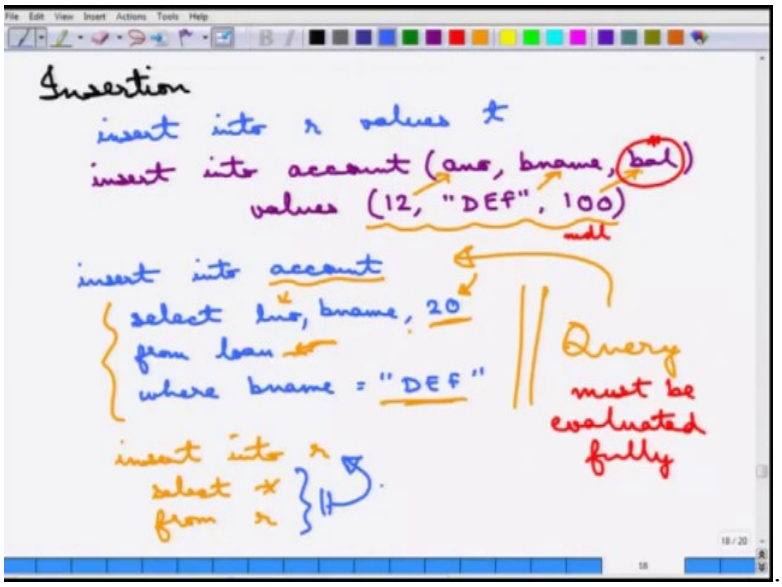
Fundamentals of Database Systems
Prof. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 11
SQL: Updates, Joins, Views and Triggers

(Refer Slide Time: 00:13)



The first thing that we will cover is the insertion. The insertion, the syntax is simply, *INSERT INTO* the particular relation name, let us say *r*, values, which forms of the values, the tuples that is formed. So, an example is, *INSERT INTO account (ano, bname, bal) VALUES (12, "DEF", 100)* we can say, essentially this means, 12 is an account number, DEF is the branch name and 100 is the balance. So, this tuple gets inserted into the account table.

If the schema is obvious, that means, if the schema of the account is actually *ano, bname* and *balance*, then this can be omitted, but it can be added. And instead of 100 etc, a particular value is not known, a *null* can be also inserted, provided balance is nullable. I mean, if the condition, you remember that while we were setting up the table name, we can define certain attributes as whether they can allow null values or not. If they allow null values, then this is fine otherwise there will be errors. Now, one thing is that, insertion can also be … so, the value of a query can also be used to insert into a table.

So, for example, this can be done.

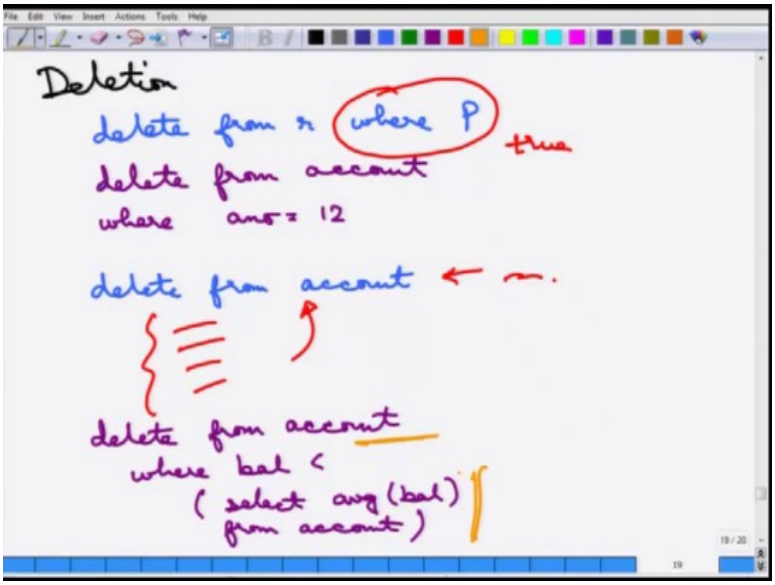
INSERT INTO account

SELECT lno, bname, 20 FROM loan WHERE bname = "DEF"

What essentially does is that, it creates a new account with the balance 20, so the balance is always set to 20, at this branch DEF. For every loan, wherever there is a loan, it creates an account and the account number is said to the same as the loan number. So, this is a way of inserting as part of that. So, this is essentially a query, which is first solved, then all the values are inserted into the account.

Now, the one thing is that the query must be evaluated fully before the insertion starts, as otherwise there can happen ... infinite insertions can happen. So, very easy example is the following. So, suppose you want to duplicate a relation. For whatever reasons, its not very clear why. But, suppose you want to do that, so you can write as queries, simply *insert into r select * from r*. Now, imagine what will happen. If this is not being null, so this must be evaluated first, so … and then all the results are inserted into it. So, this is only .. it returns 1, so this essentially duplicates the relation. That it, I mean, duplicates the tuples in the relation.

(Refer Slide Time: 02:39)

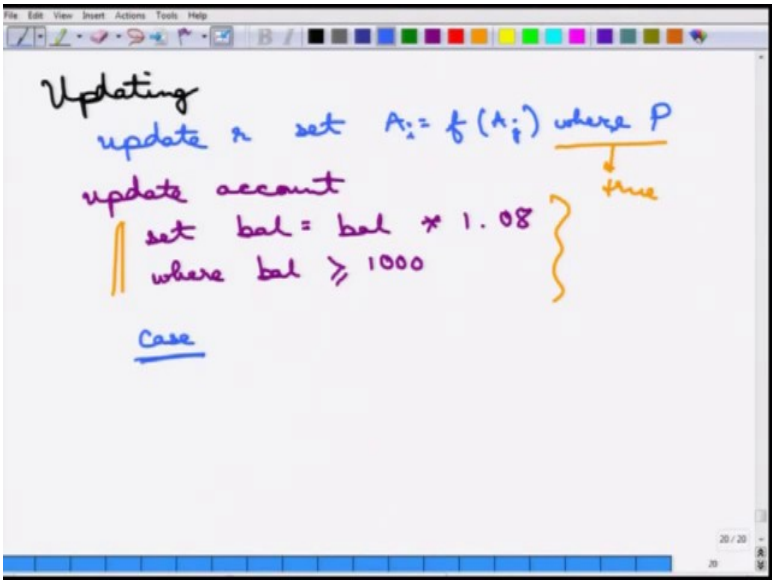


So, then, we can move on to deletion. The syntax is *delete from* relation *where* there is a predicate. So, you delete from the relation r, where the predicate is correct. So, example is, so essentially the account number 12 is deleted. And if the where is empty, so if we simply say *delete from account*, the same rule is applied, if the where part is empty, then it means true,

which means every tuple will be deleted from account. So, it ... at the end of this, it returns an empty relation nothing else is.

And just like insertion, delete can also … you can use a query as part of this delete thing. So, only where the query is evaluated and only those tuples that satisfy the query can be … are deleted. And, interesting example in this space is the following. Once more, so what it tries to do is, it deletes all accounts where the balance is less than the average. Now again, once more, the balance must ... the average must be completed first and then, the deletion happens, otherwise it will keep on deleting. You can see what will happen.

(Refer Slide Time: 03:54)



Finally, there is this updating. The syntax is *update* relation r *set* certain values, set attributes lists etc. So, set attribute lists equal to sum function of whatever, some other attribute list. So, set A_i is equal to function of A_j, where P, so the predicate. An example is,

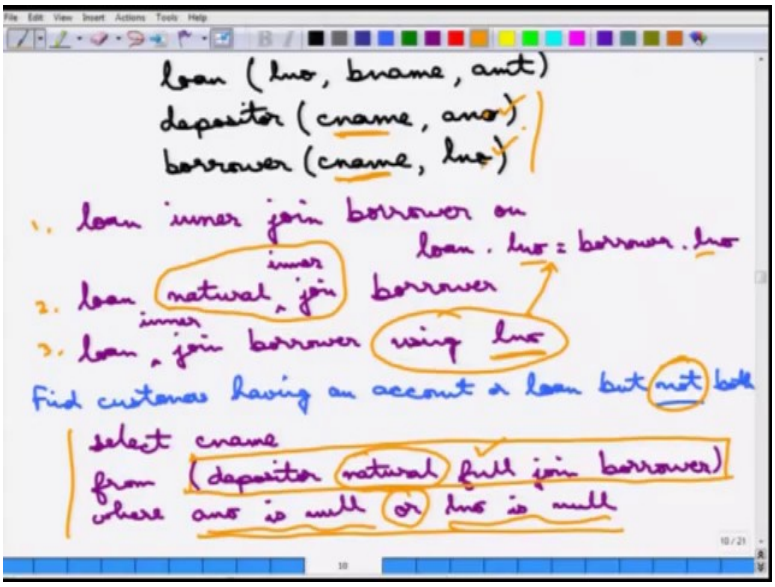
*UPDATE account SET bal = bal * 1.05 WHERE bal ≥ 1000*

So what this tries to do is, every account, where the balance is greater than some amount 1000, 8 percent interest is given, so the balance is essentially incremented by 8 percent, so this can be done.

Once more, *where*; if the *where* is empty, then it evaluates to true. So, that means, everything will be done and so on and so forth. That's the same issue as past. And again, it depends on how you are doing, so again, this can be a query of … which is first evaluated and then, this thing is done. And there can be case ... there is a case clause that can be … which is

essentially equal to the switch case kind of thing and there we can see the syntax later. So, this does … the basic queries and the database modifications are complete. We will next do another very important topic next.

(Refer Slide Time: 05:21)



So, we will next cover the join in SQL. So join, as we saw there are different types of join. Inner join. Then there is a left join, which essentially means it is a left outer join. Right join. Full join. Natural join. These (first 4) fall in one group. The natural join falls in another group. A particular join can also be set, can be set *on* a particular predicate and then, it can be also set *using* which attribute. So, the *on* is on the predicate and *using* is the attribute. So, these are the different ways one can join.

So, for example, we can say *loan inner join borrower on,* you can specify the condition, which is *loan.lno = borrower.lno*. Now this is equivalent to saying, this is *loan natural join borrower*. This is again equivalent to saying, *loan join borrower*. Okay, I am sorry, this should be *natural inner join* and this is *loan inner join,* using *lno*. So, all these three queries are equivalent and they are different ways of stating.

So, the natural inner join is the same as the inner join on that number, because it is the same attribute value. So, it is essentially, if you say, using real number, it essentially translates to this condition and it reaches the same as writing it down explicitly *on* and there is the natural join as well. So, an example of join may be, so following can be one way of solving it. Now, do remember there are different ways in which a particular query can be solved or there are

different equivalent SQL statements that can be written that will solve the same query.

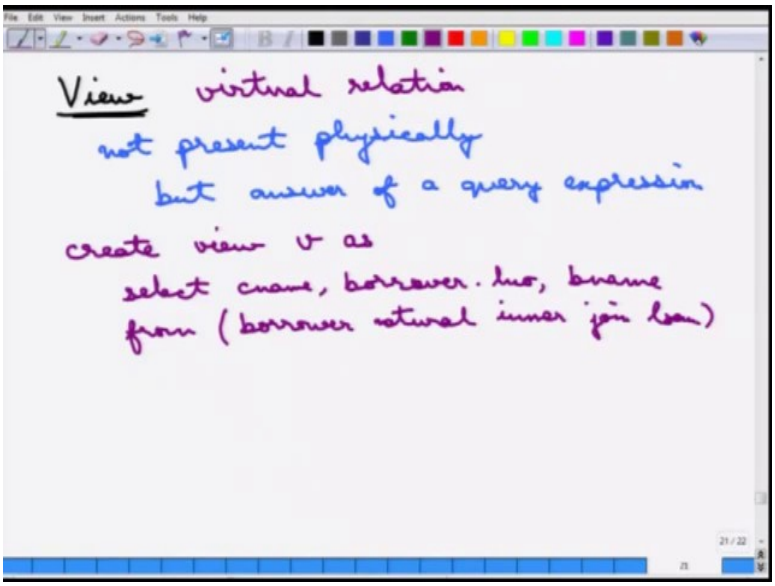
Find customers having an account or load but **not** both.

So, here is one way of doing it. *select cname from,* this is what, I am writing it down. So, *depositor natural full join borrower where ano is null or lno is null*. Now, this requires a little bit of thought. So, what it is being done is the following is that, first of all, this *depositor natural full join borrower,* this will create every possible combination of *depositor* and *borrower*, so it will find out all customers … now this is a *natural full join*, so, when you say *natural join* the *depositor* and the *borrower* table, they agree on the customer name.

So, it must have that when you are joining, the *depositor* and *borrower*, they must be the same customer name. So it will find out all ... for each customer name the account number and the *lno*. Okay. Now, so, even if there is an account number, but not corresponding *lno*, because it is a *full join* it will output them.

Now, what do we want is that we want those kind of things, where there is either an account number or an *lno*, so at least one of them is null. So, that is why this is an *or* clause, so at least one of them. So, if a customer has both account number and *lno*, then both will be valid. So, this will be false. So, that is not going to be output. So, this is the way to solve find customers having an account or loan, but not both. So, this completes the part about SQL etc and, so there are some more constructs in SQL, some … a little bit of constructs that we will cover next.

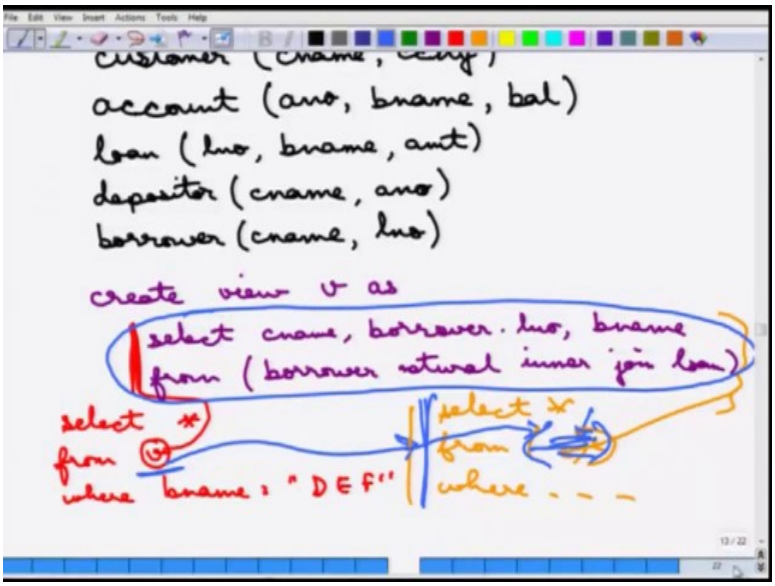
(Refer Slide Time: 09:55)



So, some important other constructs of SQL. The first one is the ***view***. So, a view is the answer of a query that is not present physically, so this is not present physically, but answer of a query expression. So … essentially a view is … a view can be considered as a virtual relation. An example. So, why are views is needed? Views help in query processing. An example may be the following is that, suppose this is being done.

So, *create view*, so this is the syntax to create a view, *v* as ... you say. So, what does this view tries to do is find all the loans of the customers, but, but it will does not bother about the loan amount. So, if we go to this example, so, so this is the view. So, this view can be later used anywhere, where this is use ... useful. So, if we do the following things.

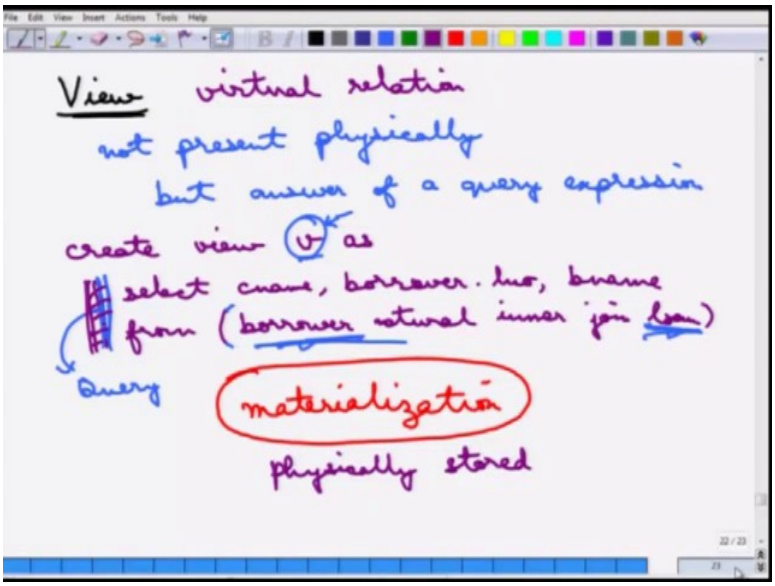
(Refer Slide Time: 11:16)



So, suppose *select * from v where* branch name is equal to DEF. So, you want the names of all customers who have an account at the DEF branch, well that's it. So, this view v is essentially this ... the part of this query. So, this is essentially the same as writing *select * from*, then this entire thing will go here, this is here, where the same clause is being done. So, important thing to note is that, how are view stored? The view as I said is not stored physically.

So, what is being stored is the query expression is being stored. So, what is stored is this following query expression. So, wherever v is used, it is actually replaced by the query equivalent query expression, so this is the query expression. So, it is actually being replaced by the query expression and at run time, this query is evaluated and the entire answer is resulted. So, why is such a case done?

(Refer Slide Time: 12:34)

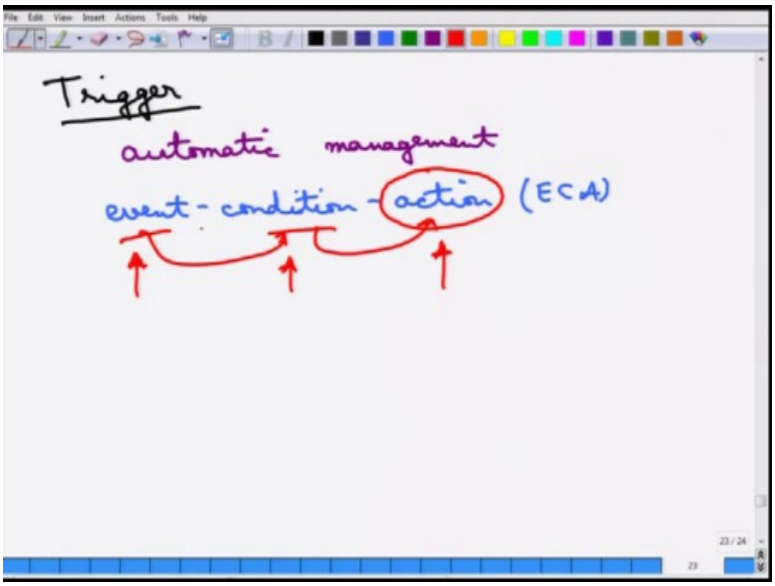


The reason is when the view is used the relation that it uses the view from may have changed. So, for example, if we go back to the query that we see, this create view v, uses the following tables *borrower* and *loan*. Now, if you store this as a table, if *borrower* and *loan* has changed, there is no way the *view v* will know that change. On the other end, if only the query expression is stored, then it is fine. Because, if *borrower* and *loan* changes it does not matter, this is evaluated again at query time … at run time, so nothing else is problem.

So, that is why there are restrictions to, what can be done, which kind of views can be done and whether a view can be updated etc. A view may not be updated because updating a view essentially means updating the *borrower* and *loan* tables right, which is not clear how to update or it may get into the problems of null tuples, null values, and all those things.

For some views, there is a term called *materialization*. So some views are materialized. So, this depends on the query engine etc., the database engine. When some views are materialized, it essentially means the view is stored physically, is physically stored. Now, why will that be done when the view is simple enough etc, and when the view is used in multiple queries, then it makes sense to materialize a view, because then, this following query is not going to be evaluated at run time etc. So, that is the view materialization issue. And otherwise, a view is not very much updatable etc. Fine.

(Refer Slide Time: 14:20)



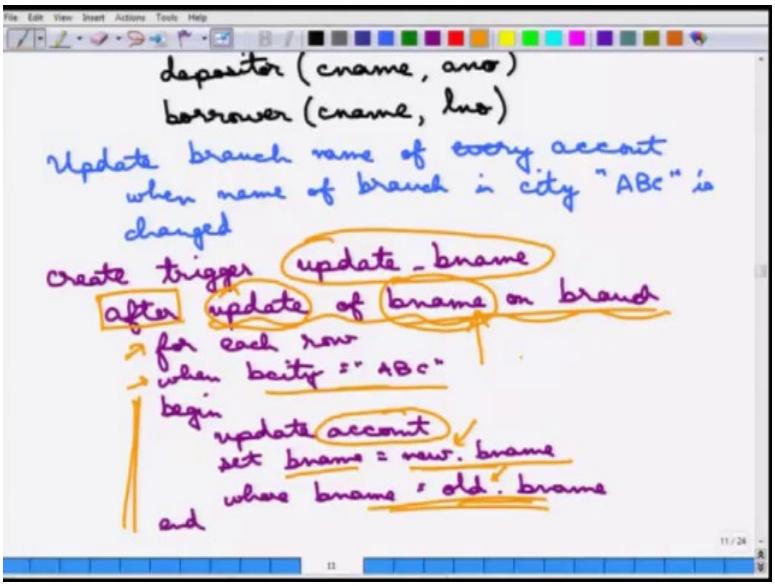
So, the last topic, that we will cover in this SQL is something called a *trigger*. So, a *trigger* ... so let me write it down what a trigger is. A trigger statement allows automatic management of database stuff. So, there is a ... so, it's an automatic … so, whenever some action takes place in one relation automatically some other statements take place and it triggers ... essentially it triggers a couple of things.

For example, whenever a grade is submitted, for example, in the in the example of student databases. Whenever a grade for a particular course is submitted the CGPI of the student is automatically recalculated and that is being stored. So, that can be written as a trigger. So, whenever a new grade is inserted, there is a trigger that will automatically recompute the CGPI. So, it essentially follows, what is called an ***event - condition - action*** model. So, this is a ECA model.

So, whenever an event happens it checks for certain conditions, if that is true, the corresponding action is being taken. So, the event is essentially a database modification, so such as an insertion, updates or deletion etc. The condition is a predicate and the action is any other database action or some even external programs can be done etc. And the action can be done either before or after the event. So, this is a modification event.

So, the action can be specified as a 'before action' or an 'after action'. So, generally it is an 'after action', but it can be also specified as a 'before action'. For example, we can think of the following query.

(Refer Slide Time: 16:01)



So, create trigger, so it says … *create trigger,* you can say update, the name of the trigger, you can give something, let us say, *update_bname*. This is a trigger. After, so, this says that, okay, after that, this is a after model, so when the event has taken place after that ... *after update of bname on branch*. So, then you can say how this update will be done for … so what is the action that can be taken.

So, *for each row* ***when*** *bcity = "ABC"*, you say you have the *begin*, there is a *begin* and *end* statement. *Update account,* in the following manner, *set bname = new.bname where bname = old.bname,* this is the *end*. So, this example covers a lot of issues, so let me go over them one by one. So, it first says *create a trigger* the name of trigger is *update_bname.* Fine. This says *after*, so that means it's an after event. *After* what is the event? Update of branch name on *branch*. So, whenever there is an update of branch name on *branch* this trigger is essentially invoked. Now, how it is done? So that … the trigger is applied *for each row* only when the branch city is ABC. This is the condition on the trigger. So, it is not for everything on the branch. Then, what is the action that is being taken? Is that, the *account* table, the *account* relation is updated in the following manner. The branch name is set to the new branch name for those things where it was the old branch name.

So, this *new* and *old* are essentially special markers. So, which says in the ... because it's an update event, so this ... the new branch name is got from here and the old branch name is got from whatever was already in the database. So, this is an example of how to work with the

trigger. So, this completes a part of SQL. So, we have covered about all the basic operators. We have covered nested subqueries. We have covered, the updates, the deletion, insertion etc and some special issues in SQL, which are the views and the triggers.