Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 15

Query Processing and optimization – II

Hello and welcome to yet another session in database management systems. In the previous session we started looking at a very crucial aspect of dbms design. Note that I am using the term dbms design and not database design. We will look at, we will address this issue in more detail a little later, what is dbms design verses what is database design. We started by looking into a very crucial aspect of dbms design namely that of query processing. We saw that nowadays or in today's world, the size of the database is no longer a problem.

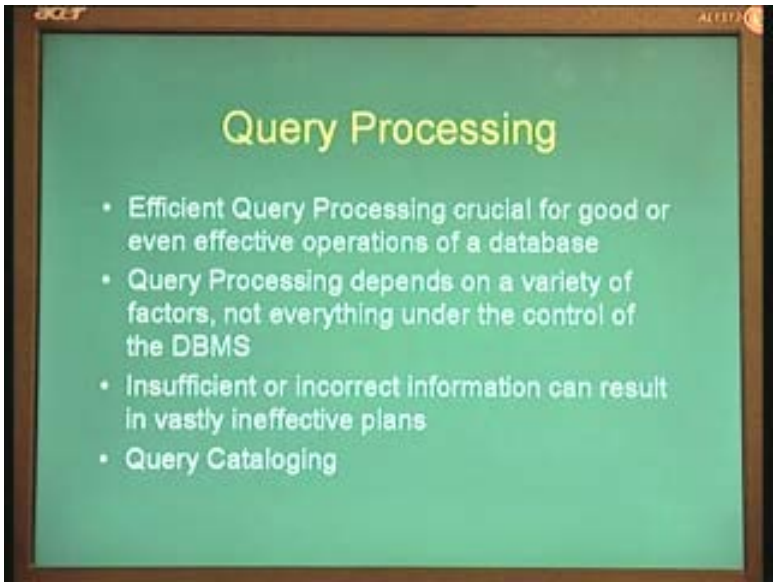
We can have huge amounts of data stored in very small amounts of space and being available at pretty cheap cost. In fact I have with me a small device which I can show it for you. This is the small device which can hold a database of size 256 megabytes of store and this cost roughly about 3000 rupees. So you can store data of the order which was not even envisaged before in very small devices at pretty affordable cost and which occupies very small amount of space. Therefore storage of data is no longer a problem, the problem today is retrieval of data.

Retrieval in the sense it is not just retrieving any data, it is retrieving whatever data that is required by the user. Therefore we saw the crucial elements today or the crucial technologies of that is going to impact database in the forth coming years are those techniques that can help in retrieving data elements as quickly as possible from databases. We saw that auxiliary files in the physical storage world which comprises of index structures and several kinds of hash structures and so on is one crucial element in making this happen, that is making fast data retrieval happen. The second crucial element is query processing, given a users query can we process the query in such a way as to return the results extremely fast.

There are several techniques that are used for making query processing interactive in nature that is remember the example of google. If suppose google were to say that when you give a web search query, suppose it were to say come back after two days for your answer it becomes unusable, nobody would use it if the other alternative that is but a search engine like google can search peta bytes of data or the search space of data is peta bytes of data and it can return you relevant results within a few seconds. Now this is possible, one of the techniques for this is to use very intelligent query processing techniques. Of course there are, if you throw more hardware into your database, if you throw faster processors it becomes faster and so on. However a crucial element is still the algorithm that is used to retrieve data. A bad algorithm can make the difference between a database that is very efficient to a database that is unusable for all practical purposes.

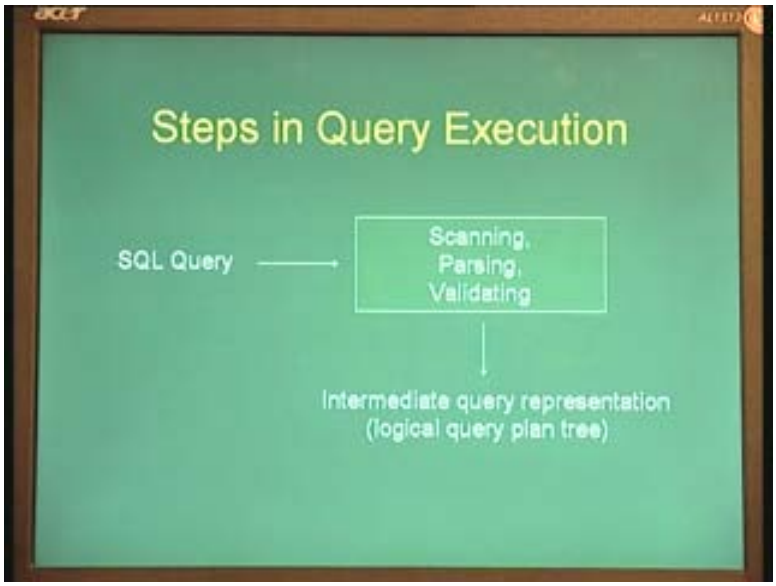
Let us briefly review what we learnt in the previous session in query processing and move on further today to look at some more aspects of query processing. Efficient query processing is crucial for good dbms design and in fact it can make the difference between a database that is operational or effective and database that is ineffective.

(Refer Slide Time: 05:14)



A bad query processor can render a database all but useless when the size of the database grows beyond a certain size, beyond a certain limit. Query processing depends on a variety of factors unfortunately and not everything is under the control of the dbms. We saw yesterday that query processing could depend on let us say the memory size or the size of the tuples or the size of the relations that are present in the database and the kind of file system that are stored and whether the file system is fragmented or defragmented and so on. So there are there are variety of factors that impact query performance and not all of these factors are within the preview or within the control of the dbms. Therefore query processing is oriented towards obtaining the best out of whatever is available at the biggest or at the control of the dbms.

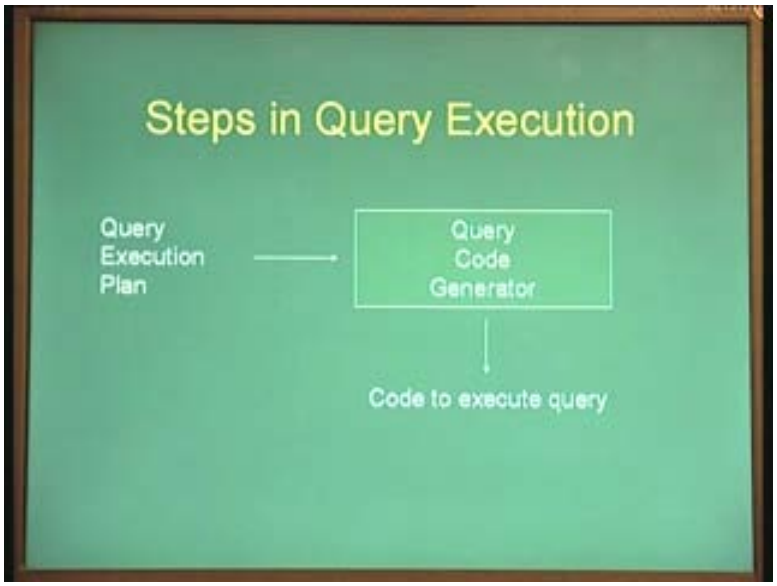
(Refer Slide Time: 06:20)



What are the typical steps that are handled or that are executed in a query execution process. Given a query, given an sql query, the query is fist scanned using a lexical analyzer which in turn takes the string which forms a query and returns a set of tokens and then these token stream is then passed to build a syntactic tree or what is called as a parse tree. And then syntax analysis and so on semantics checks and etc are all performed which comprises the validating phase of the query complier and once this is done an intermediate query representation is generated. This intermediate query representation is also termed as a logical query plan tree.

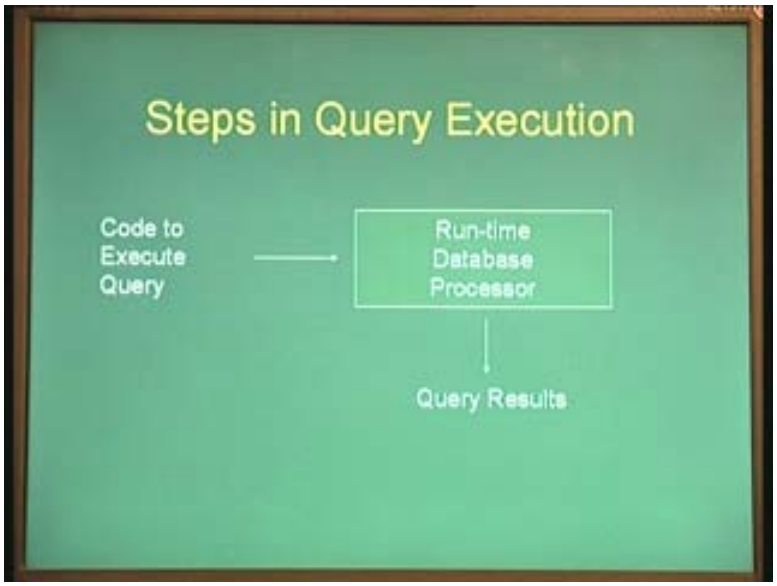
Now this logical query plan tree is usually a tree data structure that represents the relational algebra equivalent of the sql query. Using this tree data structure, there are several rules that are employed in order to systematically optimize the tree for better performance. The intermediate query representation or the query tree is then given to the query optimizer which generates a query execution plan. The execution plan is also called the physical query plan tree and in contrast to the logical query plan tree that means a physical query plan tree comprises of a query plan written in a language that is usually executed by a dbms interpreter or would be complied into machine code for execution. This physical query plan language comprises of its own constructs some of which we saw in the previous sessions something like table scan constructs and sort scan constructs and iterators and so on, in addition to all relational algebra constructs like select, project, set union, difference and so on.

(Refer Slide Time: 08:30)



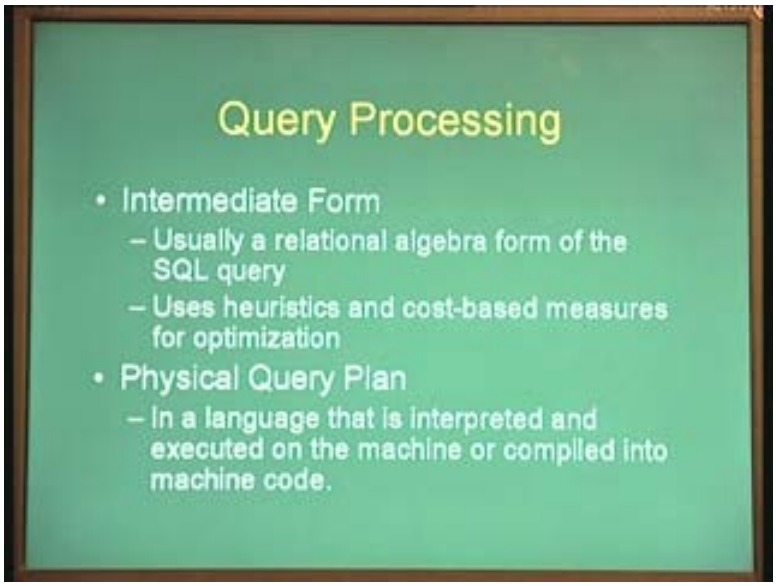
The query execution plan if it is an interpreted dbms would be executed directly and the results would be returned to the user. If it is a complied dbms engine, the query execution plan is then given to the query code generator and then appropriate machine code is generated from the execution plan.

(Refer Slide Time: 08:53)



Of course the last step would be the code if it is generated in a complied mode, it is then given to the operating system run time or the database runtime and then query results are given.

(Refer Slide Time: 09:10)



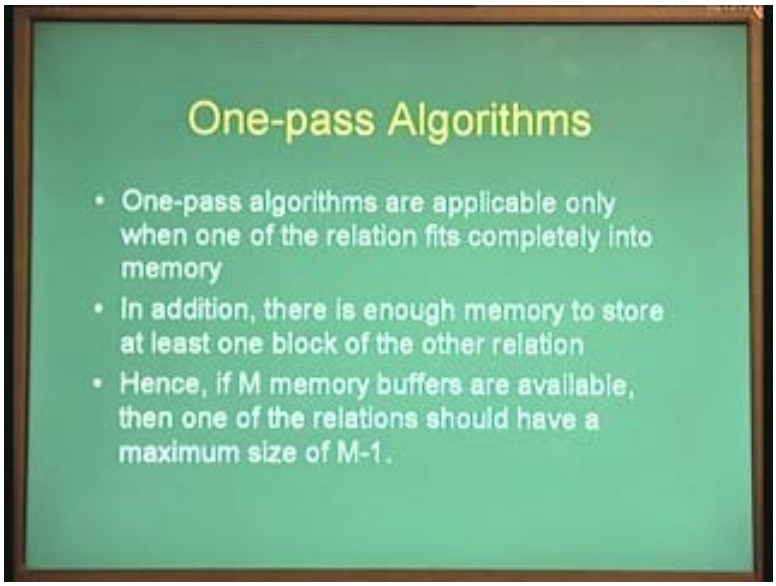
Now we are primarily interested when we are taking about query processing in two of these stages in the query execution stages. The first stage that we are interested in is the intermediate form of query representation. In fact that is something that we have deferred to a later section. We in fact went straight onto the second stage that is a physical query plan representation. We saw some constructs of the language that is used to describe the physical query execution plan and constructs that are or the physical execution plan is then either fed through an interpreter for this language which executes the query or is compiled onto machine code.

Now what are some of the operators that we saw yesterday? The physical query execution plan operators are those set of operators that define the basic physical operations that need to be performed in order to answer a query. These are also sometimes called the internal query operators or the operators that make up the internal query language. Sql or relational algebra on other hand is an external query language that is it is the query language used by the users or application programs that are using the database. But the database itself or the dbms itself uses an internal language for answering queries that accesses the file system or the storage structure used by the dbms in order to retrieve tuples in response to a query.

This of course comprises of the set of all relational operators plus some additional operators that talk about physical characteristics of retrieval. Some example operators that we saw yesterday was the table scan operator which can be implemented using an iterator operator that is an iterator objects so to say which opens a table and returns a tuple by tuple that is an iterator if you remember has three different functions an Open function, a GetNext function and a Close function. A table scan operator opens a table when the Open function is called and returns the next or the top most tuple on every getindex or getnext invocation and starts incrementing a pointer so that the next tuple is returned and so on.

So the i th GetNext invocation would return us the i th tuple in the table and then the close function would close the table. And then there was index scan that would use an index to get a particular or get the required tuple. Then there is a sort scan operator which not only returns all tuples in a table but also sorts them before returning them. We then saw some set of one parse algorithms that are built on top of these physical query plan constructs or data structures that perform various operations.

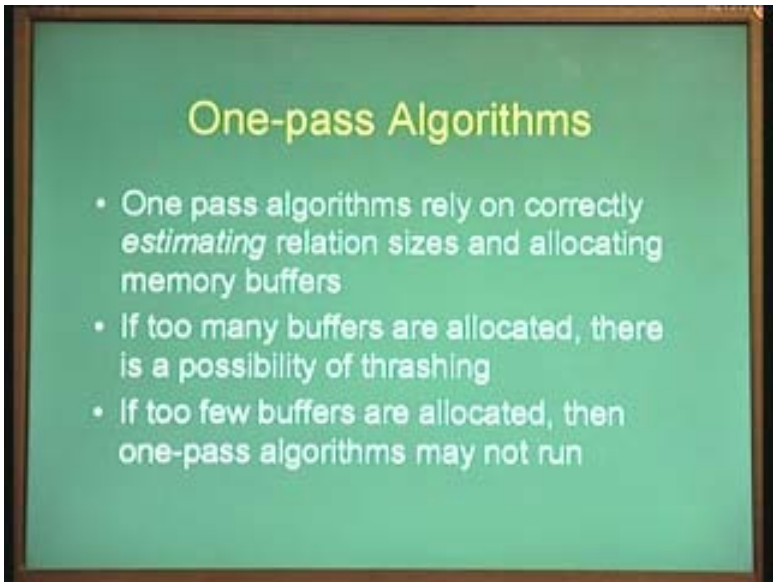
(Refer Slide Time: 12:22)



We divided one pass algorithms into three different kinds of categories. The first category was tuple at a time algorithm. A tuple at a time algorithm is something like select or project which needs to concern or which needs to be concerned only with one tuple at any given point in time. Tuple at a time algorithms can easily be executed with a single parse regardless of the size of the relation because we assume that the tuple is never large enough so that it cannot fit into main memory. We always assume that one tuple of the database can always fit into main memories.

Therefore since we are only concerned with single tuple at a time, we can always use a single parse algorithm for tuple at a time functions. On the other hand there are relations at a time algorithms that is there are functions that need to be concerned with the entire relation rather than each tuples independently. For example the function called unique in sql needs to look at the entire relation in its entirety of course rather than each individual tuple in isolation. Relation at a time functions can use one parse algorithms only if the relation can fit completely into memory. And that's not enough, in fact the relation should not only fit completely into memory but especially for binary relation at a time functions, there should be at least one block left over to read data from the other relation. Hence if a relation requires M memory buffers or rather if M memory buffers are present for the dbms then the maximum size of one of the relations of the smaller relations can be at most M minus 1. It can't be M, because we need at least one more block for data from the other relations.

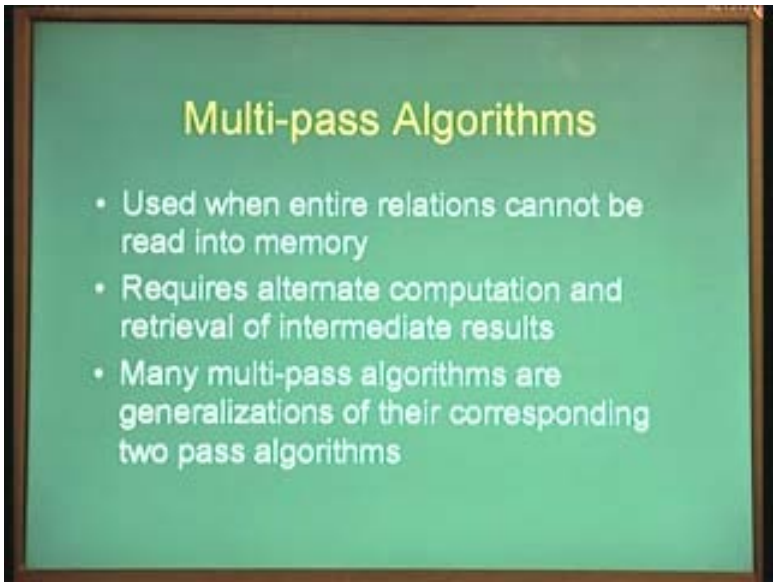
(Refer Slide Time: 14:36)



And of course that is the limitation of the one parse algorithm in the sense that you can use them only when you know that at least one of the relations can fit completely into memory. Now how do you know that a relation completely fits into memory? What if you don't know the size of the relation? In such cases you need to estimate the size of whatever data relation or whatever table that that you are using. Hence one parse algorithms rely to a very large extent on procuring good estimates of relation sizes in terms of the number of tuples. And if the estimation algorithm is wrong that is if too many buffers are allocated that is if the estimation is too high then there is a possibility of thrashing.

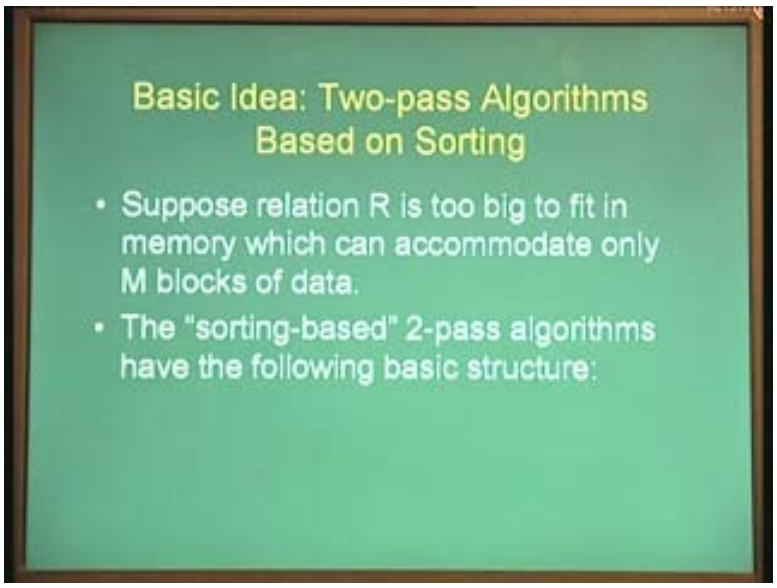
On the other hand if the estimate is too low then we may not be able to use one parse algorithms at all because the relation won't fit into memory and the strategies that we studied for one pass algorithms will no longer be applicable. What do we do in such cases that is, what do we do in cases where one or both of the relations or too big to fit in memory by itself. For these we use what are called as multi pass algorithms.

(Refer Slide Time: 16:03)



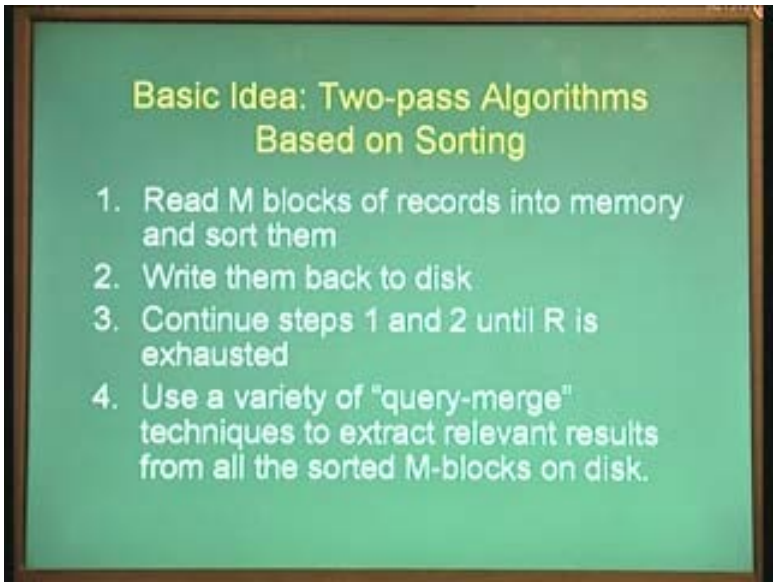
In multi pass algorithms we shall be studying today mainly about two pass algorithms. Generally many multi pass algorithms are generalizations over two pass algorithms and if you know the general strategy that is used behind two pass algorithms, it is sufficient or it would not be too difficult to generalize it to multi pass algorithms. And multi pass algorithms as the slide shows are used when relations cannot be read into memory in their entirety. Only a part of each relation can be read into memory and they usually have an alternation between reading part of a relation into memory and writing it back onto disk. That is an alternation between computation and intermediate result generation and retrieval of intermediate results.

(Refer Slide Time: 16:57)



The first kind of two pass algorithms that we are going to be looking at are what are called as sorting based two pass algorithms. We are going to be looking at three different paradigms or three different strategies of two pass algorithms, sorting, hashing and indexed based two parse algorithms. The first one that we are going to be looking at is the simplest which is called the sorting based two pass algorithms. The basic idea behind sorting based two pass algorithms is shown in the slide here. Suppose we are given a relation R. Of course that is, such that the relation R is too big to fit in memory and of course for the dbms we are given a maximum of M blocks of memory. That is M blocks of memory elements are allocated for the dbms, therefore M blocks of data can be read from the relation R by the dbms at any point in time. The sorting based two pass algorithms have the following basic structure.

(Refer Slide Time: 18:05)



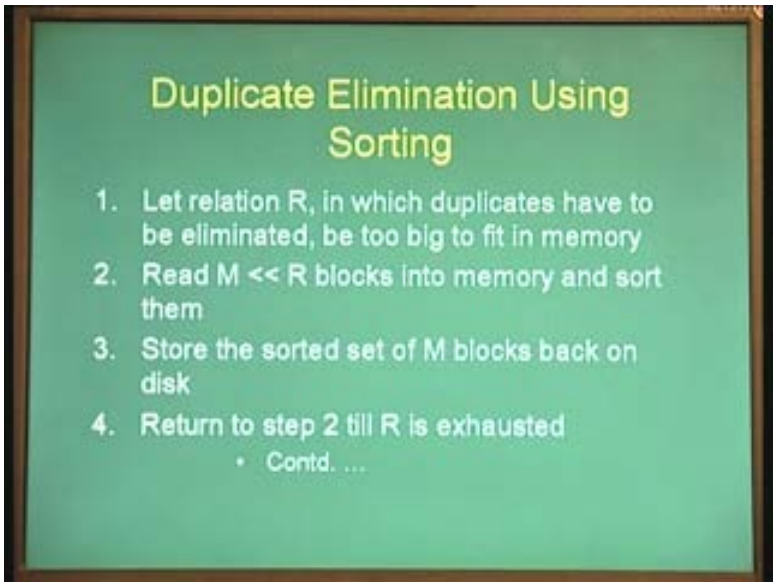
This slide show the basic or the skeletal algorithms for a two pass sorting algorithms. Essentially as we noted before, the algorithm or any multi pass algorithm alternates between reading intermediate results and writing them back. That is reading part of data from the disk and performing computations and writing them back. So the basic idea behind two pass algorithms are based on sorting is as follows. First start by reading the relations one or two relations that is we are going to concerned only with either unary or binary operators for the time being. Of course any nary operators can usually be reduced into one of these forms and it's a generalization of considering either unary or binary operators.

So let us consider that the relation or the pair of relations that we are going to be using are first read block by block. So because we have M blocks that are allocated to us, we can read these relations M blocks at a time. Now M blocks of relations from M blocks of tuples from the relation or pair of relations are read into memory and then they are sorted. Once they are sorted they are returned back into disk.

This is the alternating phase that is you read part of data from disk sort them, in this case which in this case is the computation and then write them back to disk. Now continue steps 1 and 2 until the relation is exhausted. I am just assuming here that we are dealing with unary operator that is until relation R is exhausted and it can of course mean until the pair of relation is exhausted or until the set of all relations that this query handles are exhausted. And then once these intermediate results are written on to disk that is sets of different M blocks of sorted tuples are written onto disk, use a variety of query merge techniques. Remember what is a merge technique, if we have taken a course on let us say data structures you would have come across an algorithm for merging which is usually seen in relation with sorting.

A merging basically means that given two or more lists that are sorted, can I obtain a single list that is also sorted of course as efficient fashion as possible. There exist very efficient algorithms for merging especially when the results or especially when the lists are sorted that can produce results in a linear order of time. So use a variety of these query merge techniques. Note here that in the two parse algorithms, it is no longer just merge techniques it is query merge technique. That is as and when you do the merging, perform your query or answer your query as and when you are doing the merging on this intermediate results. So use a variety of query merge techniques to extract relevant results from all the sorted M blocks of disk.

(Refer Slide Time: 21:51)

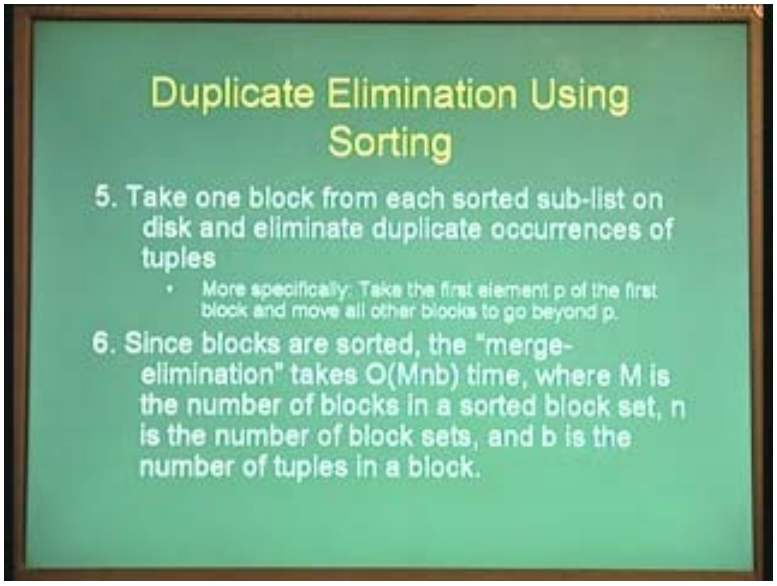


Now let us see some examples of two phase or two pass sortings, two pass sorting based methods for answering different kinds of queries. You can notice that all of these examples follow the same skeleton of the basic idea that we presented in the previous slide. So let us take first the example of duplicate elimination. Now as you can see duplicate elimination is or relation at a time unary operator. We don't have to worry about tuple at a time operators for two pass algorithms.

Why? because all tuple at a time operators can be answered using a one pass algorithm because all that we need to be concerned about at any point in time is just a tuple and each tuple can be checked in isolation with the other tuples. So let us take the first relation at a time unary operator namely duplicate elimination that is the unique construct, how do we implement the sql unique construct. Let us assume that it is relation R on which we have to eliminate duplicates. And of course we have to also assume that relation R is too big to fit in memory. Now as we know that because we are given M blocks of memory available to us, start reading R in terms of M blocks of data that is read M blocks of data into memory and sort them and the third step is store the sorted set of M blocks back onto disk.

Now continue from step two that is read the next M blocks of data from R, sort them and put them back on to disk in in a separate file name. So keep doing this until R is exhausted, R becomes empty. Then what is the next phase now? We have read R relation, sorted them and put them back into memory or rather put them back into disk. The next step now is to use one of these so called query merge technique on these intermediate results that we have generated. So what is a query here? The query here is the elimination of duplicates that is we do not want any duplicate tuples to appear in the relation, once it is output to the output buffer. So how do we eliminate duplicates from these set of sorted intermediate results? So the query merge technique for eliminating duplicates is quite simple.

(Refer Slide Time: 24:36)



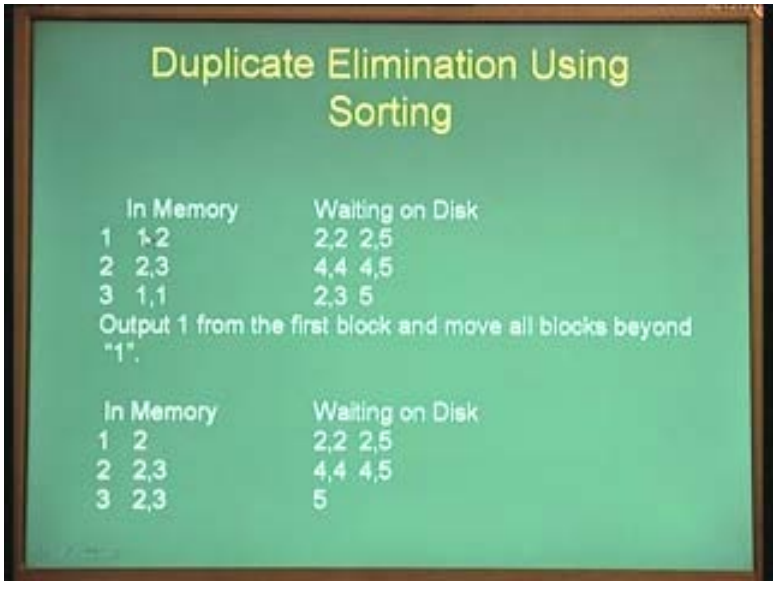
Take one block from each sorted sub list. There are several sub list of data that have been generated and each list of data is a maximum of M blocks in size that is because we have read data in terms of M blocks, except the last set of data that we wrote every other data would or every other intermediate result would be of size M blocks. The last data intermediate element that we wrote would have a maximum size of M blocks, it could also be smaller than M blocks in size.

Now take the top most block from each set of these intermediate results that we have generated and for each tuple that is take the smallest tuple among this. For each tuple just go pass the repetitions of this tuple in the other blocks. Let me illustrate this with an example in the next slide which makes it much more clearer. It is as simple as taking one tuple and then moving or rolling forward all other blocks so that they move pass the present tuple that I have taken that is if they contain duplicate tuples. If they don't contain duplicate tuples, you don't have to move them. And of course once they are all moved, put the first tuple into the output buffer.

Now this is possible or this is possible to be done in an efficient fashion because the tuples are all sorted. That is if a tuple T appears at one particular stage then all tuples that are in some way greater than T should appear below them. Therefore I just need to search each block until I find a tuple that is greater than T, when I am trying to remove duplicates. Such a kind of merge elimination takes an order of M n b time where M is the number of blocks in a sorted block set that is one of the intermediate result block set that that we have stored and b is the number of tuples in a block, n is the number of such block sets.

If you remember merging algorithms, a merging algorithm between two list of size M and N would take an order of M plus N time because we have N different lists of blocks of M different blocks. That is each list comprising of M blocks of data and each block containing of B tuples of data. We just multiply all three of them together so that we get the total time that it takes for eliminating duplicates from. Let us look at duplicate elimination with an example that makes the process much more clearer.

(Refer Slide Time: 27:50)



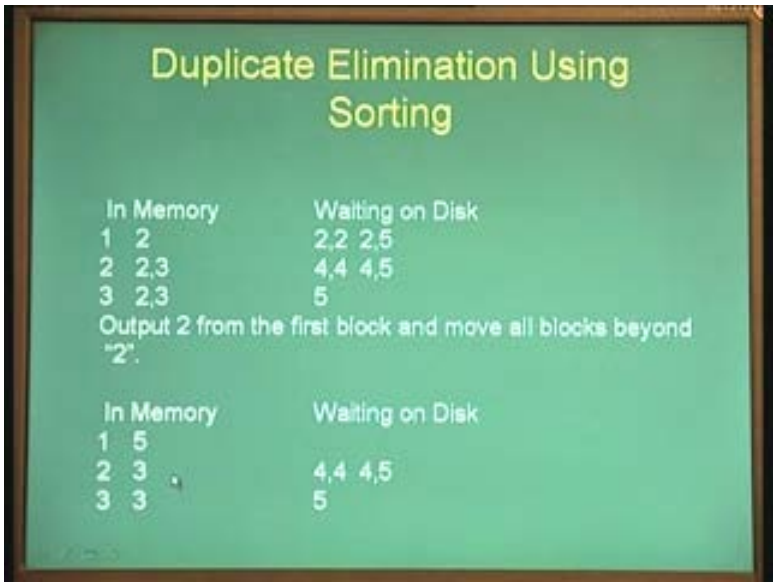
Now let us assume that as shown in the slide here, this is the first set of block set that is first set of M blocks of data that is written that is 1, 2 is in the first block, 2,2 is in the second block and 2, 5 is in the third block. Now this whole thing is a set of three blocks

that were written onto disk and as you can see here this entire thing is sorted, they are in sorted order. That is 1 is smaller than 2, is less than or equal to 2 and so on until 5. This is the second set of blocks that were written onto disk 2, 3 4, 4 4, 5 and so on. And this is the third set of blocks that were written onto disk. We now start by taking just the first blocks of each block set into memory and then considering just the first tuple in the first such block that was read.

Now the first tuple says that it is or reads as 1. When I read 1 here, all I have do to is eliminate duplicates is to eliminate 1 from all other block sets. Now if 1 has to appear in all the other set of blocks, they have to appear at the top, they cannot appear somewhere at the bottom, this is because it is sorted in order. Therefore one has to appear in the top. So all we need to do is start from the top of each block set and start moving forward until we find a tuple which has a value or whose key value is greater than 1. So therefore once we read 1 here, we look at the second block set and we read 2 to begin with and we know that 1 does not exist in this block set.

When we read the third block set we read a 1 and cancel it out, read the next tuple and this is also 1, we cancel it out and then go on to the next tuple which is a 2. So we know that we have exhausted all 1's that have occurred in this block set and we can stop this process here and output 1. So after 1 is output, the set of blocks becomes like this that is 2 is in at the front, all the 1's have gone and the block sets have been left shifted so to say in an appropriate fashion. Now the next tuple that needs to go out is 2. So you see that the first tuple here reads 2 and you have to eliminate all 2's from everywhere else. That is return this 2 to the output buffer and start moving forward until you have eliminated all 2's. So therefore you cancel the next three 2's here and end up at 5. In the second block you cancel the first 2 and end up at 3 and the same thing in the third block set as well that is you cancel the first 2 and end up at 3.

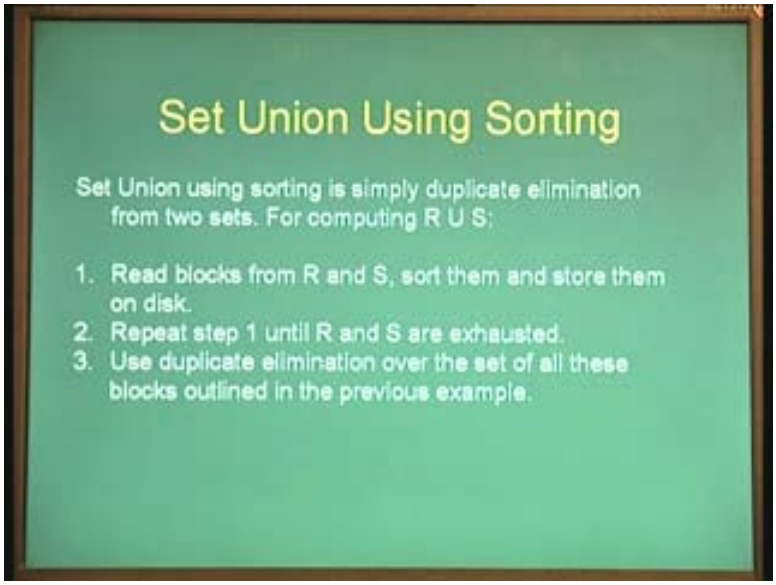
(Refer Slide Time: 31:06)



This is depicted in the second slide that is after the second tuple is put into the output buffer that is after 2 is put into the output buffer, our sets of blocks looks as it is shown here. That is all the 2's have been eliminated here and 5 has come to the top and all 3's that is all 2's are eliminated here and 3 has to come to the top and all 2's are eliminated here and 3 has come to the top here.

Now if I start to read 5 here then there is a problem because I have not read 3 as yet; 3 and 4 still exist before 5. So I cannot start rolling this block set until I go beyond 5. The simple answer to obviating this problem is to note that the first set of blocks from all the block sets are in memory. So we just choose the least element from the first set of blocks and then start rolling the blocks. Therefore we now choose 3 as the tuple to be output, to the output buffer and then start rolling each of the block beyond 3. Therefore 3 would be output from the second block set and we will roll it until we find 4 here and same thing three would be output here and we will roll it until we find 5. That is we roll the block set until we end up at 5 and so on. So in this way we can eliminate duplicates by taking the least element of the top most set of buffers of each block set and then rolling blocks until we move beyond the least element.

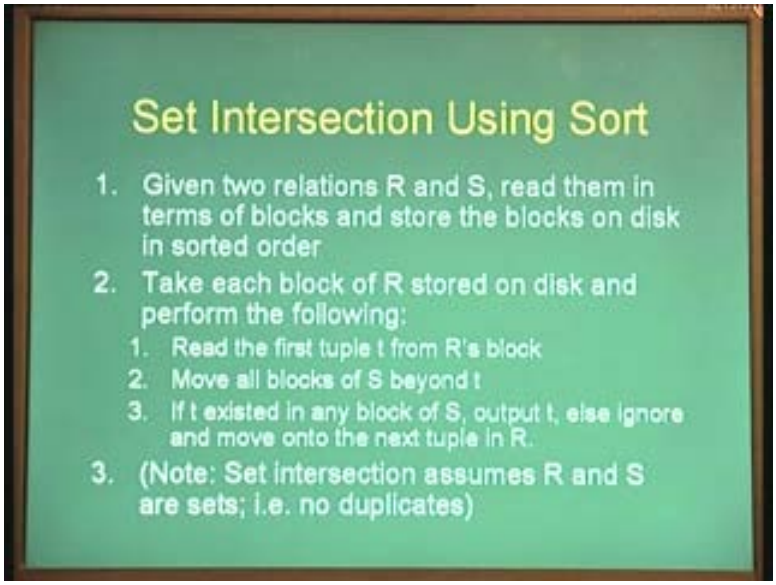
(Refer Slide Time: 33:00)



How do we perform set union using sorting? Remember these kinds of algorithms that we looked at yesterday for relation at a time operators. We looked at duplicate elimination, we looked at different kinds of set theoretic operations like union, intersection, set difference and so on. So set union is a binary operation. It is a relation at a time binary operator and because it is a set union that is I have emphasized the word called set, it means that no duplicates in the result. As you might have imagined the strategy for computing set union is very similar to duplicate elimination. When we have to compute the set union between two relations R and R and S, let us say R union S we just have to output all tuples from R and S without duplicates, as simple as that.

Therefore the set union algorithm is quite similar to that of the previous algorithm where you read blocks from R and S rather than just from single relation, sort them and store them onto disk. Now it does not matter for us whether any given tuple that we have read belongs to R or to S. We can just consider R and S to be a single relation and then start reading tuples from them, sort them and store them back to disk and use the duplicate elimination merging technique that we just looked into in the previous example for eliminating duplicates. So the output of this would be, it is clear to see that the output of this would be R union S.

(Refer Slide Time: 34:49)



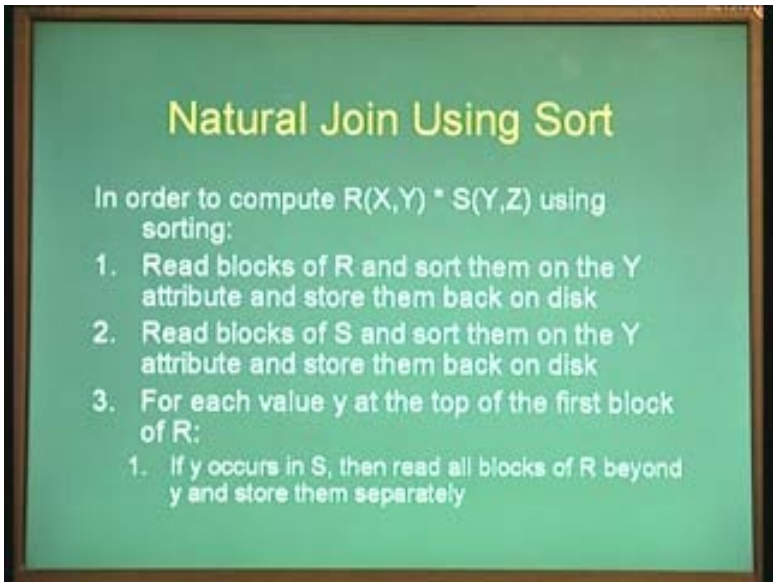
What about set intersection using sorting techniques? Set intersection between two relations R and S essentially has to return the set of tuples that are common between R and S. This set intersection is slightly different from the previous two algorithms. Why is this so? This is because here we need to be concerned about or we need to distinguish between tuples that belong to R and tuples that belong to S. In duplicate elimination as well as in union, all that we were concerned about is eliminating duplicates. That is whenever there are duplicate tuples, you just return it once and then cancel out all others. but here you have to return tuples, if and only if they appear in both relation R and relation S. therefore we need to have a mechanism of tagging each tuple as to where it belong, does it belong to the relation R or does it belong to relation S.

So the simple strategy for set intersection is shown in this slide here. Given two relations R and S, read them in terms of M blocks rather and store the blocks on disk in sorted order that is read relations R and S and sort them and store then back on disk. Now when reading it, ensure that the intermediate results of R are stored in a separate block set or in a separate buffer pool than the relations of S. So that we can easily distinguish between buffers that belong to R and buffers that belongs to S. And in order to compute the common elements between R and S, we simply do the following. We take each block of R that is from the block set and take the first block in each block set and move them into

memory and read the smallest tuple, the first tuple in this means the smallest tuple in the block set that we have just read into memory and try to roll S. That is try to roll relation S beyond this tuple that is beyond the first tuple.

If any that is if this tuple T existed in S then there should be at least one block which gets rolled that is which gets left shifted. If there is at least one left shift in S then it means that the tuple T is common between R and S, therefore it can be returned to the buffer pool or to the output pool. If no such left shift happens in S then the tuple T is not there in or is not present in S and therefore can be discarded. And note that here we are assuming that R and S are sets that is there are no duplicates in R and S for this algorithm. If there are duplicates then what we have to do is we first have to eliminate duplicates from R before shifting S. that is take the smallest tuple from R, roll all other block list in R until you eliminate duplicates of this tuple and then start looking for occurrences of this tuple in relation S.

(Refer Slide Time: 38:27)



The last algorithm that you would be looking at in the sorting strategy is the natural join function. The natural join as you know is an equijoin that operates on two relations, it is a binary operator and it is an equijoin on attributes having the same name and domain in the two relations. so assume that we have two relations R (X, Y) and S (Y, Z) where X and X, Y and Z attribute list where Y is the common component between R and S and assume that we are computing natural join between R and S.

Now a simple way of computing natural join is shown in this algorithm here. We first read blocks of R into memory that is we first read R in terms of M block and sort them on the Y attribute now. We cannot sort them entire tuple contents, we have sort them on the Y attribute. Because this is the Y attribute is the one that is common between R and S. Now sort them on the Y attribute and store them back on to the disk. And then secondly read relation S block by, that is M blocks by M blocks and again sort it on the Y attribute

and store them back on to disk. Now what is that we need to compute the natural join? Now for each relation or each tuple in R that is stored on to disk, we have to find all other tuples of S that are also stored onto disk, that can be combined with this relation. We however encounter a small problem here. What is a problem? There could be situations where a given tuple in R can be joined with every tuple in S. Now if that is the case, especially if we are talking about outer joins where we can tolerate null values for Y then we may have to take a tuple from R and join it with every tuple in S and we may not have, of course we do not have enough memory to store every tuple of S into memory and then start outputting it. So we have to do something else now. That is there could be worst case conditions where even combining the intermediate results cannot be done in memory. That is there isn't enough ram to perform the intermediate operations.

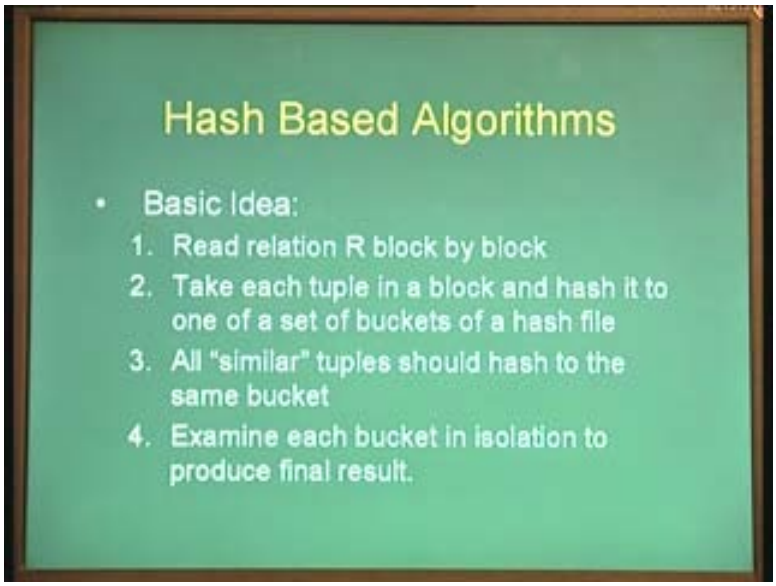
In order to tackle such issues what we do is we start with a second level of what may be termed as external merging. What we do is for each value of Y that appears at the top of the first blocks of R that is read the first blocks from each block set of R into memory, take the value of Y that appears in the first block of R and for each value of I that appears in the first block of R, note that here we cannot remove duplicates. We should have duplicates in the result unless of course it is specified that there should be no duplicates. So for each value of I that are at the top of the first block of R, start rolling S that is start reading the first blocks of S into memory and start rolling them until they go beyond the value of Y and take all these tuples and store them separately. That is let us say you take tuple T_1 from R and a set of tuples T_s from S and put all of them into another file.

Now once you have put all of them into another file, you just start combining the tuple that you have found in R with each tuple that you have found in S. So of course as you can imagine, this is a pretty, this can be a pretty slow operation because there are several amounts of disk accesses or disk writes that are happening here. The first set of disk writes are reading or the first set of disk write happens when we write the intermediate sorted relations back onto disk. The second set of disk accesses happen when we read each of the blocks, from the block sets of R into memory and then for each tuple we have to read each set of blocks in S and for all matching tuples we have to write them onto some other intermediate file where they can be combined.

Note that we cannot delete the tuples from S, after we have done this operation because there could be another tuple in R having the same value of Y which can also be joined with all these tuples in S. That is for example if we are joining let us say employee and department and let us say the join attribute is the department number that is for each department number attribute in the employee relation, find the department number attribute in the department relation and then join them together. Now there could be a case where two or more employees are working in a same department and have the same department number. So when we have found the first employee with some department number let us say D and assume that an employee relation is R and the department relation is S. Once we have found the first employee who works in a given department D in S, we take the tuple D is S and write them onto disks so that they can be combined.

However this tuple cannot be deleted from S because there could be another employee who is also working in the same department and whose tuple also has to be combined with the tuple of the department. So therefore natural join using sort is a slightly expensive technique.

(Refer Slide Time: 45:35)

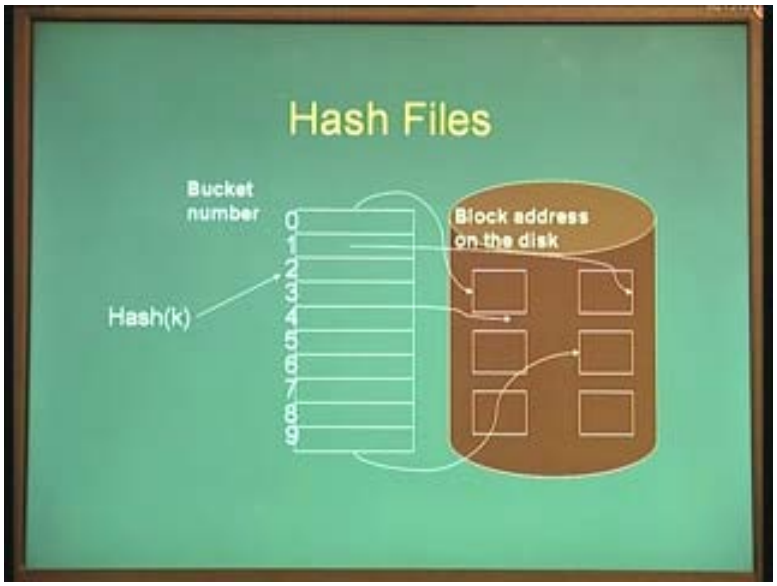


The next set of algorithms that we are going to see or the next paradigm of algorithms that we are going to see are the hash based algorithms. As the name suggests hash based algorithms use a hash table as their underlying data structure and use hashing as a overall paradigm or the overall strategy in which the algorithms are based. The basic idea behind hash based algorithms are shown in this slide here. We first, given a relation R that is too big to fit in memory of course. We read relation R block by block, we don't even have to read it M blocks at a time. If it is possible to read it M blocks at a time fine, it makes it even more faster.

Now for after we read relation R block by block, take each tuple in a block and hash it, move it through a hashing function so that it is hashed onto a bucket in a hash file. Remember what is a bucket in a hash file. A bucket is a set of buffers, is a chain of buffers that contain all tuples that are hashed in the same value for a given key. That is all tuples having the same key would be hashed onto the same bucket. There could be tuples having different key being hashed on to the same bucket and which could well be possible but what we can definitely say is that all tuples that have the same key will be hashed onto the same bucket.

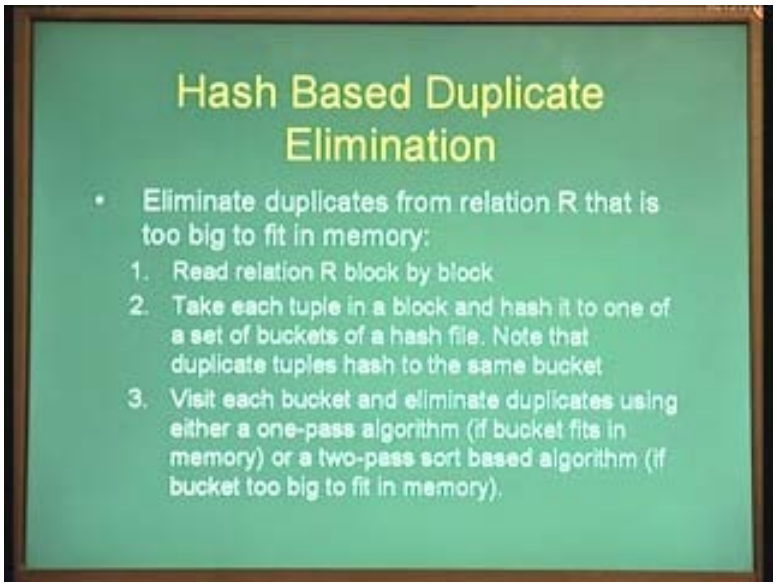
So all similar tuples should hash to the same bucket then we examine each bucket in isolation, we don't have to work across buckets anymore. In sort based techniques note that we had to work across the buffer list that were, sorted buffer list that were generated. In hash based technique we can examine each bucket in isolation of the other bucket in order to produce the final result.

(Refer Slide Time: 47:41)



So this slide is a reminder of what a hash file organization looks like. Given a tuple with a particular value key on which hashing is performed, the key then maps onto a set of buckets. This one shows static hashing and of course there are also dynamic hashing techniques. So the hashing function hashes a key onto a bucket number, the bucket number in turn points to a chain of blocks that form the bucket or where data or tuples that should lie in the bucket are stored. Let us look quickly at our usual algorithms for duplicate elimination and set theoretic operations like union, intersection and so on using hash based techniques, how can they be performed using hash based techniques.

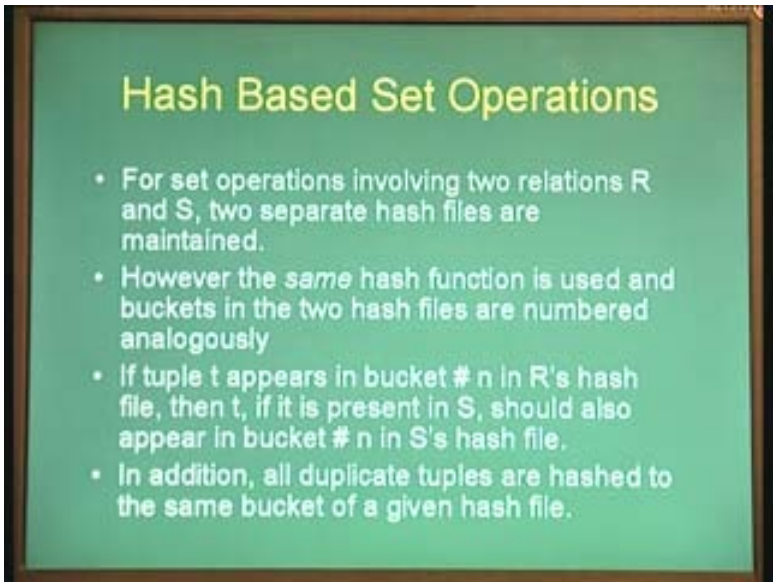
(Refer Slide Time: 48:57)



Let us first look at duplicate elimination using hash based techniques. This is quite simple as shown in the slide here. Given a relation R which contains duplicate tuples and of course which is too big to fit in memory, read the relation block by block or M blocks by M blocks. And take each tuple in the blocks that are read and hash it to a set of buckets of a hash file. And let me reiterate the fact that if there are duplicate tuples in R and important property that we have to note here is that because we are using the same hash function, if there are duplicate tuples they will necessarily be hashed on to the same bucket. So we have already cornered in a sense all duplicates into the same bucket.

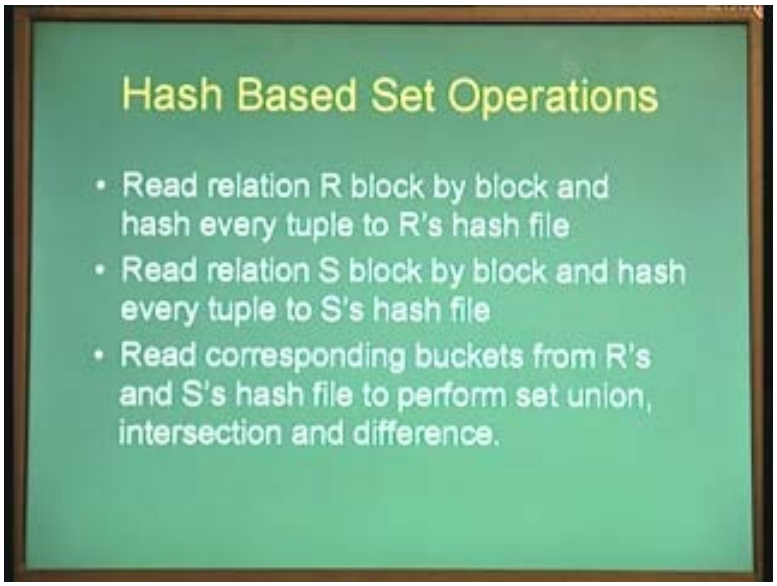
And now our job is much simpler. that is visit each bucket and eliminate duplicates using either a one pass algorithm for duplicate elimination, if the bucket can fit into memory remember how I did one pass algorithm for duplicate elimination. That is we have to maintain a internal hash or a internal index structure in memory and then eliminate all duplicates as and when we read the relation. So if the bucket can fit into memory, you can use a internal data structure or in memory data structure to eliminate duplicates or if the bucket is too big to fit in memory then we can consider this bucket as a small relation and use our previous algorithm, the two pass sort based algorithm for removing the duplicates in the buckets.

(Refer Slide Time: 50:31)



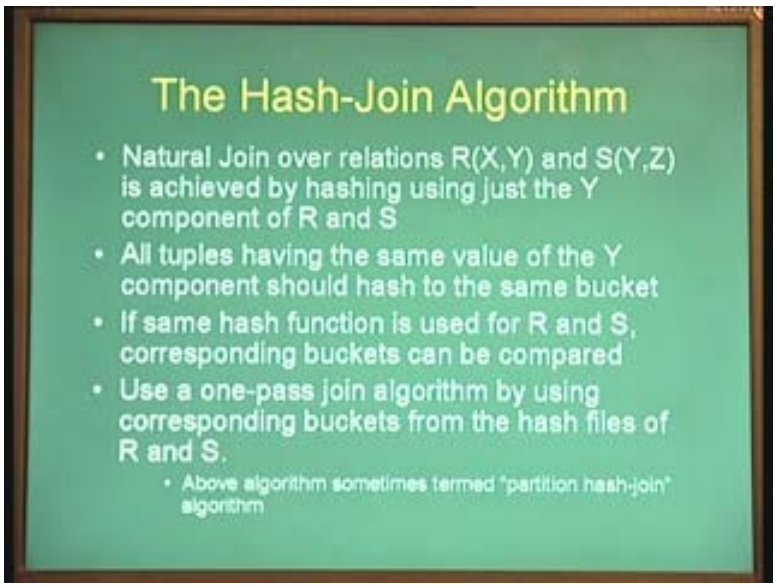
What about set theoretic operations union, intersection, set difference and so on? For set theoretic operations involving two relations R and S we maintain two separate hash files, hash based file organization. However these two hash files will use the same hash function, hash of K and the buckets are also labeled analogously that is bucket number 0 in the first hash file corresponds or is analogous to bucket number 0 in the second hash file and so on. So if tuple t appears in some bucket n in the hash file of R then if it is present in S, it should also appear in bucket number S of the hash file of S. And in addition to that the standard property that all duplicate tuples are always hashed to the same bucket in a given hash file.

(Refer Slide Time: 51:34)



So using this it is quite simple to perform set theoretic operations using hash based techniques. For example if you have to perform union, all we have do is hash each tuple of R into R hash file and hash each tuple of S into S 's hash file and take each corresponding buckets that is bucket 0 of R and bucket 0 of S and eliminate duplicates and then output the results. Similarly if you want to perform intersection, we just examine corresponding buckets bucket 0 of R and bucket 0 of S that is bucket I of R and bucket I of S and then return only the common elements and the same thing for set difference. So that is quiet straight forward. The last technique in hashing that we are going to see is what is called as the hash join algorithm.

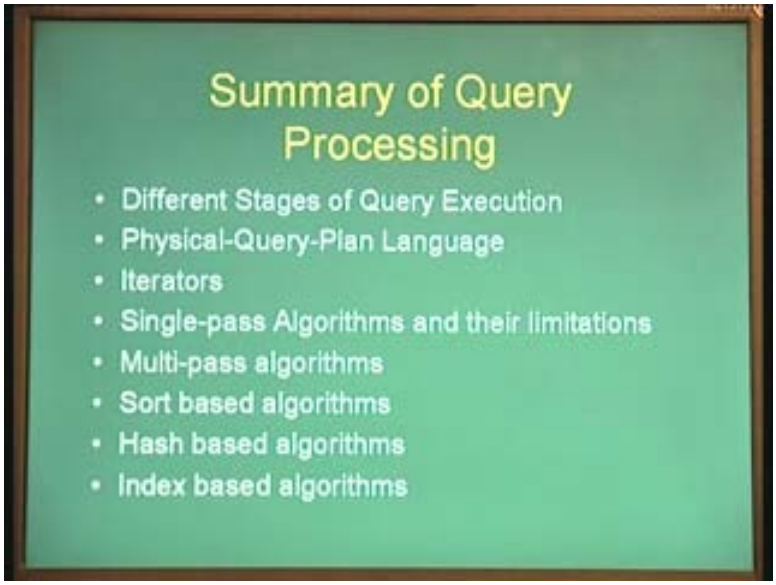
(Refer Slide Time: 52:29)



This is perhaps the most widely used technique for performing natural joins because of it is simplicity and efficiency. natural join over, let us compute natural join over relations R of X, Y and S of Y, Z where Y is the common component between R and S. Now all tuples, now we are going to first read R and S and use the Y component as the key to perform the hashing rather than the entire tuple. Now all tuples having the same value of the Y component should hash onto the same bucket if we are using the same hash function. So if the same hash function is used for R and S, we just take the corresponding buckets that is just take bucket number 0 of R's hash file and for all the tuples that you find, if they can be combined or if they can be joined with any tuple then this tuple has to necessarily exist in bucket number 0 of the S's hash file and so on.

So given this property, it is enough for us to examine just the corresponding buckets in order to perform the join operation. This makes the join operation extremely efficient because we don't have to perform too much of look ups when we are performing the we don't have to do too many look up's when we are performing the join. Such an algorithm is also called the partition hash join algorithm because the pure hash join algorithm is a slightly different algorithm from this but this is widely used algorithm for computing natural joins in many dbms systems. So let us summarize what we have learned today in the two pass algorithms.

(Refer Slide Time: 54:31)



So let us summarize query processing in general and then go back to the two pass algorithms that we have studied today. We looked into the different stages of query execution to begin with and then we saw that there are two stages namely that of the logical query execution plan and the physical query execution plan that are especially of interest. And then we saw what are some of the language constructs that make up the physical query execution plans and then we looked into algorithms that can be built to perform this internal queries or physical, to answer this physical queries using this physical query execution plan construct.

And these algorithms could be either single pass algorithms which can be used if the relations are small enough or if the query is tuple at a time query and there are multi pass algorithms which can be used if the relations are big.

Among multi pass algorithms or rather in two pass algorithms, we looked at sort based algorithms or sorting based strategies for performing several of these operations and hash based strategies for performing several of these operations. We have not looked at index base strategies which is quite analogous to sorting and hashing base strategies. We shall briefly visit index base strategies in the next session before we take up the logical query execution plans and ways by which queries can be optimized. So that brings us to the end of this session. Thank you.