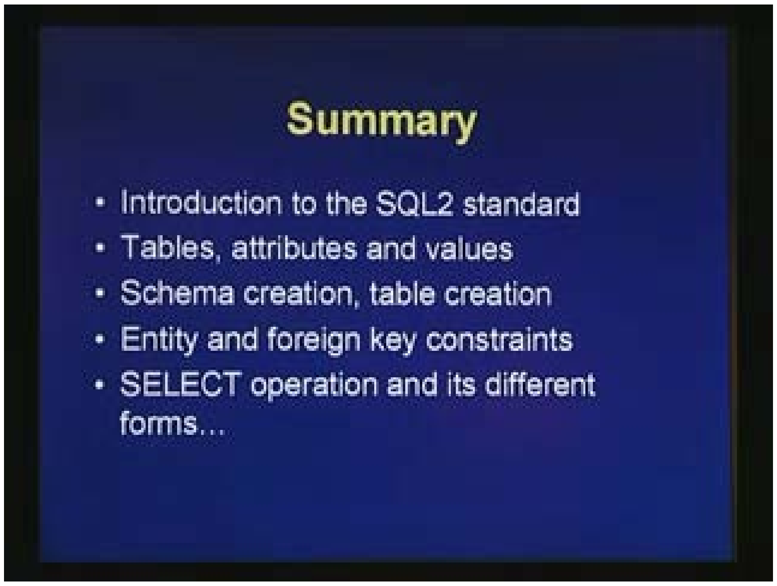
Views

- A view is always *up to date*
- Hence characterization of a view is not done during view definition time, but during query
- Views can be deleted using the DROP VIEW command
- Efficient implementation of views a tricky problem.

But this computation is done during the time of query that is when I give a query on a view this characterization is done. Views can be deleted using the drop view command just like we can delete a table. The efficient implementation of a view is a pretty tricky problem and we shall be addressing this view maintenance as a separate session in itself because how do we efficiently maintain a view and execute queries over.

So that brings us to the end of this this session. So let us briefly summarize or look at the titles of all the different topics that we have studied. We looked into a SQL two standards tables, attributes, values and constraints, entity constraints, foreign key constraints and so on. We looked at several different kinds of select operations select from where and disambiguation, aliasing, selecting from multiple tables, set operations, multi-set operations, substring operations, arithmetic operations, existence checks and null checks nested queries and aliasing and scope in nested queries and group by constructs and having constructs and so on.

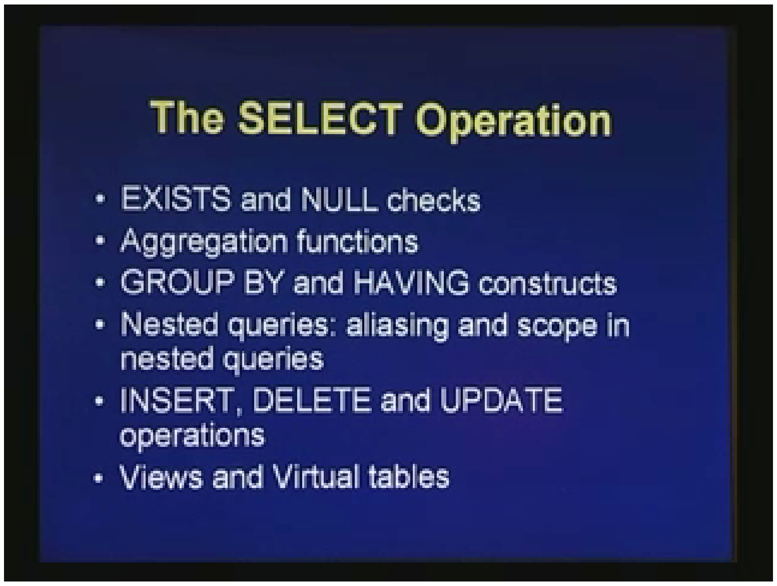
(Refer Slide Time: 00:53:14)



Summary

- Introduction to the SQL2 standard
- Tables, attributes and values
- Schema creation, table creation
- Entity and foreign key constraints
- SELECT operation and its different forms…

(Refer Slide Time: 00:03:01)



The SELECT Operation

- EXISTS and NULL checks
- Aggregation functions
- GROUP BY and HAVING constructs
- Nested queries: aliasing and scope in nested queries
- INSERT, DELETE and UPDATE operations
- Views and Virtual tables

And then we also looked at SQL statements for insertion, deletion and updation of tuples from database. Finally we looked at the notion of views or virtual tables or rather I should say that we have just scratch the surface of views or virtual tables and saw how we can specify a view in SQL using the create view command. So that brings us to the end of this session. Thank you.