Database Systems

Prof. Dr. Sreenivasa Kumar
Computer Science and Engineering Department
Indian Institute of Technology – Madras

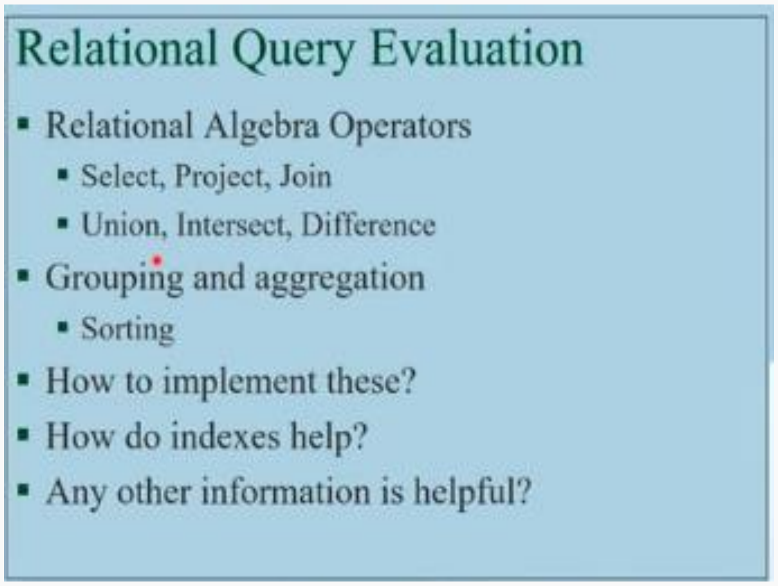
Lecture No. 33

Algorithms for Relational Algebra Operators and Query Evaluation

So I am going to start off a new module in the class. So this has to do with algorithms for relational algebra operators and we will also briefly look at how exactly query evaluation happens inside relational database management systems and what are the issues that are there. It is a relatively short module so I will give you an idea a peek into the various issues and the important algorithms.

We will discuss algorithms for Joins specifically during in this module.

(Refer Slide Time: 01:02)



Relational Query Evaluation

- Relational Algebra Operators
  - Select, Project, Join
  - Union, Intersect, Difference
- Grouping and aggregation
  - Sorting
- How to implement these?
- How do indexes help?
- Any other information is helpful?

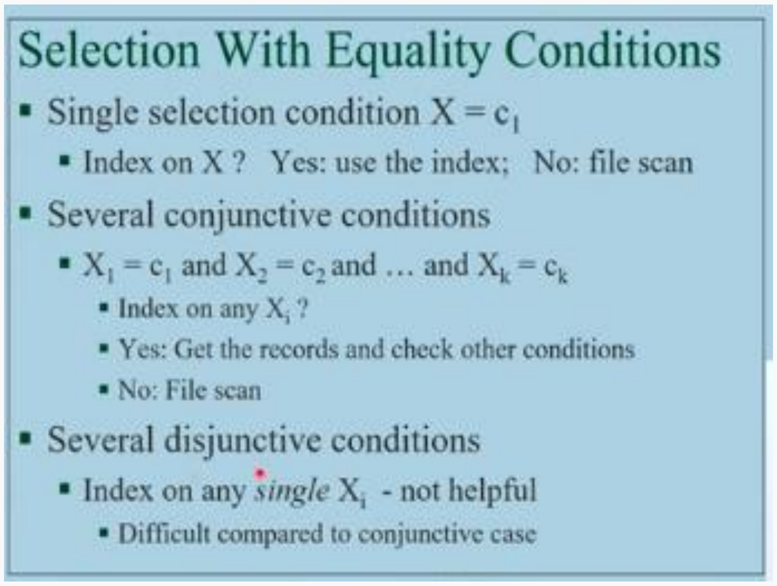
Okay so let us get going. So, if you look at the evaluation the query evaluation SQL query has to be translated into a relational algebraic expression and then that expression will have to be executed in order to give results for the SQL query. So, the relational algebra expression obviously makes use of lots of relational algebra operators that we have seen already like Select, Project, Join, Union, Intersect, Difference and all these things.

So, we have to look at how exactly these operators have to be evaluated on their operands right and then the SQL query might also have grouping and aggregation right. So, applying so grouping involves creation of the groups partitioning right and then applying some functions on those partition data. So, partitioning typically involves sorting because unless you sort the tuples you will not be able to put them into groups.

So sorting is involved so how to actually implement these things in the context where majority of these data is on the disk and so typically they are large size files and they are all lying on secondary storage and so we should consider all secondary stored algorithms that involve secondary storage right. So we will focus a lot on how do we minimize the number of disk reads how do we minimize the disk reads and in the context of disk reads since memory operations are so inexpensive we will largely ignore the memory operations.

Now when we are implementing these things obviously the indexes will help. So, the presence of index will help us a lot. So how do these indexes help and is there any other information that is helpful if that is the case how do we make use of that and how do we collect that information all these issues are there for us to look at.

(Refer Slide Time: 03:46)



Selection With Equality Conditions

- Single selection condition $X = c_1$
  - Index on X ? Yes: use the index; No: file scan
- Several conjunctive conditions
  - $X_1 = c_1$ and $X_2 = c_2$ and … and $X_k = c_k$
    - Index on any X_i ?
    - Yes: Get the records and check other conditions
    - No: File scan
- Several disjunctive conditions
  - Index on any *single* X_i - not helpful
    - Difficult compared to conjunctive case

Now let us start with simple selections. Let us focus with in selection condition can be on some attribute X and equal to some constant right. Let us in general it can be an another attribute or a very complex expression etcetera. Let us just focus on the simple condition where the selection attribute equal to some constant. Now how do we so on some relation on some certain set of tuples this kind of a selection condition has been specified.

Now one of the first things we have to check is that is there a index on the particular attribute X. If there is an index on the particular attribute X then you know that given the value of the attribute you will be able to quickly retrieve the records that have the specific value from the set of records using the index, the index will help us. It will give you record pointers and so you can go or block pointers and then you can go pick up those files.

If there is no index then we have very little we can do. Basically, we will have to do a file scan to figure out where, what are all these conditions what are all those records in which this particular condition X equals c1 holds. Of course now if so the query engine you know keeps kind of track of what kind of queries are being posed to this query engine and then you know if this kind of selection conditions coming often then it might actually look at the physical organization.

And might actually decide saying that it is probably better off to construct an index on this particular X and then keep it ready because I am getting too many selection conditions involving this particular attribute in case the index is not there. So these are the various considerations that and notice that this is a dynamic kind of a system. The system keeps learning or you know learning from the past as to how it has been doing.

It has at least the potential to do that. Now let us say there is a often times a selection condition is not just a single condition there may be a conjunction like this. X1=c1, X2=c2 and so on Xk=ck. In this case actually we are relatively lucky I can say is that if there is suppose there are some 5 number of these things. Then if an index exists on any one of these things any one of these attributes if an index exists then we can it is a conjunction.

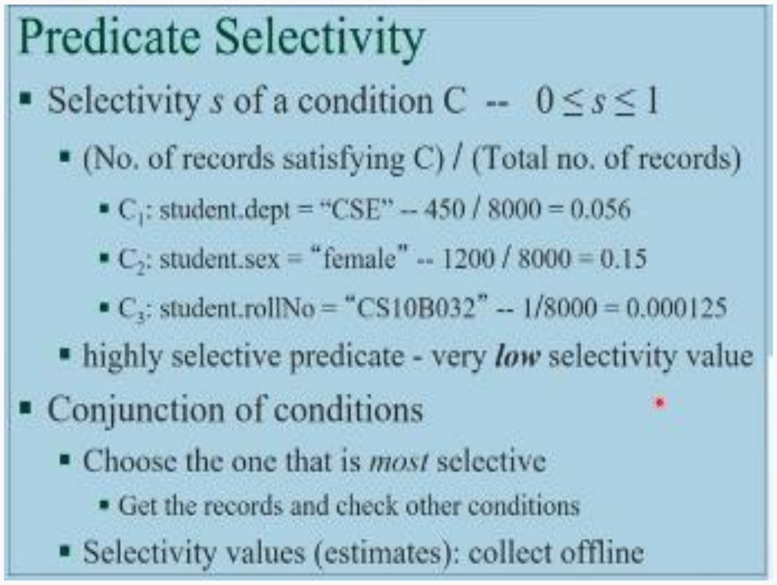
So, we can make use of say let us say on X2 there is an index. So I can go make use of the index on X2 pickup all those records whose values for X2=c2 and then in those set of records I can check the other conditions right because it is a conjunction. So no file scan will be required in case there is an index on any one of these. Since there are many it is likely that you will have a index on one of them and so we can make use of them.

Of course if you have indexes on more than one of them then you have a choice also then you can choose the index that has the better depth and since the average expected number of discrete using that indexes row like for example if you have hash indexes exist then you can make use of the hash indexes. If in an unlucky situation where there is no index on any one of these attributes, then you basically you do not have any choice other than doing your file scan and while you scan the file check this condition.

And then pick up all those records that satisfies this particular condition. Now disjunction is harder than conjunction actually if you have a disjunction there are several disjunctive conditions. So like X1=c1 or X2=c2 etcetera or Xk=ck then it is not sufficient if you have an index on anyone of them in fact you have to have index on all of them only then it will be useful. So, index on any single Xi is not helpful because it is an odd condition right.

So, you basically probably will not be able to avoid a file scan in case there are disjunctive predicates in the selection condition.

(Refer Slide Time: 08:59)



Predicate Selectivity

- Selectivity *s* of a condition C -- $0 \leq s \leq 1$
  - (No. of records satisfying C) / (Total no. of records)
    - C_1: student.dept = "CSE" -- 450 / 8000 = 0.056
    - C_2: student.sex = "female" -- 1200 / 8000 = 0.15
    - C_3: student.rollNo = "CS10B032" -- 1/8000 = 0.000125
  - highly selective predicate - very ***low*** selectivity value
- Conjunction of conditions
  - Choose the one that is *most* selective
    - Get the records and check other conditions
  - Selectivity values (estimates): collect offline

Now in let us also in this context we can also you know think about this particular quantity called predicate selectivity. How selective is a condition? So depending on that you can proceed you can basically make use of this thing. So, in case there is a index on multiple attributes then you can choose the attribute that is highly selective and then drive records that satisfy that and then go ahead and then make use of the set of records.

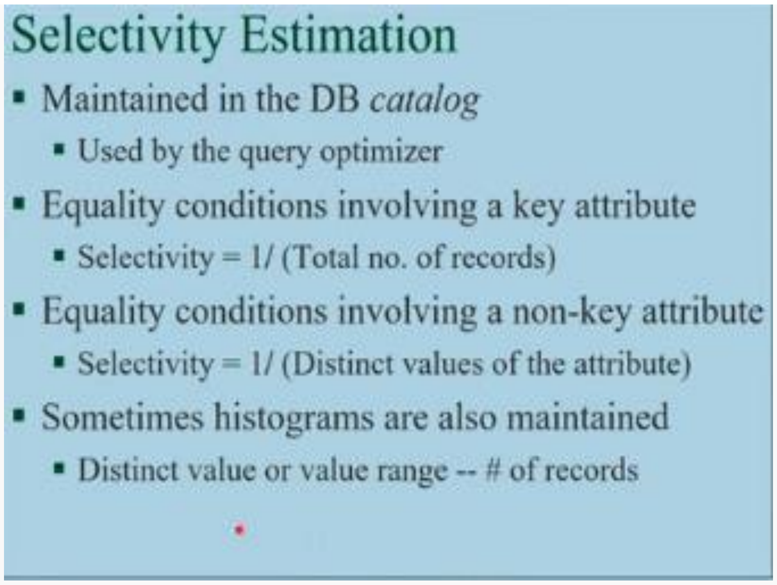
Let me tell you what is selectivity is? Selectivity is basically the number of records that satisfy that particular condition divide by the total number of records that are present. So that is what we call it the selectivity. So here are some examples. Student department equals CSE so assuming that there about 500 students and 450 students in abroad in our campus of 8,000 students the selectivity is like this and the I heard a fewer number of girls compared to boys so the selectivity of this is like this.

If you give a roll number and then look at the selectivity there is very low selectivity the low values for selectivity basically mean that you know the number of records that satisfy this is low. So, it is highly selective so in actually the lower the number we call it highly selective the condition is selective highly selective. So, your condition is said to be highly selective if the value for selectivity is low.

So if you have a conjunction of conditions then this selectivity values will help us choose the one that has most selective condition then that will give you the first number of records to handle and then retrieve those records and then do the check for other conditions. So get those records and then check the other conditions that is the strategy we use for. So how do we get these selectivity values? So, these are some additional information that we can keep track off while the system is running.

While the database is under use you can actually keep either selectivity values or their estimates in a place called DB catalog.

(Refer Slide Time: 11:54)



Selectivity Estimation

- Maintained in the DB *catalog*
  - Used by the query optimizer
- Equality conditions involving a key attribute
  - Selectivity = 1/ (Total no. of records)
- Equality conditions involving a non-key attribute
  - Selectivity = 1/ (Distinct values of the attribute)
- Sometimes histograms are also maintained
  - Distinct value or value range -- # of records

So, in a database catalog we will actually along with lots of other metadata of the database will also maintain these statistics about the various selection conditions that are being used in queries. So, this statistic will be generally used by this query optimizer. So, as the database keeps running and then answering various ad hoc queries on the side it will also keep track of these selectivities values.

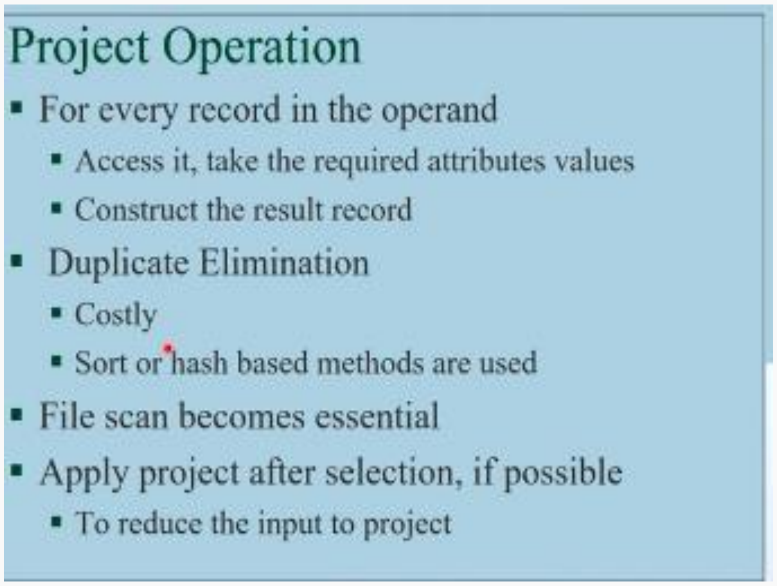
So for example if there are equality conditions involving a key attribute then the selectivity will be you know you can also use this rough formulas for finding out the selectivity even if you do not have the selectivities you can find out this selectivity. Selectivity involving a key attribute is 1 by total number of records in the record. So, if the equality condition involves non-key attributes then you can you know make some rough estimate about the selector e by taking the number of distinct values for attribute.

For in that attribute order the number of distinct values and then you can assume uniform distribution of the values among all the records and then take 1 over the number of distinct values as the estimate for the selectivity. Now sometimes the database systems actually even maintain a histogram. So, a histogram will have on the horizontal axis either a distinct value of the attribute or a range of values and on the vertical axis.

It will have the number of records that particular value or the value range in exists within that range. Okay. So, on the x axis we have either the distinct values or ranges of distinct values and then on the y axis we have the number of records that have that. So given a particular condition so you can go check the condition the constant value that has been given for that particular value.

Use this histogram and figure out how many number of records are possibly there in the data file estimated to be there. So this also will be can be you know maintained offline and then you can know keep it updated as the system is running.

(Refer Slide Time: 14:34)



Project Operation

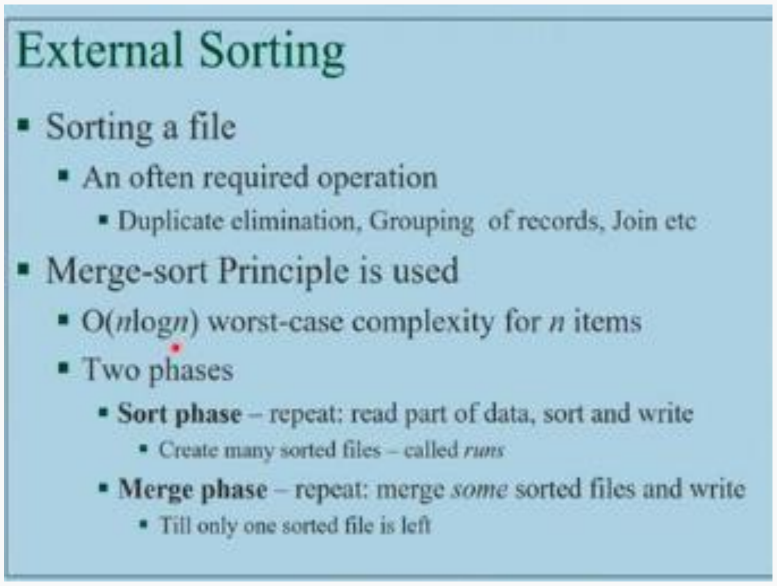
- For every record in the operand
  - Access it, take the required attributes values
  - Construct the result record
- Duplicate Elimination
  - Costly
  - Sort or hash based methods are used
- File scan becomes essential
- Apply project after selection, if possible
  - To reduce the input to project

Now moving on to projection operation, projection operation is somewhat costly actually because this requires you to choose a particular set of attributes from a record. That is what projection is right. So for every record that is there in the operand you basically have to access this and then you know take the required attributes values and construct a result record and of course so duplicate elimination is also involved.

If you know as a result of doing this in the result records if there are duplicates we will have to eliminate. So sorting so duplicating elimination is somewhat costly operation and we basically they have to use sort of hash based methods for duplicate elimination. We will shortly consider the sorting operation in detail actually. So, the file scan becomes kind of essential for project operation and the rough the one of the ideas that we can make use of is to kind try and apply this projection after the selections.

Do the selections then you have fewer numbers of records and then on them do the projections wherever possible. Essentially this will reduce the amount of input that is given to the project. Okay. Now so this is about selection projection.

(Refer Slide Time: 16:35)



External Sorting

- Sorting a file
  - An often required operation
    - Duplicate elimination, Grouping of records, Join etc
- Merge-sort Principle is used
  - O(*n*log*n*) worst-case complexity for *n* items
  - Two phases
    - **Sort phase** – repeat: read part of data, sort and write
      - Create many sorted files – called *runs*
    - **Merge phase** – repeat: merge *some* sorted files and write
      - Till only one sorted file is left

Before we move on to joins one has to you know kind of consider this sorting because it is really one of the important you know the things that are required when we are looking at execution of queries. Because sorting kind of you know appears in some step or other because either you are to do duplicate elimination or you have to do grouping or sometimes we can actually use sorting as a method of joining also, joining records.

I will talk about sort based joints especially one of the file is sorted then it might be actually advantageous to sort the other file and then merge these two files in order to do a joint. We will come to join operations in little later. First let us focus on external sorting. So external sorting basically involves that you have a huge file which is this precedent and you have been given some attributes for comparing the records and we were to sort them in increasing order of those records.

So, the merge sort principle is used. In this case merge sort principle is used. Merge sort is familiar to you. So basically, it is a order nlogn worst-case complexity algorithm for sorting n items. So, we have discussed it in the data structures and algorithms course for in memory swapping right. So, given an array how do you sort it using merge-sort principal and all that you would have studied.

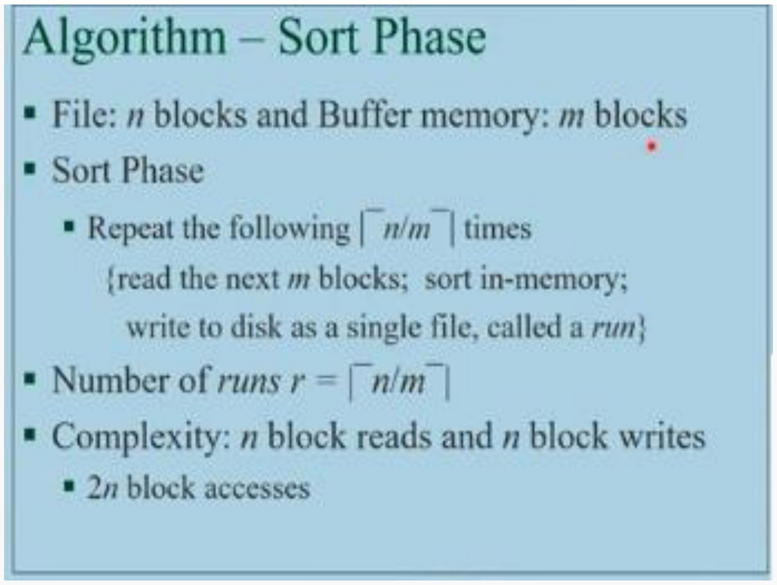
We will basically adapt that principle here but then pay attention. It is not you know completely straightforward to kind of you know do it on a disk file. So there are some issues involved. So, we will look at them carefully. So, in the external sorting well we are using merge sort principle. There are usually two phases that has to be considered. One is called the sort phase the other one is called the merge phase.

Well the sort phase you might slightly get confused, I mean if you are already sorting what is the question of (()) (19:08) you might confuse but sort phase actually is to create many sorted sub files. Actually, you create many sorted sub files of the main of the data file. These sorted sub files are called runs in the database parlance we call them a run. It is a bunch of records already sorted but it is not the whole data file, it is a part of the file.

So basically you read part of that file sort it and write it back to the disk, Then you would have created one run of the thing. So, if like this you will create several runs. Once you create so in the sort phase we create several sorted runs and then in the merge phase take some of them merge them and then produce a ultimate sorted file. So that is how the rough idea is. So, in fact what we will do is we may not be able to merge all of them together.

So, take some of them merge them and write it back and repeat this process till you are left with the one sorted file. Okay. So, let me tell you the details so I will give you a little bit more in detail now how exactly the external sorting works. Okay.

(Refer Slide Time: 20:37)



Algorithm – Sort Phase

- File: *n* blocks and Buffer memory: *m* blocks
- Sort Phase
  - Repeat the following $\lceil n/m \rceil$ times
    {read the next *m* blocks; sort in-memory;
    write to disk as a single file, called a *run*}
- Number of *runs* $r = \lceil n/m \rceil$
- Complexity: *n* block reads and *n* block writes
  - 2*n* block accesses

So the sort phase so the essential assumptions that we are making here is that the data file has some huge number of blocks, let us call it n blocks and we need to we cannot fit all of those data all of the entire file into the memory. So, we have been given some memory space so which is of roughly equal to which is equal to m number of blocks. m might be huge but still n is much larger than m. So, you have some memory spaces you know ram space which is equal to m blocks

And the disk file is having some n blocks which is a huge bigger number compared to m. So, in the sort phase what we will do is so if your n blocks we can read m blocks into the memory at a time. So, we will have in order to read all the file all the blocks of the data file we will take n over m number of iterations. Okay. So repeat the following in our number of n by m seal times. So each time what do you do. Read the next m blocks are a part of it the last time when you may be having less than n blocks.

So read the next m blocks, sort them in memory and then write the disk as a single file called run. Okay. So in this process we would have created each time. So notice that we are actually not sorting the entire data file here because we are not reading the entire data file into the main memory we cannot. So whatever part of that we can read into the memory we are reading. So m blocks we are reading at a time sorting it and then writing it back onto the disk.

So you are now creating sub files of the main data file and each of these sub files is sorted. Each of this sub files is sorted and we do not count the sorting time in memory. Okay what is the complexity of this whole thing is, so how many number of sub files you will create now this n by m number of sub files you have created. There are n number of data file data blocks you are reading m of them at a time. So, in order to finish reading all of them you make n by m number of iterations.

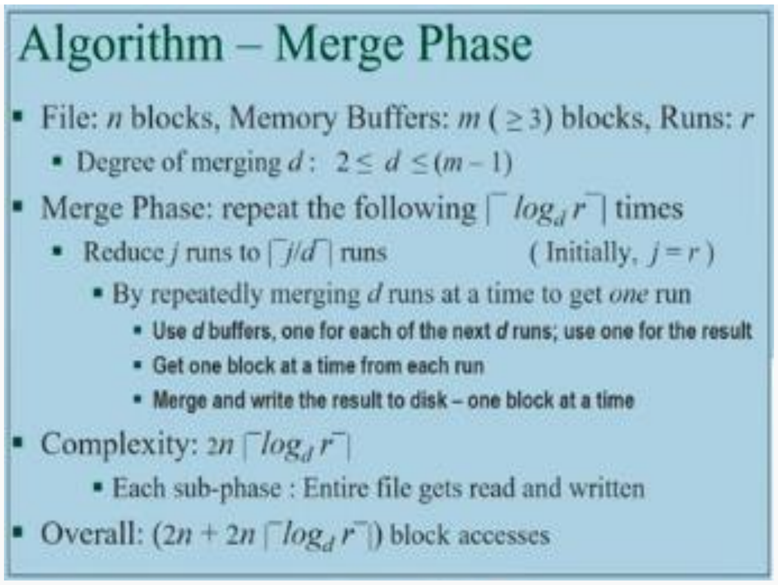
So, each alteration you will produce one sorted sub file. So, these things are call runs and so the number of runs is n by m seal and the complexity of this entire phase is to end block accesses. Why is that? Basically, you have to read all the n blocks of the file into the memory at some time or other and then basically write them all back, right. You will have to read m blocks at a time the whole data file sort them and write read back those m blocks. Okay.

So essentially you are reading all the data file blocks and then writing them back. So, it is 2n block accesses. Are you all with me? So, at the end of sort phase what we have done is to create some r number of sub files each of which is sorted. Now we can actually do merge. Remember that in the in-memory case we always assume that the whole data fits into the memory.

And so we will assume that it is some portion is already sorted in fact each record is treated as a sorted thing and then we will merge two of them together into one sort and block like that and then merge the two records into four records and so on so forth. That is the in-memory sort algorithm or merge sort algorithm but here we are looking at what is the available space if the buffer space is M blocks, so we can read that much portion of the data file sorted in memory and then write it back.

So that is the basic unit of sorted sub files we have. Now we take all these sorted files and merge them just like we did in the usual merge-sort order. Okay. So, the starting point is this number of r runs.

(Refer Slide Time: 26:25)



Algorithm – Merge Phase

- File: n blocks, Memory Buffers: m (≥ 3) blocks, Runs: r
  - Degree of merging d: $2 \leq d \leq (m-1)$
- Merge Phase: repeat the following $\lceil \log_d r \rceil$ times
  - Reduce j runs to $\lceil j/d \rceil$ runs (Initially, $j = r$)
    - By repeatedly merging d runs at a time to get *one* run
      - Use d buffers, one for each of the next d runs; use one for the result
      - Get one block at a time from each run
      - Merge and write the result to disk – one block at a time
- Complexity: $2n \lceil \log_d r \rceil$
    - Each sub-phase : Entire file gets read and written
- Overall: $(2n + 2n \lceil \log_d r \rceil)$ block accesses

So, let us focus on merge now. So if file has n number of blocks and the memory buffers are m which is at least 3 blocks or so and the number of runs is r then we can in the in-memory case we can always merge 2 things and then produce 1 but actually here we can do a d way merging. The d will be at least two but cannot actually be a large number. What is that large number, it is m-1 where m is the number of memory buffer blocks that are available to us.

Why minus 1 because we have to keep the result somewhere. After we merge, we get the result right so we have to keep the result somewhere. So, we allocate 1 block memory a 1 block for that some amount of space for that and the remaining thing remaining entire space we can afford to use for merging purpose. So that is the essential idea that is why we can actually do a d way merging. What is the d way merging?

A d way merging is look at d of these records put them in sorted order and write them back. That is a d way merging. It is a 2-way merging is look at 2 of them and then compare them and then write them in the appropriate order. That is to 2-way emerging well now we will do a d way merging where we look at d of these things at a time and then sort them and then write them back. I will write them back means I will show you how exactly we will write them back ok.

So now let us look at the algorithm for example. The merge phase has this thing repeat the following we will see how many times it will come to. So basic idea here is that if you have some runs some number of runs we will reduce this number of runs to that run divided by d number of runs. Okay. So that means if you have some 100 runs to start with and then you are doing a 4 way merging so we will reduce the runs to 25 runs.

You run you will take 100 runs and produce 25 runs. Okay. So, reduce j runs to j by d number of runs. Initially J equal the number of runs that we have. How do we do this j runs into j by d runs. That is our usual merging actually. So by repeatedly merging these d runs at a time to get one run. So how do you do. Use d buffers that is why d is at most m-1. Use d buffers one for each of these d runs that we are talking about.

And use 1 of these buffers one of the other buffer for the result. Okay. So, get one block at a time from each of these runs. Each of these runs is a huge pile away, so get one block at a time from each of these runs merge and write the result to the disk. This you can afford to do because each of those runs are already sorted. So, you can consider the top most block in each of those runs and then if necessary.

So I will show you a diagram also for that merge and write the result to the disk one block at a time. So what is the complexity of this. So how many number of times do you have to repeat this. Reduce the j runs to j by d number of runs, how many times you have to do. It is log r to the base d. You have r runs to start off each time you do a d way merge you reduce this r runs to r by d runs and the next ways you reduce it by r by d square and so on.

So, a factor of d is the reduction in the number of runs and so the number of times this whole thing has to be done is essentially log r to the base d, c that many number of times you have to do. So, if you have a thousand 24. Okay so you know what is the log function so I do not have to repeat that. Now so since you are doing this log r to the base the number of times the whole thing and what is this thing doing?

In each of these phases we are reading all the runs and then converting them into a fewer number of runs in essentially, we are reading the whole data file. In each of these sub phases we are actually reading the whole data file, because the whole data file records are actually spread over this r number of runs. We are reading all those runs and then reducing them to r by d number of runs.

But the run if you take the length of the runs in both cases if you total up the length of all those runs it is the same because you have read all the records and then written them all back you get that point? The number of runs are reducing but the number of records is not reducing. The number of records are all spread over all these runs in appropriate places. Where each of those runs is a sorted run.

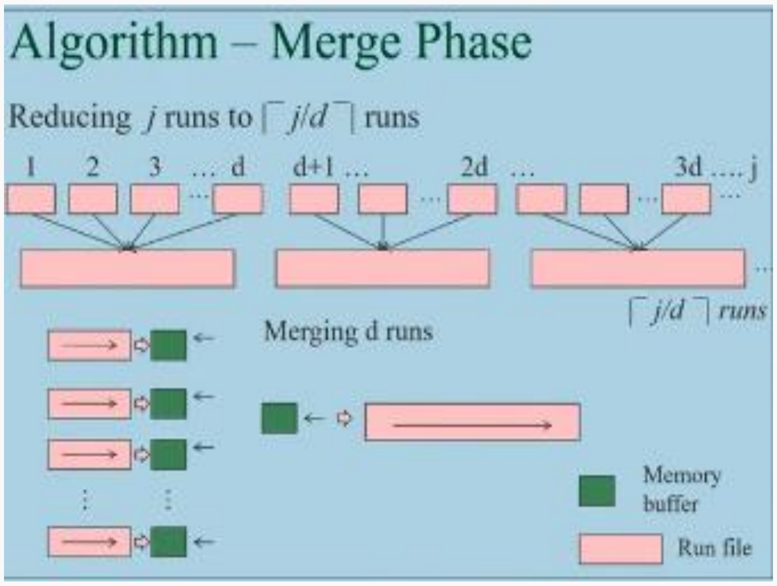
So, you are taking two of them and merging them and then producing one so the number of records is not disappear right it will all be there right. If you are taking 100 of them and then producing 25 of them the same number of records will be there in these 25 runs. So each time you are reading the whole data file and writing it back. So, you are doing 2n lock accesses. If n is the number of blocks that are there in the data file you are actually doing 2n block accesses.

And how many times are you doing this as many times as there are requirements like log r to the base d. This many number of times your reading the data file. So the complexity in terms of block accesses that is what we care about in this course block accesses. This merging actually takes some memory time but we do not care a bit about it. So 2n log r to the base d is the number of block accesses that will require.

And then this is only the merged phase what about the overall. In the sort phase we had used 2n block accesses in order to create the initial set of runs and in the merge phase we are using 2n times log r to the base d number of block accesses. So this is the overall number of block access you have to do in order to sort a file an external sorting using the most efficient algorithm merge-sort.

You know that merge-sort is one of the best sorting algorithms right. First case n log and very happy situation ok. So, let me show you a diagram here which will make this whole process a little bit easier to visualize.

(Refer Slide Time: 35:09)



So let me take the crucial step of reducing this j runs to j by d number of runs. So, at this time you have 1, 2, 3 up to j runs. So, all your records are spread somewhere here in those runs. You are taking d of them and then merging them and then creating 1 run in new run. So now because you are taking d of them together so this j runs will now become j by d number of runs in the next phase and then you have to repeat that again.

Take d of these runs and then merge them then it will become j by d, j by d square number of runs here. Now let us focus a little bit on how do we exactly merge so take each of these j runs some number of runs so some number of runs. For each of them this is a d way so there are d number of buffers that we can make use of and there is one buffer that is for keeping the result. So, d number of memory buffers these are green things are memory buffers and so here you can maintain pointers the memory pointers. Okay.

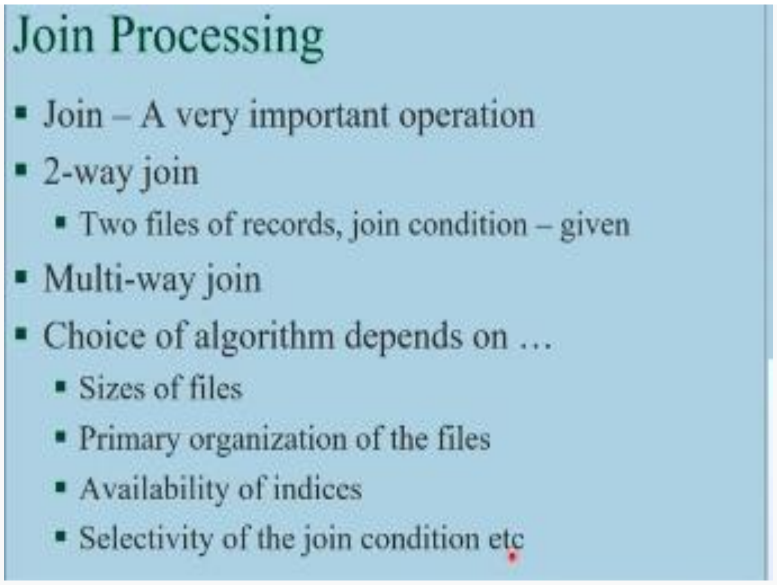
So you have seen in merge sort implementation the pointer has to be moved accordingly right. So now you move the pointer in inside the block. So now you have fetched one block here. So each of these blocks has data which is tuples records. So now you look at the topmost of all these things they are already in sort sorted order. They are all in sorted order. So now look at all of those things obviously in one of these things will be the first among them sorted.

So pick up that and advance the pointer there, wherever you pick up a record you advance the pointer there and again look at all the pointers pick up the next one advance the pointer. So in the process of advancing the pointer, advancing means when you are coming down you might exhaust the block. If you are exhausting block fetch a block from the sorted file into the buffer. So that way slowly you are getting all the blocks of the sorted file into the thing.

So in case one of the runs is smaller compared to the other one this might be become empty, so you note that pointer merge the other it works. So basically keep merging like that and as you are picking up the topmost records from this entire set of records keep putting them onto the result and the result will start filling up result block will start filling up. Once a block fills up write into the disk, keep shifting the records to the disk. So you have made your taking j runs and then you produce j by I mean you have taken some d runs and produce 1run.

Is the process clear? So, sorting even though we know the principle that we are using which is merge sort is a little involved in external methods. So, this is how you can actually do a sorting and the overall complexity as I told you is so it is pretty costly operations. So many number of block accesses are in one. Okay.

(Refer Slide Time: 39:27)



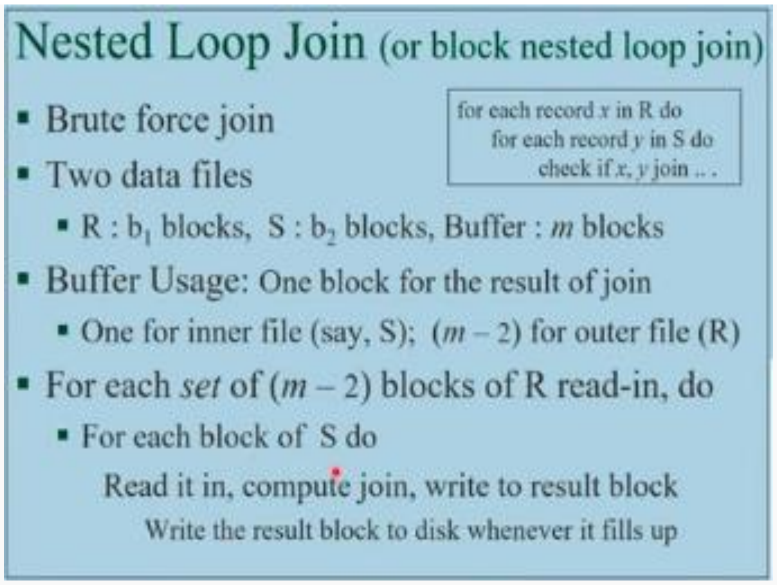
Join Processing

- Join – A very important operation
- 2-way join
  - Two files of records, join condition – given
- Multi-way join
- Choice of algorithm depends on …
  - Sizes of files
  - Primary organization of the files
  - Availability of indices
  - Selectivity of the join condition etc

So, let us move on to join is a very important operation obviously. So, let us focus a little bit on 2-way joins. So, two files of records and a join condition is given you are supposed to join. In a multi-way join you have given more than one more than two files and a join condition in order to join. Now what is the choice of algorithm for doing a join kind of depends on the sizes of the files that are involved and what is the primary organization of those files.

How are those files organized, are they already sorted files? If they are already sorted file while doing merging I can do a join actually. Something similar to merging I can keep doing and as up in that process I can actually pick up join records. We will just consider equijoin. Let us say we consider equijoin if okay and are there indices available so these are some of the questions and what about the selectivity of the join condition?

(Refer Slide Time: 40:51)



Nested Loop Join (or block nested loop join)

- Brute force join
- Two data files
  - R : b_1 blocks, S : b_2 blocks, Buffer : *m* blocks
- Buffer Usage: One block for the result of join
  - One for inner file (say, S); $(m - 2)$ for outer file (R)
- For each *set* of $(m - 2)$ blocks of R read-in, do
  - For each block of S do

    Read it in, compute join, write to result block

    Write the result block to disk whenever it fills up

```
for each record x in R do
    for each record y in S do
        check if x, y join .. .
```

So, these are the various things that are involved. Let us consider one brute force kind of a join. Let us say we do not have any indexes on those files. So, what is the brute force way of joining? This algorithm is called nested loop join or block nested loop join. So essentially this join algorithm is very simple to state for each record x in R and for each record y in S, so this is the inside loop.

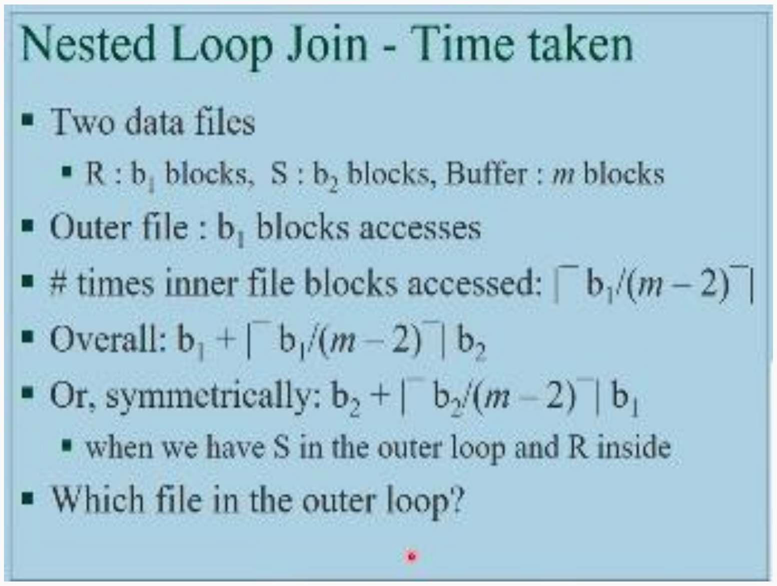
So for as you are iterating each record x in R you are looking at all records in Y and if checking x and y are joining means they are together satisfying the join condition if so produce the join. So these are simplest of approaches but then again considerations will come into picture because you have R has some b1 number of blocks or S has b2 blocks and we always have some m number of buffer blocks. Yet some amount of memory is available and we will measure it in terms of blocks because it is easier for us to handle the numbers that way.

So m let us say we have m buffer blocks. So if we have buffer blocks then basically we will keep one block for the result of the join and keep one block for the inner file let us say s is the inner file here right. So inner file we will call it inner file and the remaining m minus 2 for the outer file. m minus 2 buffers for the outer file and the algorithm is very simple. For each set of these m-2 blocks of r so instead we will obviously not be just reading records of r into the memory because we can we have to do block accesses r is a disk file.

So we will pick up m-2 blocks of r at a time. Maybe we will be able to you know we get fast access if you give large number of consecutive blocks of a file we do get quick access if you are using raid architecture and so you will specify m minus two blocks and read them so keep m minus two blocks of r in memory. Now keep reading the other file one block at a time. For each record in s right.

So we have read the entire s each one block at a time and once we read one block at a time we can actually do a computation. Figure out which of those records of s will join with any of those records of r. So, read it compute the joint write the result block. So, write the result to the in-memory block for temporarily and as long as the as soon as the block fills in you can keep actually writing it back. So that is how the nested loop or block nested loop will work.

(Refer Slide Time: 44:27)



Nested Loop Join - Time taken

- Two data files
  - R : b_1 blocks, S : b_2 blocks, Buffer : m blocks
- Outer file : b_1 blocks accesses
- # times inner file blocks accessed: $\lceil b_1/(m-2) \rceil$
- Overall: $b_1 + \lceil b_1/(m-2) \rceil b_2$
- Or, symmetrically: $b_2 + \lceil b_2/(m-2) \rceil b_1$
  - when we have S in the outer loop and R inside
- Which file in the outer loop?

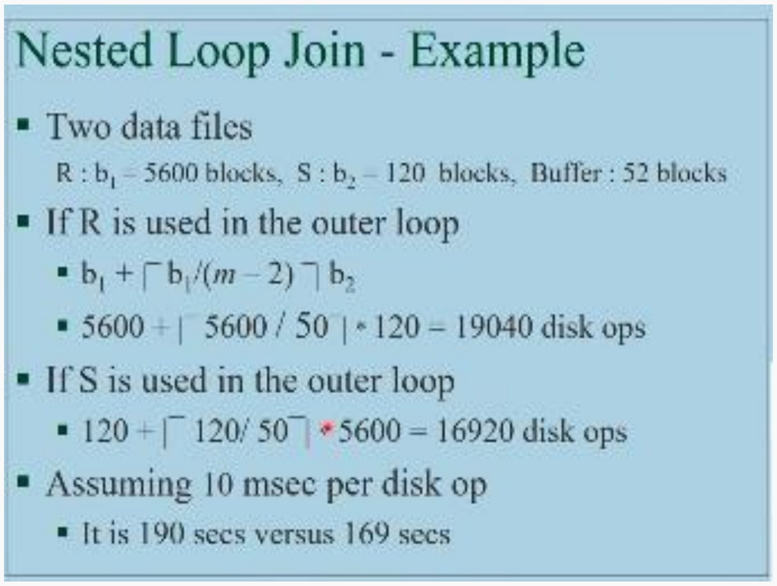
Now let me close with a small calculation here. Let us look at the time taken for doing a nested loop join. So, let us say you have two data files R with b1 blocks and S with b2 blocks and m number of buffer blocks memory blocks. So, R was the outer block right outer file. So outer file has b1 block accesses basically because you have to read the entire outer block and but the only thing is we will read them in chunks of m-2 blocks at a time.

So, but wherever we read m-2 blocks of the outer file, then we will read the entire inner block inner file. Inner file is s. We will read the whole blocks of the inner file. So how many number of times the inner file blocks are accessed. It is b1 divided by m-2. This many number of times we will be accessing the inner file s. So overall if you look at the nested loop join it will take b1 block accesses to read the outer file and b1 divided by m-2 seal times b2. b2 is the inner file.

This many block access so this file has to be repeatedly read for a chunk of the outer file you have to read the entire inner file and then you get the next chunk of the outer file read the entire inner file again. That is the block accessed block nested local. So overall list is b1 plus this many number of times and there is nothing great about r to keep it in the outer block and s to keep it in a inner block.

We can actually switch them into things and so symmetrically speaking you will get the b2 +b2 by m-2 times b1 and you can see that the second term is roughly equal. It is basically b1 times b2 divided by m-2 in both these expressions and it is this b1 or b2 that actually dictates the time. I mean also in a different shade this time so which file to keep in the outer loop basically the one that has smaller number of blocks you can keep it in the outer loop then that will be a additive value here and so you can get my rack system.

(Refer Slide Time: 47:33)



Nested Loop Join - Example

- Two data files
  R : b_1 = 5600 blocks, S : b_2 = 120 blocks, Buffer : 52 blocks
- If R is used in the outer loop
  - $b_1 + \lceil b_1/(m-2) \rceil b_2$
  - $5600 + \lceil 5600 / 50 \rceil * 120 = 19040$ disk ops
- If S is used in the outer loop
  - $120 + \lceil 120/ 50 \rceil * 5600 = 16920$ disk ops
- Assuming 10 msec per disk op
  - It is 190 secs versus 169 secs

So, I have some numbers here to illustrate that but I think I will leave it to you to figure it out. So, there is some so we will stop here for today let me probably show I will just start off with this example in the next class.