Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 14

Query Processing and Optimization

Hello and welcome to another session in database management systems. In one of a previous sessions especially when we are talking about a storage structures and index structures, we talked about what is the biggest challenge facing databases yesterday. The challenge is no longer the problem of storing data or of designing that can store larger and larger amount of data. In fact we have very small devices today that can store huge amounts of data, devices that you can probably put in your pockets are something which you can wear inside your watches and so on, which can store something like 2 giga bytes of data. Therefore the problem today is not primarily of storage of data, storage of data has become much more, the surface area required for storing data has become, has shrunk in tremendous proportions and the cost of storing data is also fallen tremendously over the years.

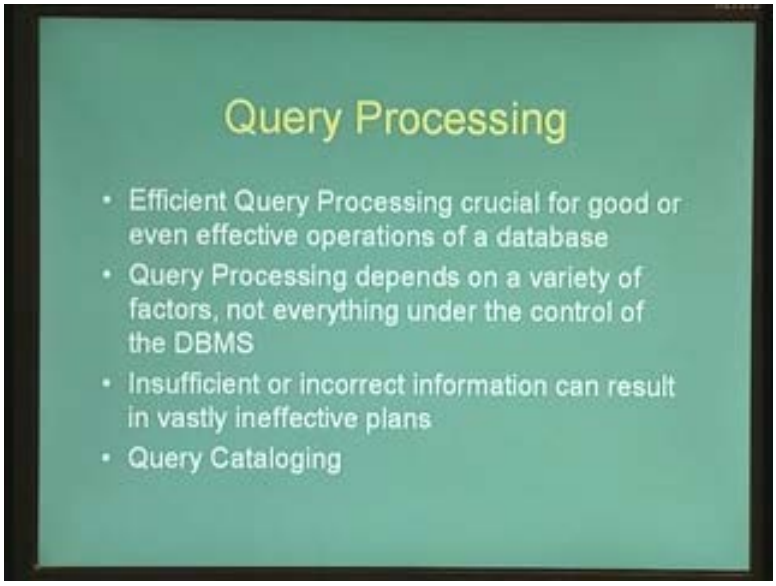
However this fall in cost or this affordability of massive amounts of data storage has resulted in a new problem or a new challenge. The challenge is that of retrieval of data, how efficiently can we retrieve data among the huge amounts of data that we have stored. We also saw how the definition of very large databases has been changing over the years. Today in, 10 years ago very large databases would probably have meant hundreds of megabytes of data, probably of few giga bytes of data. However today when we talk about very large databases, we are talking about databases that are easily into peta bytes of data 10 power 15 bytes of data. So we saw, we had mentioned these things when we are trying to motivate the use of index structures or the auxiliary files. Those are files containing metadata that can quickly point to or that can help the application program or the query to quickly retrieve the required data elements form the database.

There is however another aspect of the story using index structures alone does not help in efficient retrieval. The other crucial element in effective retrieval of data and making a data or making the difference between usability and unusablity of a database is a query execution strategies or the query execution and optimization strategies. This is the topic of this session and the next few sessions that we are going to consider. Query execution and optimization, it goes without saying is an extremely important aspect of database management and this is what is going to determine whether a particular query is going to be useful at all or not.

If a particular query execution strategy takes enormous amounts of time to retrieve a particular, to retrieve a given data element that can make the difference between whether the query is interactive in nature or whether the query is batch in nature. That is whether you have to, whether the database will have to say that please come back after two days for your query results and so on.

So let us look into what makes up a query execution and query processing and optimization.

(Refer Slide Time: 05:07)



Query processing: As we said before effective query processing or efficient query processing is crucial for good or even effective operations of database. A database can be rendered unusable if a good query execution strategies are not used. Let us do a quick calculation. Suppose I have 1 giga bytes of data and 1 giga byte as you know is 1000 megabytes of data. Even it is safe to assume that for most of the server class pc's today, we have data transfer rates of something like 1 megabytes per second. It is usually 1 mega bit per second, let us consider that it is 1 megabyte per second or 8 megabits per second. So this, if I have to access or if I have to scan through a relation let us say I have a query that select query which requires me to scan through the entire relation of 1 giga bytes of data. That means it would take about thousand seconds for me to scan the entire relation because it is of a one giga bytes of data.

Now consider a query which is given on two different tables, each of one giga byte of data so there are two giga bytes of data that is there in the database. Now a bad query execution plan would actually try to compute a cartesian product of the two tables before trying to return the results that we required. Now if I have to compute the cartesian product of 1 giga bytes of data times 1 giga bytes of data where each axis of the table is going to take me 1000 seconds then it is going to easily take me 10 power 6 that is one million seconds just to compute this cartesian product which is clearly unusable and which is clearly ineffective as far as an interactive response time is concerned. Therefore efficient query execution or trying to rehash a given query in a more effective form. For example in this, in the example that we just took up, the query might be able to figure out that what the user really wants is a natural join for example or some kind of an equi join rather than a cartesian product.

If it is able to figure that out then probably you would get a response in a little more than a 1000 seconds which is much better than a million seconds for query execution and query processing depends on a variety of factors and not all of these factors are under the control of dbms. What are this factors that can affect query execution? Let us take a few examples. One of the factors that affect query execution is for example whether the storage media is fragmented or defragmented. If the storage media is let us say fragmented, if you remember what is meant by fragmented storage, in a fragmented storage contiguous block on the storage media belong to different files. Now if I have to access a relation if I have to scan through a relation, let us say in response to a select statement and these blocks are divided or distributed all across the storage medium then the response time would be increased considerably for this query. So, not all query execution or not all factors that affect queries or query execution times or under the control of the dbms.

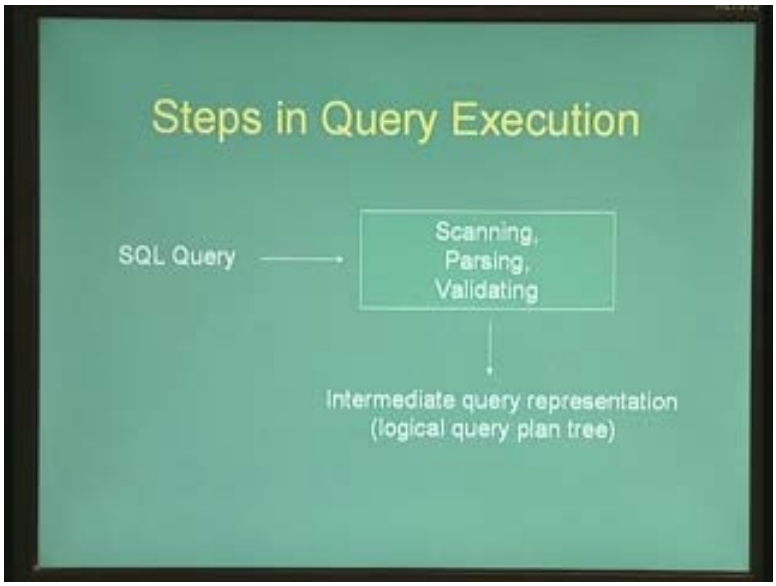
The example that we took right now is not purely outside the control of the dbms also. This is because several different, several kinds of database management systems would override the operating system mechanisms and then start to deal with devices directly. For example they create their own file system that can ensure that the file system is never fragmented at all and so on. Therefore some high end database management systems would try to overrule the underlying operating system and try to access the hardware directly in order to speed up query execution and speed up or decrease response times.

And as we saw earlier insufficient or incorrect information about factors that can, that affect query plans can lead to vastly ineffective queries. For example if a query execution plan estimates that the size of a table is let us say few kilobytes but the size of the table is actually a few giga bytes then whatever execution plan that it uses for a few kilobytes will not work for the few giga bytes table because the entire strategy changes when the scale of the problem changes. A few megabytes can probably be or few kilobytes can probably be stored in main memory whereas a few giga bytes cannot be stored in main memory also.

And query executions usually use what is called as catalogs. We shall be looking into catalogs in more detail in a later session. They use what is called as query cataloging that help in estimating several factors or several kinds of information about the database. This can include the size in terms of bytes of a particular table, the size in terms of tuples or the number of tuples in a table and estimate of the number of tuples and an estimate of the number of distinct values in a tuple that can help in building indexes for example.

Query catalogs hence play a very crucial role in deciding the query execution plan that is ultimately used on the database management system. How does a typical process or query execution process or query processing, process look like? The typical steps in a query execution process is quite similar to the execution steps in typical compiler, the way a compiler complies a given high level language construct into machine language and executes it.

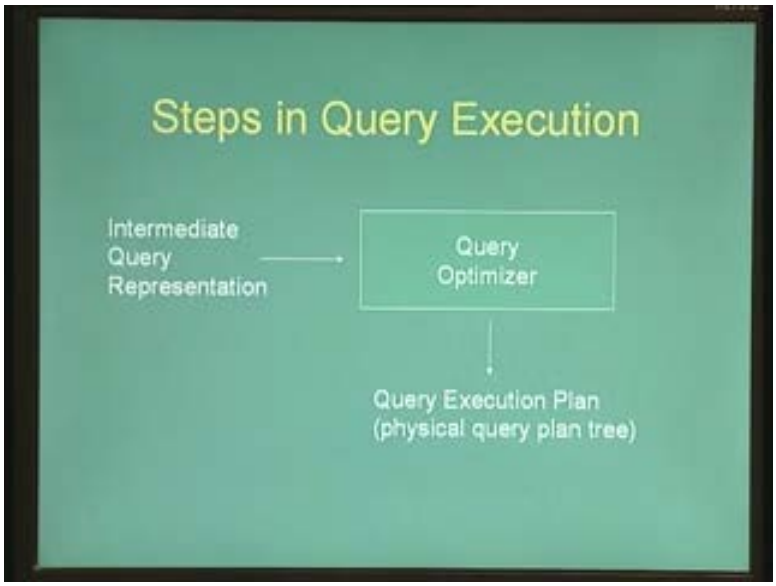
(Refer Slide Time: 12:08)



We start with the user description of the sql query. The sql query is then read and parsed that is the sql query is then read by a query complier that performs scanning where lexical analysis is performed that is the sql query is read character by character and then tokens out of the characters are recognized. And then a token stream is given to the parser which in turn parses the query, constructs a syntax tree and then the tree is validated for semantic checks for types and interoperability and so on. And once this is done, an intermediate form of the query is generated and which is called the logical query plan tree.

This intermediate form of the query is usually a relational algebra representation of the sql query. Now once this intermediate query tree is generated, a series of heuristics and cost based searches are used to rewrite this tree or rewrite this query execution tree. In order to make it more optimal or in order to make it in order to use a better equivalent query for whatever execution, whatever query has been requested by the user.

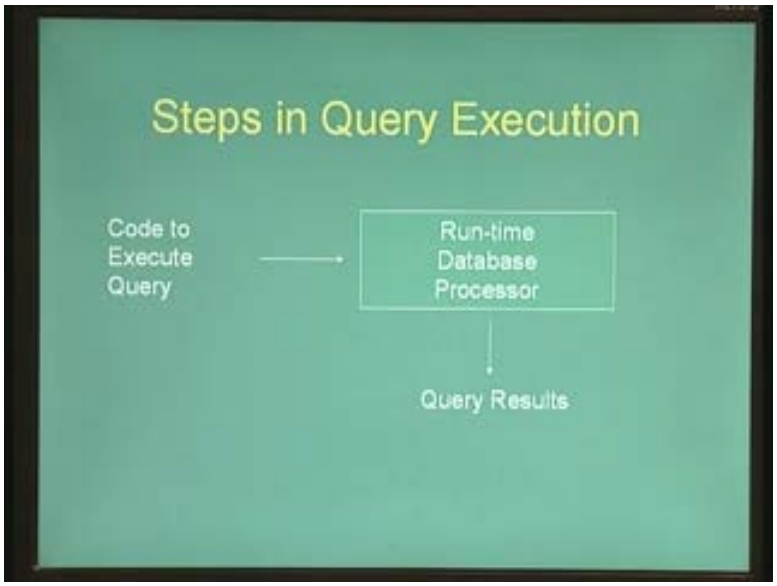
(Refer Slide Time: 13:33)



This intermediate query representation is then given to the query optimizer which in turn generates the query execution plan. That is it rewrites the tree in order to reorder few of the operations and then a final physical query plan tree is created. We shall be looking into a query optimization strategies in a later session. However there are optimization strategies can be broadly divided into either a heuristics based optimization strategy which are essentially some kinds of thumb rules which have been known to yield better strategies for query execution. And then there are what are called as cost based optimization strategies where an estimate of the cost that is required to execute one query plan against the other is used in order to utilize the best, potentially the best query execution plan.

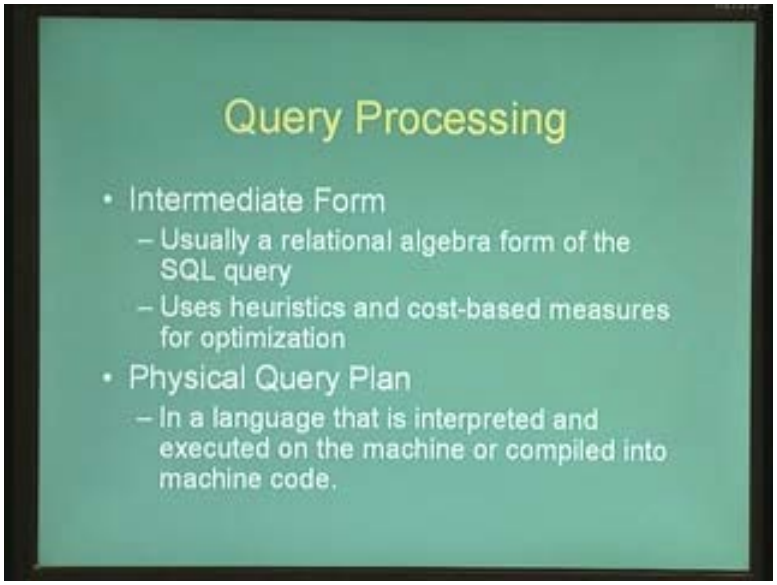
The physically query execution plan is written in a separate language not necessarily the machine language but there is a separate language that uses its own construct or its own algebra for representing what are called as internal queries. That is the query that are actually performed on the storage structures on the physical files that are stored onto disks. The query execution plan is then given to the query code generator which either executes the query as it is that is starts giving results directly which is called the interpreted mode of query execution or it generates machine code which is called the complied mode of query execution which can then be used to actually perform the physical operations required to answer the query.

(Refer Slide Time: 15:.31)



The code, the machine code that is generated in a compiled mode of query execution is then given to the run time database processor which executes the code and returns the query results.

(Refer Slide Time: 15:49)

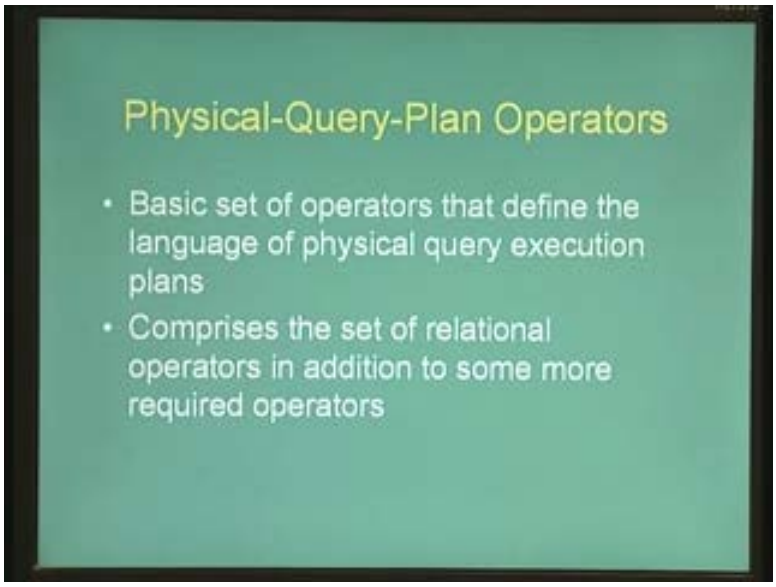


In these steps of query executions, two aspects are important and interesting. These are the intermediate form of the query and the physical query plan. The intermediate form of the query as we mentioned before is usually a relation algebra equivalent of the sql query that the user has given. The intermediate form of the query is in the form of tree structure which is also called as an expression tree where relational algebra operators are on the

non-leaf nodes and the actual domains form the leaf nodes. That is the actual relations on which query is executed from the leaf nodes. The tree is then rewritten based on a set of rules that are derived from either heuristics or cost based optimization to generate an equivalent tree which produces the same query but preferably in or hopefully in much lesser time with much lesser overheads.

The physical query plan is written in a separate language which has its own construct and that is either interpreted or complied into machine code. Let us have a look at what constitutes the physical query execution plans and what are the constructs that make up this physical query execution plan. The logical form or the logical query execution plan or the intermediate form, we shall be looking into greater detail in later session.

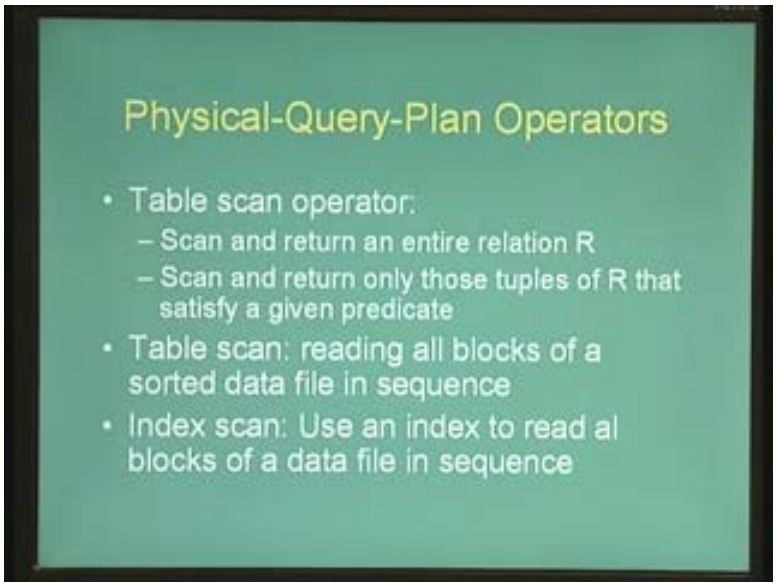
(Refer Slide Time: 17:24)



The physical query plan comprises of a basic set of operators that define the language of physical query execution. Now what should this operators be? Obviously the operators at the lower level or the inner query or the internal query should contain all the operators of relational algebra itself. That is if the relational algebra says select on this condition, the internal language should also be able to support an operator that can select a particular tuple based on a set of tuples on this condition.

However in addition to the relational algebra operators, there are several other operators which talk about a physically accessing tables and iterating through them or several other physical operations. Note that relational algebra itself does not concern itself with the physical implementation of the database.

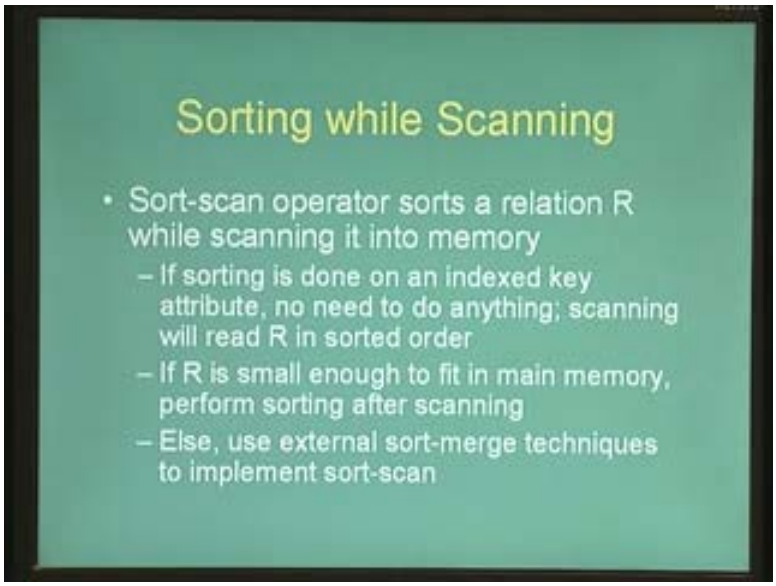
(Refer Slide Time: 18:23)



Let us have a look at a few candidate physical query plan operators and see how they work that gives us a flavor of how does the physical code actually look like or the query execution code look like. The first operator that you are going to see is the table scan operator. A table scan operator as the name suggests just simply scans a particular given table that is it scans and returns an entire relation R or the operator can be parameterized in the sense that you can give certain conditions to the table scan operator that scans a given relation R and returns only those tuples that satisfy the given condition or the given predicate.

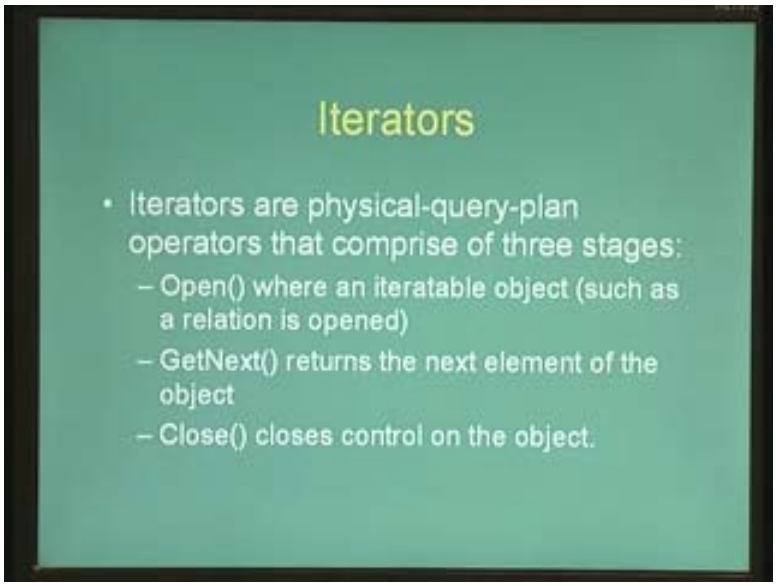
The main operations that are performed by table scan is to read all blocks. Note that blocks is the physical component in terms of which the records are stored. So a table scan contains a code or the operator for table scan contains code by which blocks belonging to a particular file or a particular table are read in sequence and the block is and the table is returned. There is also an index scan operator that makes use of an index file in order to read all blocks of a data file in sequence.

(Refer Slide Time: 19:50)



Another operator that is used usually in the physical query plan language is a sort scan operator. The sort scan operator scans a relation R and sorts these results before returning it to the higher level whichever called it. If the relation is already stored in a sorted form and the sorting is also required on the same ordering attribute, no sorting needs to be done separately by the sort scan operator. And if the relation is small enough to fit in memory then sorting can be done directly in memory. However if the relation is too big to fit in memory then external sorting and external sort merge techniques have to be used in order to sort the given record.

(Refer Slide Time: 20:41)

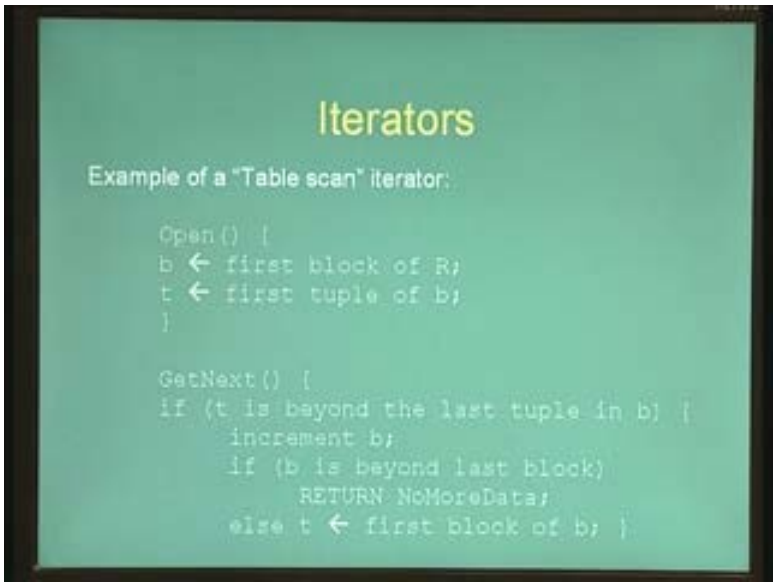


The next physical query plan operator that is quite frequently used is what is known as the iterator. Iterator is an important concept in managing or in the physical management of data records. If you have probably let us say programmed in a, done programming c plus plus let us say using the standard template libraries on unix environments or the active template libraries on Microsoft environments, you would have come across the term iterator in several places. What does iterator do? The iterator is an operator that functions on an operand which is a composite operand.

For example an iterator operates on a hash table or a linked list or a tree or something like that, so the iterator operates in a way that iterates through each element that forms the composite operand. That is it starts from the first element and it comprises of GetNext function which can get you the next element, until you reach the end of the operator or the data element.

So iterators typically contain three different functions as shown in this slide here. The first function open will open the iterator object that is the composite object on which the iterator function has to be performed. The next function called GetNext is going to get the next logical block that or the next logical record or next logical node or whatever it is in this composite object that has to be returned. And then the last operator called close, closes control on the object.

(Refer Slide Time: 22:35)

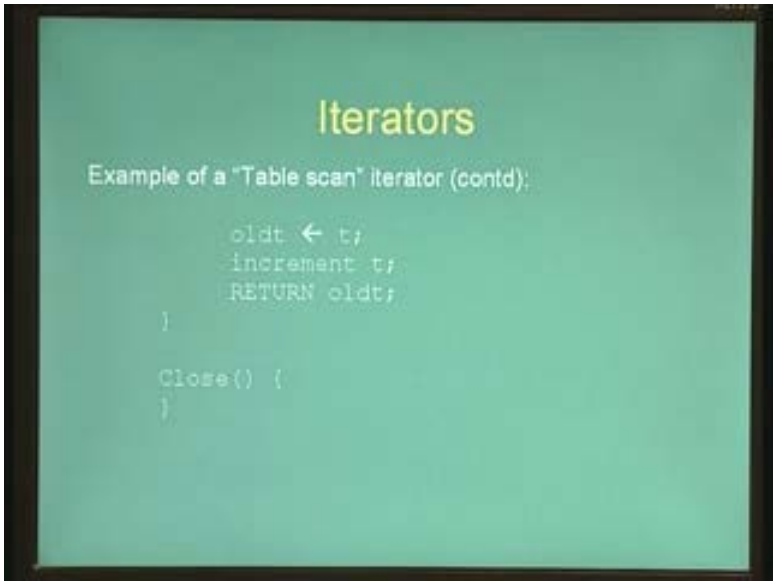


This slide shows an example of the iterator function, example of a table scan iterator that is how we can implement the table scan operator using an iterator. As the slide shows there are three different functions that have to be implemented. Open, GetNext and Close which is shown in the next slide here. The Open function let us say given relation or a given file let us consider relation to be stored in a file or a table to be stored in a file, the open construct initializes two variables, a variable called b which points to the first block of the relation and a variable called t which points to the first tuple in inside b.

The GetNext function just iterates through this variables that is before we go in to GetNext function, let us try to ask ourselves what should the GetNext function do. Ultimately for the program which is calling the table scan iterator, all that is required is the set of tuples one after the other. The GetNext function however has to deal with two different, two different things the blocks that is the physical data stores and the tuples that is the logical data stores.

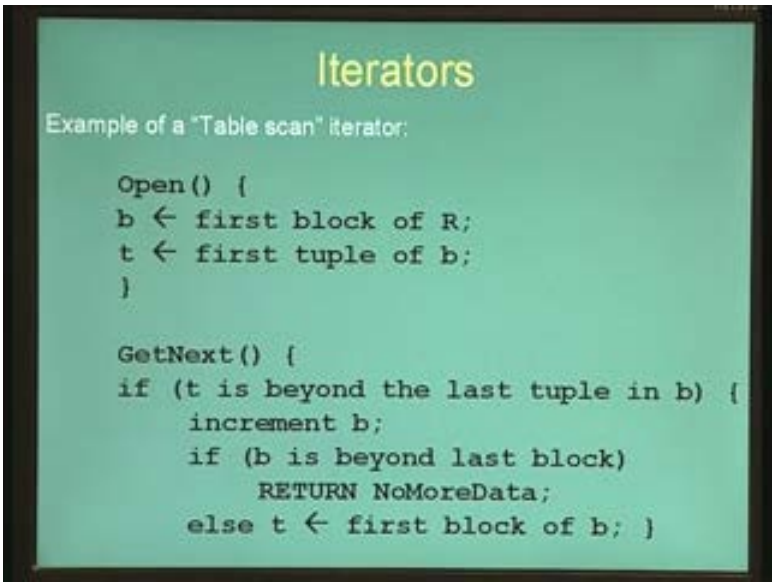
Now tuples can be iterated within blocks but when a block is exhausted, the blocks themselves have to be iterated across the files. That is the next logical block in the file has to be chosen. So the GetNext function performs precisely this set of function. That is if t is beyond the last tuple in b, that is if the current block b has been exhausted then we increment b that is point b to the next logical block in this sequence in the file. And if b is beyond the last block in the file then you return no more data that is it is exhausted in the record or the file or else the else condition here states, is basically t is beyond the last tuple in b but b is not beyond the last block in the file. That means set t to the next or set t to the first tuple in b that is the next b is been incremented and set t to the first tuple in b or the new block.

(Refer Slide Time: 25:10)



And then increment t and return the old value of t which t was pointing to. So return, that is we first assign oldt equal to t and then increment t and then return oldt. For close we don't have to do anything because we have already returned, we already returned from the GetNext operator if we have reached end of the file. Therefore close in this example is redundant, however usually the close function performs some kinds of a cleanup operation where if some data structures were opened during the course of the iterator function, these data structures are closed and the corresponding memories freed and so on. Let us implement or let us look at an example of the iterator function. We shall implement the table scan operator that we saw earlier using the iterator function.

(Refer Slide Time: 26:16)

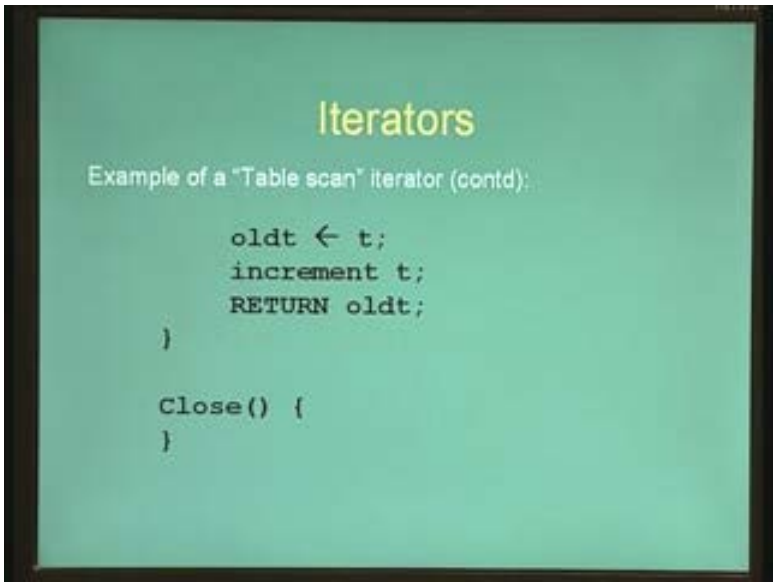


As we saw before an iterator has three different functions the Open, GetNext and Close. Assume that the table is implemented or is contained within a file and the file is organized as a sequence of blocks. That is there are several blocks that make up the file. So when we open the iterator that is open the table scan iterator, we need to initialize a few things. This is shown in the slide here, that is the slide, the open function initializes two different variables b and t where b points to the first block in the file of the record and t points to the first tuple in the block.

The GetNext function, note what the GetNext function should return here. The table scan operator should return tuples after tuples that is the first tuple, second tuple and so on. however at the physical level, we are concerned not only with tuples but also with blocks that is we actually read and write in terms of blocks and in not in terms of tuples. Therefore initially what we do is we first check to see if the tuple is beyond the last tuple in b.

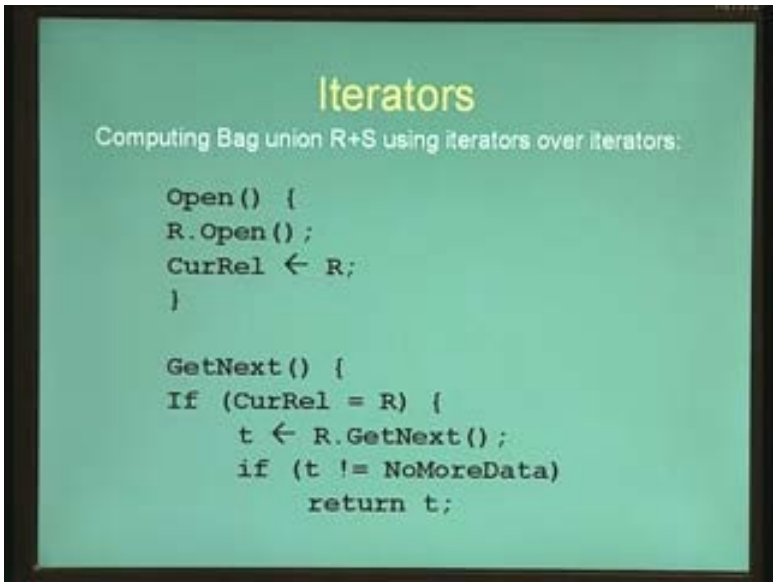
If this is the case we have to increment b and if b is beyond the last block in the file then we just return no more data. That is we say that is all, there is no more tuples to return or else that is the else is for the inner if, so else t is then set to the first tuple in the next block that is we have incremented b and then we just set t to the first tuple in the next block. And if none of these is the case then we just return the next tuple that means we first copy the corresponding tuple to be returned into a new variable which is called oldt in this example and then we increment t and then return oldt.

(Refer Slide Time: 28:10)



For the Close function in this example, we don't have to do anything because we have already returned in the GetNext function when we have reached the end of the file. But usually in a Close function, we use the Close function to clean up whatever mess we have created so to say. That is whatever data structures that we have opened, whatever memory we have allocated which are not useful anymore have to freed up and so on. So the Close operator or the Close function is called at the end of the iterator which performs all this cleanup operations. Let us look at another example using iterators on how to compute the Bag union of two relations. What is the Bag union?

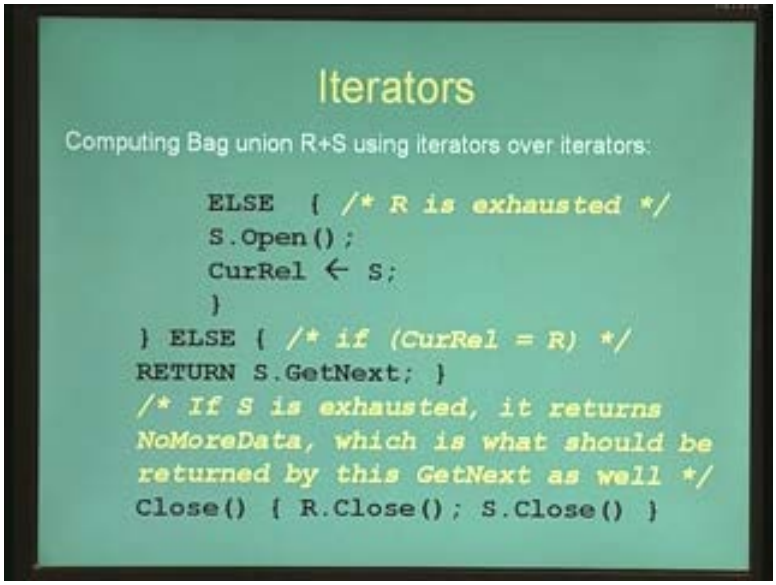
(Refer Slide Time: 28:58)



Remember we have talked about a considering relation as Bags rather than sets. A Bag is just a collection of tuples or collection of elements without regard to whether there are duplicates in the collection. So it is also called a multi set. A multi set union or a Bag union is simply a Bag that is made of two different Bags. That is you just empty contents of one Bag into the other and you have got a multi Bag union or union over Bags. So this is denoted by the operator plus or the disjoint union operator R plus S.

So an iterator for performing the disjoint union, here we are considering that both R and S which represent relations for us are now in the form of iterators themselves. That is we are abstracted away a relation form being a file to being an iterator that is we know it is just a data structure which we can open and call the GetNext function and then close once we are done with the data structures. So as far as we are concerned, both relations are just iterators. So in the open function of our disjoint union iterator, we just open one of the relations. We say R.open and then we point the current relation to be R. In the GetNext function we say that if current relation equal to R then we have to call GetNext on the current relation that is we just say current relation dot GetNext. And if GetNext returns no more data that is if there is no more data that is returned then start or set current relation as S and then call S.Open. And then in the subsequent GetNext operations, you just call S.GetNext instead of R.GetNext.

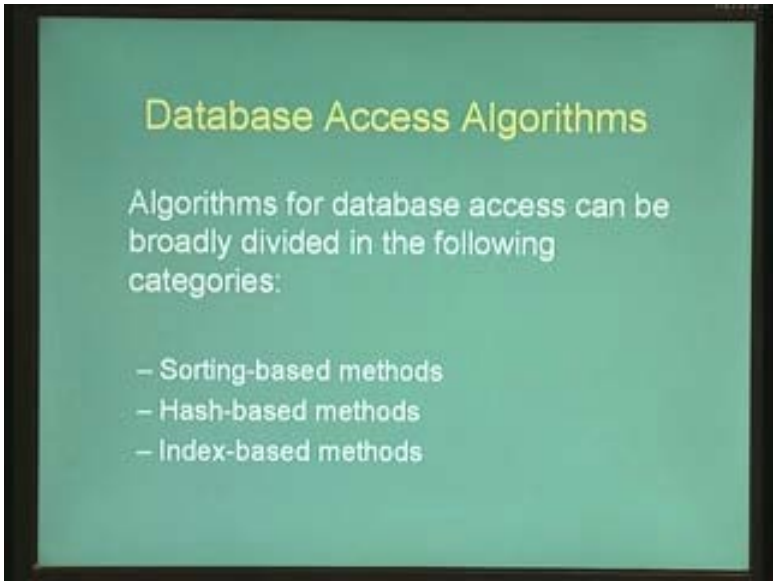
(Refer Slide Time: 30:53)



So what we have effectively done is we have exhausted one of the records by calling GetNext as many times as possible. That is whenever GetNext is called on us, we call GetNext on the current Rel that is the CurRel relation. So once we exhausted one of the relations, we open the other relation and start calling GetNext on that function. So when S is exhausted it returns no more data which is what should be returned by the GetNext as well and in the Close function we just close both of these iterators. That is we call R.Close and S.Close. Well, R.Close and S.Close don't do anything which we saw in the previous example but in case they do certain cleanup operation. It is always a good

measure to or it is always a good programming practice to call the corresponding Close operators in our Close operator.

(Refer Slide Time: 32:13)



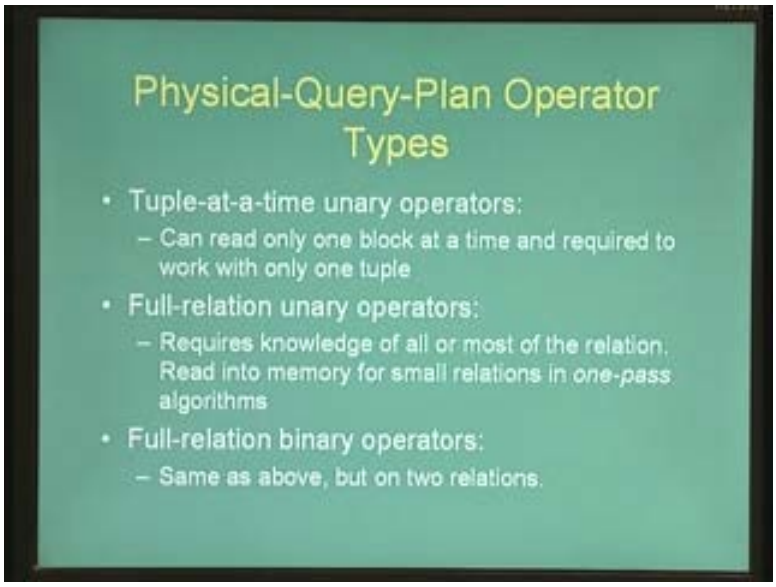
So we just went through some of the elements of the physical query plan programming language that is it contains elements of table scan, index scans and iterators and so on. Now let us have a look at some algorithms that are built around these data structures or around this constructs of the physical query plan that can help us in understanding how a given relational algebra operator, let us say like select or project or something of that sort is actually executed inside the database system.

We can broadly divide algorithms for a data access into one of the three following categories. We call them sorting based methods, hash based methods and index based methods. These methods as you can see here are typically meant or oriented towards increasing the effectiveness of search. In a sorting based method the relations that are scanned using the sort scan operator that is they are sorted as in when they are scanned and because they are sorted the property that the relations are sorted would help in performing certain other relational operators like say join in an efficient fashion.

Similarly hash based methods use some kind of a hash function to quickly search for whatever tuple or data element that is being asked for within this relations. An index base methods resort to index structures like trees or balance trees and so on for searching the required data element. We can also divide algorithms for data access based on what kinds of data access requirements that they pose. We can divide the kinds of data access requirements into one of these three kinds of requirements. The first requirement is what is called as a tuple at a time unary operator that means the query requires or requires to contend with one tuple at time.

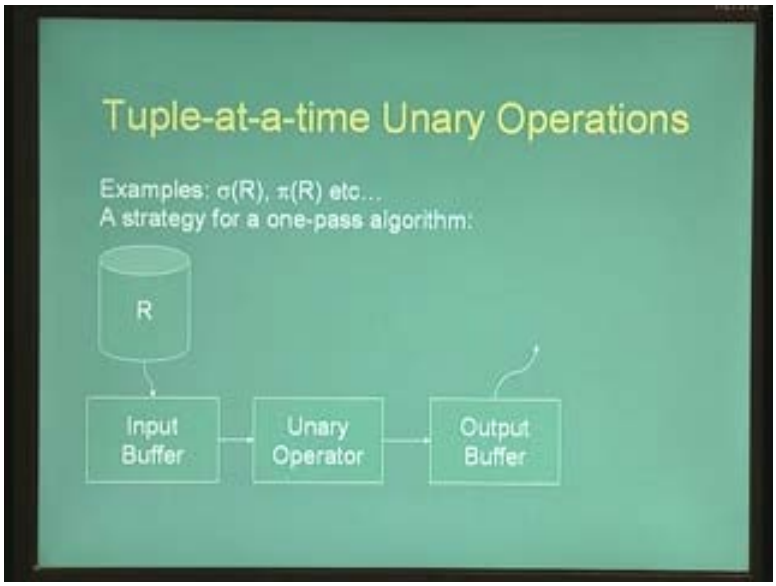
For example select and project operator. Every time select is called, select has to be or the condition for select has to be checked against each tuple in the relation that is tuple after tuple. So at a time one tuple is being accessed and this is a unary operator that is it is just one relation on which a particular tuple is being accessed. Then there are full relation unary operators where the entire relation has to be searched. For example if I have to return something based on or return the value of some relation or if I have to compute let us say set theoretic operations like not of something and so on or any kind of set theoretic operation that are unary in nature. And the last kind of operations are full relation binary operators. These are operators that again have to compute or that again requires a complete relation as their query result and they are not just unary, they are binary that means they have two relations to contend with.

(Refer Slide Time: 35:52)



That is some examples or something like set theoretic operators like union, intersection and so on which require two relations and the entire relation has to be scanned the entire relation has to be returned.

(Refer Slide Time: 36:10)

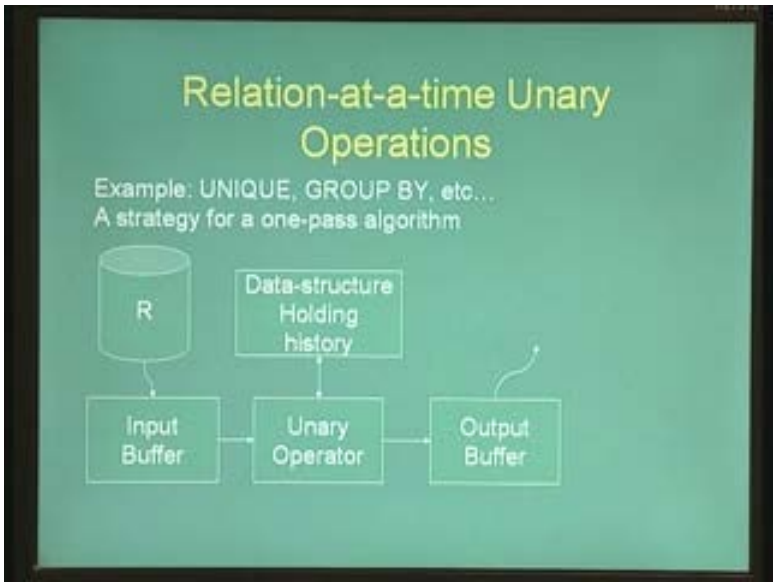


Let us see how each of these kinds of query execution requirements can be met using some algorithmic strategies. In this session we are going to be looking at a kind of strategies what are called as one pass algorithms. What is a one pass algorithm? A one pass algorithm is an algorithm that performs at most one pass over the entire database that is over the entire relation of interest. It does not access the relation multiple times. very important and many times limiting assumption in most of the one pass algorithms that we are going to see here is that it assumes that the relation that we are looking for is small enough to fit in main memory. In many cases this is the reasonable assumptions but in many other cases, it is not a reasonable assumption. That is even a single relation could be so huge that it may not fit into main memory.

So how does, how can we use a single pass or a one pass algorithm to perform a tuple at a time unary operation? Let us take some example like select or project as shown in the slide here. let us say select some condition over R or project some condition over R. All we have to do is scan through this R that is use the table scan iterator for scanning through this relation tuple after tuple and store this relations or store this tuples that have been scanned in a input buffer, perform a unary operator and output it to the output buffer. So this is schematically shown in the diagram here.

That is this is the relation iterator and this relation iterator returns tuple after tuple which goes into the input buffer and in this case this input buffer can be as small as one tuple long. That is we can allocate just enough memory in the input buffer to store just one tuple. So each tuple, after tuple is put into the input buffer and checked against the unary operator and either discarded or sent to the output buffer, so as simple as that. This is quite simple that is within a simple single pass, we have been able to answer a tuple at a time unary operation.

(Refer Slide Time: 38:49)



What about relation at a time unary operations? What are some examples of relation at a time unary operators? One example is that of let us say the unique function. In the select, in the sql statement suppose I say select name from employee or select unique name from employee that means given me the set of all unique employee name's without repetitions. This as you can see is a unary operator that means it operates on just one relation. However it is a relation at a time operator that is it requires to have the entire relation, knowledge about the entire relation before being able to return the required value.

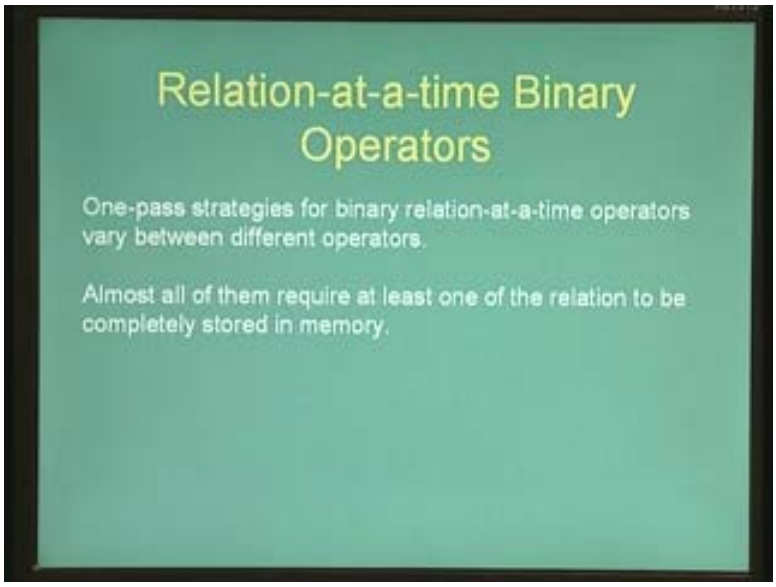
So the general strategy or a general one pass algorithmic strategy for relation at a time unary operators is shown in the slide here. R is now familiar table scan iterator which returns to a tuple after tuple which goes into the input buffer. Now the input buffer is then read into the unary operator whatever be the unary operator, whether it's unique or group by, for example group by is another relation at a time unique operator. Now this unique operator will either output this tuple into the output buffer, if it is safe to do so or otherwise will put the tuple into a data structure holding the history of whatever relation has been read until now.

For example in the unique operator, all we need to do here is have a hash table that contains one entry each for each unique entry that we have found until now in the database. So whenever I read a new name, let us say whenever we read a new tuple into the input buffer and check out the name attribute, we just check the hash table here the data structure holding history. We just check the hash table to see if this name was already encountered, if you are already encountered this thing. If you have already encountered this name then we just discard this new tuple. Otherwise we add this new name into this hash table here and then output the tuple. So a single pass algorithm for a relation at a time is also quite simple except that we need to have an augmenting data structure in the form of usually an index tree or a hash table or something like that can

hold the history that is required. Now one more thing that is to be noted here is that suppose the unary operator that we are concerned with is the groupby operator.

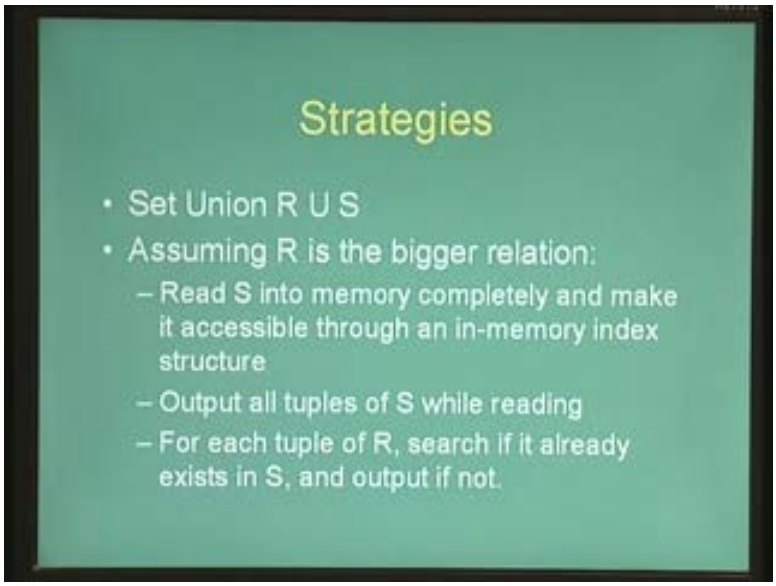
Now the groupby operator cannot return any output until the entire set of relations or the entire set of tuples in this relation is read and the performing grouping is formed using this data structure. That means this space allocated to this data structure should be large enough to hold the entire relation. Therefore such an algorithm cannot be used for relations that are too large to fit in memory because we are concerned only with one pass algorithms in this session here. We assume that the relation can be held in memory so that the entire relation or the entire history of what we have read can be held in the data structure.

(Refer Slide Time: 42:35)



Let us look at one pass algorithms for relation at a time binary operators. Now one pass algorithmic strategy is vary depending upon on what is the binary operator that we are looking and almost all of the algorithms for binary operators require that at least one of the relation be read completely into memory before we start reading the other relation. And obviously if we have two relations and one is much smaller than the other, it makes much more sense to read the smaller relation into memory and iterate over the larger relation. So let us look at a few examples and which will make this clear.

(Refer Slide Time: 43:19)

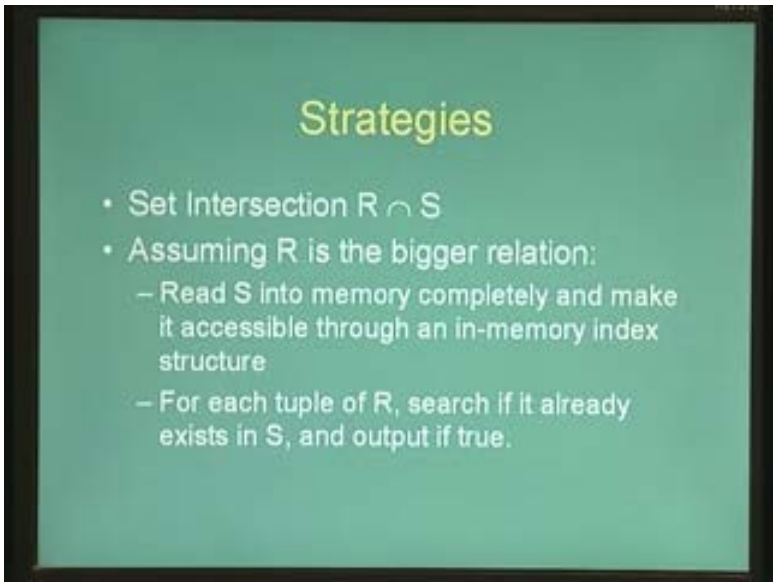


Let us see what is the strategy what is a one pass algorithmic strategy for computing the set union of two relations R union S. I have explicitly used the word set union here instead of just union that means this is not a Bag union. That means to say that we have to compute R union S without returning any duplicate entries in the result. That means we have to remove all duplicates while returning R union S. Assuming that among R and S, R is the bigger relation, here is a very simple strategy to compute R union S. First read S into memory completely using the iterator on S, retrieve all tuples from S and place it into memory and place it in some kind of data structure like an index or hash table by which we can access each tuple of S as efficient of as possible.

Now as and when we are reading S, keep outputting the tuples of S because anyway R union S should contain all tuples of S. Then once S is completely read into to memory and indexed in a data structure, start reading R that is the next relation and for each tuple of R that is read into memory, check whether it already exist in S. If the tuple already exist in S then just discard the tuple because we do not want duplicates in the output result. Otherwise if it does not exist in R then or if does not exist already in the relation then just output the tuple.

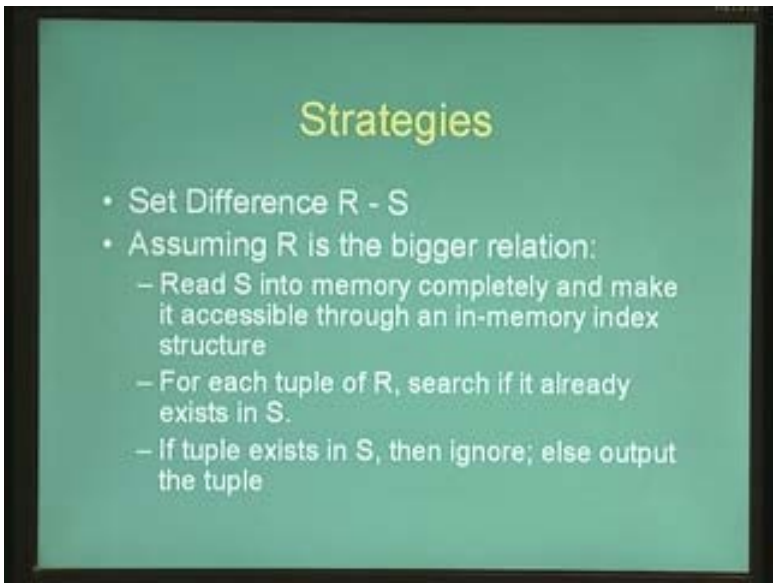
Now here we are also making another implicit assumption that R and S are sets themselves and they are not multi set. That means there are no duplicate tuples in R itself. Therefore it is sufficient for us to check for duplicates against S, otherwise we need to also store tuples in R so as to check the duplicates within R itself. If we assume that R and S are sets, the set union operator can be performed using the strategy that we outline just now. The next binary operator that we are going to look at is the intersection operator. The strategy for the set intersection operator is also quite similar to that of the union operator.

(Refer Slide Time: 45:56)



Assume that we have to perform the set intersection between R intersection S. And assuming that R is the bigger relation, we first read S into memory and then store tuples of S in an in-memory data structure or in memory index or hash tables structure that can help us access the data elements of S quite efficiently. Then start reading R into the memory tuple by tuple using the iterator for R, then for each tuple of R if and only if the tuple also exist in S, output the tuple of R in to the output buffer otherwise discard the tuple.

(Refer Slide Time: 46:38)

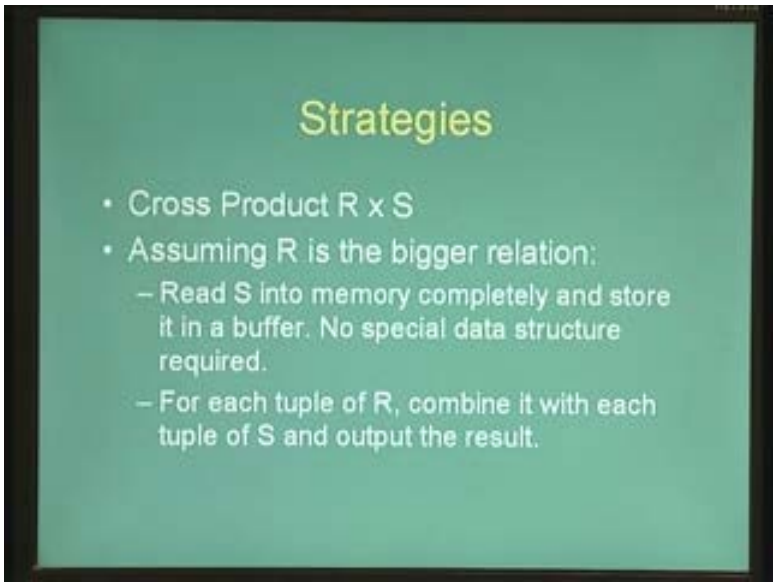


What about set difference? Set difference if you see differs depending on whether we are computing R minus S or S minus R because set difference is not a commutative operation. Now suppose let us say without loss of generality, let us say we are computing R minus S and that R is the bigger relation. So we are computing R minus S and the first relation R is the bigger among the two relations. That means we read S into memory as usual that is read the relation S using the S iterator into memory and put S into an in-memory index structure or a hash structure. And for each tuple of R check to see whether it already exists in S. If it already exist in S then discard the tuple or if it does not exist in S then output the tuple as simple as that. But what happens if we compute S minus R, that is R is the bigger relation and it is right hand side of the difference that is instead of computing R minus S, we are computing S minus R. Because R is the bigger relation it is always more efficient to read S into memory rather than R.

Now if we read S into memory, how does the algorithm change? Let us have a look at that. So this slide shows set difference S minus R instead of R minus S and assuming that R is the bigger relation. Now if R is the bigger relation, have a look at the steps closely for this slide here. If R is a bigger relation, all we have to do is first read S into memory completely that is read the complete, use a S iterator and read all tuples of S into memory and place them into an index structure or a hash table. Now for each tuple of R, what should we do? That is we are computing S minus R that is S minus R is the set of all tuples in S that are not in R. So for each tuple of R, check to see if it already exists in S. And now what happens if it already exists in S?

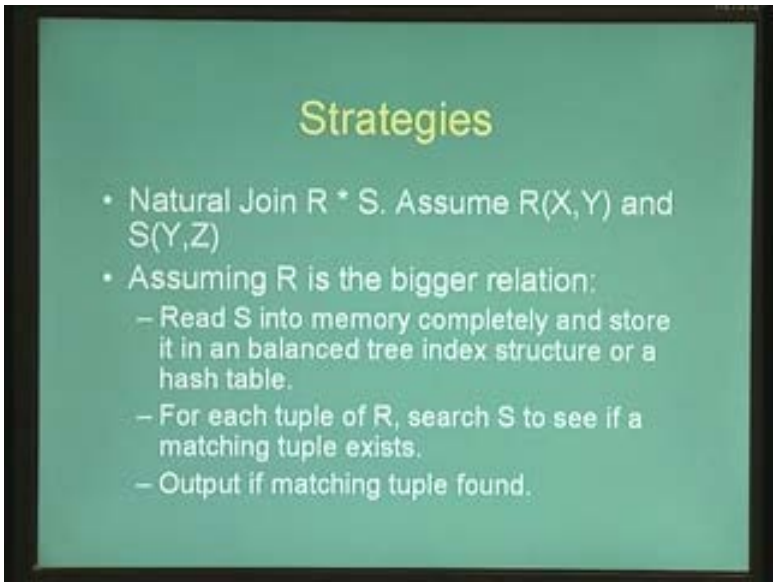
If it already exist in S then these tuples should not be there in the final output. And the tuples that should be in the final output are those tuples of S that are not in R. So what we do here is for each tuple of R, for which we find a matching tuple in S we cancel them out that is we delete the tuple in S from the index structure. That is we delete it from the index structure or the hash table that we have been using. Now once R is exhausted that is once we have finished reading through the relation R and we have deleted all common tuples, whatever is left in the S data structure that we have read into memory is the output. That is that we can push them onto the output buffer.

(Refer Slide Time: 49:47)



What about cross product? R times S. Cross product is simple as far as the algorithm is concerned and quite expensive as far as the performance is concerned. That is again assuming that R is the bigger relation just read S into memory and we don't need any data structure here, we can just store S in its contiguous sequence of memory locations and for each tuple of R combined with every tuple of S and start returning it, as simple as that.

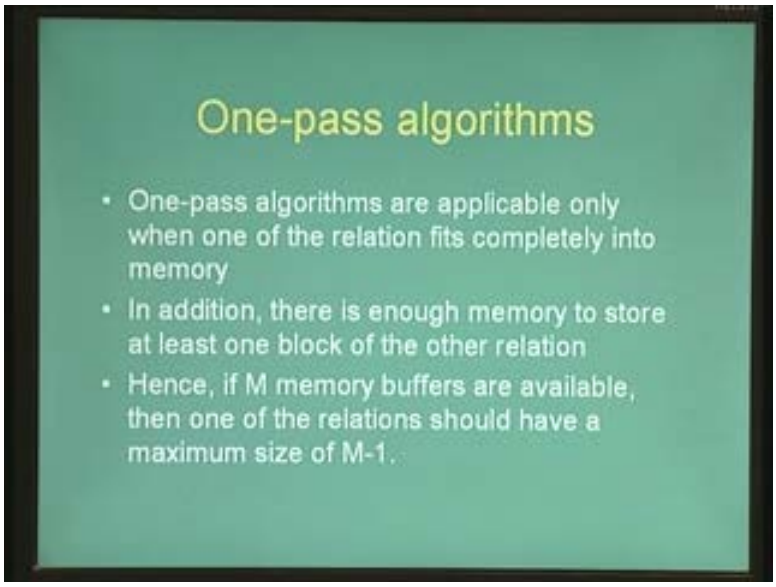
(Refer Slide Time: 50:18)



The last one that we are going to be looking at is a one pass algorithm for natural join. What is a natural join? A natural join is an equi join on two relations that equates

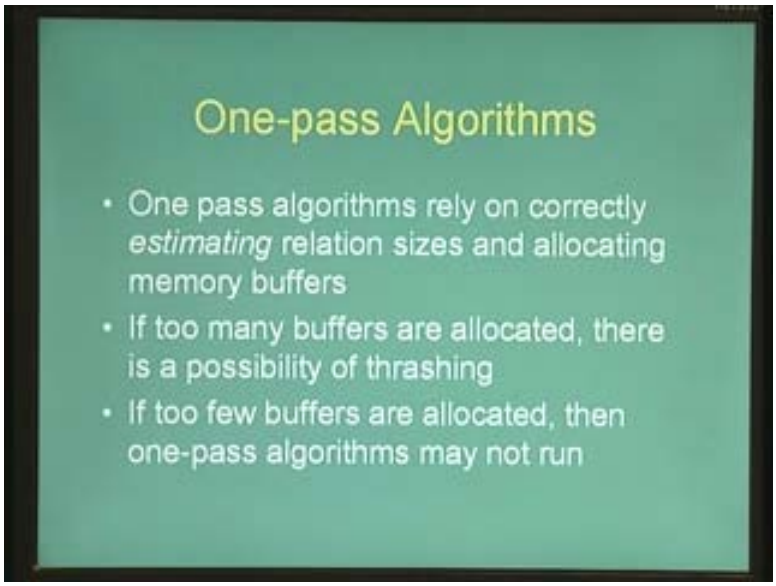
attributes having the same name and domain. Now assume that R, X, Y and S, Y, Z are being subjected to a natural join. That means Y is the common set of attributes or subset of attributes between R and S. now assuming again R is the bigger relation, read S completely into memory and then index or place X in a hash table or an index, so that it can be searched efficiently. Now for each tuple of R what we have to do is search through the hash. Now, this hash table or the indexing structure should be done based on the common attributes. That is based on Y here, even if Y is not the key attribute. So, we perform the indexing or the hashing based on the common set of attributes Y and then using which we can search for every tuple of R that is read whether there is a matching tuple in S. If there is a matching tuple, match the two tuples and output it to the output buffer if not just discard the tuple of R.

(Refer Slide Time: 51:44)



So that was a brief overview of the one pass algorithms that can be used by using the physical query plan operators like say iterator and sort, table scan, sort scan, etc using which we can develop a strategy for performing relational algebra operations like R union S, R intersection S, select, project, unique, groupby and so on. But what are the constraints of one pass algorithms? One pass algorithms are applicable especially for binary relation at a time binary operators. They are applicable only when one of relation can fit completely into memory and it is not just it fits completely into memory, we should also have extra blocks of memory for the other relation. That is let us say if I have M memory blocks available for me, then the smaller relation can be at most M minus 1 in size, M minus 1 blocks in size. It can't be M in size because we need at least one more block to perform book keeping for the other relation that we are reading from disc.

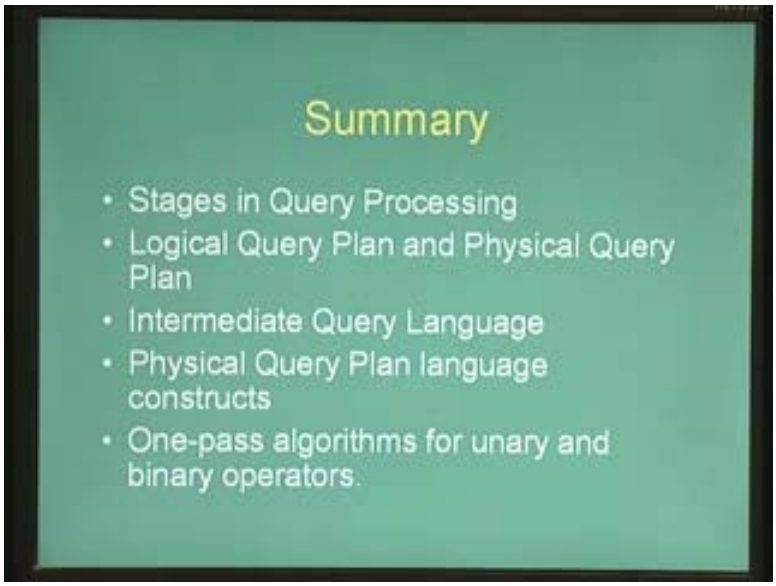
(Refer Slide Time: 53:00)



And one pass algorithms rely to a great extent on correctly estimating the size of relation. If a query execution plan in a dbms engine decides to use a one pass algorithm for performing a particular query, relational query, relational algebra operation then it depends for a large part large part on the estimate of the size of the relation that it has. Now if the size is wrongly estimated then it can for example if we allocate two few buffers thinking that the size would be small then the one pass algorithms will be unusable, the query execution plan is unusable, we can't use the one pass algorithms at all.

On the other hand if we allocate too many buffers thinking that the relation size is big then we may end up in a possibility of thrashing where memory blocks have to be swapped on to disk and so on. So it is quite crucial to obtain a good estimation in order to use one pass query execution plans.

(Refer Slide Time: 54:15)



So let us summarize what we have learnt in this session. We have kind of a scratch the surface of an important and crucial area of data base management systems called a query processing and optimization. We have seen the different stages in query processing and two important intermediate steps in query processing namely, the logical query plan and the physical query plan and in this session we have concentrated more on the physical query plan. That is a physical query plan is a set of language constructs that perform low level operations using, that performs low level operations directly on the storage media that can physically access files into memory for performing any data base related operations.

We also saw what a physical query plan language would look like or rather what kinds of constructs it would have. It would obviously have relational constructs, in addition it would have constructs like iterators, tables scans, sort scans, etc. We have looked at the variety of one pass algorithms using these physical query plan constructors using which we can perform a variety of relational algebra operations like select, project, unique, groupby and many other set theoretic operations. In the next session we shall be looking into the big question of handling joins in query execution plans and also have a look at the logical query execution plans. So this brings us to the end of this session. Thank you.