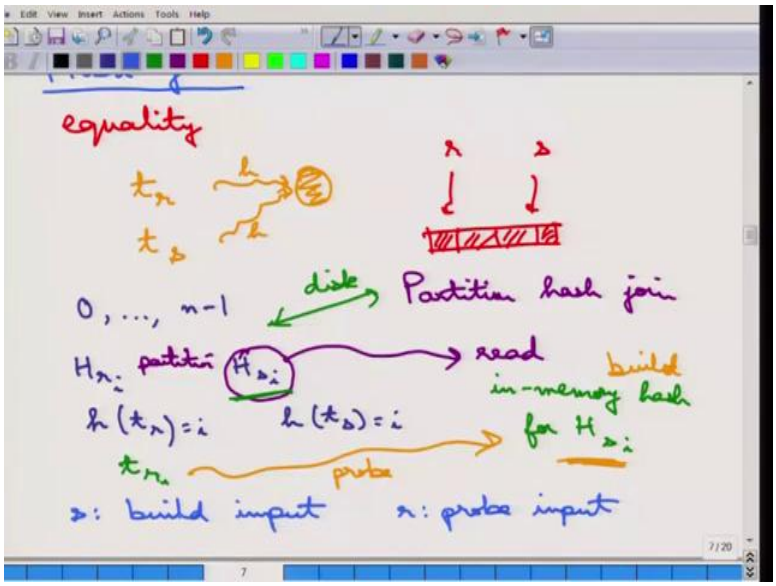
Fundamentals of Database Systems
Prof. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 23
Query Processing: Hash Join and Other Operations

(Refer Slide Time: 00:21)



We will move on to the next algorithm which is Hash join, this is a very, very important algorithm and a little more complicated than the previous ones. Once more this is applicable only when the condition is equality, that is, it is an equality join. So, the idea is the following, take a record t r and take a record t s. Since, it is an equality join they must be equal; that means they must hash to the same position. So, if this is a hashing function that is applied, they must be hash to the same position.

So, what is being done is the following is that, first of all both the relations r and s are hashed to the same hash table and then, from each of these hash buckets the matching records are picked up one after another. So, that is the entire algorithm and there is a hash function to partition the records. So, if the hash function, if the relation does not fit into memory then this hash function needs to partition it and then it is called a partition hash join and just a little bit more detailed on that algorithm, what it being done is that…

So, suppose the hash table entries are from 0 to n minus 1, so there are n entries. So, each record, also each record of r and s, so say this is H r i and H s i these are the partitions of

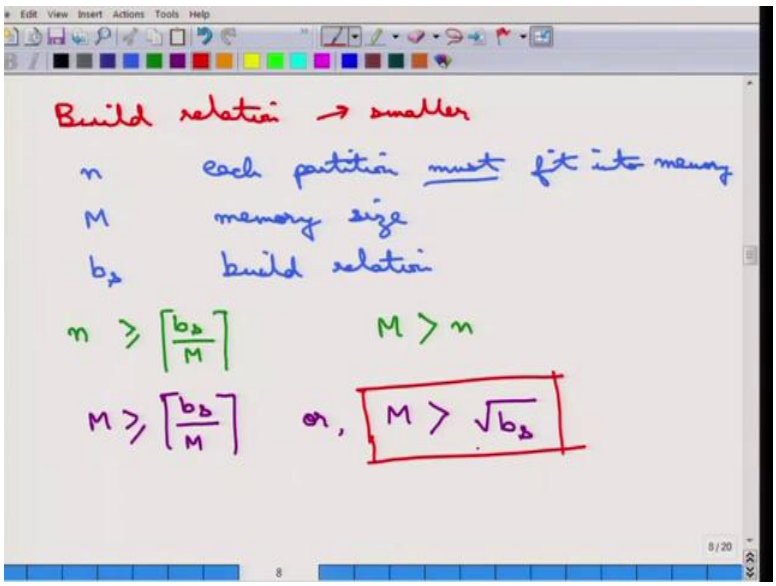
r and s. So, this means that essentially hash function on t r is i, then it goes to the partition H r i and hash function of t s is i, then it goes to the hash function of, the hash bucket of H s i. Once that is done, the partition H s i is read, so these are partitions.

Once this is, this table is done, this partition H s i, one of the partitions from H s i is read. So, this is read into memory and in memory hash index is built, so remember that this partition hash function is the out of disk. So, this is in operating at a disk level, so this is partitioning the tuples into a disk based partition. So, each partition is in separate position in the disk, once that is being read, an in memory hash is built for this partition only, each memory hash for H s i only.

So, all the tuples in H s i, there is an in memory hash function that is built and then, what happens is that for every record t r i which is in this hash partition H r i. So, this is called the probe, so this is probe to find out the corresponding matching tuples. So, this is a built, because an in memory hash is built and t r i is probed, so that is why there are some nomenclature that are used.

So, s is called the build input, the reason is s is used to build an in memory hash function, while r is called the probe input, the reason is tuples from r are used to probe this in memory hash function that is built for the other relation, so this we should remember. So, couple of things is that, each of the partition of the build relation must be in memory; otherwise, this whole procedure does not make any sense.

(Refer Slide Time: 04:00)



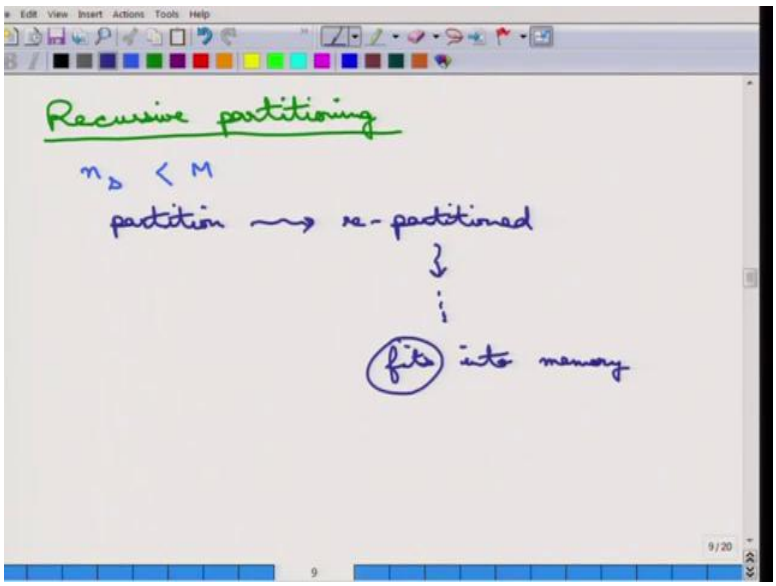
And therefore, the build relation should be smaller. So, the build relation should be the one that is the smaller one. So, if between r and s, s is smaller, s should be the build relation. Now, let us analyze the sizes of this, so suppose there are n partitions, now each partition must fit into memory, this is the one invariant that we require. Suppose, the number of blocks in a memory, this is the memory size in number of blocks.

And suppose the build relation has got b s blocks for the build relation. Now, what do we want is that, this n should be greater than equal to b s by m. Why is this required? Because, the total number of build relation is M and if each of them fits in M, that gives you the number that many partitions can be handled and n should be more than that. So; that means, if n is more than that if the actual number of partitions is more than that, then each partition the size of each partition will be less than what can be fit.

So, the other condition of course, is that M should be greater than n, because each partition must fit. So, everything inside the partition must be fitting in the memory and the number of memory sizes. So, from each partition a block must be read, so the one block from every partition can be read, so that is M should be greater than n. Now, combining these two what we can then get is that M is greater than equal to b s by M.

So, this is what the condition we were getting or we can write it as M should be greater than or equal to square root of b s. Now, this is one thing that must happen. So, the memory size must be greater than the square root of the size of the build relation.

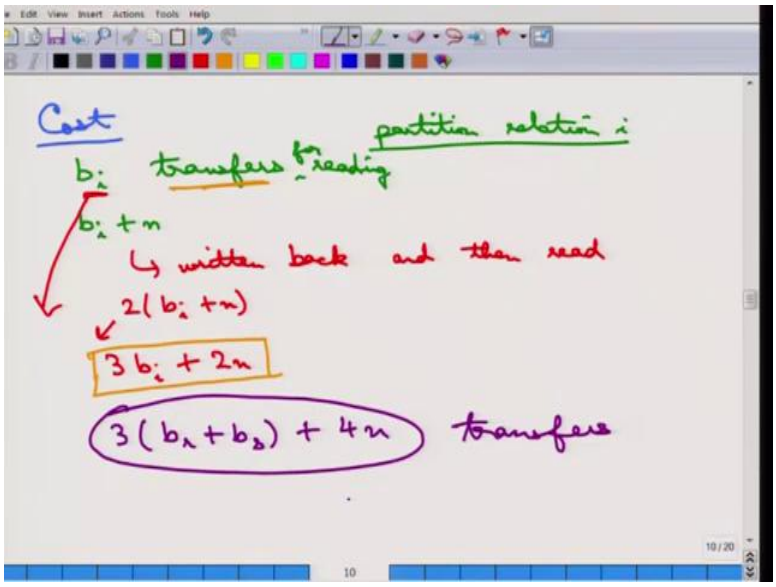
(Refer Slide Time: 06:16)



Now, if this happens then a single partition is good enough. If this; however, does not happen, then the strategy called recursive partitioning needs to be employed. The recursive partitioning, the intuition of the recursive partitioning is the same as the external mod sort that we saw is that, if things cannot be done in the one pass, then whatever can be done in the one pass is done, then things are delegated to the next pass.

So, the same thing if this is not done, so initially what will happen is n s is chosen such that this is less than or equal to M and then, each partition or partitions are built first. So, partitioning is done at the first level and then each of this portioning is then again, but now each of these partitions cannot fit into memory. So, each of this partition is then again repartitioned and this goes on till finally, the partitions can fit into memory. So, this goes on till it fits into memory that is what the recursive partitioning does. So, this is the important thing, this must fit into memory and there is a hybrid hash join which just retains the first partition all this, so that is fine.

(Refer Slide Time: 07:23)

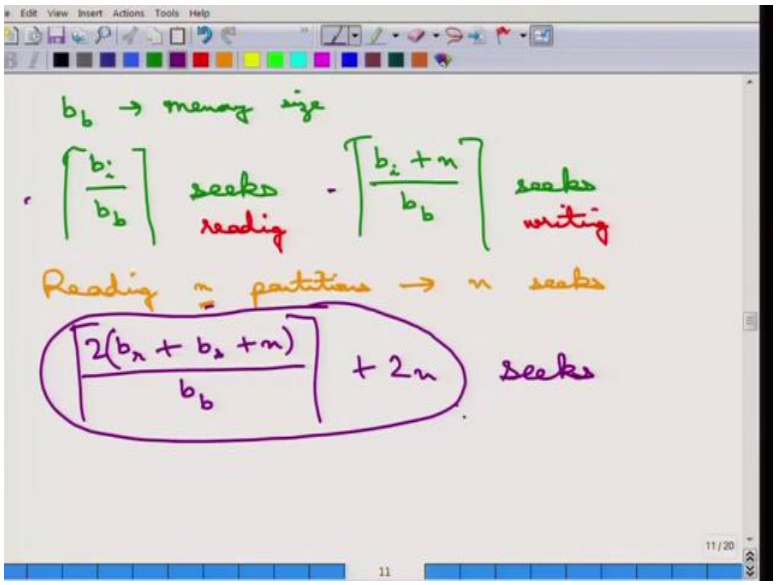


Now, the important thing is to analyze the cost of this hash join strategy or it may be recursive partitioning etcetera, etcetera. So, relation r contains b i blocks, so initially it requires b i transfers to read relation r. Now, the total number of blocks after the partitioning, so this is trying to partition relation i, so this is what the operation that we are doing, so b i transfers read for reading. Now, this may, what may happen is that after the partitioning is done, it may produce b i plus n blocks, because each of this is just one

extra tuple is there, so it fits does not fit into the block etcetera. So, there can be b i plus n.

So, what may happen is that, so this needs to be written back and then needs to be read, written back and then read again, because each partition is going to be read. So, this uses 2 b i plus n and then there is this b i, so if we add these two numbers are for each partition this is 3 b i plus 2 n that is the number of transfers, this we are only worried about transfers now, so this is what each relation i. So, the total number of transfers therefore, is 3 b r plus b s plus 4 n, this is the total number of transfer that is required for the initial phase or just the required for the just the initial phase.

(Refer Slide Time: 09:23)



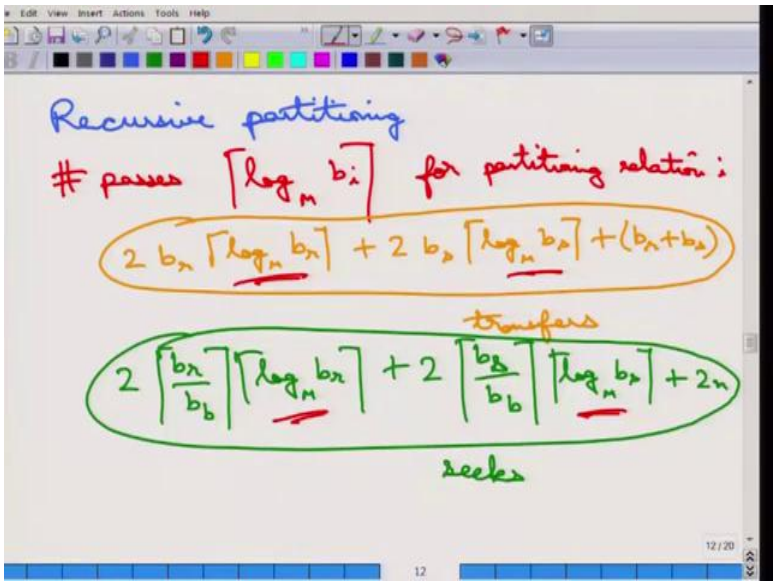
So, what are the number of seeks if we go ahead using the same thing? The number of seeks for reading this, making the partition is that again suppose b b blocks fit into memory the memories, so b b blocks. So, memories are in the b b blocks, b b blocks from each relation fits into memory. So, then there are b i by b b seeks that are required to read all of them, then similarly b i. So, this is for reading and then b i plus n by b b seeks for writing plus that same kind of thing. So, then reading the n partitions require n seeks, because each partition is in a separate thing, reading n partitions that requires n seeks.

So, the total number of seeks that is required is your b r plus b s plus n, I mean of course, twice of that by b b. This is that these two terms plus of course, this answer plus 2 n, this

is the total number of seeks that is required for doing this hash join. Now, this is of course, assuming that there is no recursive partitioning, because once you read and then reading this n partitions is good enough to do the hash join. One important thing also that you should must highlight here is that the output cost is not taken into care of.

So, it will require some cost to flush the output to the disk to write down the output to the disk, but that is not taken care of. Generally, output is just assumed to be given back to the user in the console or some other manner.

(Refer Slide Time: 11:19)



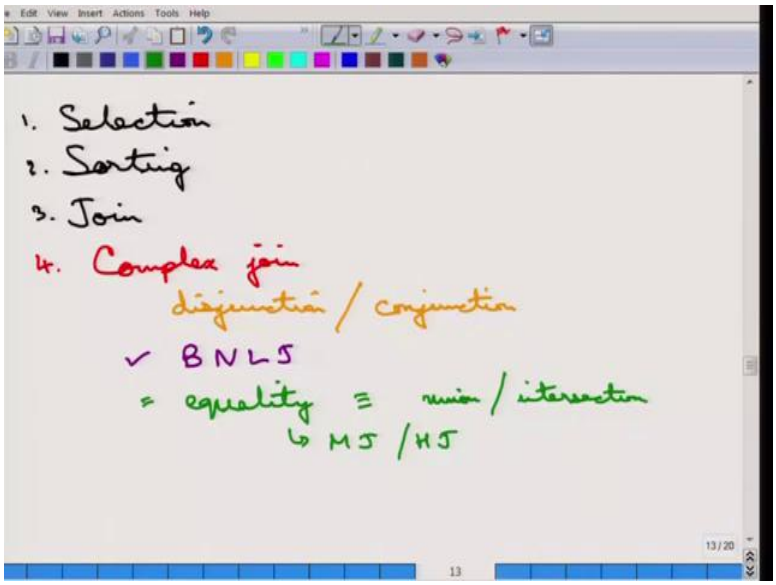
Coming back to the cost of hash join, this the previous case was assuming there is no recursive partitioning. If; however, recursive partitioning is required, then the situation is a little bit different and we can analyze it in the following manner. So, the number of passes, the first thing to enter is the number of passes that is required, that is the first important thing is log m times b i. Because, that is the m, m is the memory size, so that many passes are required.

Now, for simplification if we ignore half fill partitions, so total number of transfers therefore, is this has to be multiplied, because in every of this pass the entire relation is read, etcetera. So, this is of course, and there is a similar term for s and plus your b r plus b s this is for finally, reading the two partitions all the things. So, there is the total number of this is for transfers, this is the total number of transfers that is required and the total number of seeks that is required will be instead of b r this will be b r by b b that

many seeks are required and then it is the same formula otherwise. So, plus 2 b r, once more I want to highlight that I am ignoring the half fill partition.

So, I am assuming there are no half fill partitions, so all partitions are nicely behaving, this of course, may not happen. So, plus n, etcetera part of these things needs to be added, so this plus 2 n, so that many seeks is required, so this is for recursive partitioning, because this is the number of passes that is required for relation. So, for partitioning relation i, but it cannot be done in one pass, that is the whole point. So, there is an extra cost that goes into each of these factors. If this is the number of passes that cost, for each of these passes everything is read and written, so that is the idea of a recursive partitioning.

(Refer Slide Time: 13:40)

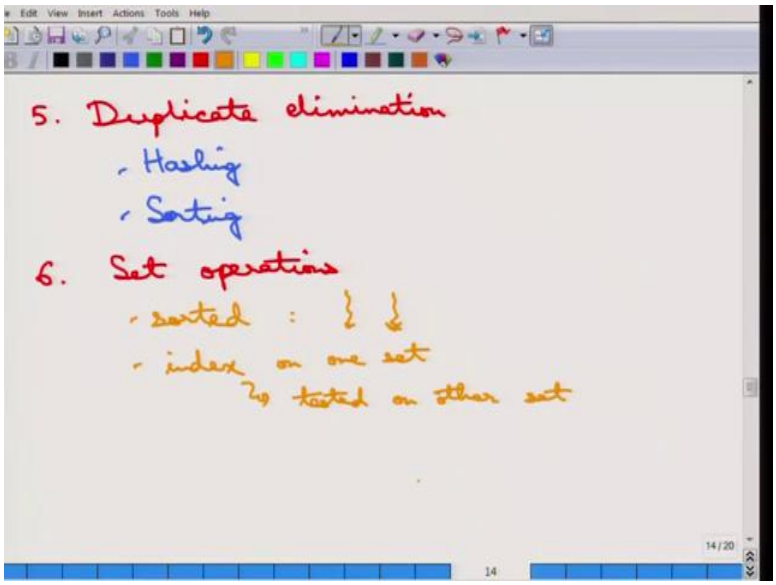


So, let us take a stock of what we have done so far. So, we have handled selection, we have handled sorting and we have handled join. So, these are the three main operations that we have handled so far, there are some other kind of operations. First operation is the complex join, so it is… What is a complex join? I mean it is not... So, the join condition is a disjunction or conjunction, etcetera. So, disjunction or a conjunction, it is not a very simple one attribute to another attribute join. So, there is a little more complicated predicate, predicate is a little more complex.

Now, of course, the block nested loop join can be applied, because the block nested loop join does not depend on what the predicate is, it just takes up two tuples and applies the

join condition. So, this can be always done if the disjunction and the conjunction is only based on equality, then this hash join or merge join may be done depending on the equality and then those indexes, union and intersection needs to be done. So, depending on whether it is disjunction or conjunction, union or intersection can be done on top of your merge join or hash join, so that can be handled.

(Refer Slide Time: 15:05)



So, this is for the complex join, there is one another important operation that the database needs to do sometimes it is called duplicate elimination. If you remember SQL by default is a multi set whereas, relational algebra is set and there is a way of asking the SQL to produce a set by specifying a keyword distinct and when that is being done, what it essentially tries to do is it finds out the duplicates and eliminates them.

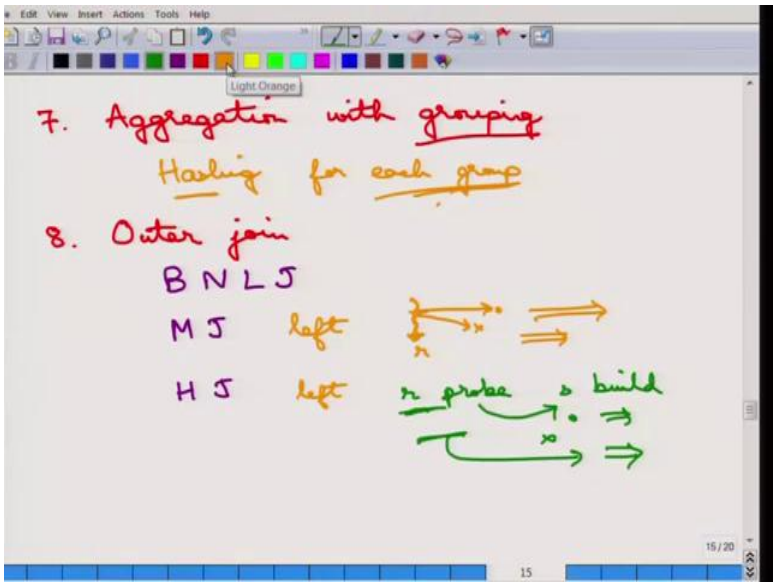
So, how is this duplicate elimination done, well the first thing that comes to mind is hashing of course, hashing can be done. So, all the tuples can be hashed to find out and then whenever the two tuples has to the same location they can be checked attribute by attribute to say if they are done. The other, a little non standard way of doing it is sort it and then go over the sorted manner. So, each tuple is compared with only the next tuple, so these are the two ways of doing the duplicate elimination.

So, the analysis of duplicate elimination is the same as doing the hashing or the sorting. So, that is the same thing, then there are certain set operations that the relational algebra and SQL needs to handle. So, what are the different set operations that can be done,

union intersection difference all those things. Now, if the relations are sorted already then this is easier, because then it just needs to be scanned the two relations or the two sets etcetera, just needs to be scanned in order and this can be applied.

The other way of doing it is that and index can be built on one of the sets, index on one set and the other can be then tested on the other side. So, you see that this set operations etcetera is kind of like the join and the idea is instead of two relations there are two sets, but then the intersection union operations are being done in that almost the same philosophy using the same philosophy, so all of those things can be done.

(Refer Slide Time: 16:59)



The next important operation is aggregation. Now, aggregation can be done with or without grouping. So, if it is without grouping then it is easier, because then you just need to go over all the tuples and just find ((Refer Time: 17:16)) function. So, what is more interesting is aggregation with grouping. So, the important part is how to do the grouping then the of course, the first and the most basic idea is to use the hashing for each group and then for each group essentially becomes an entity by it becomes a relation by itself.

I mean it can be treated like a relation by itself and then it can be aggregated. So, that is the aggregation with grouping the hashing is required to partition the relation into this group, a very important that we have left out is the outer join. So, whatever we have been talking about so far is on the inner join. So, we have been assuming that the operations

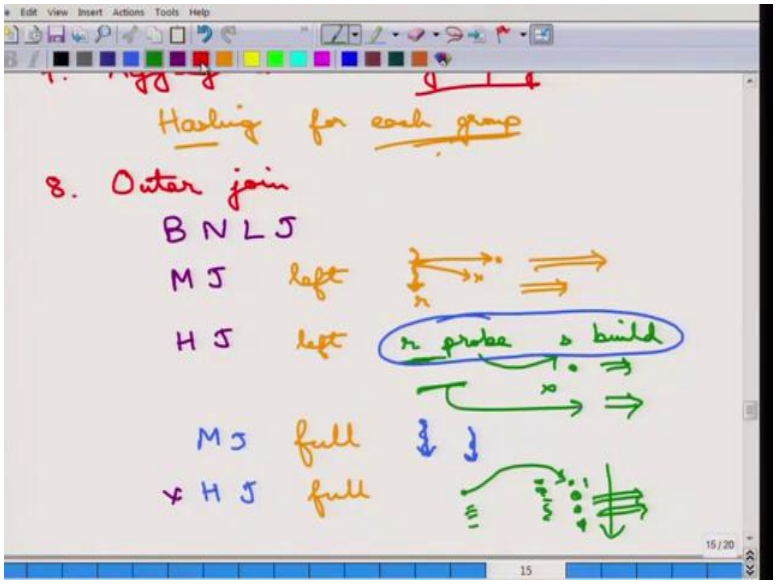
are inner join, it is now outer join to recall even if your t r and t s do not agree on the equality, suppose it is a left outer join then the t r must be output as well, so that is the whole use of outer join.

So, how to handle that? So, once more the block nested loop join requires no modification, because this is a most generic algorithm and if the t r and t s match it is output it; otherwise, it is padded with suitable null values etcetera can be done. For the merge join what can be done is that suppose it is a left outer join then when r is being scanned even if... So, if there is a matching tuple in s fine this is output, if there is no matching tuple even then this is output. So, when r is being scanned it is made sure that everything on the r side is being output.

So, the merge join can also be handled, the question then is if there is a left join, suppose it is a left join the question is which one should be the build relation and which one should be the probe relation for the hash join. So, suppose it is a hash join, so hash join can work in the same manner, suppose it is the left thing and one of this r can be used as probe relation or the s can be used as a probe relation.

So, the question is which one should be used as the probe relation, if it is a left outer join then r should be the one which should be used as the probe relation. The reason is s can be used for build and every tuple from r is probed if it is matches then that is output, even if it is does not match then the r side is fine and the r side is simply output, this is all fine for the left outer join and similarly the strategy can be used for right outer join. However, what happens when the operation is a full outer join.

(Refer Slide Time: 20:02)



So, for full outer join of course, the merge join can be used easily. Because, everything is output for the hash join it is a little bit more complicated then you see that this probe and build relation is not completely useful, because suppose there is an s that is kept on. How do we know that s has been output or not. So, generally h the hash join algorithm cannot be is not used for the full join; however, what can be done is the following is that for every relation in s for every tuple in s there is a marker.

So, if s is probed one of the tuples in r probes it then this is marked as 1 and by default it is otherwise 0. At the end of the whole output at pass is taken over this s an everything that is 0 is output. Why is this? Because, if this is 0 meaning nobody from r has caught it and r is suppose the left outer join it is done in the left outer join manner. So, everything from r is actually handled, so but this s needs to be also handled, so this is outputting.

But, you see that this is a little bit more clumsy and not very efficient algorithm. So, this is generally avoided and simply a merge join or a block nested loop join is actually the one that is preferred by all of these things, because it is the most generic and if the outer join contains conjunction, disjunction etcetera then you can see how complicated the algorithm can become. So, this finishes the module on query processing and we have seen how the different query algorithms can be handled etcetera, next we will start on the query optimization.