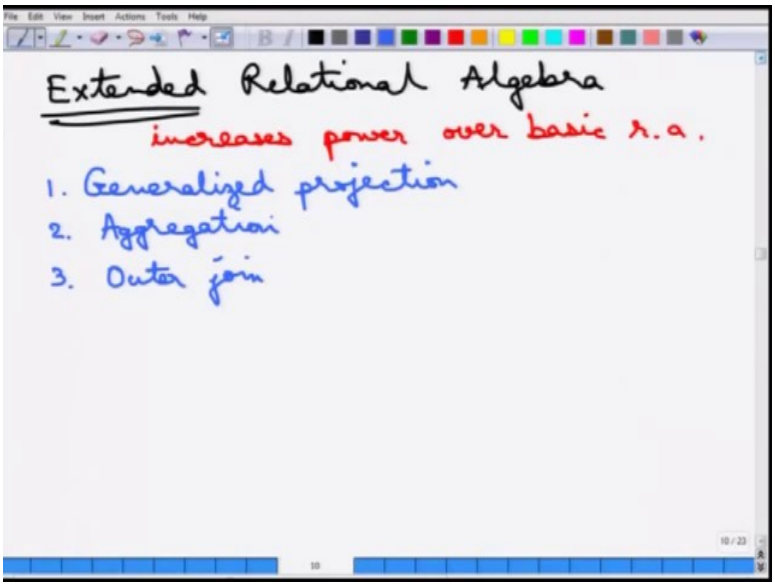
Fundamentals of Database Systems
Prof. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 06
Relational Algebra: Extended Relational Algebra

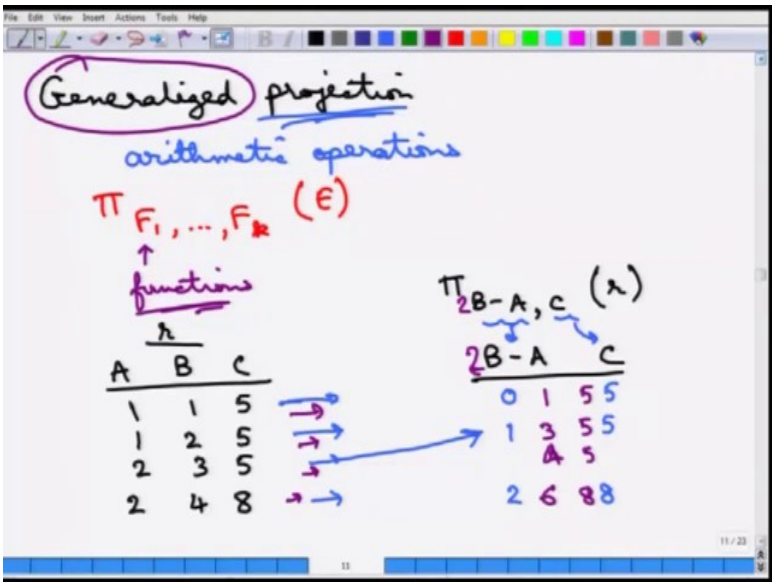
(Refer Slide Time: 00:14)



Let us move on to something called an **Extended Relational Algebra**. So, this is extended, so it does increase the power. So it increases power over basic relational algebra. So, the operators here is there are three operators, first is called a *generalized projection*, the second one is *aggregation* and the third one is called *outer join*.

So, at least from the names it can be guessed that this is a generalized projection; that means, it has got something to do with the projection and tries to make it more generalized, so it tries to extend its power. Similarly, outer join is some form of join, but with something more and we will see exactly, what all of those things. Of course, aggregation is something new. So, let us go over each of this in a little bit more detail.

(Refer Slide Time: 01:32)

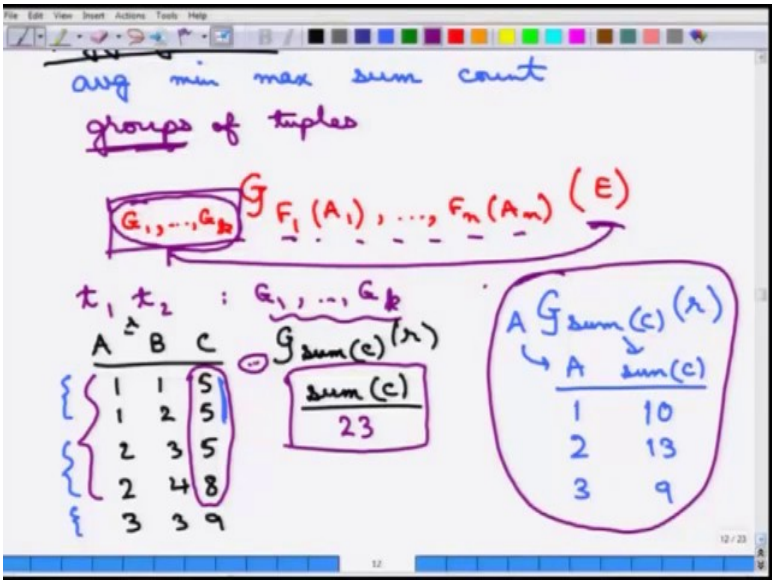


So, the first one is **generalized projection**, so what it does is that it takes the projection operator, the normal projection operator. So, the normal projection operator allows only it names certain attributes and just selects out those attributes, it just projects out only those columns. So, what the generalized projection operator does is that it allows arithmetic operations on those projections. So, arithmetic operations on the projected columns and what do I mean by that, so it essentially says that it is defined like this, $\Pi_{F_1,\ldots,F_k}(E)$ this is a function F_1. So, you can define some *k* functions over the expression, so these are instead of just names these are functions, so these are generalized arithmetic operations or generalized arithmetic functions. So, for example if we take this example suppose there is *r*, which is defined as (A, B, C). Now suppose, what I do is, I do something like this, so $\Pi_{B-A,C}(r)$. First of all, the schema for this becomes $B - A$ and C. Why is that? Because, this schema these are copied, essentially this is the first thing and this is the second attribute and $B - A$.

So, B minus A, so it is done from here $B - A$ is (0, 5) fine, then this is 2 minus 1 this is (1, 5), then this is again (1, 5), so this goes here, so there is nothing new is output and this outputs (2, 8). So, these are the answers, then now, well you can change a little bit on this, so instead of this thing you could have said $2B - A$, then this could have become $2B - A$ by C. So, from this, this would have produced (1, 5), the second one would have produced (3, 5), the third one would have produced (4, 5) and the fourth one would have produced (6, 8).

So, you can see that any operation can be done, now instead of $2B - A$, you can do lots of other, other things etcetera. But, this is how the generalized projections works; This is what the generalized part is, because it works on any function. So, generalized projection that is how this works.

(Refer Slide Time: 04:08)



The next one is **aggregation**, so it let us one aggregate, so it can use certain aggregation function, such as *average*, *min*, *max*, *sum*, *count*, so that is what it does. So, it can be applied on tuples or more importantly, it can be applied on groups of tuple. So, I mean this can be applied on the entire relation or on certain groups of tuples and these groups can be defined, The grouping is done. So, the generalized form of this aggregation operation can be written in this manner.

So, there is this generalized *G* function and let me try to write it there and you are applying some function F_1 on A_1 and so on, so forth. So, you can apply some n functions on this and a grouping is defined using some grouping operators, grouping values, which is $G_1, \ldots, G_k$ of course, this is on some expression here. So, what it means is that, this expression is first taken and then, the grouping is done using this G_1 to G_k.

Then, for each group where each group according to this G_1 to G_k, each of these functions are applied, then $F_1(A_1)$, F_1 on the attribute A_1 which is part of the G_1 to G_k, $F_2(A_2)$,

$F_n(A_n)$; that is when applied.

Now, how is this grouping done? This grouping is done, so tuple t_1 and t_2 are in the same group if they agree on all G_1 to G_k; They must agree, so for all of them. So, essentially this everything has to be same for t_1 to t_2.

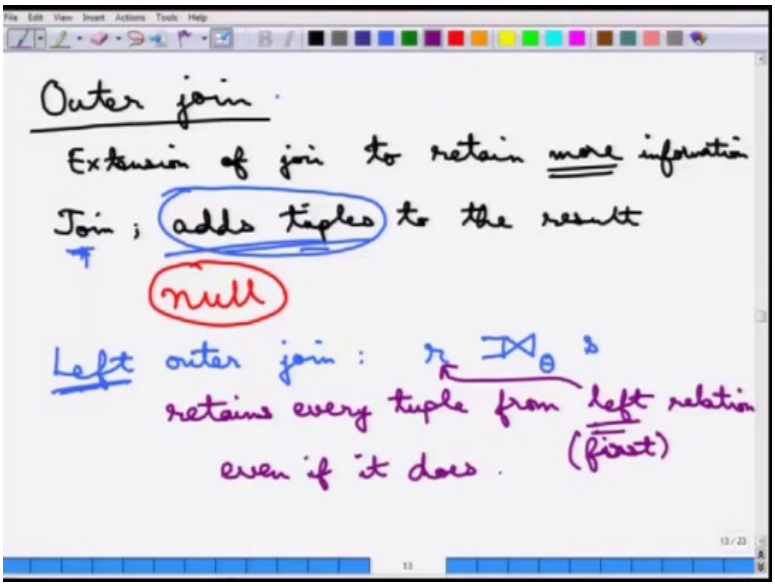
Now, what does that mean? Let us again take an example that is the best way of understanding it. So, let us say this is (A, B, C), there is a relation *r*(A, B, C), so now, you are doing this operation, let us say $G_{sum}(C)$; that is all that is being done. First of all the schema that is produced is simply $sum(C)$; that is what is being done and what is being done is that just the sum over. So, let me explain what is it, there is a grouping here defined; that means, that every tuple is in the same group, so everything is in the same group and essentially just the column C is summed up, so the answer to this is 23, that is it, this is the answer to this.

Instead if you try to define a little more complicated query for us, suppose you say *G* is being done $sum(C)$; that is fine, but with an A. So, by the way I should say that this is of course, some *r* and this is also of course on *r*, so what is being done is that this is the following, ${}_AG_{sum(C)}(r)$

So, first of all the schema of this is *A* and then, $sum(C)$. Why is that? Because, this grouping is done on *A*, so all the tuples must agree on the *A* value and that *A* value is written here and then, the corresponding sum is done. So, how do we do that? So, first of all, all the tuples that agree on the *A* value; so that means, these two are in one group and these two are in the second group.

Now, for these two the *A* value is 1 and the sum is just this, which is 10 for these two this is two and the sum is 13. Now, suppose there were another tuple, which is what would have happened is this is in a group by itself, so these would have been simply (3, 9). So, this is the answer to this query of the aggregate operation with this grouping; that is, what that it has being done.

(Refer Slide Time: 08:24)

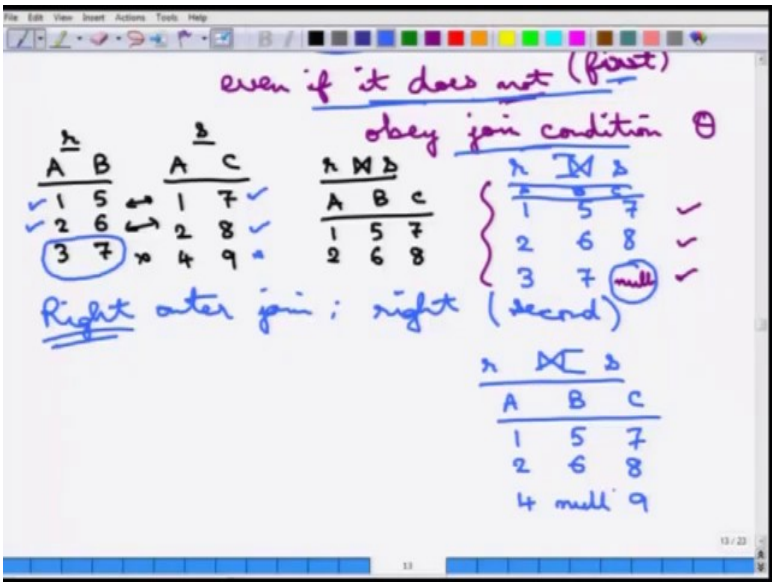


Let us see the final operator in this space is that of **outer join**. So, in outer join, what happens is that it is an extension of join to retain, so it is an extension of the normal join that you see, join to retain more information. So, how do you retain more information is that, it first does the join and then adds tuples to the result. Now this seems to be very interesting, that how can it add tuple to the result, which is not defined as part of the join.

So, for the first it computes the join; that is given as part of the condition about that join, it adds all those tuples that are part of the join they need to add certain more tuples. Now, what more tuples? This requires the use of something called a **null** value and we define **null** earlier in some connection that **null** is a part of every domain etcetera. So, null can be considered to be the value for any attribute of any tuple, this is the specialty of **null**.

Now, let us, define one at a time what is it. So suppose we first define, what is called a *left outer join*, so this is called a left outer join. So, this is denoted by r, this is an interesting operator, this is the join operator with theta, because this is left outer, there are two strokes on the left, so this is $r \; ⟕_\theta \; s$. So, what it does is that, it retains every tuple from left relation. Now which is the left relation? Of course, this is the left relation, because this is on the left of this, left or; you can say the first relation it retains everything from the first relation.

(Refer Slide Time: 10:52)



Even if it does not obey join condition, even if it does not obey join condition θ. Still probably not clear us to, what is happening, but what is happening is essentially this. So, let us probably take an example; that is the easier way to understand. Now if we do a natural join of $r \bowtie s$ simply natural join. Now, what happens is if you do a left join of s $r \mathbin{⟕} s$, what it does is that, it first completes the join which is (1, 5, 7), (2, 6, 8).

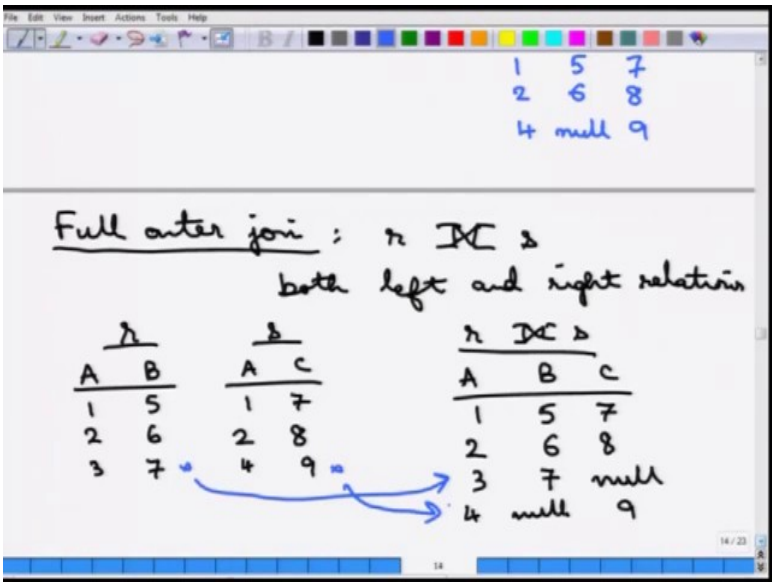
And then, it retains every tuple from the left relation, even if it does not obey the join condition. Now which is the tuple that it does not .. So this tuple has already been captured as part of the join condition this tuple has already been captured as part of the join condition, but 3 and 7 is not captured. So, it must retain (3, 7). Of course, this is (A, B, C). (3, 7.. The question, then comes is, what will happen to this value of C? It has got nothing defined, but then the power of *null* comes it essentially says if nothing is defined with this *null*.

So, the answer is (1, 5, 7), (2, 6, 8) and (3, 7, null), so this is a special value that is being used and this is a left join. So, now, that the left outer join has been defined it is probably easier to define the *right outer join*, which is essentially the same thing except the left of the first relation is replaced by the right relation or the second relation and everything else is analogous. So, now, to just complete the example the right outer join for that same example is this way.

So, that first of all the operator is written this way $\bowtie$, again and it has once more it has the same schema as (A, B, C). Now, once more first of all, the first thing to do is to complete the

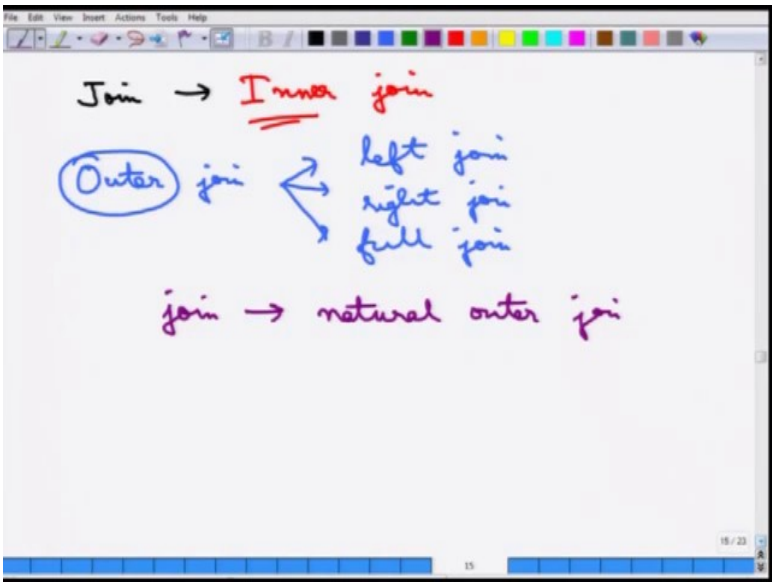
join, which is (1, 5, 7), (2, 6, 8), then, what is left out. So, this is captured from the right relation this is captured from the right relation this is not captured, so this needs to be copied down. So, 4 and 9 and this is added with null, so that is the *right outer join*.

(Refer Slide Time: 13:28)



Now, that right outer join is defined, so there is something called a *full outer join*, which is the combination of both right outer join and left outer join. So, this is the operator for this ⟗ and it captures from both left and right relations. So, once more going to, that if you go back to that example of (1, 5, 2), (6, 3, 7), then the complete outer join $r \text{ ⟗ } s$ is defined as (A, B, C) the first we complete the join, which is (1, 5, 7), (2, 6, 8) and then, we complete the left and right join. So this is (3, 7, null) and (4, null, 9), so everything is completed from. So, these were the things that were left out here and left out there, so both find their way in the *full outer join*.

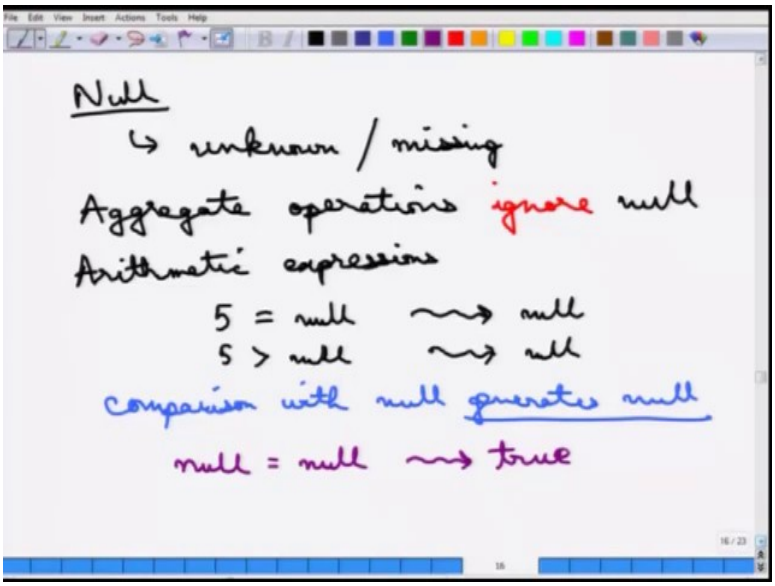
(Refer Slide Time: 14:39)



Now, that these outer joins are defined, so that *normal join* is sometimes called an *inner join*. The word inner comes is now understandable, because there is an outer join. And sometimes the word outer join this outer part outer join the outer part is omitted. So, then what do we essentially get is the *left join*, because there is no left outer join a *right join* and a *full join*. Because, there is no left outer join right outer join and full join.

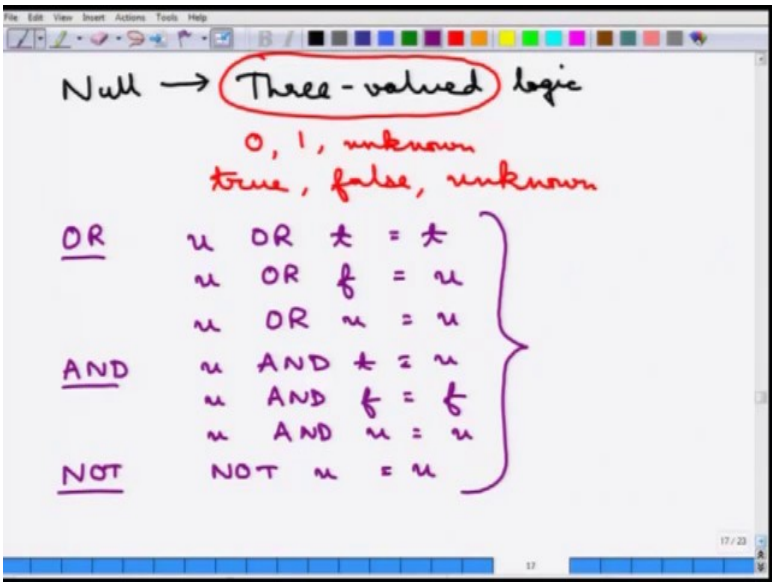
And, when no theta condition is specified, when it is just the word join is used when simply the word join is used, then it translates, it is generally meant as a *natural outer join*. So, these are just part of the terminology. So, then there is a little bit left about these relational algebra queries, which is the special value about ***null***. So, we have already seen, what how ***null*** are useful etcetera and ***null*** is a special value.

(Refer Slide Time: 15:51)



So, it essentially denotes an unknown or a missing value. We saw in the examples in the outer join, that when A and B are joined, we do not know the value of C is, so it is a unknown or missing and that is being denoted as *null*. So, now, this null and let us see, what the aggregate operation operators ignore null, so aggregate operations simply ignore null. But, what happens with arithmetic operations arithmetic operations may require comparison such as 5 equal to null, 5 greater than null etcetera. So, what happens is that, so for example, 5 equal to null this evaluates to *null*. Or 5 greater than null etcetera, so all of these things evaluate to null. So, a comparison with a null, generates another null. There is one very interesting exception, which is null equal to null this evaluates to *true*.

(Refer Slide Time: 17:19)



So, null equal to null is actually true. So the ***three valued logic***. So, using null, null generates to a, what is called a three valued logic. So, we have been all used with binary logic, which is using 0 and 1. There is a three valued logic. The three values are 0, 1 and *unknown*. So, essentially *true*, *false* and *unknown*. So these are the three values and let me write down the rules for all of this three valued logic.

So, first of all the OR table, so OR is, unknown OR true evaluates to true, because there is a true and it does not matter what the unknown is. unknown OR false evaluates to unknown, because the false does not matter and this is it, unknown; that is it. unknown OR unknown that evaluates to unknown.

AND is similar. So, unknown AND t is unknown. unknown AND false is false. unknown AND unknown is equal to unknown.

The NOT is there. So, NOT of unknown is equal to still unknown.

So, this is the three valued logic table that we require to move forward. The select operation treats *unknown* as *false*. I mean if there is something unknown, then the select operation will not select it; That is what it will be meant.

So, that completes the part about the queries of this relational algebra and we will later go over, how to use relational algebra for database modifications.