

Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 16

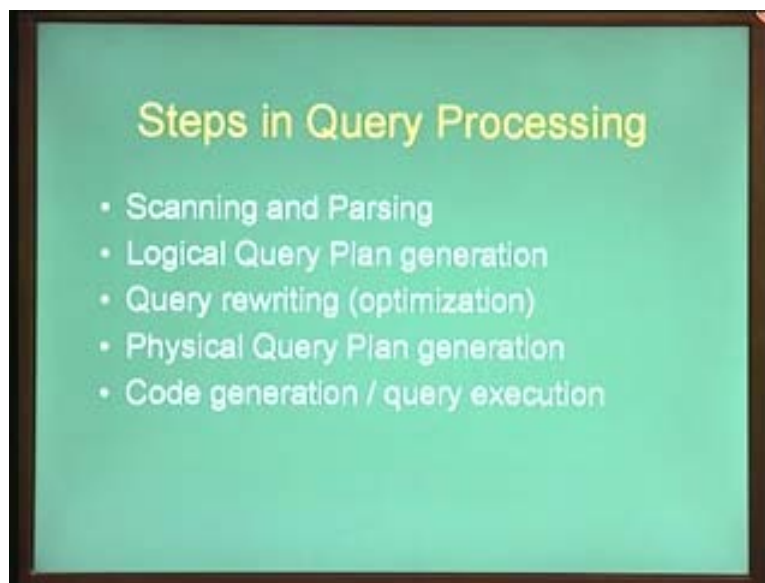
Query Processing and Optimization – III

Hello and welcome. In today's session we shall be continuing with whatever we have been exploring in the past two sessions namely query processing and optimization issues. As we have seen query processing is a very crucial element in a dbms design that is this is not about database design like I had mentioned in previous session that is database design is the term that is used to denote activities like schema design normalization and so on. That is how to design a database such that a dbms can be used to handle this, the data in the database efficiently as possible.

On the other hand query processing issues concern design issues of the dbms itself that is how can we build a dbms that can efficiently process a given user query and even if in many cases, even if the query is not formulated in a form that is the slightly to be the most efficient, can the database or can the dbms detect it and rewrite the query in such a way that the query becomes much more efficient. And we saw that a query processing is so crucial that it can make the difference between usability and un-usability of the dbms.

Let us briefly have an overview of the different topics that we have studied in query processing before we move on to today's topic that is of query optimization.

(Refer Slide Time: 02:47)



A typical query processing, a typical process of query processing takes several different steps. When the user gives an sql query, it is first passed through a scanning and parsing phase where the query is first scanned so that the query becomes or query is divided into a stream of tokens and these sets of tokens are then parsed to build a parse tree or a query parse tree that gives the syntactic structure of the formulated query that is given by the user.

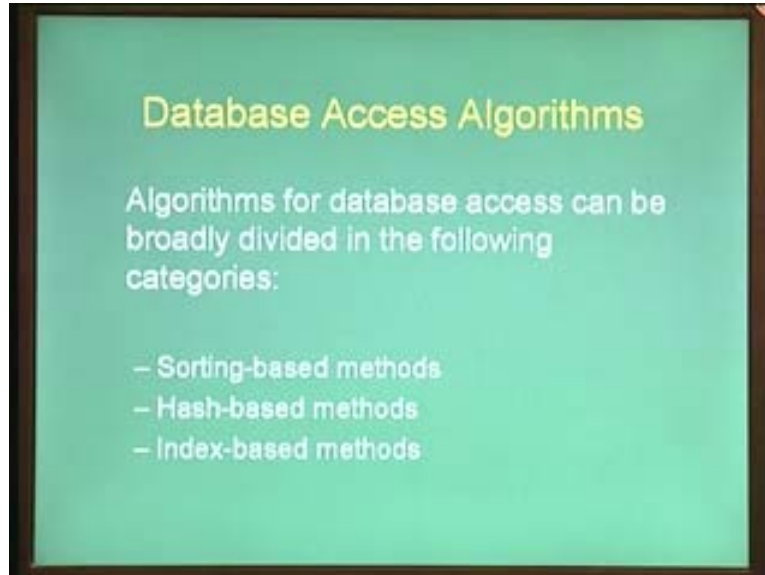
Now from this parse tree, a logical query plan is generated that is the parse tree is rewritten based on certain rules, heuristic rules and several different rewritings of this parse tree are possible and one of them is chosen based on some criteria like the cost estimation for this particular query tree and so on. So using this, the logical query plan is generated. The logical query plan of course is re written like I said and we optimize, the dbms optimizes the query automatically to a certain extent based on rewriting the query.

Then from this the physical query plan is generated. The physical query plan is a plan written in an intermediate language that is either interpreted that is executed directly by dbms or is compiled into machine code. And in the last two sessions we saw what would be the or what are the typical building blocks of this physical query plan language. There are, the physical plan of course should support all kinds of logical query operations like select, project and so on.

In addition, it should also support some physical aspects of query processing like how to iterate through different tuples in a given relation or things like sort scan, table scan and index scan and so on. How do we retrieve based on a particular index and so on. We also saw some algorithms that implement the internal queries that is if you remember the internal queries are formulated or all those queries that are formulated by the physical query plan language that is it is the query that the dbms uses to access data from the file system. The external query is the query that the users use or the application program uses to access data using the dbms. We saw different kinds of access algorithms based on the physical query plan constructs. We saw different, we basically divided this algorithm into one pass algorithms and multi pass algorithms. One pass algorithms are those algorithms which make at most or exactly one pass over the required relation but one pass algorithms have a limitation in the sense that if we are using a one pass algorithm for a relation at a time operator.

What is a relation at a time operator? An operator that requires the entire relation to be present in order to answer the question that is being posed by the query. For example operators like removing duplicate that is the unique operator in sql or order by or group by in sql and so on. So all this require the entire relation to be available before the query is answered. Unless the entire relation is processed, even the first tuple of the answer cannot be returned. Even before answering the first or outputting the first tuple of the result, the entire relation must be processed at least once.

(Refer Slide Time: 06:10)



In single pass algorithms, we can apply single pass algorithms only if or if we use let us say tuple at a time queries where we need to be concerned only with one tuple at a time rather than the entire relation at a time or we can use it for relation at a time queries as long as at least one of the relations can be fit into memory completely. That is it not only fit into memory but there should be more memory space left over for at least one block of the other relation if we are using any binary operator.

In the previous session we saw what are called as two pass algorithms and we also said that multi pass algorithms are basically generalizations of the two pass algorithms. Two pass algorithms are used when, of course they are primarily used in relation at a time queries because tuple at a time query do not need two pass algorithms. We can just use a one pass algorithm. So a two pass algorithms are used for relation at a time queries where the relation size is too big to be able to fit into memory. And we saw the basic structure of a two pass algorithm. The basic structure of a two pass algorithm has an alternating computation and intermediate storage handling phases.

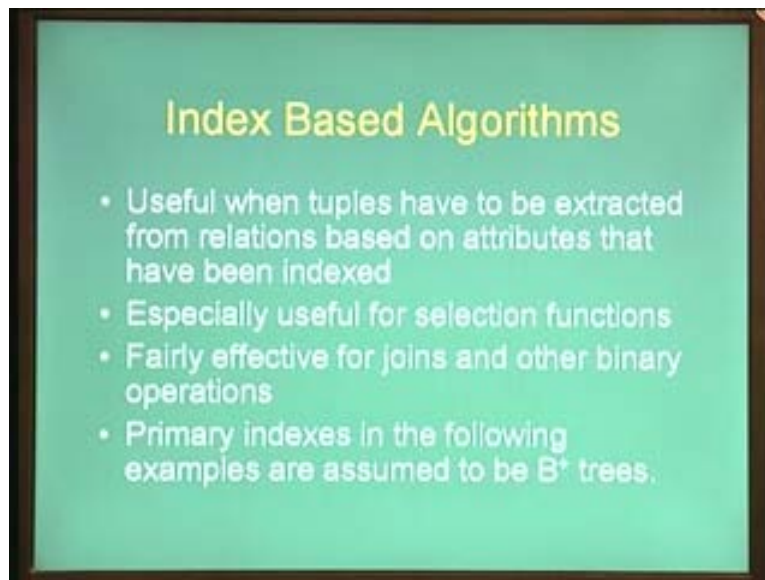
That is take one set of blocks from relation, perform some computation on them like sorting or hashing or indexing which we are going to see today and then write them back into disk. And then read back all this intermediate results before producing the final result. So we divide a two pass algorithms into three different strategies or three different paradigms so to say what are called sorting based methods which we saw in the previous session where the computation that is done when a chunk of blocks is read from a relation is the sorting function. That is a chunk of blocks is read from each relation and they are sorted and they are placed on to disk.

The next kind of algorithms that we saw was the hash based methods where we read a chunk of blocks from the relation and start hashing each tuples based on certain criteria. If it is a natural join for example we hash it on the common attribute or if it is a group by

or if it is a unique function then we hash based on the entire tuple and so on. Today we are going to look at the last method or we are going to see just an overview of the last method namely the index based methods. That is where the computation that is used is an indexing function that is adding or searching on a index. And for our purposes, we are going to assume a sorted index like a B plus tree.

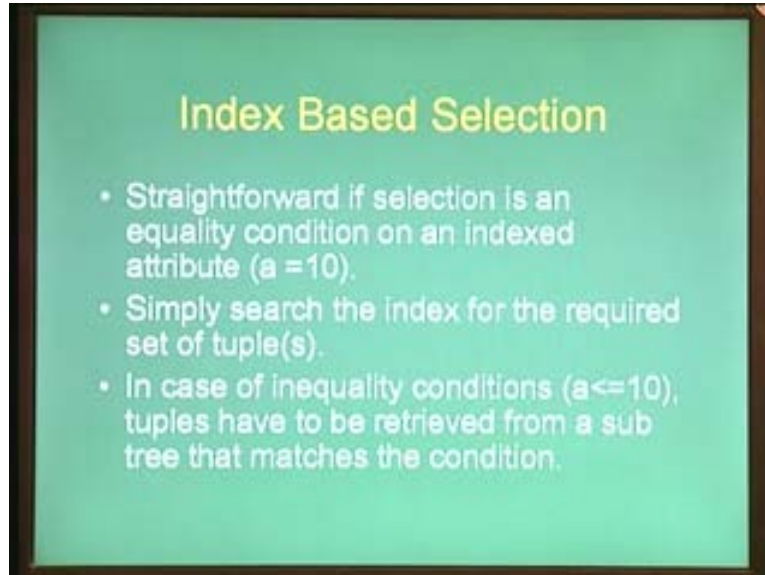
A sorted index is something like where we can use the index structure not only to retrieve a tuple based on a key value but we can also retrieve tuples based on a sorted order of all the key values that are present in the relation.

(Refer Slide Time: 10:24)



So what are index based algorithms? Index based algorithms can be contrasted from sorting and hash based algorithms in the sense that they use a index instead of sorting and hashing for the computation phase. And they are useful when tuples are to be extracted on attributes that have been indexed and they are especially useful for selection functions. Especially of course the, that is when the attributes that are being searched for index. And they are fairly effective for joins and other binary operations. As you will see there is another join function called zig zag join which is also quite popular as hash join and in terms of efficiently computing joins between two relations.

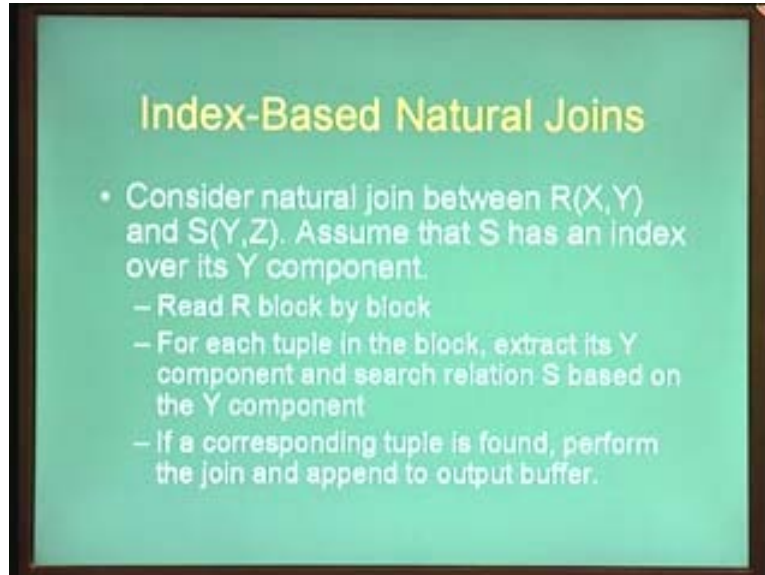
(Refer Slide Time: 11:19)



Let us look at the first algorithm in index based methods. How do we perform select or how do we perform a select operation using an index? We are of course assuming here that the select operation involves a condition that is over an attribute that has been indexed. Of course, if the select is over a condition which is not been indexed then we cannot use index base selection, we should either use one of the other methods that we have seen. Now it is fairly straight forward if the selection condition is an equality condition on an attribute because that is what indexes are meant for. Let us say searching a B tree based on the value of a key is simply saying that select key equal to this value from the relation on which the B tree is maintained.

So the simple algorithm is to search for the index for the required set of tuple or tuples. In case of inequality conditions that is something like a less than or equal to 10, we have to retrieve a sub tree from the B plus tree index that is we cannot retrieve just one node of the index. And we are also making another assumption here that the equality and inequality condition involves the key under constant and not another attribute that is we are saying a equal to 10 or a less than or equal to 10. We are not saying something like a equal to b where b is another attribute which has not been indexed but and we have to search for all tuples in such a case where the **index indexing** indexed attribute is equal to some other attribute that has not been indexed.

(Refer Slide Time: 13:15)

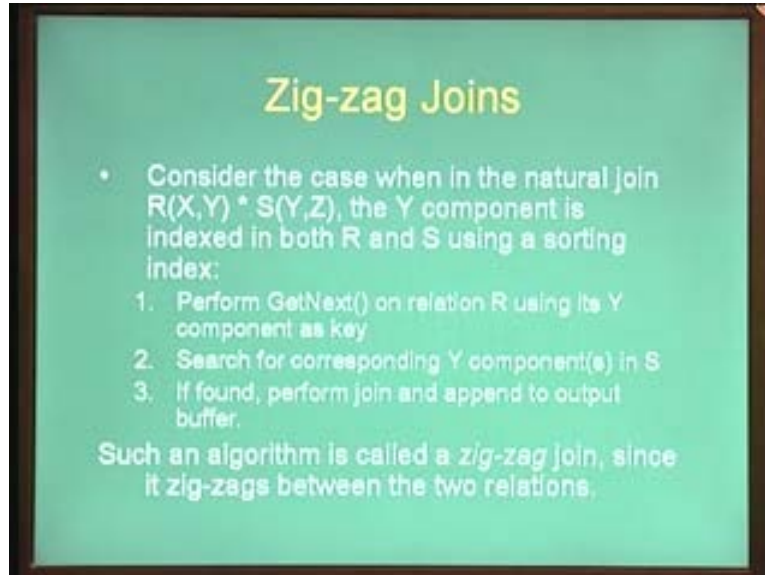


We shall not be covering all the other algorithms say set theoretic algorithms on index based methods. After we have seen the general pattern of how set base operators like union, intersection and set difference are performed using sorting and hashing, it is fairly straight forward to design algorithms using index based methods also. Let us look at the last query that we looked at the other algorithms namely joins or natural joins. How do we perform natural joins using index based algorithms? Now consider a natural join as usual between two relations $R(X, Y)$ and $S(Y, Z)$ where Y is a set of attributes that are common between R and S . That is Y is the attribute over which the natural join is going to be performed.

Now assume that suppose, that is suppose Y is not only indexed that is the Y attribute is not only indexed, it is also the primary key in S . Assume that S has as index over its Y component. If it is a primary key then it becomes even more simpler but even if it is not, it is just an indexing attribute or set of attributes we can use the following algorithm. We just start reading R that is the first relation in this chunk of blocks that is chunk of M blocks. So read R block by block and for each tuple that we have read from R extract its Y component and search relation S based on the Y component.

That is we start sequentially reading one of the relations and for each block that we read we do a index search and because index search is much faster, we find the set of all relations let us say if it is a primary key then we find exactly one relation and if it is not a primary key we have to use some kind of clustering index and we find a set of tuples that may contain this value of Y that we are looking for. Now all that we have to do is join this tuple with how many ever tuples that we have found in S and we have computed the natural join over R and S . So if a corresponding tuple is found then perform the join and push this to the output buffer. Using index based methods we can also perform another kind of natural join which is called a zig zag join.

(Refer Slide Time: 16:05)

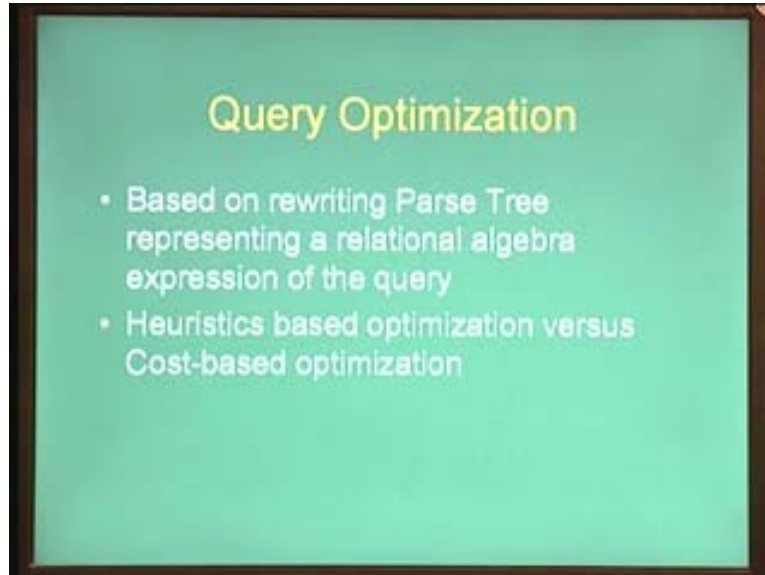


The zig zag join can be performed when there is a sorting index that is available for both R and S. That is assume that we are performing a natural join over two relations R and S where R is (X, Y) and S is (Y, Z) where Y is the common set of attributes between R and S. Now also suppose not only that Y is indexed over S, it is also indexed over R and this index is a sorting index that is it is a something like a B plus tree where I can access all keys available in the relation in a sorted form.

Now all that we have to do here for performing such a join is shown in the set of three steps in this slide. We just use or we just keep calling getNext function, note that we can represent relations R and S as iterators. So we just open both relations in iterators and let us say we use R and start calling getNext function on the iterator, R iterator.

Now this get next function get the next logical key or the next sorted key in R and for each key that is found in R, we start searching for corresponding tuples or matching tuples in S. And if matching tuples are found, perform the join and append the output to buffer. Such an algorithm is called a zig zag join. This is because even though logically we are going in a sorted form physically, the control would actually be going in a zig zag fashion over on a storage disk. That is we don't know or we are not sure that the sorted form in which we are accessing the keys is indeed the same form in which records or tuples are stored on disk. So such an algorithm is called a zig zag join. However it is quiet an efficient algorithm because we are using index on both set of attributes that is both R and S.

(Refer Slide Time: 18:24)

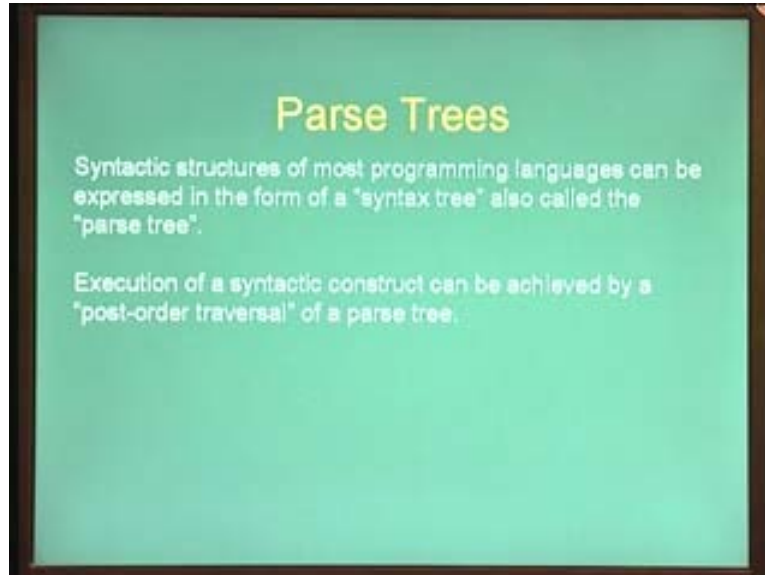


Let us move on to the main topic of today that of query optimization. Until now we have been looking at the last step in a query execution process that of managing the physical query plan languages. That is how do we perform internal query answering using physical query plan languages. Let us move on to the next or the upper level before this that is of logical query plans that to primarily that of parse trees that are generated after a query is scanned and parsed by the query compiler.

We are going to see whether or how or whether we can re write a parse tree in a sense that we can make the query answer in a much more efficient fashion than it was formulated by the user or the application program. So query optimization techniques that are independent of the semantics of the query or the semantics of the application is what we are going to be looking at here. And these techniques are based on rewriting the parse tree representing a relational algebra expression of the query. That is the parse tree would actually convert a given sql query into a relational algebra and then represent it in the form of expression tree which can be optimized. And there are two kinds of optimization, what might be termed as heuristics based optimization and cost based optimization.

A heuristics based optimization is essentially a set of thumb rules using which we make assertions that the resulting parse tree is a better or more efficient than the original parse tree. However given a parse tree usually there might be more than one parse trees that could be generated based on which heuristics are applied and in what order they are applied. Now among these different parse trees, we might have to choose one of the parse trees for a particular query instance. And we can perform this choosing by assigning trying to estimate cost that each of the parse tree incurs. So usually in practice a combination of heuristics and cost based optimization are used before a query execution plan is finally chosen for execution.

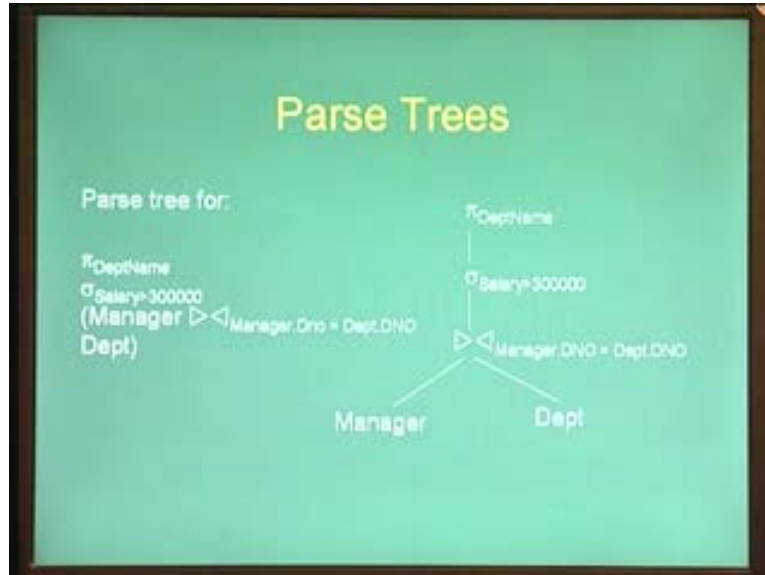
(Refer Slide Time: 21:04)



Now we have been talking about parse trees and rewriting parse trees. So what are parse trees? A parse tree as you might have come across especially in the context of compilers or syntactic structures that of most programming languages that can be expressed in the form of a tree structure or tree data structure. A tree data structure represents a hierarchical structure. This is also called a syntax tree or a parse tree and execution of a syntactic structure is usually done by what is called as a post order traversal of the parse tree. That is a post order traversal simply says that given a tree having a root or 2 or more sub trees or 1 or more sub trees. Execute the sub tree first and then execute the root. And the same rule recursively applies to all of the sub trees. We shall not be going into details of how to build a parse tree and a traversals and so on but we shall be concentrating on how to modify a parse tree. That is in the context of relational algebra that is context of queries that are expressed relational algebra.

This slide shows an example of a parse tree. the left hand side of the slide shows a small relational algebra query which says project department name from select salary greater than 3 lakhs from again manager join department where the join condition is manager dot D number equal to department dot D number. That is it is a natural join between the manager and department relations. And from this natural join we are selecting the set of tuples where the salary field is greater than 3 lakhs and then projecting the department name. That is we are looking, we are querying for all departments who or which pay their managers a salary greater than 3 lakhs.

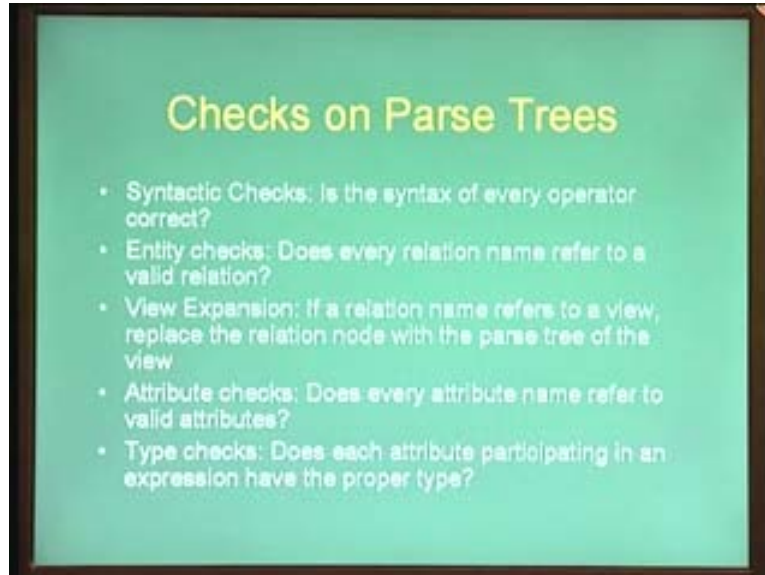
(Refer Slide Time: 22:20)



The corresponding query can be represented in the form of a tree data structure that is shown in the right hand side of the slide. As the tree shows the top most operation project becomes a root node of the tree. Now the project is a unary operations therefore it has one child which is the select query. So the select query becomes the child of the project query. The select again is unary operations which is being performed on the join operator here.

So the join operator becomes a child of select and the join operator is a binary operator where it has two children and these two children are its two arguments that is manager and department relations. So, logically as you can see, the queries expressed the hierarchical structure that is inherit in the query is made explicitly by using the parse tree. Once a parse tree is generated by a query compiler, there are several kinds of checks that are performed on the parse tree before it is optimized. some of these checks are shown in this slide here.

(Refer Slide Time: 24:31)



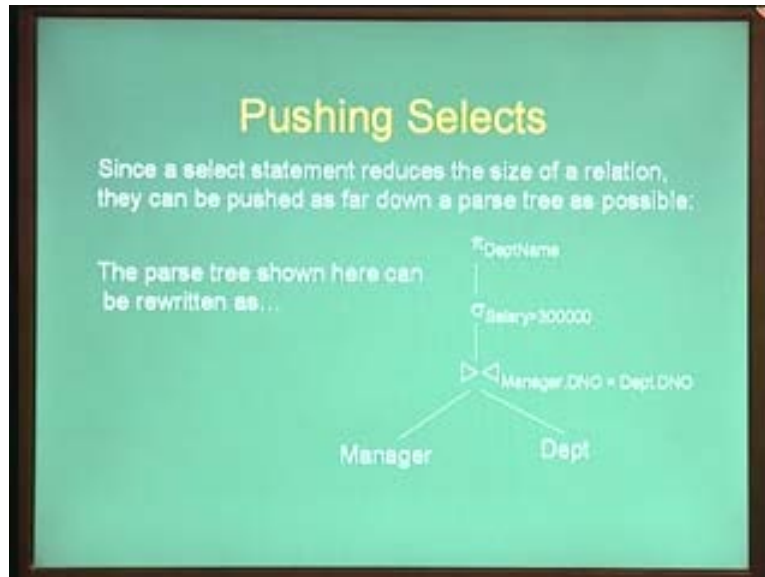
Syntactic check: The syntactic check simply says that is the syntax of every operator correct. For example we cannot have two children on a project operator, project and select are unary operators. and similarly we cannot have a single child on a join operator, it is a binary operator and so on. Entity checks: entity checks basically check whether every relation name that is specified in this query actually exists. That is, **is it there on disk**, that is, is there a relation called manager and department on disk and so on. In some cases a relations may not, a relation that is named in a query may not actually exist on disk but it could be derived that is it may not be a base table but it could be a derived view that is it could be a view that is there in the schema. In such a case the compiler would expand the view. Remember, a view is again another query and the contents of the view are not actually on the database.

So the view is again expanded and the parse tree for the view is joined or is hooked to the parse tree of the overall query wherever the view name appears. So this is called view expansion and then there are attributes checks that is it does every attribute name refer to valid attributes and then there are type checks. That is, does each attribute participating in an expression have a proper type that is we cannot say something like salary less than cats and so on.

I mean it has to be, if salary is numeric then the other attribute should also be numeric and so on. So there should be type compatibility between attributes within an expression. Once these checks are performed, parse trees are rewritten based on certain heuristics. So these heuristics are also called as rewrite rules and which specify several conditions and corresponding actions that need to be performed when these conditions hold. and a parse tree should be expanded to its maximum extent before rewriting that is for example view should be, view should be replaced by the relevant parse trees when the parse trees is being, before the parse tree can be optimized.

And some rewrite rules could be situation specific and we are not going to be looking at such rewrite rules here anyway. And they work only if certain conditions hold on the data set that is they make certain assumption about the dataset based on which the parse trees can be rewritten.

(Refer Slide Time: 27:35)

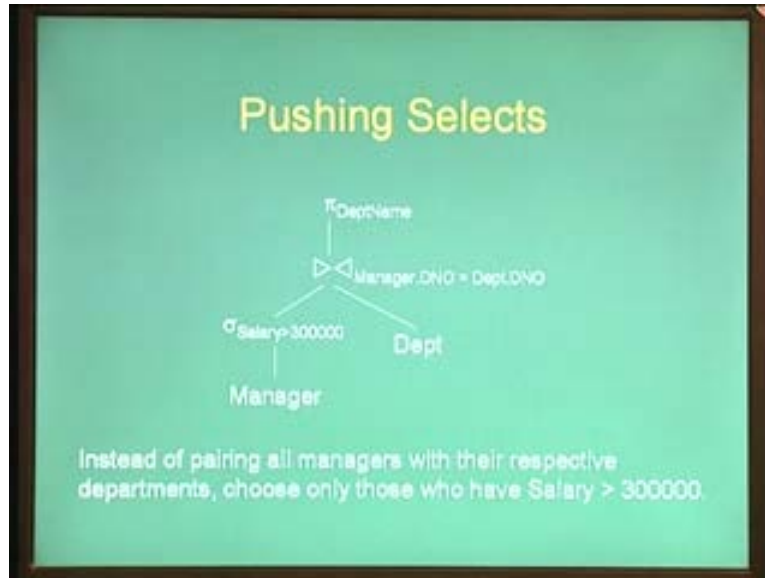


The first rewrite rule that we are going to see today is what is called as pushing selects. The pushing selects basically says that try to push a select operation as low down in a tree as possible without altering the semantics of the parse tree or without altering the correctness of the parse tree. Why is this important or why is this beneficial? Remember a select or the cardinality of a select operation is less than or equal to the cardinality of the input relation for the select. That is a select basically removes certain tuples from a relation before returning the output. And one thing to note here is that whenever we talk about relations, always think of very relations that is megabytes or probably giga bytes of tuples present in a relation. And if a select operation is such that it requires only a small fraction of the tuples there is a great amount of optimization that is performed already. That is huge number of data need not be handled after the select is over and we can start working with a much smaller dataset.

So, select based optimizations are the most common optimizations that are used in query rewriting. So the slide here shows an example of pushing selects based optimization. let us look back at the parse tree that we just generated previously that is a project department name from select salary greater than 3 lakhs from natural join between manager and department. Now here you can see that the select operator here that is salary greater than 3000 is being performed after all possible managers or join with all possible departments. Now if we are going to anyway look at a managers whose salary is greater than 3 lakhs, we don't have to consider joining all possible manager tuples with all possible department tuples.

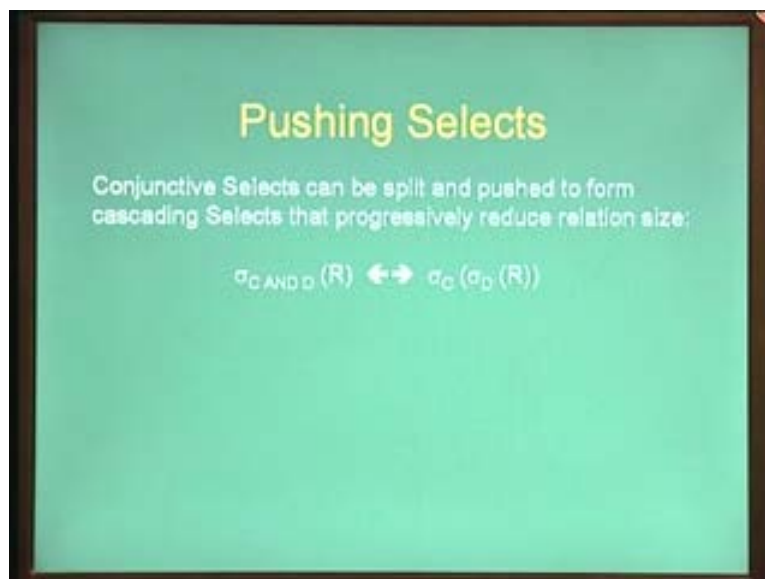
You might as well say that we are going to join tuples of manager where the salary field is greater than 3 lakh. So that is what is performed in this tree here.

(Refer Slide Time: 30:06)



That is the select operation is pushed down in the tree so that first we select the set of all manager tuples where salary is greater than 3000 and then join this with a greater than 3 lakhs and join this with the corresponding tuples of the department relation and then project just the department name. so if, let say if there are about 1000 managers in a company and only about 50 mangers have a salary greater than 3 lakhs then instead of joining or instead of looking at 1000 different manager tuples, we need to content ourselves with only 50 manager tuples.

(Refer Slide Time: 30:58)

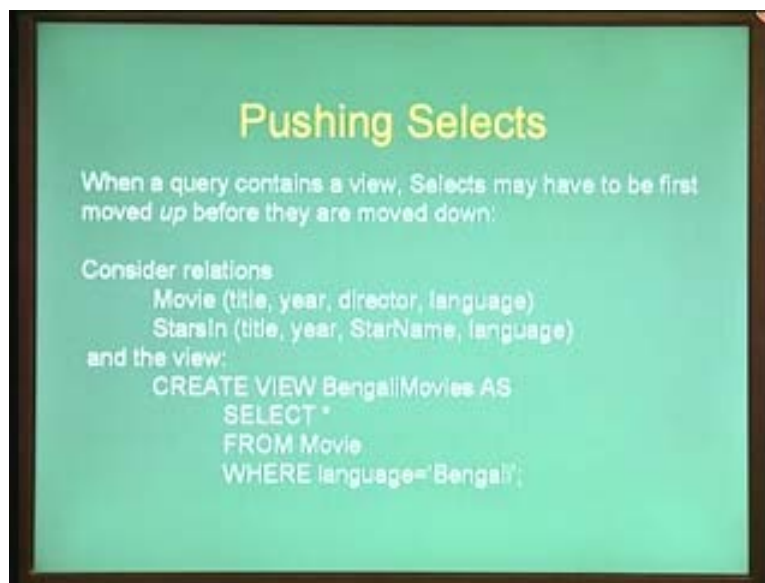


The second form of rewriting rules is also related to pushing selects and which is called the cascading select or conjunctive selects. That is a select which has a conjunction can be split and cascaded into several different select operations and which can progressively start reducing the search space as we go along. This slide shows a small example which illustrates this point.

Suppose we have a select operation that says select C and D over R that is select where C and D here are some logical condition. Now we are selecting those sets of tuples from R where both condition C and condition D holds. If this is just the relation here, now this could be, the conjunction could be even more that is it could be C and D and E and F and so on. There could be many more AND conditions here. now if we leave this as it is, then the entire set of relations are searched for this condition that is the entire set of conditions are matched on the entire set of relations.

On the other hand we can see that the right hand side of the relation or the right hand side of this equivalence condition says that select C from select D from R. that is it is a cascading select that is shown in the slide here where instead of using C and D over R, we first say select D over R that is this could be much more efficient if D is an indexed attribute over R. So if D is an indexed attribute over R or rather D is an condition over an indexed attribute over R, we can quickly or efficiently retrieve all those tuples that match the condition D. From this much smaller set or hopefully much smaller set of tuples that have been extracted, we then perform a select C or we then look for the condition C. So the search space for C is much smaller than the search space of D and if D is a condition over an indexed attribute, the search space of D or the inner select can also be very fast that is because we are retrieving based on index searches.

(Refer Slide Time: 33:41)

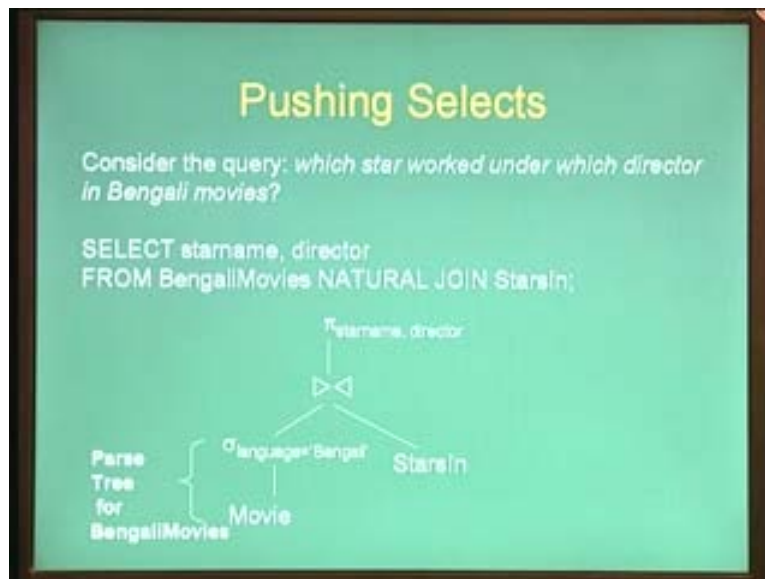


There is an exception however to the pushing select thumb rule that is push selects as far down a relation as possible thumb rule and this exception occurs whenever there are

views that are there in a relation. When a view is expanded, it might be necessary sometimes that selects are actually pushed up beyond the view before they can be pushed down. Let us look at an example to illustrate this. This slide shows an example having two relations movie and Starsin. Movie is a relation that has the following attributes,

The title of the movie, the year of the release of the movie, the director of the movie and the language in which the movie is made. And then Starsin is an attribute that says which film stars acted in which movie. It says it has an attribute title which is the title of the movie the year which is the year in which the movie was released, the name of the film actor and the language of the movie. And now of course we can see that among the two relations title, year and language are common. That is we can perform a natural join using the three different attributes. And now we create view called BengaliMovies where which just says select star from movie where language equal to Bengali. That is we are interested only in those tuples of the movie where language equal to Bengali.

(Refer Slide Time: 35:27)



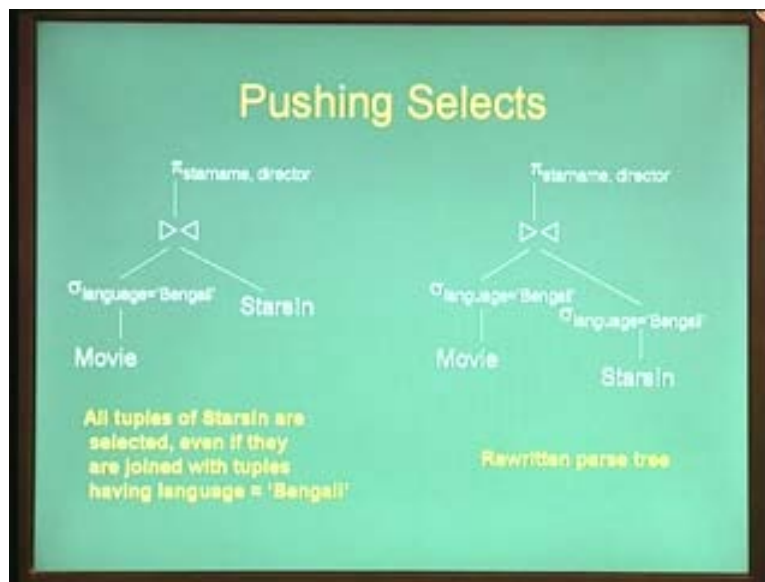
Now let us see if we give a particular query and what happens. Let us say we have a query which says which filmstar worked under which director in Bengali movies that is we have to bear each film star with a director as long as the language of the movie is Bengali. So how do we go about or how do we express this query. This is quite simple from a sql point of view that is we just say select starname and director from BengaliMovies NATURAL JOIN Starsin.

That is we join we perform a natural join between BengaliMovies and Starsin. note that Bengali movies also have the same structure as the relation called movies. That is you can join based on title, year and language attributes. So the corresponding parse tree for this is also shown in the slide here that is we are projecting starname and director from a natural join between BengaliMovies and Starsin. However this relation Bengali movies is a view.

So we have also expanded the view here. The view for BengaliMovies is basically select language equal to Bengali from movie. So this view is expanded and Starsin is a base table which is kept as it is in the parse tree. Now, if you see here even though the left hand side of this tree that is the parse tree that talks about Bengali movies contains tuples where the language field is Bengali. It does not contain tuples having any other language field. However while performing the natural join we are still considering all tuples in Starsin even if it does not contain the language called Bengali.

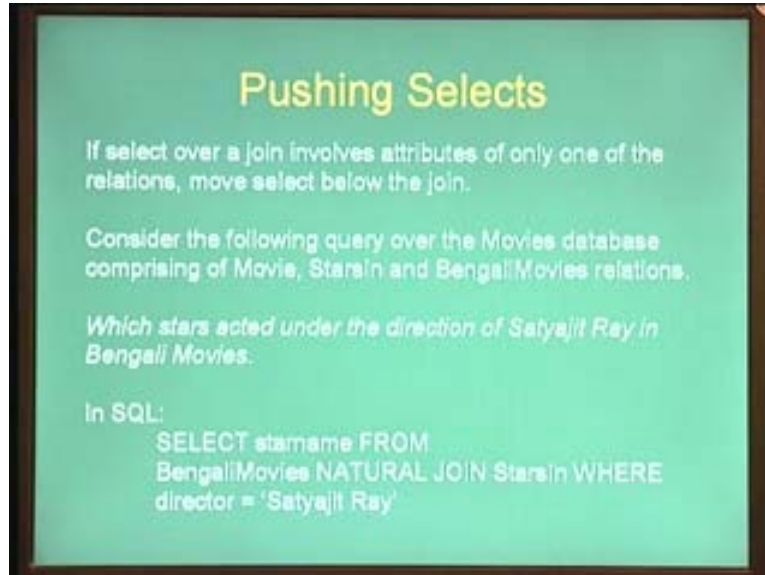
That is we take a particular actor and we look at the set of all his or her movies regardless of whether the language is Bengali or not. This is clearly wasteful. Because after all we are only interested in which star worked under which director in Bengali movies. So, one way to optimize this is to put a select operator here. That is we need to select those tuples from Starsin where language equal to Bengali. Therefore this select operation here which is in a view or which defines the view should first be taken up before it is brought down. This is what is shown in the next slide here.

(Refer Slide Time: 38:08)



That is all tuples of Starsin are selected even if they are joined with tuples having language equal to Bengali. So here we just select all tuples which is not really necessary. Therefore we need to take this select tuple that is the left hand side of the relation selects a particular or performs a particular selection condition which is also applicable to the right hand side of this sub tree. So we move up, we move this selection relation up as much as possible before moving it down and to bring it here. That is so this would be the rewritten parse tree. That is there are two separate select statements one for movie and one for Starsin which finally forms the query.

(Refer Slide Time: 38:58)



Pushing Selects

If select over a join involves attributes of only one of the relations, move select below the join.

Consider the following query over the Movies database comprising of Movie, StarsIn and BengaliMovies relations.

Which stars acted under the direction of Satyajit Ray in Bengali Movies.

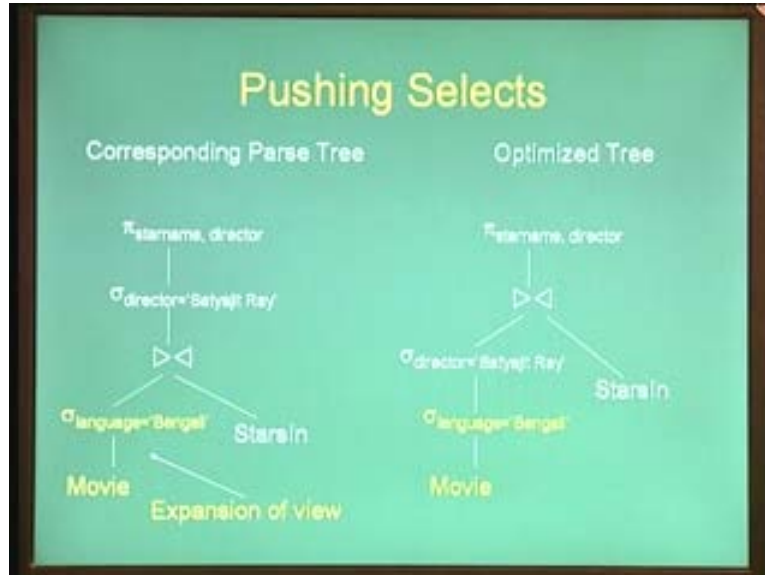
In SQL:

```
SELECT starname FROM
BengaliMovies NATURAL JOIN StarsIn WHERE
director = 'Satyajit Ray'
```

Some more thumb rules involving selects. Some times in a join function we perform a join blindly and then select a set of tuples from this joined set of tuples based on a condition that applies to only one of the relations. We have actually seen such a example earlier but let us revisit again in order to make this, bring out this rule. We didn't apply this rule however because we applied some other rule. Now consider the following query over the movies database. Again the same movies database comprising of movie Starsin and Bengali movie relations.

Now the query is we are only looking for which actors or which stars acted under the director Satyajit Ray in Bengali movies. We are not just pairing up all sets of actors with all sets of directors under Bengali movies, we are just looking for who all acted under the direction of Satyajit Ray in Bengali movies. So the sql statement for this again is quite simple. We just say select starname from BengaliMovies NATURAL JOIN Starsin WHERE director equal to Satyajit Ray.

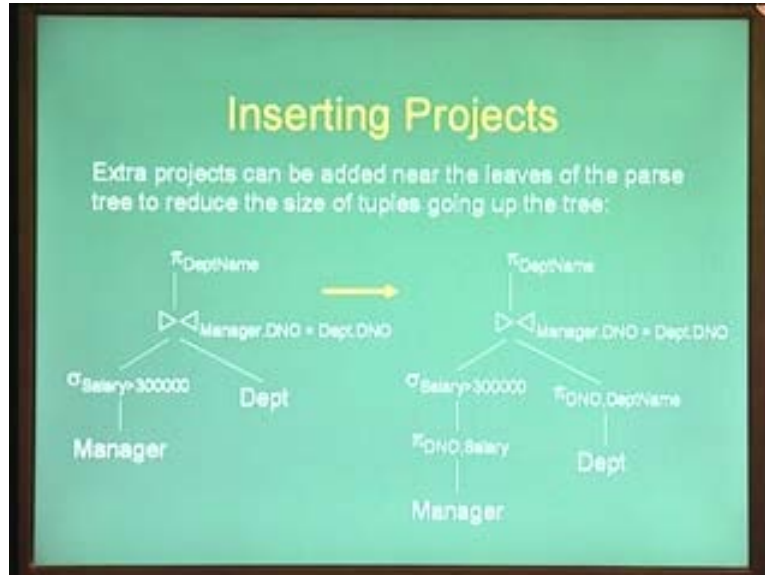
(Refer Slide Time: 40:30)



The corresponding parse tree if you build the corresponding parse tree here this would look like this. At the lowest level is the join operator between BengaliMovies which is expanded here that is shown in yellow and the Starsin relation. Now after these two relations are join, we select for director equal to Satyajit Ray and then project starmame and director name. Of course we need we need to only star name, there is no need to project director.

So what we are doing here is that we are joining two relations and even if let us say that even if we move this select up and bring it here, we are still joining two relations based on just the factor that language equal to Bengali. However we can note that in the second relation here, we are finally interested in those tuples where the director field equal to Satyajit Ray. We are not interested in any other directors whose records are available in this relation here. So we can as well move this director equal to Satyajit Ray select below the join here that is which is above the join here is now below the join function. And of course in addition to this we can also perform the previous optimization which is not shown here that is move this language equal to Bengali up and bring it down so that it comes before Starsin. So there are two different optimizations that are possible in this particular query tree.

(Refer Slide Time: 42:17)



The next kind of thumb rule we are going to be, that we going to consider is what is called as inserting projects. Now note that if the output of a query involves projection of just a few attributes over a large relation containing let us say tens of attributes or different or tens or even sometimes of hundreds of attributes, a large tuple containing many different attributes, there is no need to work with so many tuples or so many attributes when all we require is just what is given by the overall projects and the conditions that of select operations that are given in the query.

Have a look at the examples shown in the slide here. the left hand side of the slide shows a query tree which says going back to our manager and department example, project department name from that is select salary greater than 3 lakhs from natural join between manager and department. That is this is also optimized that is the select operator which is here is now brought here so that it is optimized.

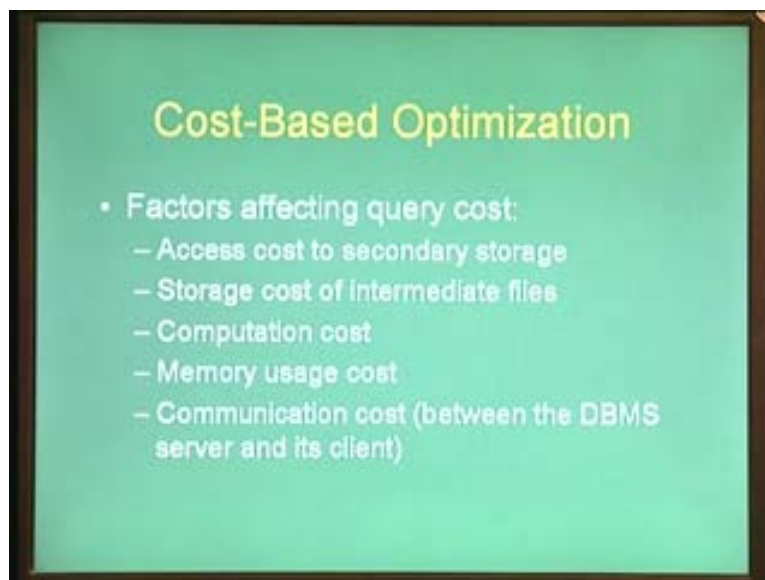
Now if we notice the output of this tuple is just the department name, the output of this query is just the department name. The user or the application program is not concerned about any other field of the database. That is anything like the department id or the number of people in the department, the location or whatever else that is there in the department relation that is not really important.

However we cannot just throw away all other attributes other than whatever is requested because these attributes may be required for some conditions. Now which are the other attributes that are required for some conditions. Here that is department dot dnumber equal to manager dot dnumber is another condition that is required for the natural join that is shown here. So, the only set of attributes from department that we are concerned about here are the department name and dnumber that is the department number.

Similarly the only set of attributes that we are concerned about for the manager relation is just the salary and the department number. we are not concerned about the name of the manager, the age of the manager, the date of birth, the employee number, the address, nothing. So what we can do is we might as well insert extra projects down in the query tree so that the output relation becomes smaller and more focused to what is required for as part of this query.

That is we have inserted a project dnumber and salary before manager because those are all the fields that we require above and similarly we have inserted a project dnumber and department name because those are all the attributes that are required for these query from the department relation.

(Refer Slide Time: 45:42)

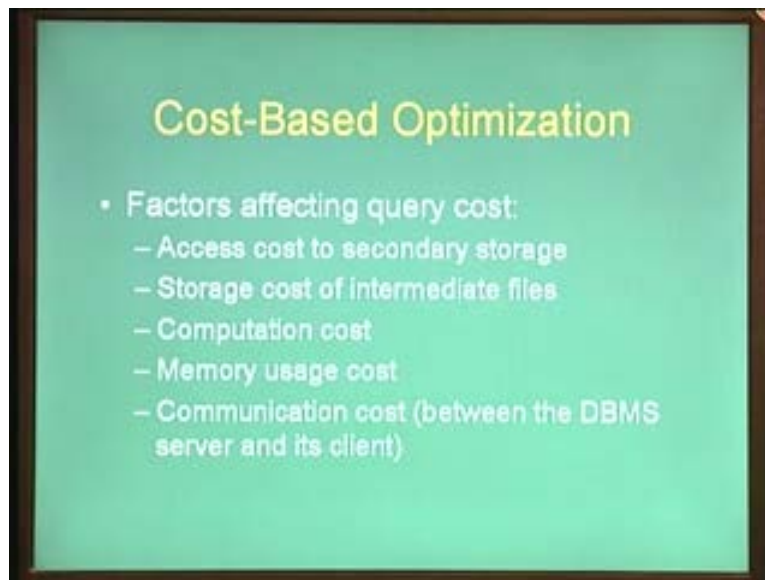


So that was the brief overview of some of the thumb rules that can be used to rewrite query parse trees in order to make them work better. Let us look at the other aspect of query optimization that of cost based optimization. A cost based optimization is essentially used to choose between one or more or choose between two or more different candidate parse trees. that is given a query parse tree if I have used the thumb rules and generated several other candidate parse trees, each of which claim to be more efficient which is the one that I have to use.

In order to do that we assign a cost for each of the different components or a cost estimate for each of the different components of a parse tree. And the overall cost of a parse tree is an algebraic expression over each of these cost estimates. It is either a sum of all these cost components or multiplication or whatever depending on what exactly is the tree. Now what are the factors that affect execution cost? This slide shows some possible candidates that affect execution cost, say access cost is second storage.

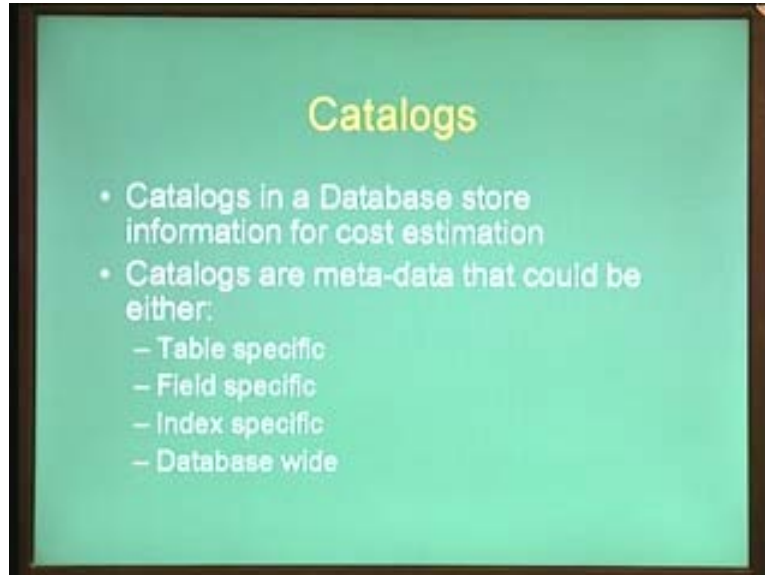
How costly it is to access data from disk or any other storage medium that we are using. then what is the storage cost, how much storage do we need especially for storing the intermediate files and what is the computation cost, how much of processor time that we require and so on. What is the memory usage cost, how much memory, primary memory or ram does it take. And how what is a communication cost especially between the dbms server and the client, if they are situated on different machines. And communication cost becomes very profound, if we are considering distributed database that is how many communication sequences are required between the different dbms servers that form this distributed databases. So all of these factors affect the overall query execution cost.

(Refer Slide Time: 47:59)



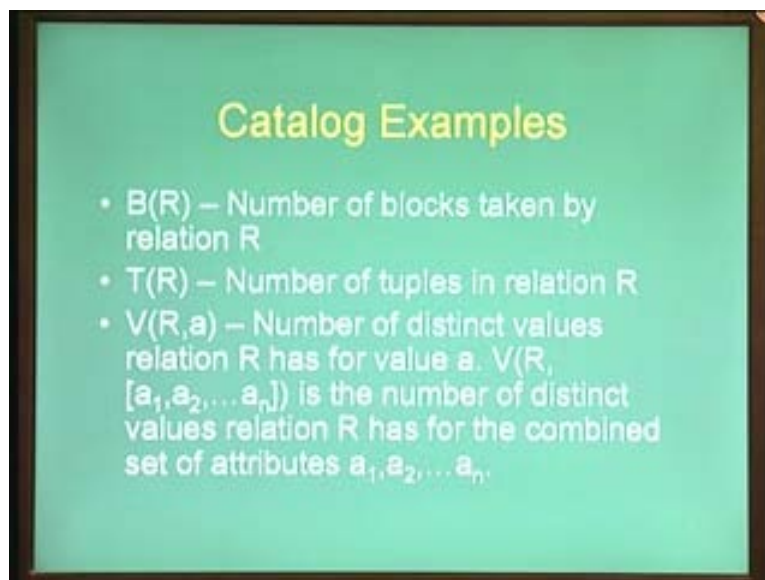
A database catalog is a set of value estimates or some kind of meta data or meta information that are stored in a dbms that are useful for cost estimation.

(Refer Slide Time: 48:13)



And catalogs are metadata that could be either say table specific metadata like an estimate of the number of tuples in table, the size of the table and the number of blocks that are occupied by table and so on. And they could be field specific like the number of distinct values of a particular field, an estimate of those numbers of different distinct values of a field and so on or they could be database wide tools or they could be index specific information and so on. So let us look at some typical kinds of information that is stored in a catalog and see how we can estimate the cost of different operators.

(Refer Slide Time: 48:57)



Some typical information that are stored or depicted here let us say B of R is the notation used for the number of blocks that are taken up by a relation R . Similarly T of R the number of tuples that exist in a relation R and similarly V of R, a is a field specific attribute that is it is an estimate of the number of distinct values that attributes a has in relation R . And of course for example if attribute a is the gender then the number of distinct values is only 2. On the other hand other hand if attribute a is something like distance between something and something else, it could take on a range of several other values. And of course V of R, a_1 till a_n is the set of is the number of distinct values of the combined tuple or that is formed by a_1 to a_n .

(Refer Slide Time: 49:57)

Cost Estimation Examples

Estimating the cost of selection.

Consider a select of the form: $S = \sigma_{A=c}(R)$, where

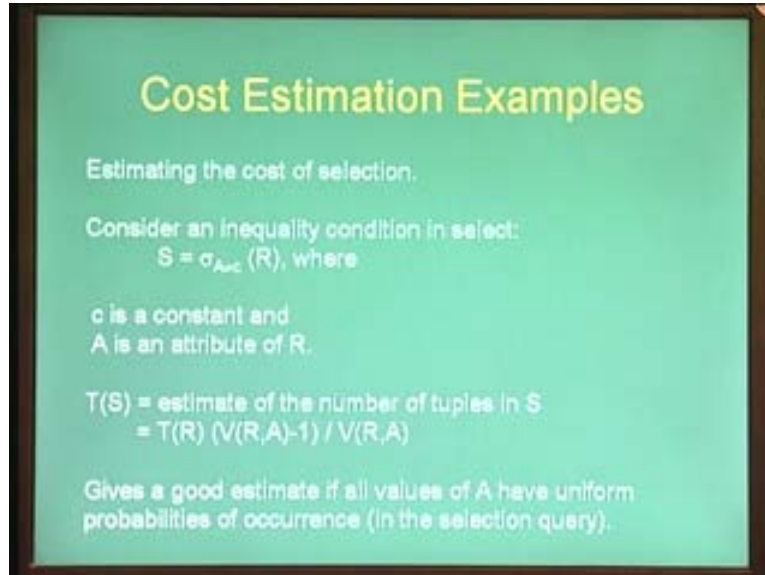
- c is a constant and
- A is an attribute of R .

$T(S)$ = estimate of the number of tuples in S
 $= T(R) / V(R,A)$

Gives a good estimate if all values of A have uniform probabilities of occurrence (in the selection query).

Let us look at some simple cost estimation techniques and there are many more cost estimation techniques other than this but we are only looking at examples that show the bigger picture behind this, the representative example. So suppose we have an equality selection that is select, that is s equal to select A equal to C from R where A is an attribute name and C is a constant. Now A is an attribute name and C is a given constant and we have an estimate of the number of distinct values that the attribute can take which is given by V of R, A then the probability of A equal to C is simply 1 over V of R, A and because the number of tuples is T of R , the slide here shows the estimate that is T of A is equal to T of R divided by V of R, A which is an estimate for the number of tuples that are there in S . And this is a good estimate if all values of A have fairly uniform probability or that is in the selection query. Now if the dataset is queued then it may not really be a good enough estimate.

(Refer Slide Time: 51:27)



Cost Estimation Examples

Estimating the cost of selection.

Consider an inequality condition in select:
 $S = \sigma_{A \neq c}(R)$, where

c is a constant and
 A is an attribute of R .

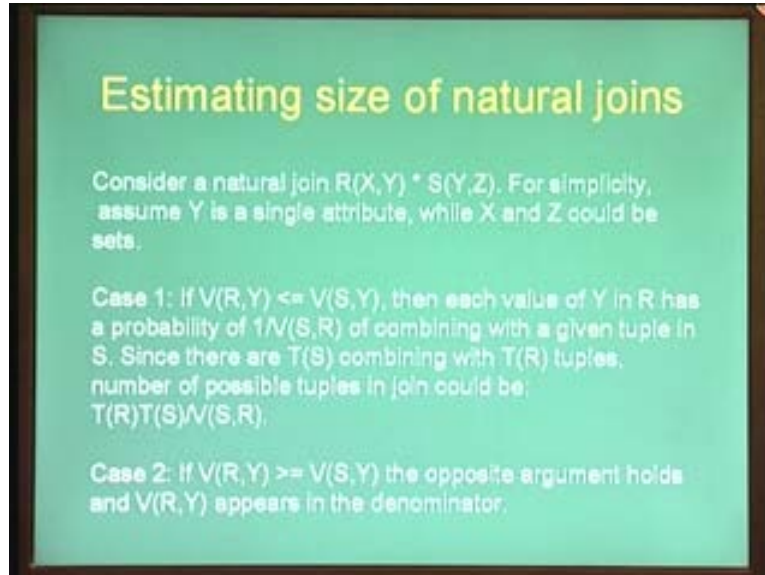
$T(S)$ = estimate of the number of tuples in S
 $= T(R) (V(R,A)-1) / V(R,A)$

Gives a good estimate if all values of A have uniform probabilities of occurrence (in the selection query).

Consider the inequality condition that is select A not equal to C from R which is assigned to S and C is again a constant. now this is again quite simple because suppose the number of distinct values of A is given by V of R, A then the probability that A is not equal to C is simply V of R, A minus 1 divided by V of R, A that is 1 over 1 minus 1 over V of R comma A . Now multiply this with T of R which gives us an estimate of the number of tuples that could be in S . What about a composite condition something like select C or D that is condition C or condition D over R and assign it to S .

Now let us first, suppose we have estimated that P tuples in the relation satisfy condition D condition C and q tuples satisfy condition D and there are n number of tuples in the relation. now the probability that a given tuple will match C or D is shown here that is 1 minus, the multiplication of 1 minus P over n and 1 minus q over n . That is this is the probability of a tuple not satisfying C and this is the probability of tuple not satisfying D and the multiplication of this is the probability of tuple not satisfying both C and D and 1 over this or 1 minus this is the complement of this. This is typically the De Morgan's law of which says, which gives us the probability of tuples that satisfy C or D . Now multiply this with N or the number of tuples which gives us the size estimate of the query.

(Refer Slide Time: 53:29)



Estimating size of natural joins

Consider a natural join $R(X,Y) \bowtie S(Y,Z)$. For simplicity, assume Y is a single attribute, while X and Z could be sets.

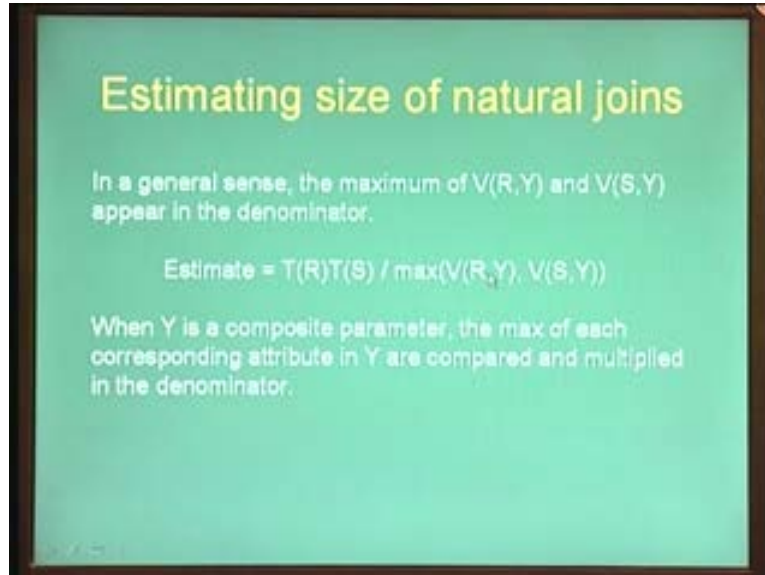
Case 1: If $V(R,Y) \leq V(S,Y)$, then each value of Y in R has a probability of $1/V(S,Y)$ of combining with a given tuple in S . Since there are $T(R)$ tuples in R , the number of possible tuples in join could be: $T(R) \cdot T(S) / V(S,Y)$.

Case 2: If $V(R,Y) > V(S,Y)$ the opposite argument holds and $V(R,Y)$ appears in the denominator.

The last estimate that we are going to look at is trying to estimate the size of a natural join. Consider a natural join between R of X, Y and S of Y, Z . Initially for simplicity let us assume that Y is a single attribute, all though X and Z could be composite attributes that is their sets. Now also let us assume that V of R, Y that is the number of distinct values of Y that are in R is less than the number of distinct values of Y that are in S . Now each tuple in R can be combined with corresponding tuple in S with the probability of 1 over V of S, Y because that is the number of values that is S, Y rather than there is a small bug here, V of 1 over S, Y . Now that is the probability of finding given value of Y .

Since there are T of S tuples combining with T of R tuples, the size estimate of the total of this join would be T of R times T of S divided by V of S, Y here. Now this is true if the smaller relation is R that is the V of R, Y is less than V of S, Y .

(Refer Slide Time: 55:04)



Estimating size of natural joins

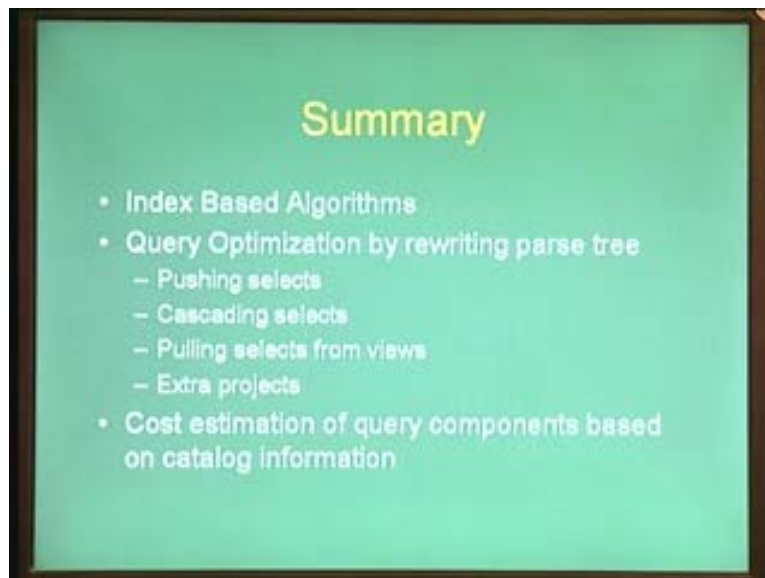
In a general sense, the maximum of $V(R, Y)$ and $V(S, Y)$ appear in the denominator.

$$\text{Estimate} = T(R)T(S) / \max(V(R, Y), V(S, Y))$$

When Y is a composite parameter, the max of each corresponding attribute in Y are compared and multiplied in the denominator.

If the opposite is true then V of R, Y appears in the denominator. Therefore we just take the maximum of V of R, Y and V of S, Y in the denominator and the numerator remains the same T of R times T of S which gives us the good enough estimate of the number of tuples in the natural join that is if Y is a simple attribute. If Y is a composite attribute that is it contains of many different attributes then we have to consider the max of each corresponding attribute that is common between R and S . We shall not be looking into this in more detail here.

(Refer Slide Time: 55:40)



Summary

- Index Based Algorithms
- Query Optimization by rewriting parse tree
 - Pushing selects
 - Cascading selects
 - Pulling selects from views
 - Extra projects
- Cost estimation of query components based on catalog information

So that brings us to the end of this session and also the end of the topic on query processing and optimization that we have studied. We have in some sense just scratch the surface of this vast and crucial aspect of dbms design especially, namely query processing and optimization. So in this session we looked at index based algorithms for physical query plans. And we also looked at different query optimization techniques based on pushing selects, cascading selects and pulling selects out of views and inserting extra project operations.

We also looked at several kinds of cost estimation techniques which can be used in combination with heuristics of query rewriting in order to select the best or in order to select what is considered to be the best query execution plan. That brings us to the end of this session. Thank you