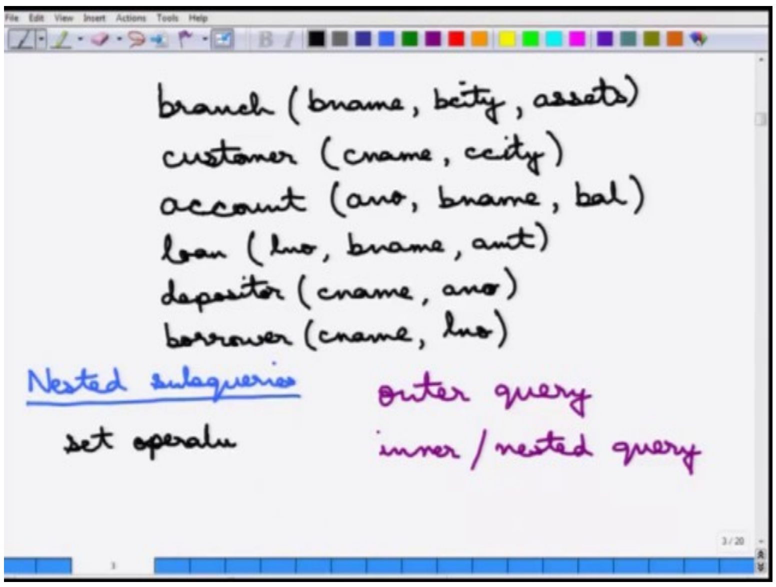
Fundamentals of Database Systems
Prof. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 10
SQL: Advanced Queries

(Refer Slide Time: 00:09)



branch (bname, bcity, assets)
customer (cname, ccity)
account (ano, bname, bal)
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

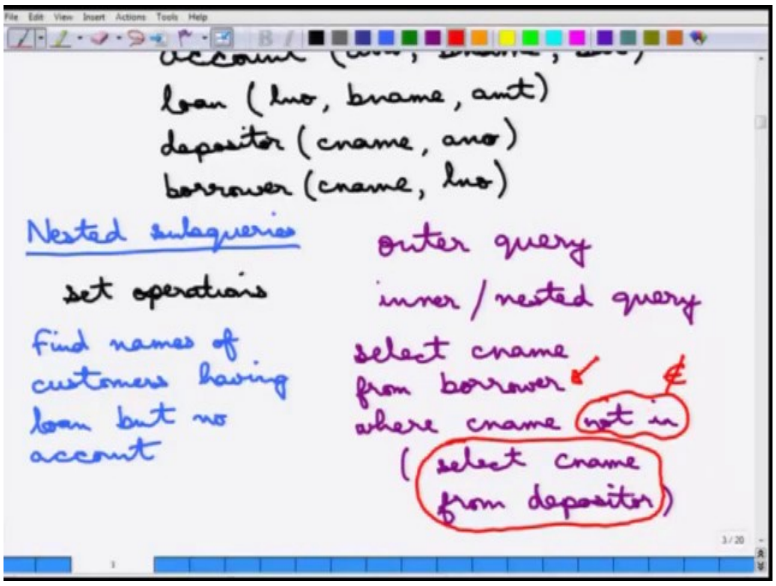
Nested subqueries
set operatn

outer query
inner / nested query

So, this is a nested sub-query. So essentially, what happens is that a query is used in the FROM and WHERE clause of another query. The query on which the FROM and WHERE is used is called the outer query and the query that is used inside is called the inner query. So inner query or sometimes it is also called the nested query. An example I will give immediately, but this is mostly useful for set operations.

So, if there are set operations, then these nested queries are very useful. So, for example, suppose, we want to find out names of all customers having a loan, but not an account. This can be solved in the following manner.

(Refer Slide Time: 00:58)


loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

Nested subqueries

set operations

Find names of customers having loan but no account

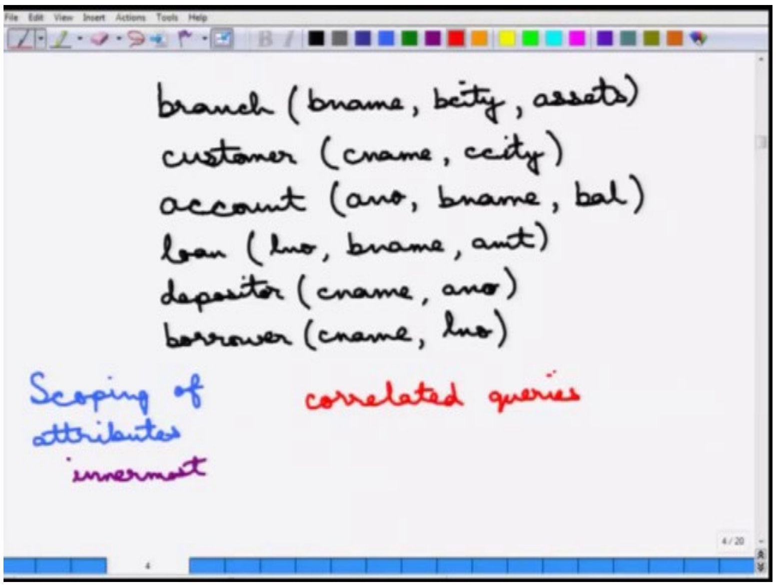
outer query

inner / nested query

select cname
from borrower
where cname not in $\notin$
(select cname
from depositor)

This is an inner query, SELECT cname FROM depositor. So let us try to understand what is happening the ... how the query is being deciphered. So, first of all, this inner query is run *SELECT cname FROM depositor*, so it essentially gives the names of all the customers, who have a depositor account. And then, it tries to select those names from the borrower, that is who has a loan, where cname is not in ... this is a set, so this is essentially equivalent to the not of set membership and this is WHERE nested sub query is very useful.

(Refer Slide Time: 01:46)


branch (bname, bcity, assets)
customer (cname, ccity)
account (ano, bname, bal)
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

Scoping of attributes

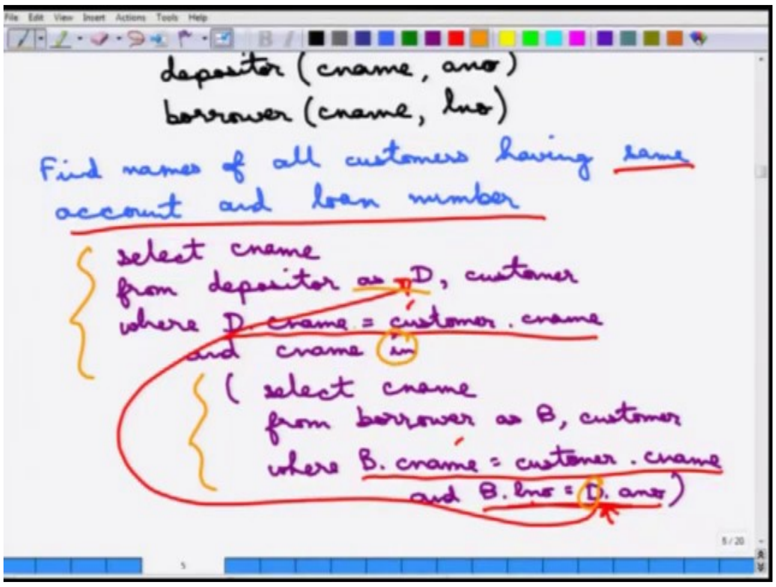
innermost

correlated queries

And a related issue for this nested sub-query is the scoping of attributes, so the scoping. So, this is the same issue as in other programming languages, such as C. When a block of statements is used inside another block of statements. And the same name is used, which scope does it apply to? And the same default rules apply. So, if nothing is qualified it applies to the innermost query, otherwise it can be qualified.

And, a nested query … so then there is some other terminology that we are going to use. When a nested query refers to something in the outer query,that is called a correlated query. Because essentially these two queries are not independent of each other, but they use the results of each other.

(Refer Slide Time: 02:33)



```
depositor (cname, ano)
borrower (cname, lno)
```

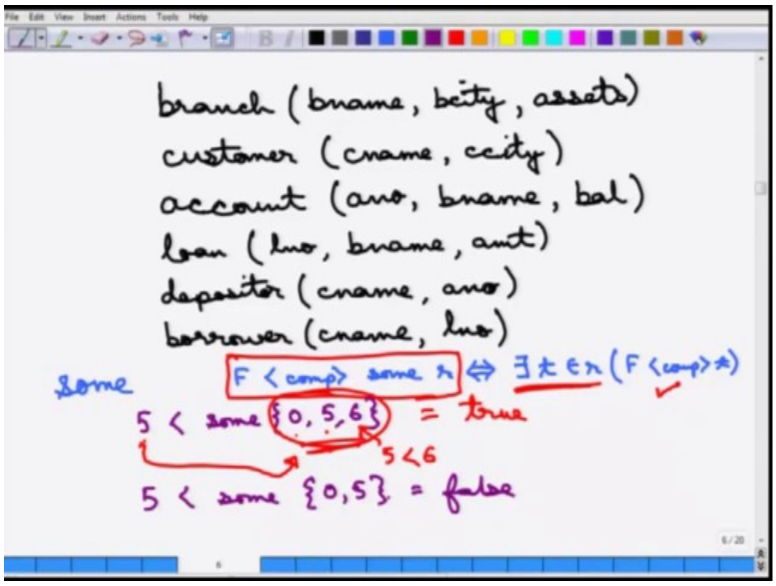
Find names of all customers having same account and loan number

```
select cname
from depositor as D, customer
where D.cname = customer.cname
and cname in
    ( select cname
      from borrower as B, customer
      where B.cname = customer.cname
      and B.lno = D.ano)
```

So, the query is, so this is solved using a correlated query and the answer to this solution can be written in the following manner, so this is the most important part of this. So, this *B.lno* is equal to *D.ano*, so this essentially this *D* refers to the one in the outside query, so this is why it is a correlated query. Let us also go over the query to see how this solves the particular. So how does it solve find names of all customers having same account and loan number.

So, let us first see, what is being done in an inner query. So, the inner query says it selects all customers, who has a loan such that of course, the name is the same, this is just to ensure that the join is meaningful and whose loan number is equal to some other account number D, where… What is D, where D is the depositor, so D is the depositor which has the same customer name. So, essentially it tries to connect *B.lno* with *D.ano*. And because of these two clauses *B.cname* is equal to *customer.cname* and *D.cname* is equal to *customer.cname*, it connects them with the same customer. This is a little complicated query, but if this is being understood one at a time, first the inner query with this scoping to the outside and then, it is easier to understand. And it uses different concepts, first is a correlated query and of course, which is a nested query then this renaming and then also this set membership operation *IN*.

(Refer Slide Time: 03:59)



branch (bname, bcity, assets)
customer (cname, ccity)
account (ano, bname, bal)
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

some

$\boxed{F \text{ <comp> some } r} \Leftrightarrow \exists t \in r\, (F \text{ <comp> } t)$

$5 < \text{some } \{0, 5, 6\} = \text{true}$

$5 < 6$

$5 < \text{some } \{0, 5\} = \text{false}$

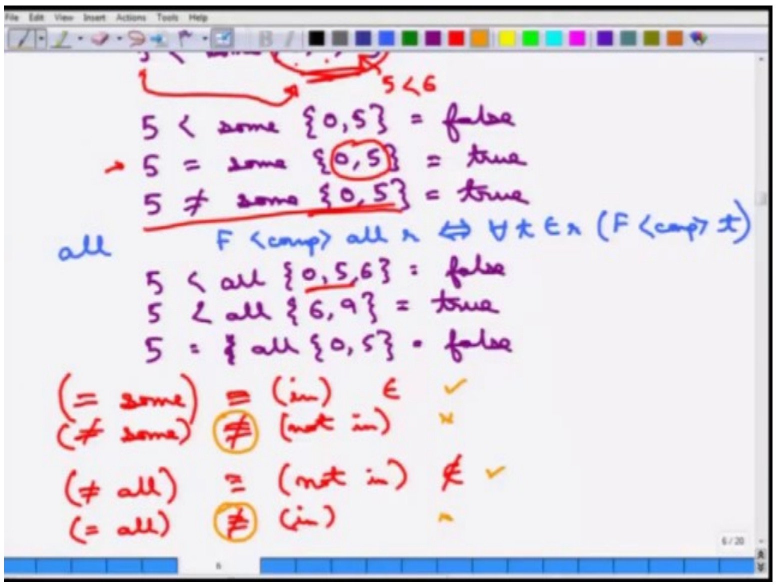
So, now, let us go over some of the set operators. The set comparison operators. The first set comparison operator that we will talk about is called *SOME*. So, *some* and here is the definition of *some*, so this is F compared to some comparator operator with *some* r. This implies that this is equivalent to … There exists some tuple in the relation r, such that F compares to t. *F <comp> some r* $\Longleftrightarrow$ $\exists t \in r$ *(F <comp> t)*

So, what does it mean? So, let us take an example, so if we take this following example 5 is less than some of, let us say the set is {0, 5, 6}.

So, what does it mean? That we must find some D in this relation and if we can find some D in this relation such that 5 is less than that, then this is true. So, how it is being solved is that 5 is tested with 0, it is not true, that does not matter. 5 is tested with 5, which is not true. But, 5 is less than 6, so that comparison goes correct. So, this evaluates to true. Because there is some element in this, such that 5 is less than … this comparator is less than one of those elements.

So, there must be at least one element, such that this holds. Then you can say that together if this holds, this comparator with some r holds. So thats the meaning of some operator. And we can see some examples. For example, if you say 5 less than some {0,5}. This will, however, evaluate to false, because there is no element such that 5 is less than that. So, this goes through and then, it can be the ... equivalent of this can be defined.

(Refer Slide Time: 05:57)



$5 < 6$

$5 < \text{some } \{0,5\} = \text{false}$

$5 = \text{some } \{0,5\} = \text{true}$

$5 \neq \text{some } \{0,5\} = \text{true}$

all $\quad F \text{ <comp> all } r \Leftrightarrow \forall t \in r \ (F \text{ <comp> } t)$

$5 < \text{all } \{0,5,6\} = \text{false}$

$5 < \text{all } \{6,9\} = \text{true}$

$5 = \text{all } \{0,5\} = \text{false}$

$(= \text{some}) \equiv (\text{in}) \quad \in \quad ✓$

$(\neq \text{some}) \not\equiv (\text{not in}) \quad ✗$

$(\neq \text{all}) \equiv (\text{not in}) \quad \notin \quad ✓$

$(= \text{all}) \not\equiv (\text{in}) \quad ✗$

So, let me just complete this. So, 5 is equal to some {0, 5}. This is true, because there is one element, where 5 is equal to true. Very importantly, and this is a little counterintuitive, 5 not equal to some {0, 5} is also true, because there is some element in the set that is not equal to 5. So this is a little counterintuitive, but this is how the thing works. So, let me go ahead with the next operator, which is similar operator, which is *all*. The *ALL* is the defined in the same manner.

So, F comparator with *ALL* r is equivalent to for all t belonging to r this must hold. *F <comp> all r* $\Longleftrightarrow \ \forall \, t \in r$ *(F <comp> t)*

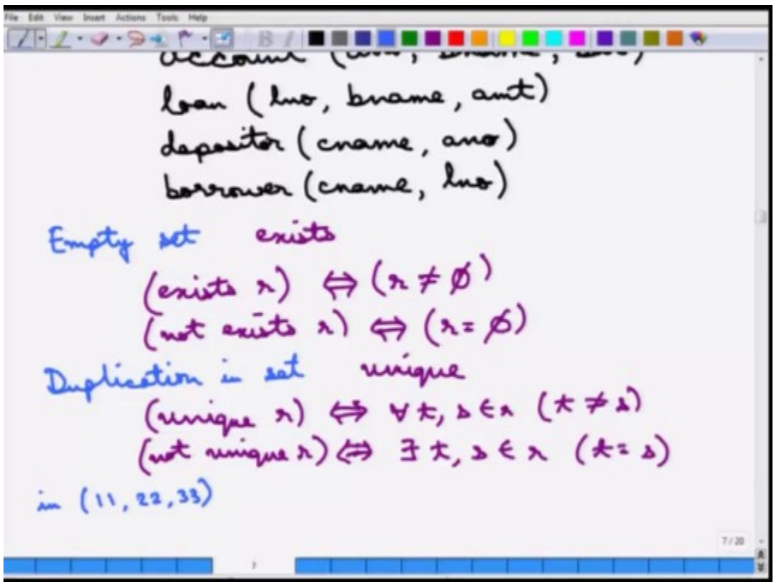
So, all the elements must follow this. So, then, if we say *5 < all {0, 5, 6}*; that is false, because there is some element 6 here, where 5 … there are some element 0 and 5, where this is not *true*, so this is *false*. And, similarly we can go ahead with the other examples, which is with ... if this is 5 less than all of, let us say {6, 9} this is true. Then, 5 is equal to all of {0, 5}, this is *false*, and so on, so forth.

So, essentially, what happens is, now we can decode *some* and *all* in terms of their set membership. So, *some*, if you say equal to *some*; that is the same as saying ... so this operator equal to *some* is the same as saying whether *in*. So, when you say for example, 5 equal to some; that means, is there any element in this set; such that this equality holds, so that is essentially the in operator this is the *in* operator.

However, not equal to *some* is not equivalent to not *in.* This is something to be remembered. Because this is not equivalent. Because ... again we see from this example this can be clear that it does not mean that it is not in. On the other hand, if we now look at all, so the all, we can break down *all* in the following manner. Not equal to *all*, if I write not equal to all, that is equivalent to not *in*. So this is essentially this. So, all of the set members must not be equal to this.

So, none of the set members can be equal to this. That means, this is not in that set. However, using the same logic, if you say equal to *all*, that is not equivalent to *in*. So these are just two of the ... two important things to remember. This goes through, this goes through, but these two do not go through. So, this is why these two are equivalents. Okay. Then there is a little bit issue with the empty set.

(Refer Slide Time: 09:09)



loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

Empty set exists

$(\text{exists } r) \Leftrightarrow (r \neq \emptyset)$
$(\text{not exists } r) \Leftrightarrow (r = \emptyset)$

Duplication in set unique

$(\text{unique } r) \Leftrightarrow \forall t, s \in r \ (t \neq s)$
$(\text{not unique } r) \Leftrightarrow \exists t, s \in r \ (t = s)$

in (11, 22, 33)

And, let me go to the next page, so there is an issue with the empty set. So, what is an empty set, how … so a set can be empty of course, and there can be tested with the ... So, there is a operator called *exists*, which tests whether the set ... whether there is an element in the set, which is essentially saying whether the set is empty or not.

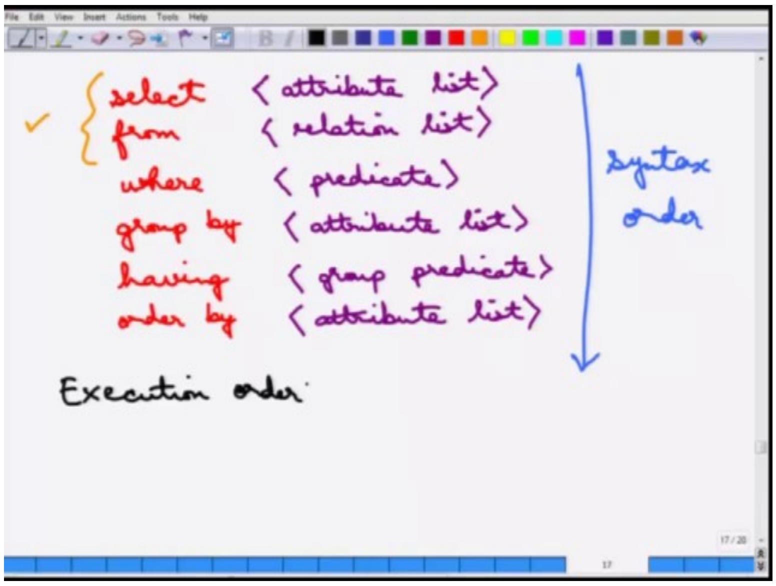
So, exists r, which essentially is equivalent to saying, r is not to equal the null set, which is this thing. *(exists r)* $\iff (r \neq \phi)$. And … the complimentary is not exists r, which is equivalent to saying r ,whether r is equivalent to phi or not. *(not exists r)* $\iff (r = \phi)$. And again, this can be used in different set operations etc. Similar to empty set, there can be duplication in sets. Because remember that SQL uses multi-sets. So duplication in a set can be tested by this operator called *unique*.

So, essentially testing whether there is something unique in that. So if you say *unique* r, that means you are testing it all the tuples in the set r is unique or not which is ... it can be written down in the following manner. *(unique r)* $\iff \forall\, t, s \in r(t \neq s)$. And then, there is not unique, meaning there is at least one tuple t, which is duplicated. So, that means, that *(not unique r)* $\iff \exists\, t, s \in r(t = s)$. So, that is how the duplication is done.

So, duplication is useful in the examples such as, find names of customers, who have utmost one account at a particular branch. So, essentially you are testing whether for that customer you find out all the accounts in that particular branch and say if the customer name is duplicated or not. If it is not you return it, otherwise you do not have to do that. So, this is the way to do it. And in SQL, just to complete the story, sets can be explicitly denoted using this kind of operator, so this is a set with this three things {11, 22, 33}.

So, something can be tested, so one can test whether a particular account number is within this or whatever. So, this is the basic idea of a SQL format. So, let me just highlight the format of this SQL, this is essentially … consists of up to 6 clauses. They may be nested.

(Refer Slide Time: 11:51)



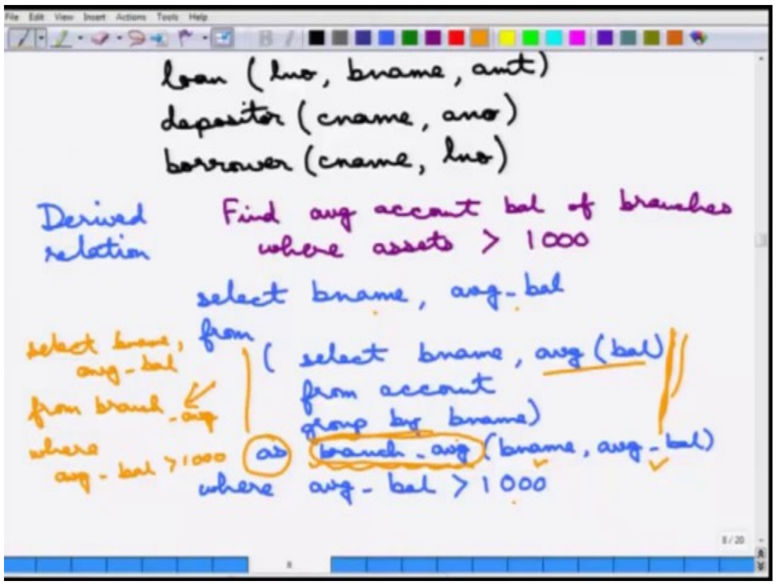
select ⟨attribute list⟩
from ⟨relation list⟩
where ⟨predicate⟩
group by ⟨attribute list⟩
having ⟨group predicate⟩
order by ⟨attribute list⟩

And these are ... 6 clauses are: *SELECT, FROM* these two are mandatory. So, these two are mandatory, this must be there. And the others may or may not be there. And it must be specified in this particular order. *SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY*. So, this is ... *SELECT* essentially is on a particular attribute, this is a summary of the attribute list, *FROM* is the relation list, *WHERE* is predicates, *Group By* is again on attribute list, *Having* is again group condition / group predicate, and *Order By* is again on an attribute list.

Note that, this is the syntax order. SQL query must happen in this syntax. However, this is very, very different from the execution order. So, what I mean by the execution order is that, suppose a particular SQL query with these following 6 clauses are being queried. Now, the question is how does the database engine solve it? Does it do the select first? It really cannot. Because unless a *FROM* it knows, which relation the *FROM* has to come, it cannot do it.

So, in these 6 clauses it is … the syntax order is not the same as the execution order. And I leave it to you as an exercise as to figure out, what can be the execution order of the 6 clauses of the basic SQL query. Alright. So, let us move ahead to this something else called the derived relations.

(Refer Slide Time: 14:09)



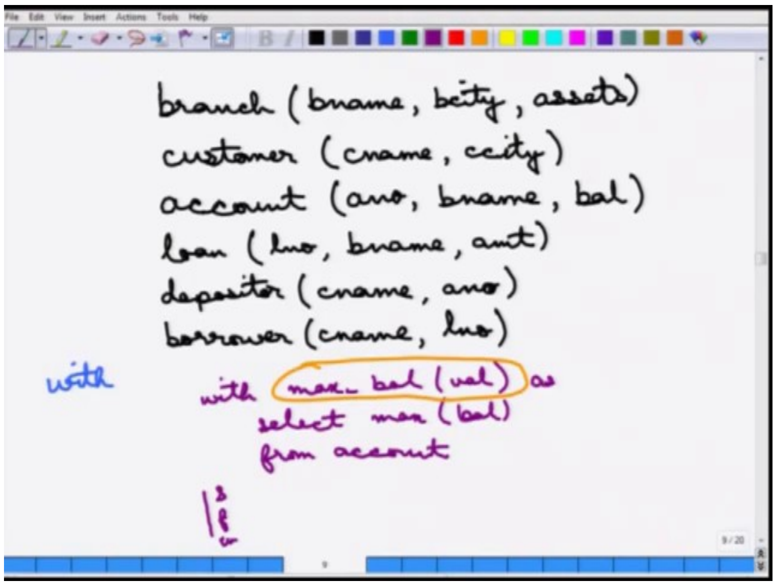
```
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

Derived      Find avg accnt bal of branches
relation        where assets > 1000

              select bname, avg-bal
select bname,  from
  avg-bal        ( select bname, avg(bal)
from branch_avg    from account
where              group by bname)
  avg-bal >1000   as branch_avg (bname, avg-bal)
                 where avg-bal > 1000
```

So, a derived relation … something that is used in the middle of a query. So, same like a nested query etc, And ... so, for example, let us takes this particular example. Okay. So, how do we solve it? Essentially there is a ...we can do it in the following manner. So, let us see how this is being solved. So, first of all, let us look at the inner query. the inner query essentially just selects the branch name and the average balance of that branch name. So it is grouped by the branch name, for each branch name it finds the average balance.

Now, this as, the answer to this query is branch average is the derived relation. So, this is a new name that is given to the answer to this query, which is the *branch_avg*. With the attributes name as *bname* and *avg_bal* So we are essentially renaming that stuff. So, this query, now becomes of the following format, *SELECT bname, avg_bal FROM branch_avg WHERE avg_bal > 1000.*

So, this boils down to SELECT, the same thing, branch name, average balance FROM this branch average, is what we have been using, where average balance is greater than 1000. So, this boils down to this, because a relation is being derived as part of the query answering. First this relation is derived, then that is being used in an outer query.

(Refer Slide Time: 15:42)



branch (bname, bcity, assets)
customer (cname, ccity)
account (ano, bname, bal)
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

with

```
with max_bal (val) as
  select max (bal)
  from account
```

And, just to go over a little bit more of the syntax of this. There is a *WITH* clause in SQL, which defines a temporary relation. So, you can define *with* … so you can write other query, some SELECT FROM, WHERE etc. Alright. So, that completes the query part of the SQL. So, we have covered all the query parts. Next, we will cover the updating. So insertion, deletion and updating. How to modify the tuples in SQL?