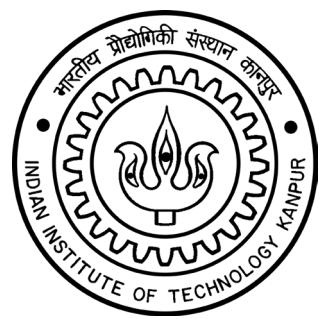


# FUNDAMENTALS OF DATABASE SYSTEMS

**Prof. Arnab Bhattacharya**  
Computer Architecture  
IIT Kanpur



# INDEX

S. No	Topic	Page No.
	<b><i>Week 1</i></b>	
1	Introduction to Databases	1
2	Relational Data Model	6
3	Relational Algebra Basic Operators	17
4	Relational Algebra Composition of Operators	26
5	Relational Algebra Additional Operators	31
6	Relational Algebra Extended Relational Algebra	41
	<b><i>Week 2</i></b>	
7	Relational Algebra: Database Modifications	51
8	SQL: Introduction and Data Definition	60
9	SQL: Basic Queries	67
10	SQL: Advanced Queries	79
11	SQL: Updates, Joins, Views and Triggers	89
12	Normalization Theory: Motivation	100
	<b><i>Week 3</i></b>	
13	Normalization Theory: 1 NF and 2NF	108
14	Normalization Theory: 3NF	116
15	Normalization Theory: BCNF	123
16	Normalization Theory: MVD	129
17	Physical Design	138
18	Database Indexing: Hashing	149
	<b><i>Week 4</i></b>	
19	Database Indexing: Tree-based Indexing	158
20	Query Processing: Selection	167
21	Query Processing: Sorting	175
22	Query Processing: Nested-Loop joins and Merge join	184
23	Query Processing: Hash join and other Operations	194
24	Query Optimization: Equivalent Expressions and Simple Equivalence Rules	206
	<b><i>Week 5</i></b>	
25	Query Optimization: Complex Equivalence Rules	212
26	Query Optimization: Join Order	220
27	Query Optimization: Heuristics and Sizes	230
28	Database Transactions: Properties and Failures	240

29	Database Transactions: States and Systems	249
30	Recovery Systems: Deferred Database Modification	258

### ***Week 6***

31	Recovery Systems: Immediate Database Modification.	267
32	Recovery Systems: Checkpointing and Shadow Paging	272
33	Schedules: Introduction	281
34	Schedules: Conflict Serializability	287
35	Schedules: View Serializability	293
36	Schedules: Result Equivalence and Testing for Serializability	301

### ***Week 7***

37	Schedules: Recoverability	308
38	Concurrency Control: Locks	320
39	Concurrency Control: Two-phase Locking Protocol	327
40	Concurrency Control: Timestamp Ordering Protocol	334
41	Concurrency Control: Validation-based Protocol	345
42	Concurrency Control: Multiple Granularity for Locks	352

### ***Week 8***

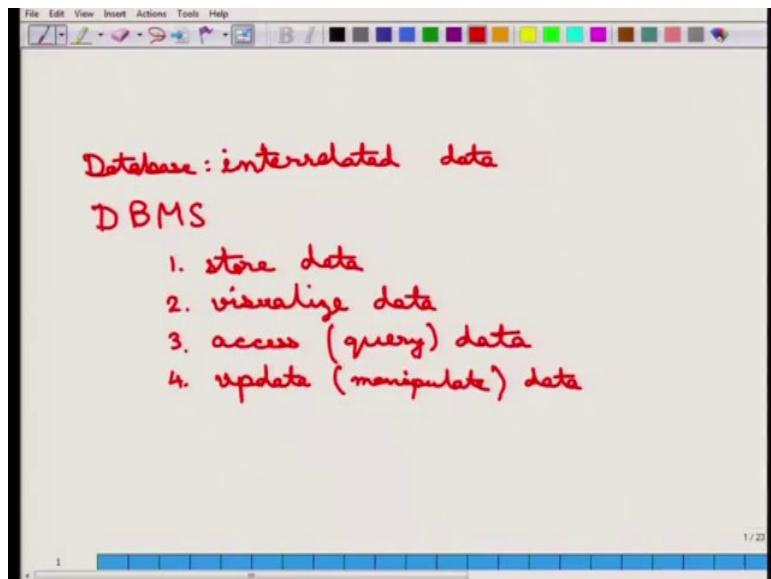
43	Concurrency Control: Deadlock Prevention and Deadlock Detection	364
44	Concurrency Control: Deadlock Recovery and Update Operations	372
45	NoSQL: Introduction and Properties	381
46	NoSQL: Columnar Families	389
47	NoSQL: Different NoSQL Systems	398
48	Big Data	407

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

**Lecture - 01**  
**Introduction to Databases**

Welcome all of you to this course on Fundamentals of Database Systems. This is mostly an Undergraduate course for Computer Science and Engineering as well as for Information Technology students. So, let us start off with, *what do we mean by a database*. A database is essentially a collection of data.

(Refer Slide Time: 00:32)



But, very importantly, it is not any data, it is a collection of *interrelated* data. So, the data fields or the data points must have some connections with them. So, it is a set of *interrelated* data. We all have some idea of what databases are and where they can be used. For example, in an institute, the entire records about an institute constitutes one database. So, it will have different entities such as faculty members, students, staff, etc, as well as other kinds of things, such as courses, etc. And everything constitutes one database.

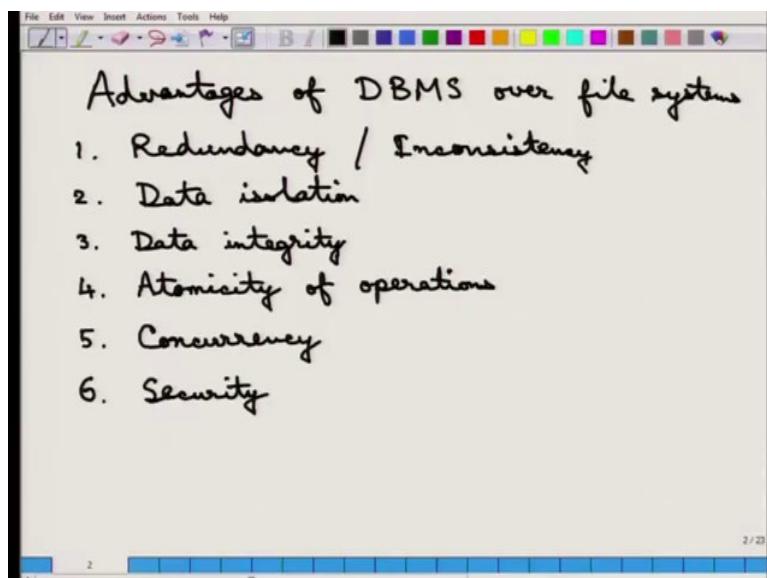
So, this is what is a database. What is a DataBase Management System or a DBMS? It is a system that provides an environment to handle a database. So, it must be efficient and convenient to use and essentially, it contains some programs and interface to store data.

So, the first point is *store data*; number two is, it must be able to *visualize data*, then *access*

*data* or what is also known as *querying the data* and the fourth one is *update or manipulate the data*. So, these are the four tasks of a database management system. So, this is over the database. The question comes then is that, the normal file system can also do all of these things. It can of course, store data. So, we store files etcetera. It can visualize data. So, well it is not really clear what does visualization mean, we will see that later, but you can at least see, what the files and folders are etcetera. You can of course, access data, you can access a particular file within a particular folder etcetera and it let us you update the contents of the file, contents of the folder, etcetera.

So, how is a database different from a file system or how is it more important than a file system? So, there are different advantages over the file system that let us go through.

(Refer Slide Time: 02:54)



## Advantages of DBMS over File Systems

The first one is that, it *reduces the data redundancy and inconsistency problems*. So, let me go over each of these points a little bit. So, what does data redundancy mean? Essentially data redundancy means the following. There is a same piece of data, the same information that is stored in multiple places. And a well designed database does not really allow that.

So, if there is an employee or if there is information about a student, the name of the student, the roll number etcetera generally should be stored in only one place. This is the big difference about file systems, where most of us replicate files, copy files from here to there,

etcetera, etcetera. What does that copying also do is that it runs into this problem of inconsistency. So, what does inconsistency actually mean is that you have two versions of a file and in one version you have changed something and you have inadvertently forgotten to make the same modification to the other file, or there are two people making two different versions of the file etcetera. So, the information in these files two files may be inconsistent. So, one of them may not be correct, because the other one has been updated and that update has not yet come to this piece.

So, database, by virtue of storing each piece of information only once and in one place, reduces this inconsistency. So, there is no redundancy and there is no inconsistency; that is the first point.

The second point is a very related point, it is called *data isolation*. So, here what happens is that, the data is isolated in the form that there is only one. So, the data is stored in an internal format and there is only one interface to access the data. So, the essentially the data is stored in some kind of a binary format, which is not the headache of the final user. The user wants the piece of data and there is an interface, the database interface, that will let that user access the data in a particular format. So, the other important advantage of this is that, because the data is isolated and stored in a format that the database handles, the problem of formatting of data is not there. And this, we often run into with file systems, for example you may be working with an excel file, which is a spreadsheet and once you take it to for example, Linux system or Mac system, that excel file may not open correctly. Because, it is not in that open document format and the programs to open them do not behave consistently; that is the problem, but the database isolates that.

The third important thing is called *data integrity*. So, what data integrity means essentially is that, there are many pieces of data, where there is a semantics to the data or there is a correctness condition. For example, the CPI or the Cumulative Point Average or your marks, the grade point average, whatever you call it, they are generally considered to be between 0 to 10 in our Indian systems at least. But, suppose you are storing all of these information in a file or in an excel spreadsheet. Now there is no check. So, if you, for example, by chance entered somebody's CPI as -1 or 12, there is no check when you try to put in that data. However, a database will let you put in some integrity constraints. So whenever you define the CPI field, you can say that, it is between 0 to 10 and so, any attempts to enter our data value, for example -1, will result in an error and it will not let you enter. The logic is in the

database system itself not by the access. So, once you define the database of that CPI, you can say that, it is between 0 to 10 and that logic is within the database system. So, this is the 3rd point.

The 4th point is called the *atomicity of operations*. So, this is probably one of the most famous uses of database and as an example, let us take the bank transfer case. So, certain amount of money, so let us say 100 Rupees, is transferred from person A's account to person B's account. So, the atomicity of transactions defines that either the money is transferred completely or no money is transferred at all. So, what do I mean by that is that, it cannot happen that Rupees 100 have been deducted from A's account, but it has not been credited to B's account or the other way around, that rupees 100 have been credited to B's account, but it has not been debited from A's account. And you can see, what problems will happen if inconsistent credits and debits can take place. So, a database system encodes that entire operation of debiting from A's account and crediting to B's account as one operation or one transaction. We will see more about transactions later in the course. It mandates the atomicity of it. So, either this has happened completely. So, either 100 has been debited and credited both or it has not been debited credited either. So, this is the other important part.

Let us come to the 5th point. The 5th point is *concurrency*. So, very simply, concurrency means multiple operations can take place together in the database; the database generally takes care of those things as long as there are no conflicts in that particular data item. So, if A is transferring money to B and C is transferring money to D, they do not need to wait for each other and both the transactions can happen together. In file systems, it may not happen if all the transactions are opened in the same file. So, the same file needs to be first edited for A to B, then saved and then C to D should happen, but it depends also on the file system that is being used.

Probably the last important thing about databases or why it is useful over file system, is that of *security*. So, security of the data is paramount in databases and the database or the data fields can be made secure in multiple ways. So, first of all, for example, the employee database is secure, in the sense that, an unauthorized users may not gain access to any of this. So, that is one part of the security. The other and probably the more important part of the security is that, there are different access control levels in the database for different fields and different roles. For example, an employee may be able to see her own record, but not records of others. It may happen that way. It may also happen that student may see a CPI of herself,

but not of others, but a student may change her home address, but not her CPI. So, even if the record is about the student, even within a record, there can be attribute level security, attribute level access privileges, etcetera. And the security mandates that nobody, unless it is allowed by the database system (and these roles have to be defined very carefully while defining the fields of the database), will be allowed to change that. These are the six important points for the advantages of DBMS over the file systems.

Just to go over it, the first one is redundancy or inconsistency, then it is data isolation, then it is data integrity, the 4th one is atomicity of operations, 5th one is concurrency and the 6th one is security.

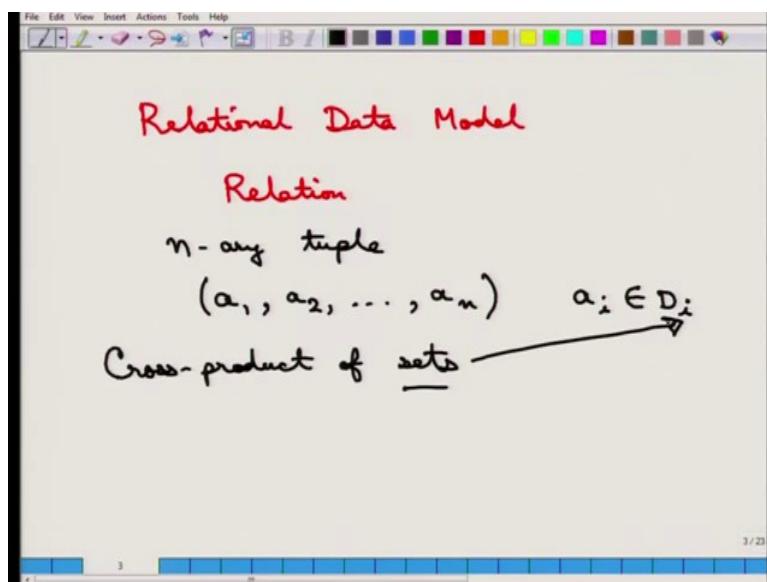
So, well, that ends the first part of the module, in the next module, we will cover the relational data model.

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

**Lecture - 02**  
**Relational Data Model**

Let us start on the next topic which is on Relational Data Model.

(Refer Slide Time: 00:13)



So let me start off by defining what a relation is. Well, very simply we all understand what a relation is. It encodes the relationship between different types of entities. So, it is defined as an  $n$ -ary tuple, so what does it mean is that, for example, a particular relation may be  $a_1, a_2$  up to some  $a_n$ . So, this is one relation. And each  $a_i$  comes from a domain  $D_i$ . So, relation is an  $n$ -ary tuple, it has got essentially  $n$  attributes, each attribute  $a_i$  comes from a domain  $D_i$  and a particular relation is formed of  $a_1$  to  $a_n$ . So, a relation is the cross product of the sets of  $D_i$ , These sets are corresponds to these  $D_i$ . So, this is the cross product of sets.

(Refer Slide Time: 01:37)

$a_1: \text{Name} = \{A, B, C\}$

$a_2: \text{street} = \{1st, 2nd\}$

$a_3: \text{city} = \{\underline{\text{Kolkata}}, \underline{\text{Delhi}}\}$

$r = \{ (A, 1st, \underline{\text{Kolkata}}), (A, 2nd, \underline{\text{Delhi}}), (B, 2nd, \underline{\text{Kolkata}}) \}$  | unordered

Set Table

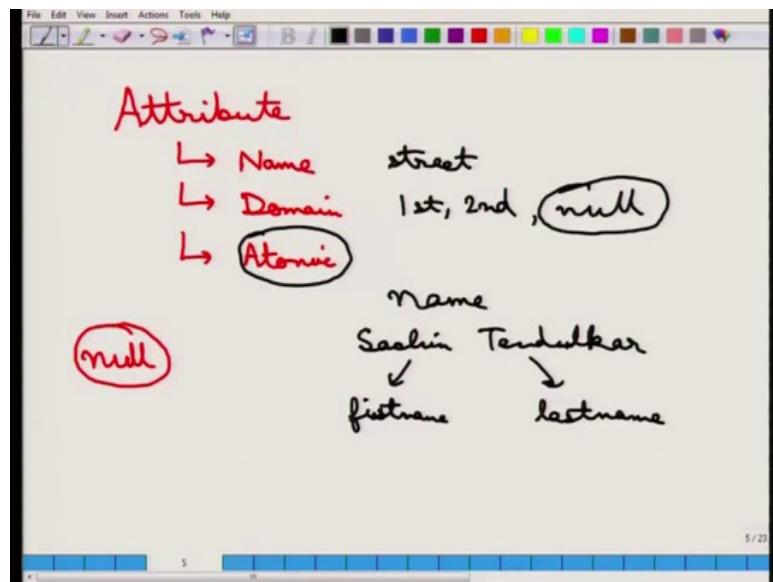
name	street	city
A	1st	Kolkata
A	2nd	Delhi
B	2nd	Kolkata

So, for example, let us take a particular example and say, we have *name* as the first attribute. So, this is one attribute, this is let us say,  $a_1$ , this attribute is name. And, Let us say, in the *name*, the particular elements that we are talking about are A, B and C. Let us take another attribute which is called street, which is again very simply say 1st street, 2nd street, 3rd street so on, so forth etcetera. Then let us take third one which is city and let us for example, say this is Kolkata, then Delhi so on so forth. This can be a particular relation. So, it is the subset of the cross product, so any combination can be part of a relation, so for example, I can say that the first n-ary tuple in this relation is  $(A, 1st, \text{Kolkata})$ . So, what does this mean, that there is an entity, whose name is A, whose street is 1st street and the city is Kolkata, this simply does mean that. And of course, it has it can get many such things it can have  $(A, 2nd, \text{Delhi})$ , it can have  $(B, 2nd, \text{Kolkata})$  and so on so forth and this simply defines the relation. Now, note that every member of this relation must be part of the cross product of the domain that we have defined, but it is not necessarily that everything is part of the relation. So, for example,  $(A, 1st \text{ and } \text{Delhi})$ , this does not define any relation. So, there is no such tuple like A, 1st and Delhi, This may happen. But all it means is that if you say that there is A, 1st and Kolkata this must come from this, so that is the definition of a relation.

So, that is the first thing, the other thing is relations are unordered, so it does not matter if we write A, 1st, Kolkata then A, 2nd, Delhi then B, 2nd, Kolkata or if we write A, 2nd, Delhi then A, 1st, Kolkata or B, 2nd, Kolkata. So, this is the set of course, and this is unordered, so the ordering has got no meaning and generally what happens is that, generally this is depicted

as a table. So, we have all have some idea of what a database looks like and generally what we do is that we depict it as a table. But, this is just a depiction, the relational data model does not depend on how you depict it, this is just a general depiction. So, this is how we will do, so this will go here A, 1st then this is Kolkata then there will be A, 2nd, Delhi then B, 2nd, Kolkata. So, this is just the same relation if we can write it as a table, so that is all about relations.

(Refer Slide Time: 05:16)



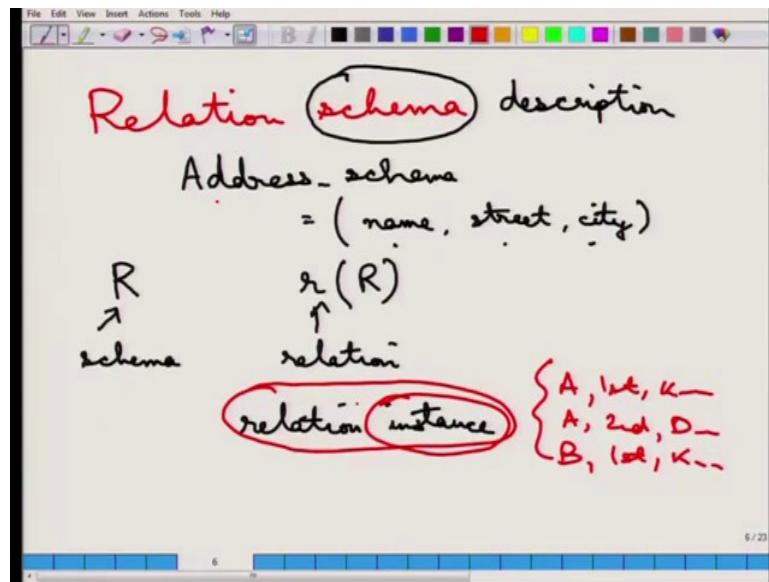
And the next important term that we define is the attribute. So we have been talking about attributes, but we will go a little bit more detail into what an attribute is. So, an attribute has got a name, so it is of course, coming from a domain, so it has a domain and it has got a name. So, for example, street is the name of that attribute and name is the name of that attribute itself and then, there is a domain for each attribute. So, if we go back to the last things, so this is street is an example and the domain here was first and second, then generally these attributes are atomic.

Now, what does atomic mean, this needs to be defined a little bit, atomic means that this cannot be subdivided any further. So, the the first street cannot be subdivided anything further, street may not be doing anything further, but let us take an example of name. So, the name of a person is generally, let us say we can take this name Sachin Tendulkar, this is actually not atomic, because this can be broken into a first name as well as last name.

So, it depends on whether you want to call it atomic or not, but generally attributes are

supposed to be atomic. So, that is of one thing, so these are indivisible and these are not set. So, that is the other meaning of atomic and then there is a special value called null, null is part of every domain, so domain of the street can be null. So, for a particular relation, for a particular row, for a particular entity if we do not know what is the value of the street, we can always depict as **null**. So, **null** is a special value that is part of every attribute.

(Refer Slide Time: 07:24)



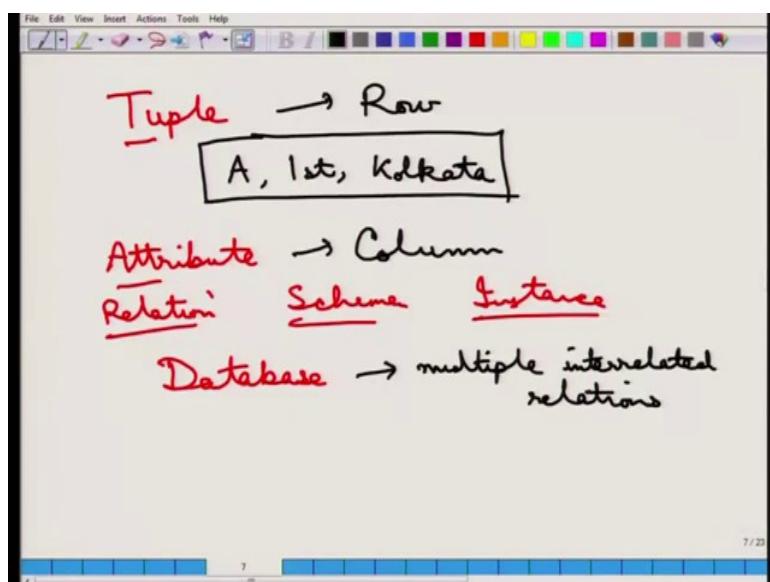
So, the next concept that we need to define is called a **relation schema**. so the sets that we defined earlier define the relation schema. So, for example, we can say, going back to our example, address schema. What is an address schema? An address schema says that it must have three fields, the first field is name, the second field is street and the third field is city. So, note that what a schema means, a schema essentially means that this is the description of the relation. So, schema means essentially a description and something more, but very roughly it means the description. And the description of the address is done by this address schema, which is, it contains a name, it contains a street and a city. And the relation is defined over a schema. So, if the schema is for example, if the schema is R, a particular relation r is denoted to be from this schema R.

If the R is the schema and a particular relation is r which within brackets, we have to say that schema of it. So, then it is very clear that the small r, the relation r is part of the R which is the relation schema and a relation instance, is what we saw last time. So, a relation instance, so let us highlight it, this is a relation instance, this is what we saw the table last time some

we saw this example A, 1st, Kolkata, etcetera, A, 2nd, Delhi, etcetera and B, 1st, Kolkata, this is a particular instance.

Because, in another relation it may happen in that there is A, 2nd, Delhi and B, 1st Kolkata and so on, so forth. This is a particular instance. So, this is called a relation instance. So once more a relation schema is a description of how the relation should be defined. For example, this address schema and the relation must say which schema it comes from and then there is a relation instance.

(Refer Slide Time: 09:51)



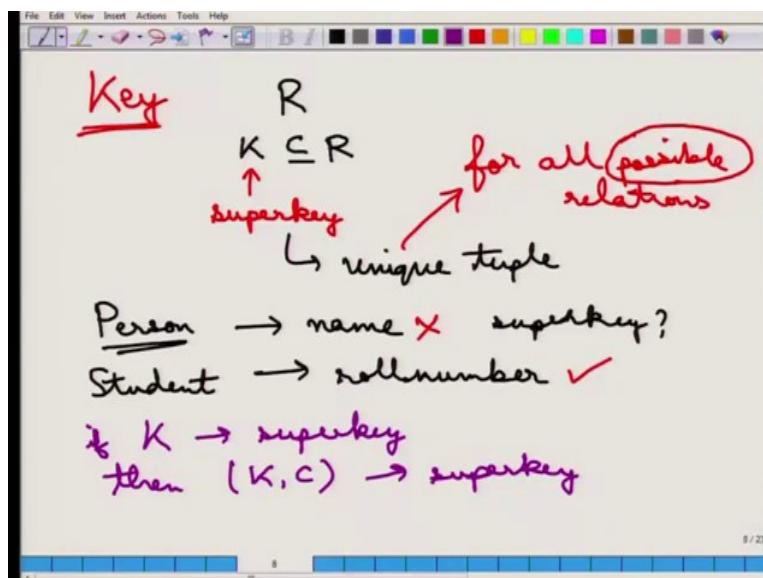
If we then go forward, then there is something called a tuple, this essentially corresponds to a row in a table, this is the one tuple. For example, is that (A, 1st, Kolkata) together consists of one tuple, this is essentially one row, so this is one particular value of the relation that is the tuple and we all understand probably what a tuple means, And just to complete this thing, a tuple corresponds to a row while what we are talking about an attribute that corresponds to a column.

So, these defines about a tuple, attribute, so far what we have got is we have got a tuple, we have got an attribute, we have got a we have defined a relation its schema, its instance, etcetera and now we will formally define what a relational database is. So, a relational database consists of multiple interrelated relations, so a database is a collection of multiple interrelated relations.

So, each relation of course stores about a particular relationship and between two or more relationships there is a connection and there is a relationship, which is also part of the database description. So, the database is not just the collection of the relations, but also the connections between the relations, so that is very important. So, this entirely defines what a database means and it must handle these problems as we went over in the last module, the problems of data isolation, data repetition, etcetera. So, a database tries to handle all of those things.

So, the next very important concept that we will do is called the key.

(Refer Slide Time: 12:04)



So, let us consider a particular relational schema  $R$ , now a subset  $K$  is called a super key of  $R$  if and only if the values of  $K$  are enough to determine all the values of the attributes of  $R$ . So, to repeat, suppose there is a tuple in  $R$  and you do not know all the attributes of tuple in  $R$ , but you know, what the super key attributes of the tuple  $R$  then it is enough to determine the other values of the tuple, so this is the unique identifying subset of attributes.

So, if you know this then you can complete the entire set of attributes, so a super key essentially identifies a unique tuple. So, to turn it around, we can also say that each tuple has an unique value of the super key.

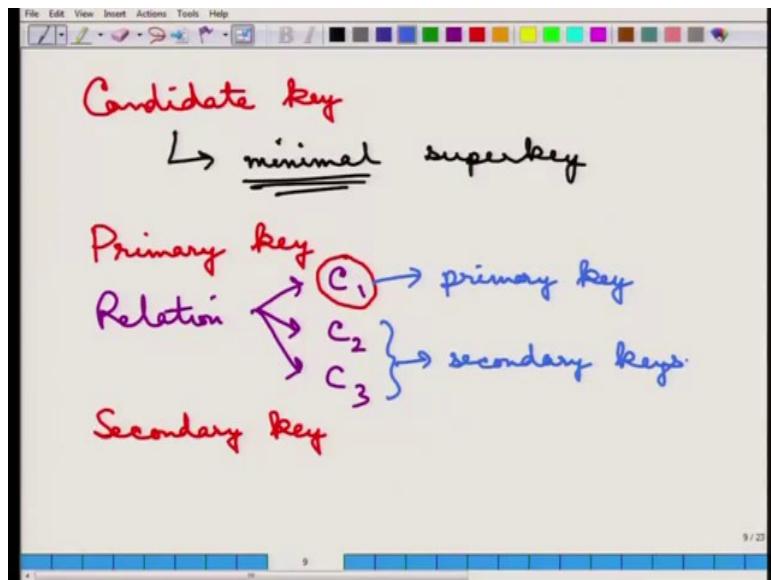
So, some examples. Suppose we consider a person as a relationship and let us consider some attributes, let us consider name as the first attribute, now is name a super key? Well, it is not.

Why? because two persons can have the same name. Now, very importantly and this is a very important point that you must understand, while we are talking about a database and what is a super key etcetera in the context of relation, it must be that the attribute, for example the name here should be able to distinguish the persons for *all possible relations*. So, this super key this is a unique tuple for all possible relations, so what does that mean is that, for a particular set of people for example, it may happen that none of them has a repeated name. So, the name is actually unique for them, but it is still not correct to say that the name is the super key. Because, it may happen that for another collection of persons with the same schema with the same set of descriptions it may happen that the two persons have the same name. So, this is very important. This is for *all possible relations*, so that is why name is not a super key. On the other hand suppose there is a student database and you take roll number. This is a super key, because the roll numbers are designated in that particular manner that every student has a roll number. So, no matter what set of students one takes which university or whatever you go to which institute etcetera, generally the roll number is unique. So, roll number is a super key for the student.

Now the other interesting thing about super keys is that, if K is a super key then anything mixed with K anything after K is also a super key. So, any super set of a super key is a super key as well, so this is much easier to understand.

So, if roll number uniquely identifies the student, roll number plus name will also uniquely identify a student, roll number plus street will also identify a student, roll number plus street plus city will also uniquely identify a student. Because, there is a roll number that uniquely identify, so a super set of a super key is also a super key.

(Refer Slide Time: 16:12)



So, that is the concept of a super key.

Then comes the concept of a *candidate key*, so as you can understand, the definition of super key is a little bit problematic, because any super set of a super key is also super key. So, what people generally tend to work with is what is called a candidate key. Candidate key is a minimal super key, It is a minimal, it is very important. It is a minimal super key. What does that mean? So, a candidate key is of course, a set of attributes. Do remember that every key when we talk about is a set of attributes.

So, a candidate key is again a set of attributes. Minimal meaning if you take any subset of that set it is not a super key any further. So, all the attributes of the candidate key are required to make it a super key. So, the candidate key, in some sense, is the tightest possible set of attribute that one can identify to make it unique and a super set of a candidate key is not a candidate key, because the super set of a candidate key means that it has got something more and; that means, it is not minimal.

So, this is the candidate key essentially gets rid of the problem of the superset of super key. So, that is the called a candidate key, so this is what is called a candidate key and the next important thing that comes is the definition of what is called a *primary key*. But, before we go to the primary key let me highlight one thing that a particular relation may have multiple candidate keys. So, it may have a candidate key 1, it may have a candidate key 2, it may have a candidate key 3, etcetera, etcetera, etcetera it may have multiple candidate keys. Now, note

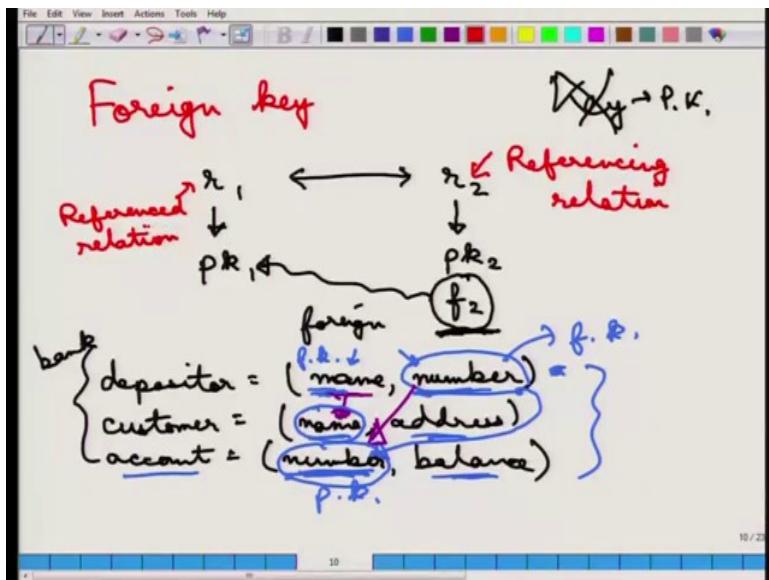
I am still talking about candidate keys of course, it can have multiple super keys, but it can also have multiple candidate keys.

Now, what is a primary key? Versus there is something called a primary key and then there is something called a secondary key. So, both the primary key and the secondary key are candidate keys, and one of the candidate keys is designated by the database designer as a primary key. So, now there is no theory behind this, this is what the database designer, designates one of the candidate keys as the primary key, the rest then becomes secondary keys. So, to repeat, a relation may have a multiple candidate keys one of the candidate keys is designated as a primary key and the rest are secondary keys. Now, if it happens that the relation has a single candidate key that candidate key is of course, a primary key with no secondary keys.

A question will now be coming is that is it guaranteed that every relation has a candidate key, the answer is yes, not in a very strict sense, but for most cases the answer is yes. So, in the worst case or what will happen is the most general cases if one takes all the attributes of a relation that is an uniquely identifying value of the relation, because generally relations are considered to be not repeating. So, the tuple values are not repeating, so there is this problem, there is this concept of the non-redundant information.

So, generally all tuples are unique, so which means that if all the attributes are taken together then that uniquely determines a tuple. So, in the worst case all the attributes of a relation together constitutes its candidate key and that is it I am so a primary key if there is only one candidate key that is a primary key.

(Refer Slide Time: 20:11)



So, next we move on to the last bit on the key thing which is called a foreign key, before we try to understand what a foreign key is we highlighted that relationships have different connections between them. So, suppose there are two relationship  $r_1$  and  $r_2$  with some connections between them and let us say the primary key of  $r_1$  is called  $pk_1$ . And this has got some attribute  $pk_2$  and then this has got some other attribute  $f_2$ .

Now, what may happen is that this  $f_2$  comes from the  $pk_1$  of the other relation. So, what I am trying to say is this attribute the  $f_2$  of  $r_2$  is the primary key of another relation. So, then this is called a foreign key, because this is a key of a foreign table, this is essentially the key in colloquial language means like a primary key, although there is nothing called a key, this is an important point just to note down there is nothing called a key, this key there is nothing called a key it, but generally people mean it is a primary key.

So, this  $f_2$  is a primary key of something else, so that is why it is called a foreign key. So, let us consider an example of a bank accounting system, where there is a depositor relationships. So, a depositor relationship has got all of these. Let us say, the depositor has a name and a number and let us say there is a customer relation. So, this is all part of the bank database, there is a customer which has got a name and some address and then there is an account which has got a number and the balance in that account, this is an example of a bank schema.

So, let us try to decode what does this mean and the account is about the different deposit accounts that we got. So, it has got an account number just like our savings bank account

number and then the balance and the customer is just like you and me, everybody, we have got a name. And let us assume for the time being that the name is the primary key and we have got an address. Now, the depositor, this, defines a relationship between which customers have which accounts.

So, these numbers essentially mean that customer one whose name is whatever something has this account number. So, this refers to this account, so this is essentially a foreign key, because this is a number here, so this is a primary key and this number essentially refers to this number. So, that is why it is a foreign key and similarly this name essentially refers to this name, the customer name essentially refers to this name, so that is why these two are foreign keys.

So, this is also a foreign key and this is also another foreign key. This may happen. So, this is why the concept of foreign key is important and we will see later much more about these foreign keys, essentially just a very quick digest is that if a particular name does not happen in customer, it cannot happen in the depositor and if a particular number does not occur in this account database it cannot happen in the number. Because, I mean there cannot be information about an account for which there is no account balance.

So, that is the foreign key and just to complete the story about foreign keys. So, this is called the *referencing relation*, the one which has got the foreign key and this is called the *referenced relation*.

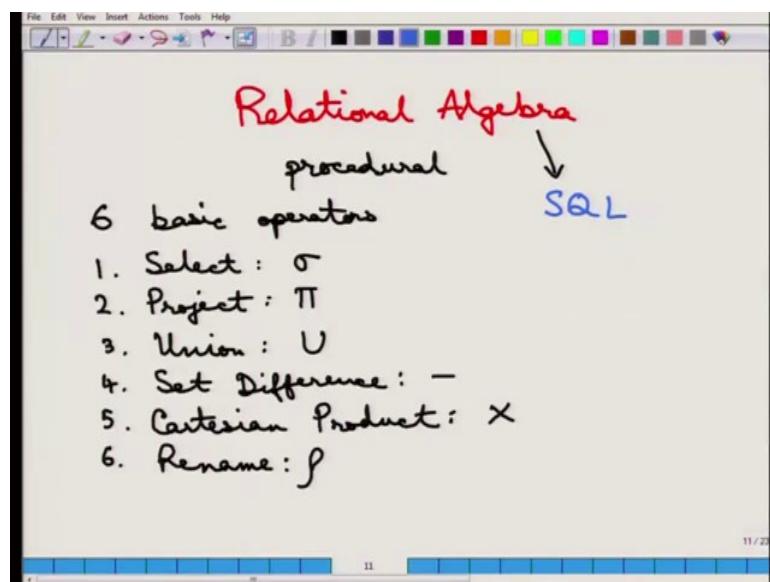
So, that completes this module on the relational data model, so just to recap we studied about what relations are, what tuples are, what attributes are, how is the database defined in terms of those things and then very importantly, about the different keys. So, we started off with super keys, then we defined candidate keys and then we defined primary keys and secondary keys and finally, foreign keys and how is the foreign key helpful or useful.

Thank you, so in the next module we will talk about relational algebra.

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

**Lecture - 03**  
**Relational Algebra- Basic Operators**

(Refer Slide Time: 00:15)

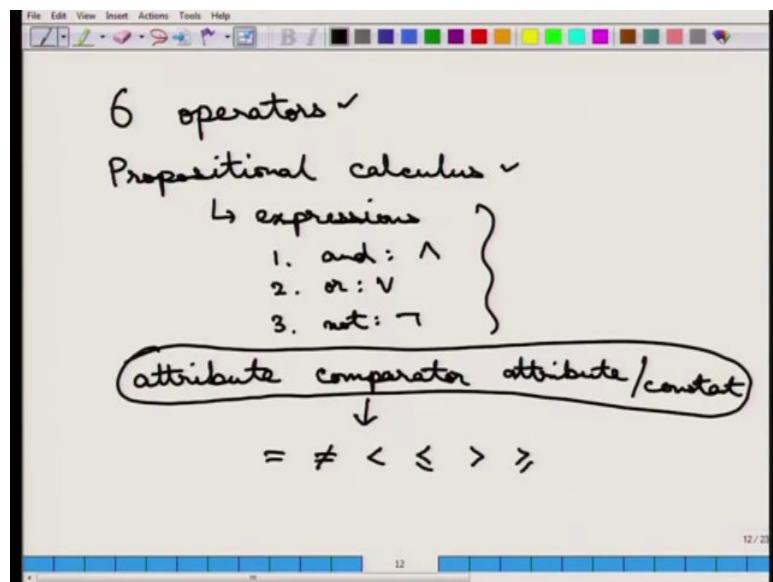


Welcome back, so we will start on our next module, which is on **Relational Algebra**. So, this module is on relational algebra and what do we mean by that, is that, this is a procedural language, to define the database queries. So, they are very important distinction first of all, that this is a procedural language, so this is to define the database queries.

Essentially we will first talk about 6 basic operators; that is the first module that we will be talking about. And the 6 basic operators are, the first one is called **select**, which is denoted by sigma. The second one is **project**, which is denoted by pi, then **union** or the set union essentially, **set difference**, then **Cartesian product** and the last one, the sixth one is called **rename** operator.

Now, before we go forward let us understand that relational algebra is what forms the basis of the SQL or the Structured Query Language that, we will study later. So, relational algebra defines the theory behind SQL, it is not exactly the same as SQL, but it defines the theory. So, let us go over.

(Refer Slide Time: 02:01)



Other than the 6 operators, so there are these 6 operators that we are going to study, then relational algebra uses propositional calculus. So, this consists of expressions, in the propositional calculus, there are expressions. The expressions are connected by these three basic operators AND, OR and NOT. So, each term is of the form attribute, there is a comparator with an attribute or a constant.

So, each term is essentially an attribute is compared with another attribute or a constant. And the comparators are the standard 6 comparators, which is equal to, not equal to, less than, less than equal to, greater than and greater than equal to. So, this defines the entire relational algebra, the basic relational algebra. So, we have these 6 operators, then the propositional calculus of these three expressions and the terms are consisting of this form attribute comparator with attribute and constant.

(Refer Slide Time: 03:35)

So, let us go one by one over the 6 basic operators, the first one is the **select** operator. Now, just to note that this is different from the select that we understand in SQL, we will go over all of them in much more details, but, the definition of the select is the following. Now, here I am trying to write down a formal definition of select, remember that  $r$  stands for a relational instance, so it is a relational. So, sigma is the select operator and  $p$  is the predicate on which the selection is done by the way, so this selects all tuples  $t$ ; such that of course,  $t$  is the part of relation. So, it selects all tuples from the relation and  $p$  of  $t$ . What does this mean? This means that, the predicate, if one applies the predicate on this tuple  $t$  that is correct, so this is called the predicate, so this predicate is correct. So, just once more to say, this essentially selects all predicates this selects all tuples from  $r$  on which the predicate is satisfied.

One important thing to notice that, if you do it on a relation  $r$ , then it does not change the schema of  $r$ . Now, let us take an example; that is the best way of understanding this, so here is the example. So, suppose a relation has got four attributes  $A, B, C, D$ , which are these values, let me just write down some values. So, there are just four tuples in this relation  $r$  and now, suppose we are applying the following selection on this  $A$  equal to  $B$  and  $D$  is greater than 5 on this  $r$ . So, then what is the answer, so first of all, when we say that it does not change the relationship schema.

So, the first thing to note down is the answer will be of the form  $A \ B \ C \ D$  as well again,

nothing will be changed. So, what is the way to do it is that, let us test the first tuple. So, we test whether A equal to B, which is A equal to B is correct and B is greater than 7, so this is correct. So, this becomes part of the answer set.

In the next one again we test A equal to B, no A is not equal to B, so this is wrong this is not part of the answer set, this one 2 equal to 2, which is fine A equal to B, but 3 is not greater than this, so this is also not part of the answer set. And the fourth one 2 is equal to 2 and 6 is greater than this, so this is part of the answer set. So, the answer consists of these two tuples of the same form (A, B, C, D) with (1, 1, 2, 7) and (2, 2, 8, 6). So, that is the select operator, let us move ahead with the next operation, which is called the project.

(Refer Slide Time: 06:35)

A	B	C
1	1	5
1	2	5
2	3	5
2	4	8

A	C
1	5
2	5
2	8

Now, **project** does change the schema of the relationship that even we apply it on. So, project essentially is of the form (Refer Slide Time 06:48). So on a relation r the project operator is applied on certain attributes  $A_1$  to  $A_k$  and it essentially just selects out those attributes from the projections. And duplicate rows are removed, so duplicate is removed this is one important thing with the basic projection, duplicates removed. This is done, because relations are sets, so that is why duplicates are removed.

So, let us take an example, so suppose there is this r is your (A, B, C) with the following four tuples (1, 1, 5), (1, 2, 5), (2, 3, 5) and (2, 4, 8). Now, suppose the following projection is done A, C on r, so the projection the schema of this is the same as whatever is projected on, so if A and C are projected on, the schema is A and C. And then, it is simply selected out, so from

here this simply comes as (1, 5), then here this again should have come as (1, 5), but then it merges. So, it does not produce any more tuple. From here it produces (2, 5) and from here, it produces (2, 8); the answer for the project is simply this.

(Refer Slide Time: 08:25)

<u><math>r</math></u>		<u><math>s</math></u>		<u><math>r \cup s</math></u>	
A	B	A	B	A	B
1	1			1	1
1	2	1	2	1	2
2	1	2	3	2	1
				2	3

The next operation is the **union** or the set union. So, set union of the  $r$  union  $s$  is all set of tuples; such that  $t \in r$  or  $t \in s$ , this is very easy to understand. This is exactly the same as the set union, once more duplicates are removed, because these are rows. So, one important thing is that  $t$  and  $r$  and  $s$  should have the same attributes, if they do not have then some renaming needs to be done, such that  $r$  and  $s$  follows the same thing.

And let us just work out an example, so suppose this is  $A$ , this is  $(A, B)$ , this has got (1, 1), (1, 2), (2, 1) and  $s$  has got the same schema  $(A, B)$ , which has got (1, 2), (2, 3). So, the union of  $r$  union  $s$  produces again  $A B$  and this is simply copied, so (1, 1) is copied, (1, 2) is copied from  $A$  and (2, 1) is copied from  $A$ . Now, then  $(A, B)$  from  $B$  is not copied any further, it duplicates it, this is duplicated, so (2, 3) is simply copied, so it forms this. So, these are simple things to understand, so union again has the essentially just they set union, nothing more.

(Refer Slide Time: 09:52)

Set Difference

$$r - s = \{ t \mid t \in r \text{ and } t \notin s \}$$

<u>r</u>		<u>s</u>		<u><math>r - s</math></u>	
A	B	A	B	A	B
1	1			1	1
1	2	X		2	1
2	1			2	1

16 / 27

The next operation is also the simple set operation which is the **set difference** and this is again, what we understand from normal set operation is nothing much difference. So,  $r$  minus  $s$  is the set of all tuples; such that  $t$  is in  $r$  and  $t$  is not in  $s$ . So, once more this does not change the schema and renaming etcetera is done and set, so because these are sets duplicates are removed and an example is the best way.

So,  $r$  this has got  $A B$ , which is  $(1, 1), (1, 2), (2, 1)$ .  $s$  has got  $(A, B), (1, 2), (2, 3)$ , so  $r - s$  everything that is in  $r$ , but not in  $s$ . So,  $(1, 1)$  is not here, so  $(1, 1)$  finds its way through.  $(1, 2)$  it is here, so  $(1, 2)$  doesn't find its way through.  $(2, 1)$  is not there, so  $(2, 1)$  finds its way through as well. So, it is just  $(1, 1)$  and  $(2, 1)$ . So this is the set difference.

(Refer Slide Time: 11:08)

So, then the next operator is **Cartesian product**, so Cartesian product is a little bit more interesting. So, it takes out two sets  $r \times s$  the first thing that it does is that the schema is changed. So, the tuples that have produced have attributes from both  $r$  and  $s$ . So, what it does is that it produces tuples of the form  $t q$ ; such that the  $t$  part of it comes from  $r$  and the  $q$  part of it comes from  $s$ .

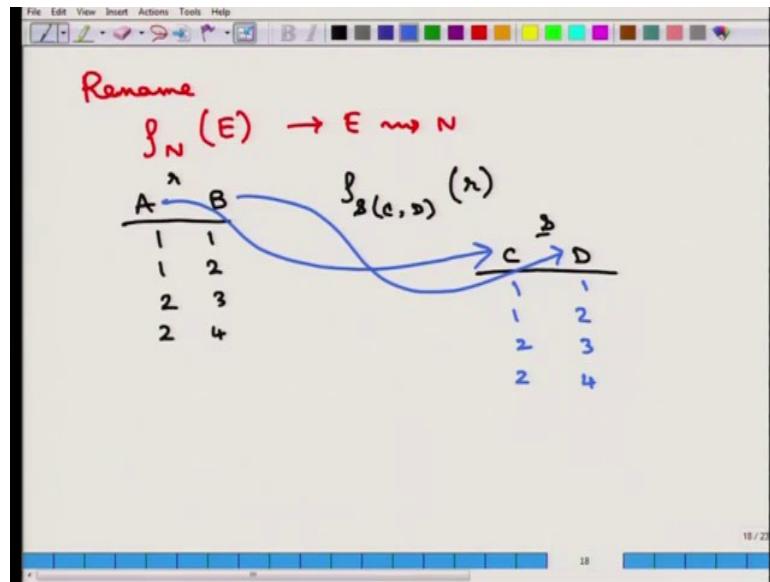
So, very importantly this is of the form  $(t, q)$ , which is not the same as either  $t$  or  $q$ , so it changes the schema. And, if the attributes are disjoint, then there is no problem it is just an addition of the schema, the addition of the attributes of  $r$  and  $s$  and if these are not disjoint then some renaming must be done. So, what do we mean by that is that, we will go over all of these things in probably much more detail later, but just to highlight it, what it means essentially is that if both  $r$  and  $s$  have the same attributes  $A$ , then it should be called  $r.A$  and  $s.A$ .

Now, here is an example, a complete example on  $r$  and  $s$ . The schema of  $r$  is  $(A, B)$  and suppose it contains, simply  $(1, 1)$  and  $(2, 2)$  and suppose  $s$  the schema of  $s$  is  $(C, D, E)$  and it computes  $(1, 2, 7)$ ,  $(2, 6, 8)$  and  $(5, 7, 9)$ . Now,  $r \times s$ , the first thing to notice, what is the schema of going to be,  $r \times s$  now since the attributes sets are disjoint. So, the schema is essentially all of them, the union of all of them, it is  $(A, B, C, D, E)$ .

Now, what are the values of these, the first thing is that essentially, what is done in a Cartesian product is that first tuple from  $r$  is taken and it is applied and the attribute union is

applied with all from s. The first tuple that it produces is (1, 1, 1, 2, 7), then it is (1, 1, 2, 6, 8), then it is (1, 1, 5, 7, 9). Now, similarly the second one is taken and all of these things are done, so it is (2, 2, 1, 2, 7), (2, 2, 2, 6, 8), (2, 2, 5, 7, 9), so that is all about the Cartesian product, so it changes the schema.

(Refer Slide Time: 13:56)

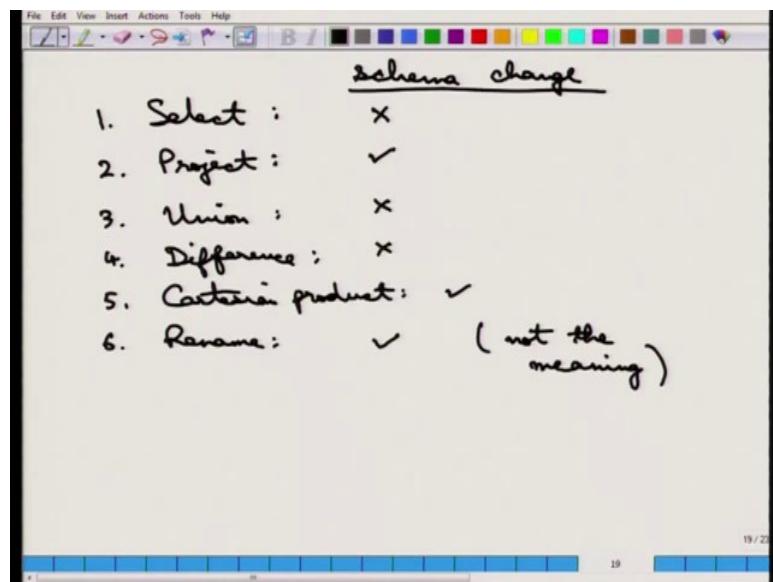


The next basic operator is called **rename**, it is a very simple operation it essentially just renames, so it renames it simply returns E, but now its name has been changed to N that is all. So, it simply renames it and it does not do anything else, so the very simple example of this first thing is that suppose this is (A B). So, this is (1, 1), (1, 2), (2, 3), (2, 4) does not matter, this is r and the operation that is applied is on r, which is called, let us say you change it to (C, D) on r.

So, what it returns is s with these values (C, D) this is the same s, so it simply copies on (1, 1), (1, 2) there is nothing more, but what it has done is that A has been changed to C and B has been changed to D. Rename of the operation has been done. And, so the schema is actually changed, but not the meaning of it.

Now to summarize all of these things, so let us see.

(Refer Slide Time: 15:16)



So, the first one was select. “Schema Change”. Select does not change the schema. The second one was project it does change the schema. The third one was union it does not change the schema. The fourth one was difference, it does not change the schema. The Cartesian product, of course, does change the schema. And rename actually changes the schema, because instead of (A, B) it returns new name (C, D), but not the meaning. The meaning is not changed. So, that is all about these things and then, we can take all of these operators and can compose them, so we can apply them one after another and that is what I mean by composition of operators.

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

**Lecture - 04**  
**Relational Algebra: Composition of Operators**

Welcome. Today we will talk about Composition of Operators. So, in the last time we saw some of the basic operators and today to understand how more than one operators can be used in the single query and we will also see certain queries for that.

(Refer Slide Time: 00:25)

r		s		$\sigma_{A=C}$ ( $r \times s$ )	
A	B	C	D	E	
1	1	1	2	7	✓
1	1	2	6	8	✗
1	1	5	7	9	✗
1	2	1	2	7	✓
1	2	2	6	8	✗
1	2	5	7	9	✗

So, this is called a **Composition of Operators**, so the operators can be applied one after another and it has to be defined in what way the composition can take place. So, it essentially uses multiple operations. So, for example, suppose  $r$  is  $(A, B)$  and  $(1, 1), (1, 2)$ . This is the same example that we saw earlier. And  $s$  is  $(C, D, E)$  with  $(1, 2, 7), (2, 6, 8)$  and  $(5, 7, 9)$ . Now, suppose the operation that we are working on is this  $\sigma_{A=C} r \times s$ . Now this has to be understood in which way it is happening. So, this happens first this one is being done, this is number 1 operation then after that this is done, so this is number 2 operation. So, essentially what is being done is, initially  $r \times s$  is produced. So,  $(A, B, C, D, E)$  that is being produced and that is the entire  $(1, 1 \dots$  let me just write it down to make it complete  $\dots 2, 6, 8), (1, 1, 5, 7, 9), (1, 2, 1, 2, 7), (1, 2, 2, 6, 8), (1, 2, 5, 7, 9)$ . So, this is the  $r \times s$  that is produced then the  $\sigma_{A=C}$  is produced on the same thing.

So, the first one, the first  $r \times s$  changes the schema, the second one does not change the schema and the second one produces the answer which is on  $A = C$ . So the first one is correct, this one is right, this one is wrong, this one is wrong, this one is again right. So, (1, 2, 1, 2, 7) this is wrong, this is wrong that is it, this is the final answer for the query of  $r$ . So, well that completes the set of basic operators, so we will go over the examples next.

So, let us work on with some of the examples for the six basic operators that we just saw.

(Refer Slide Time: 02:57)

Banking

- branch (bname, bcity)
- customer (cname, ccity)
- account (ano, bname, bal)
- loan (lno, bname, amt) ←
- depositor (cname, ano)
- borrower (cname, lno)

Find all numbers loan of £ 100 or over

$\pi_{lno}(\sigma_{amt > 100} (loan))$

So, here is a banking example that we will use many, many times, so let me just explain the example. So, there are six different relations in this, the first one is the *branch*, so the *branch* has the attributes, branch name, branch city. So, where is the branch name and which city it is in, the second one is the *customer* relationship, so all the customer. So, the customer name and let us say the customer city, so which city the customer is in.

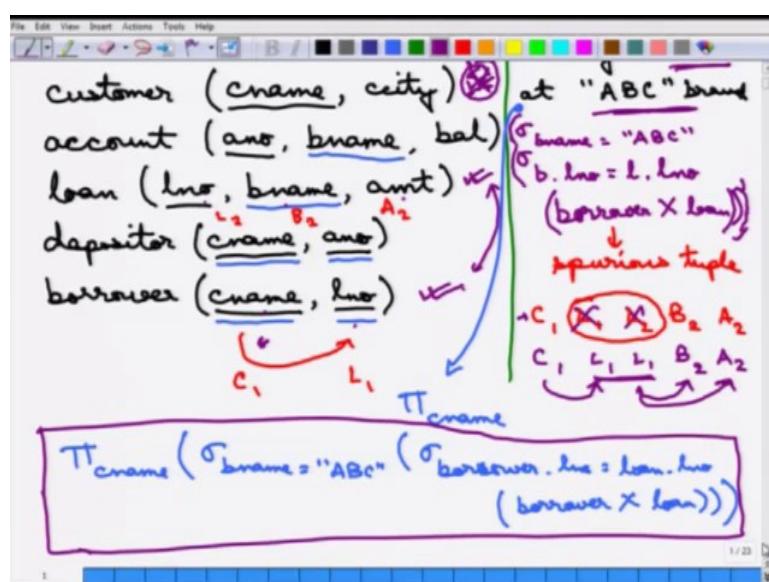
And by the way, I am underlining primary keys, so branch name is the primary key and customer name is the primary key. Then there is an *account*. So, the *account* is about all the accounts that are there, so the account name is the primary key and which branch it is in. So, the branch name, so branch name as you can see is a foreign key, and balance. So the foreign key let me highlight it using the blue color and then let us go to the next thing which is *loan*. Now, loan is about a same kind of thing like an account, but it is a loan number and a branch name and the amount of the loan. Once more, this is the primary key and this is the foreign key. Finally, there are two things which are, *depositor*, depositor has who is the customer that

took this customer name and which account number that is it. So, both are primary keys together both are primary keys and both are actually also foreign keys. We will come to nuances of this later. But, suppose and the last one is the *borrower*. borrower has a customer name, so, who borrowed what and which loan that is borrowed, So once more these are both primary keys as well as foreign keys.

So, with these things let us try to solve certain types of queries, so let me use this part to use this thing. So, the first query that we will try to solve is, find all loans of Rupees 100 or over. So, how do we solve it? So, this is first of all we need to understand that this is a query about a loan, so this is the loan table that we need to look into.

And essentially there is a amount attribute that we need to use and using that we can simply do a this thing. So, select all tuples from loan, where this amount is greater than equal to 100. That is it. So that solves this query, so find all loans of Rupees 100 or over. So, this returns everything loan numbers, so find all loan numbers of loans which has got rupees 100 or over. So, it is kind of the same thing, but except now what needs to be done, so it has to be the same kind of thing. So, you first find out all loans that are greater than 100 or over, but we only need the loan number. So, you project it on the loan number, that is it. This is the important part we just project it on the loan number. So, this is the important part that is the change from the previous query.

(Refer Slide Time: 06:47)



So, let us now move on to a little complicated query, so the query is the following. Find

names of all customers, having a loan at let us say ABC branch, so the branch name is ABC. So, now, the important part is to do that we need to find the names of all customers, so the first thing that we need to use is this table, the customer table. But, unfortunately the customer table by itself does not contain any information about the loan. So, for the loan we need to use this table, where the branch name is there. But again what happens is that, the loan table will contain the branch name, etc and we only need to find out the names of all the customers. So, instead of this table we may use this table, the borrower, because that contains the customer name. So, we can simply use instead of this table, we can use this table and we need to do whatever borrower and loan, so this is the Cartesian product that we need to take. So, if we take the borrower Cartesian product of loan we get the information of customer name, loan number, loan number, branch name and amount for all the possible loans.

But, this will generate what is called a *spurious tuple*, if we just do this Cartesian product this will generate what is called a spurious tuple, because what will happen is that. So, there will be some customer name corresponding to some loan number, let us say customer name C1, corresponding to loan number L1 and then there is a loan number L2 here with some let us say B2 and A2 this will generate tuples of the form (C1, L1, L2, B2, A2) which is not useful, because the loan numbers are different.

So, what we need to do is to ensure that this is the same loan number. So, for that what we will do is, instead of just doing this, what we will do is that we will select from this table everything, where the borrower dot loan number. So, let me just write it short hand  $\sigma_{b.lno=l.lno}$ , so this we need to select it on this table. So, this will then get rid of tuples like this and it will essentially only select tuples of the following form. So, this is what we require and this essentially means the customer C1 has the loan L1 and corresponding to the loan L1 this is in this branch B2 and has the amount A2. Now, we only need to find out the loans, where this is at ABC branch, so what we need to do is to do another selection on top of this. So, this is one part, this is another part, so then we need to do sigma over, everything where the branch name is ABC, so this you have to select on branch name is equal to ABC and that is applied over this entire thing, so this is what it is done, so this is the entire answer to this query.

So, if we now want to find names of all customers, then this is still not complete then we need to put in some more level here and let me just try to do it here. So, this is another. There are three brackets here, so this will be somewhere here, this we use to do a  $\pi_{cname}$ , because

we only need to do the customer name. So, essentially if we go down a little bit, this is a way to write down the entire query, this is

$$\pi_{cname}(\sigma_{bname='ABC'}(\sigma_{borrower.lno=loan.lno}(borrower \times loan)))$$

and let us

complete that, borrower cross loan. So, this is the answer to the entire query that we were doing.

So, this is an example that shows you how quite a complicated query, like find names of all customers having a loans there are many, many things here can be broken down into small, small parts and then solved one at a time and we can do many more such queries. So, for example, we can use this to say well instead of having a loan at ABC branch find me all customers having a loan at ABC branch, but not any account.

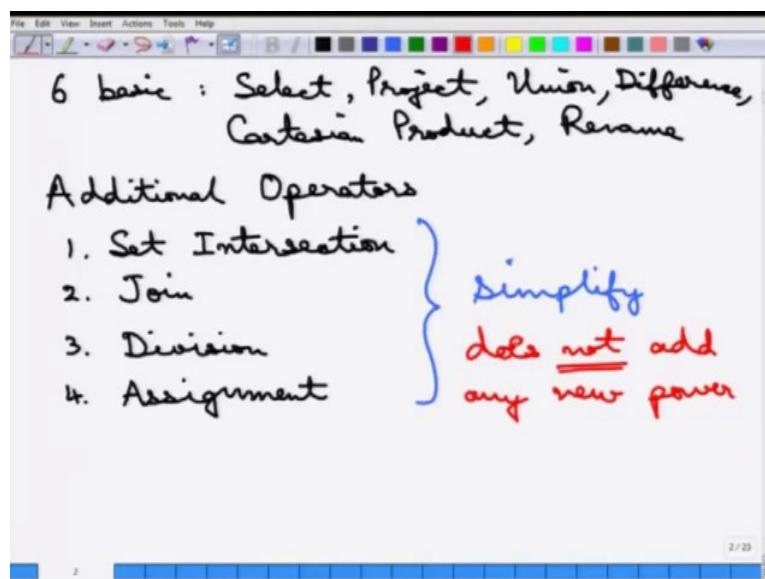
So, what you then need to do is, you need to ensure that the customer does not have an account. So, you need to take the set difference from these depositors, so the depositor needs to be set difference done with that. So, we will leave this as an exercise to you, but the basic idea is, how to use only this six of these operators, the basic operators to solve different queries. So, if that is all done, so in the next module we will move into the additional operators.

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

**Lecture - 05**  
**Relational Algebra: Additional Operators**

Last time we saw six basic operators.

(Refer Slide Time: 00:12)



Now, we will introduce some additional operators. Let me first enumerate them and then I will go over what does it mean. So, first one is **intersection** or the set intersection, then the second one is **join**, third one is called **division** and the fourth one is **assignment**. So, essentially what these additional operators let you do this simplify the queries, but does not add any new power to the basic relational algebra.

So, any of the queries that can be solved using these new four additional operators can also be solved using the six basic operators. So, these four operators do not add any power to the type of queries that can be solved. However, this simplify writing the query very heavy and we will see examples of that later, but let me first go over each of these operators to see what they mean.

(Refer Slide Time: 01:42)

Set Intersection

$$r \cap s = \{t | t \in r \text{ and } t \in s\}$$

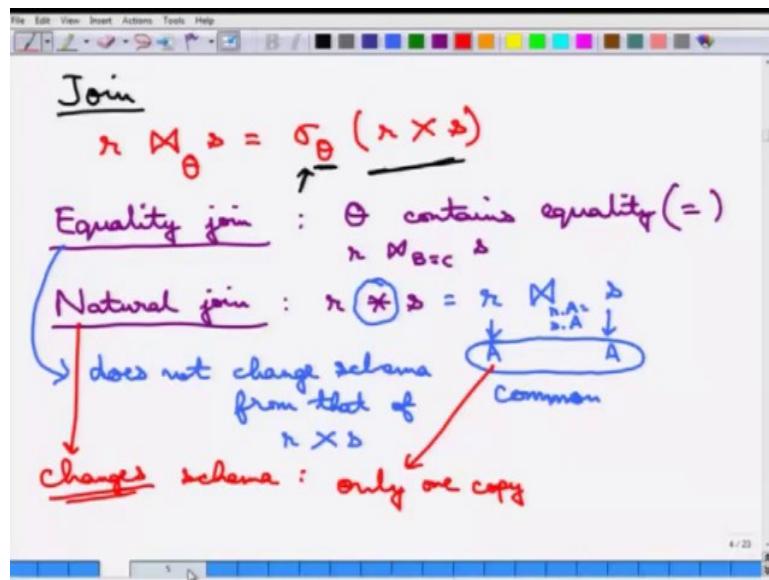
<u>r</u>		<u>s</u>		<u><math>r \cap s</math></u>	
A	B	A	B	A	B
1	1	1	2	1	2
1	2	2	3		
2	1				

$$r \cap s = r - (r - s)$$

So, the first one is set intersection. Set intersection is very simply the set intersection that we all understand for normal sets. So, essentially  $r \cap s = \{t | t \in r \text{ and } t \in s\}$  so it has to be both the cases and just like the set union and the set difference, they have to have the same schema etcetera, etcetera and it is quite simple actually. So, just to complete let us have an example, so suppose this is  $(A, B)$  with  $(1, 1), (1, 2), (2, 1)$  and  $s$  is the same schema  $(A, B)$  with  $(1, 2), (2, 3)$  that is  $r \cap s$  will produce again  $(A, B)$ . So, the thing that is common in both the cases is simply this, so the only answer is  $(1, 2)$ .

Now, the question is we are claiming that set intersection is not a new operator, it can be defined using the other operators and the answer to that how do we do that, because essentially you can see that set intersection,  $r \cap s$ , can be written in terms of the set difference operator. So, use the set difference operator, the set intersection can be written, so it does not really add any new power.

(Refer Slide Time: 03:08)



So, let us go to the next operator, **join** and this is a very, very important operator as we will see, but again it does not really add any new power, but let us first define what join is. So, the symbol for join is this  $\bowtie$ , so this is like two triangles facing each other and we define there is a condition  $\theta$ , so this is essentially  $\sigma_\theta(r \times s)$ . So, now, we can understand why I have been saying that this does not add any new power.

So, essentially what it does is it takes a Cartesian product and then select certain tuples based on the predicate theta. So, it is a selection using theta on the Cartesian product and we have already seen one example. So, the when we took a *borrower*  $\times$  *loan*, when we took that Cartesian product and then applied a theta which is *borrower.lno* = *loan.lno*, this is essentially a join using this condition.

Anyway, so let us see an example, so join is actually very useful, so there are some versions of join that are defined, the first one is called an *equality join*. So, equality join essentially is that the  $\theta$  condition contains equality (=), that is the equal to operator. So, for example,  $r \bowtie_{B=C} s$ . So, this equality operator is there, that is why it is an equality of join.

The next very important part is called a *natural join*. So, this will come up many, many times when we are studying SQL, etcetera later on and by join generally people do mean only natural join. So, natural join by the way has its own operator, this  $r$  it can be just denoted by this star operator (\*), this is essentially equal to  $r \bowtie s$ . Now, to define a natural join between

$r$  and  $s$ ,  $r$  must have an attribute  $A$  and  $s$  must have the same attribute  $A$ . So, I mean, so  $r$  must have some attribute which has the same name and the same schematic meaning as  $s$ , so it is essentially saying  $r.A = s.A$ . So, when there is the same attribute between two relations  $r$  and  $s$  and an equality join is proposed between  $r$  and  $s$  on that attribute on only that attribute, then it is called a natural join. So, there has to be a common attribute. So that is the thing there has to be a common attribute and that is why it is defined.

Now, a couple of interesting points about this is that equality join for say does not change the schema, or any join does not change schema from that of  $r \times s$ ; However, natural join changes schema.

So, how does it change the schema? It essentially has the common attribute only one copy of the common attribute. So, it does not store both  $r.A$  and  $s.A$  which is obvious, because  $r.A$  must be equal to  $s.A$ , so it keeps only one copy, so it does change the schema little bit, it essentially removes the redundant attribute. So, that is one thing to remember and let us see some examples of join.

(Refer Slide Time: 06:56)

A	B	A	C	A	B	C
1	1		1	1	1	2
1	2		2	1	2	2
2	1		3	2	1	3

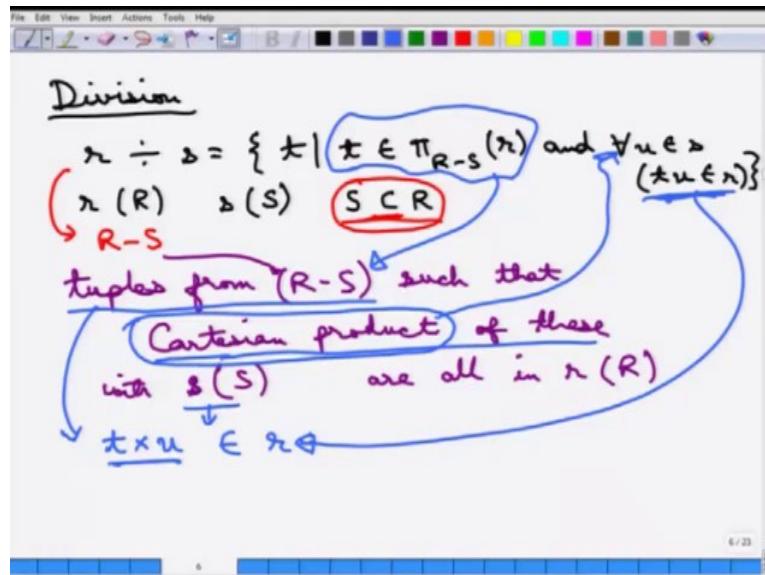
So, suppose this is  $r$  with the schema of  $(A, B)$  and these are the tuples there  $(1, 1)$ ,  $(1, 2)$ ,  $(2, 1)$  this is  $s$  with the schema of  $(A, C)$  and  $(1, 2)$ ,  $(2, 3)$ . Now, if we do  $r * s$  this can be also denoted as  $r \bowtie s$ , so this also stands for natural join, so this is natural join, so the answer for this is that. So, the common attribute is first identified which is  $A$ , so  $A$  is the common attribute, so only one copy of it is kept, then the rest are copied, so  $B$  and  $C$  and this is the

schema of (A, B, C) then only those tuples are taken into account where A is same.

So, between this the A is the same, so this is copied, so only one copy of one is kept, B is 1 and C is 2. So, this is that between these two, these join does not happen, because the A is not the same, then we move on this joint does happen, because this 1 is equal to 1 and then the copy that is kept is 1, for B it is 2, for C it is 2 that is the thing. And similarly, this does not happen, because the joint conditions do not agree, this does not happen, because the join condition do not agree and this does happen, because 2 is the same and this produces (2, 1, 3).

So, this is the answer for the natural join, so this is once more the natural join. So, this is one example of how to define natural join, etcetera.

(Refer Slide Time: 08:43)



So, let us now move on to the next operator which is called **division**, now division is a slightly complicated operator. Essentially what division tries to do is, let me give you the first big definition, the formal definition of division,

$r \div s = \{t \mid t \in \pi_{R-S}(r) \text{ and } \forall u \in s, (tu \in r)\}$  . So, this is really complicated as it sounds, but let me tell you what it does, essentially let me write it down this following way. First of all the schema of r. So, suppose r follows the schema R and s follows the schema S.

Now, for the division operator to be applied it must be that, the schema  $S \subset R$ , this is an important condition, this must be holding. So, this must be S is a subset of R, this must be

happening; otherwise, this division operator cannot be applied. The number two is that the schema of  $r$  divided by  $s$  essentially becomes  $R - S$  that is why this is important. But, more importantly what does  $r \div s$  does is that it chooses tuples from this particular schema  $R - S$  such that Cartesian product of these with  $s(S)$  are all in  $r(R)$ .

So, suppose you take a tuple from  $r - s$  suppose the tuple is  $t$ , so Cartesian product of this with  $s$  is suppose this tuple is  $u$ . Now, the Cartesian product... So,  $t u$  must be part of your  $r$  relation  $r$ , because this is what the Cartesian product you have taken, so this is what this condition says. Now, this is for all  $u$ 's, the Cartesian product, because it is a Cartesian product this captures all  $u$ , this is the all  $u$  part and this part is the  $t$  belonging to  $r$  and this is essentially saying what is the schema of this, this is from this part, so which is what it is doing. So, it is a little complicated example and let me go over an example to say what I have been saying.

(Refer Slide Time: 11:31)

<u><math>r</math></u>				<u><math>s</math></u>		<u><math>r \div s</math></u>	
A	B	C	D	C	D	A	B
1	5	2	7	2	7	1	5
1	5	3	7	3	7	1	6
1	6	3	7			2	6
2	6	2	7			3	6
2	6	3	7				
3	6	2	7			3	5
3	6	3	7				
3	5	3	7				

So, here suppose  $r$  is  $(A, B, C, D)$ . This is the schema of  $r$  and this is your  $s$  which is just  $(C, D)$  which is just  $(2, 7)$  and  $(3, 7)$ . So, first of all  $r \div s$ , so first of all we must decide what the schema is, now the schema is, as I told you, this is essentially you take this schema and this schema and it is the of difference. So, the schema is only  $(A, B)$  that part should be easier to understand, now suppose you have a particular tuple  $t$  here.

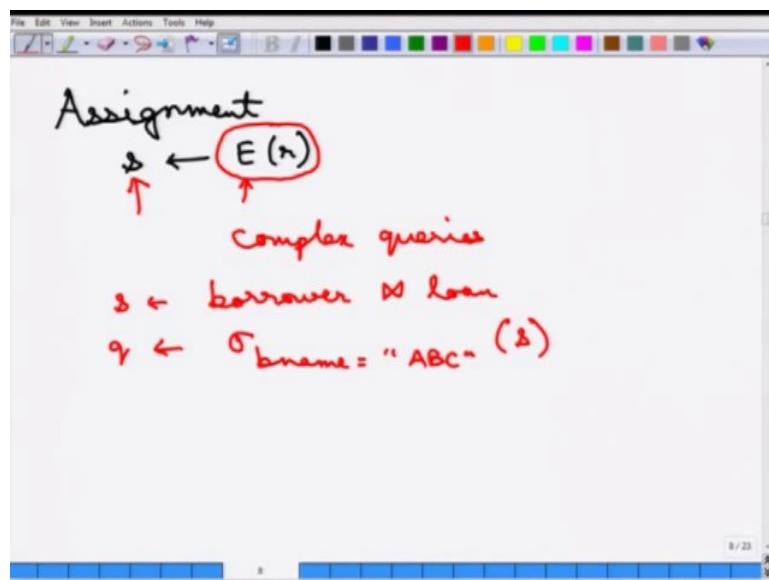
So, this is particular tuple we will fill it up later, but this  $t$  when joined with these two. So, that is taken the Cartesian product both of them must be in the original relation  $r$ , so both of

them must be in this relation  $r$ . So, let us try and fit some bills. First of all, let us see if does  $(1, 5)$  is fit the bill, if  $(1, 5)$  fits the bill then  $(1, 5, 2, 7)$  must be part of that which is true, then  $(1, 5, 3, 7)$  will be part of that. So,  $(1, 5)$  is an answer set, so  $(1, 5)$  is correct.

Now, let us take  $(1, 6)$ . Now  $(1, 6)$  again it must be joined with both of these, so  $(1, 6, 2, 7)$  is not part of the answer set, so this is gone, this is not part of this thing. So, then let us take  $(2, 6)$ . Is  $(2, 6)$  part of it? well  $(2, 6, 2, 7)$ ,  $(2, 6, 3, 7)$  are there. So, this answer correct answer. Then is  $(3, 6)$  there?  $(3, 6, 2, 7)$  and  $(3, 6, 3, 7)$  are both there which is correct and  $(3, 5)$  no, because  $(3, 5, 2, 7)$  is not there. So, the answer essentially is then simply let me write it down, the answer is  $(1, 5), (2, 6), (3, 6)$ , so this is the answer with A and B of course.

So, now, the question is how are this kind of division such a complicated operator useful. So, we will see some examples where the division operator will be used, but just a very quick basic idea to start making you thinking is that if this part of the division then it must happen. So, essentially you take the relation  $r$  and you divide by  $s$ , so you divide such that... So, if whatever 2 and 7 is here, so you select only those tuples where  $(2, 7)$  and  $(3, 7)$  are both present, so you get out that part and you just select. So, this is what the idea is, but we will see an actual example later.

(Refer Slide Time: 14:11)



So, we will move on to the last general operator which is the **assignment**. This is denoted simply by this  $s \leftarrow E(r)$ . This is almost similar to the renaming operator, but what it does is that, so, this is an expression, the expression ( $E$ ) has been applied on  $r$  and you temporarily

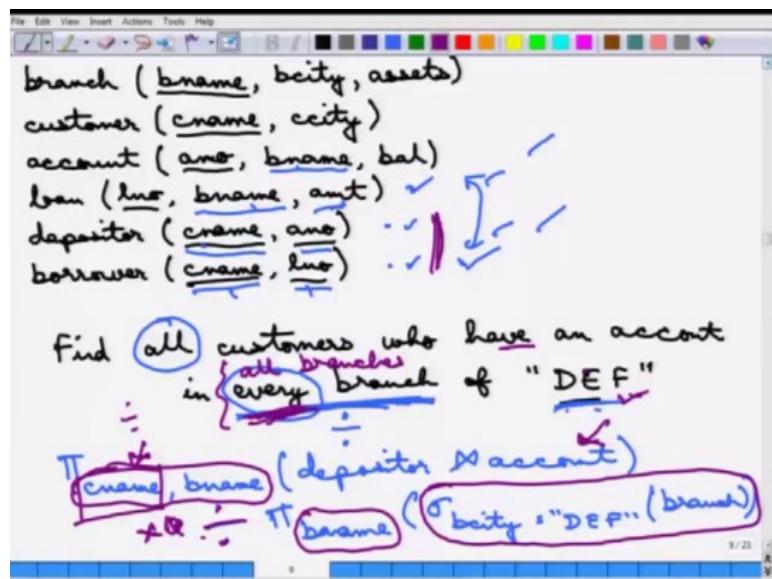
assign it to a variable  $s$ . So, this is very useful in complex queries, where instead of just writing things on top of each other, you assign to  $s$  and then you write it down.

So, for example, in the previous query what some queries back, so you can write  $s$  to be that whatever you say. So, that you can say this is  $borrower \bowtie loan$  and then you can say the answer to my query is  $\sigma_{bname=ABC}(s)$ . So, you can see that this does not add any power to this, but it essentially simplifies writing the query.

So, that finishes the basic definitions of what the additional operators are. We will go over some examples next.

Now, let us go over some examples that uses the additional operators that we learnt.

(Refer Slide Time: 15:40)



So, coming back here this is the same banking example that we have been following. So let us start solving some queries. So here the operative word is both. So, which means that this is a set intersection query and the way to solve this is essentially well we are just doing this let us say borrower intersection with depositor. Now, this is interesting, because although we say find all customers essentially we meant to say find names of all customers. So, that is the, because customer name is the identifying information for customer. So, we can just do this.

Now more importantly the borrower, so every customer which has a loan must be in this borrower table and every customer who have an account must be in depositor table. So, we

can just use these two relations and solve this query. More importantly this projection of customer name must be done before the intersection is done. Because, borrower cannot be directly intersected with the depositor, the reason is they do not have the same schema. So, we must first ensure that the schema is the same and then the intersection can be taken. So, if this is done.

Let us move on to the some other queries. So this kind of queries wherever there is a find all customers, who have an account in every branch. So, the operative word is every branch, wherever there is a every branch kind of a query, this points out to the division operator, because we need to find out, So, the every branch that is what the division queries try to solve. So, essentially the way to solve this is that we need to first find out this customer name and branch name from the depositor natural join account. So, what does the depositor natural join account do? This depositor natural joint account gives the information. So, depositor this with account gives the information about all the customers and everything. Now, we are selecting only the customer name and the branch name, now this branch name must be in the every.. So, the branch name city is DEF, so that is what we need to do. So, this when we divide it with the branch name of everything where the branch city is DEF from of course, the branch table.

So let me do it, a little bit one at a time. So, what does this mean? So, this branch city is equal to this, from the branch this, selects all the branches in every, so essentially it is every branch in the city DEF. Now, we are selecting only the branch name out from this, that is what we need and here it selects a customer name and the branch name from every possible depositor. And then, if we do a division, then it essentially selects out this customer names, such that this has an account in every branch. Because, this division operator makes you sure that this, so this customer name will be part of it only when this entire join, entire Cartesian product is present here. So, this is how to solve a query where there is this every branch, so whenever there is a every branch or you can say all branches of ... So, whenever this kind of query then the natural way to think of it is using the division operator. We will have many more examples etcetera later for your practice, but this is what the way to think.

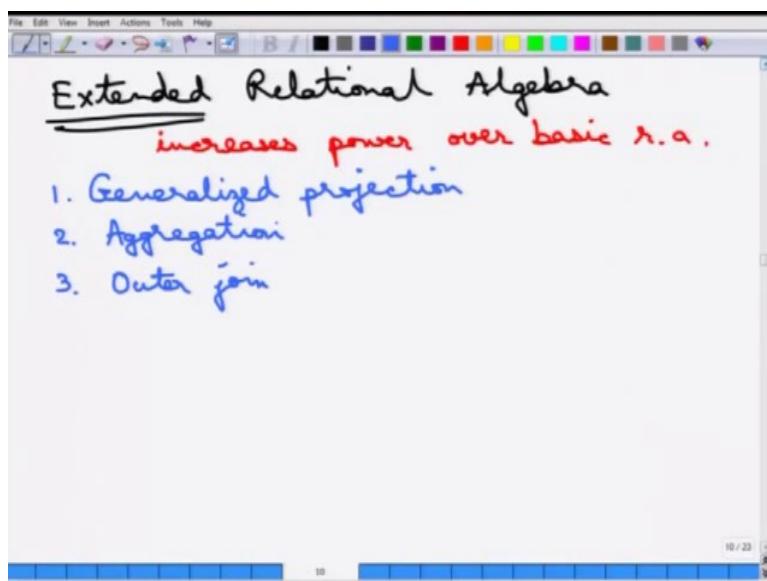
So, this finishes part of this normal relational algebra, where we have seen six basic operators plus four additional operators. Just to remind you that the additional operators do not increase the power, but it makes the solving some of the queries easier. Next we will go over an extended relational algebra, so we will define some more operators, where it does actually

increase the power of relational algebra.

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

**Lecture - 06**  
**Relational Algebra: Extended Relational Algebra**

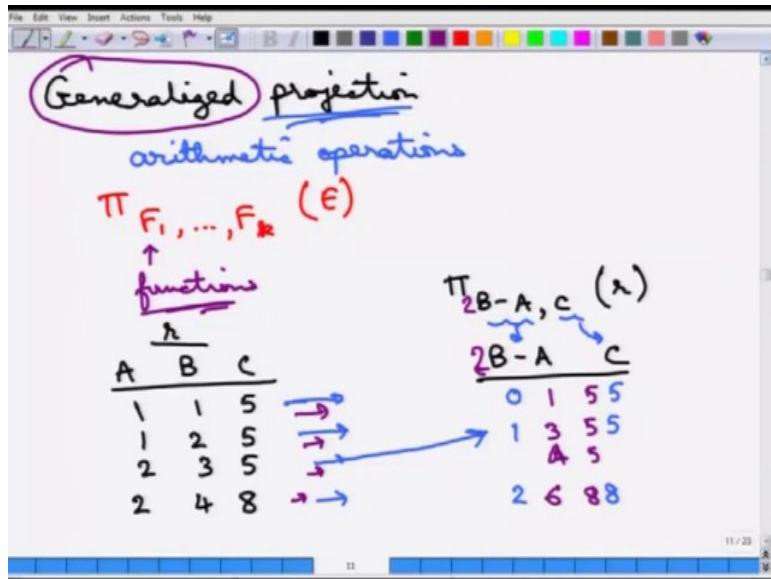
(Refer Slide Time: 00:14)



Let us move on to something called an **Extended Relational Algebra**. So, this is extended, so it does increase the power. So it increases power over basic relational algebra. So, the operators here are there are three operators, first is called a *generalized projection*, the second one is *aggregation* and the third one is called *outer join*.

So, at least from the names it can be guessed that this is a generalized projection; that means, it has got something to do with the projection and tries to make it more generalized, so it tries to extend its power. Similarly, outer join is some form of join, but with something more and we will see exactly, what all of those things. Of course, aggregation is something new. So, let us go over each of this in a little bit more detail.

(Refer Slide Time: 01:32)

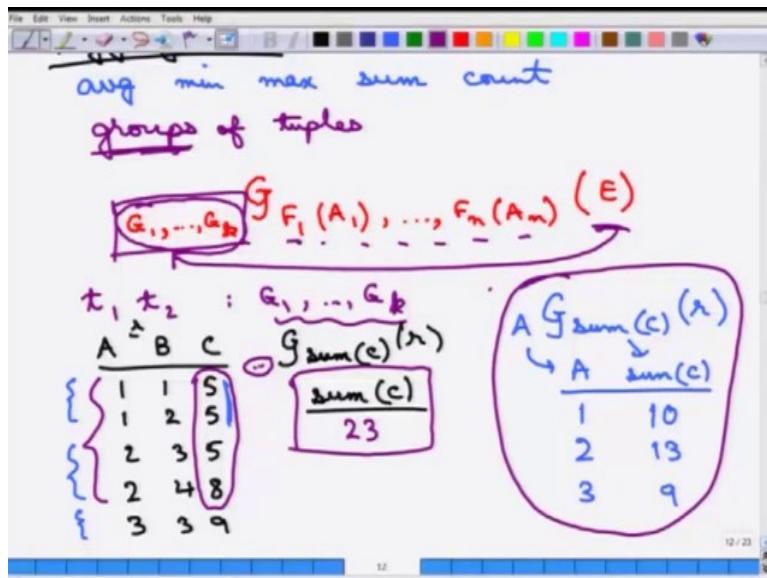


So, the first one is **generalized projection**, so what it does is that it takes the projection operator, the normal projection operator. So, the normal projection operator allows only it names certain attributes and just selects out those attributes, it just projects out only those columns. So, what the generalized projection operator does is that it allows arithmetic operations on those projections. So, arithmetic operations on the projected columns and what do I mean by that, so it essentially says that it is defined like this,  $\Pi_{F_1, \dots, F_k}(E)$  this is a function  $F_1$ . So, you can define some  $k$  functions over the expression, so these are instead of just names these are functions, so these are generalized arithmetic operations or generalized arithmetic functions. So, for example if we take this example suppose there is  $r$ , which is defined as  $(A, B, C)$ . Now suppose, what I do is, I do something like this, so  $\Pi_{B-A, C}(r)$ . First of all, the schema for this becomes  $B - A$  and  $C$ . Why is that? Because, this schema these are copied, essentially this is the first thing and this is the second attribute and  $B - A$ .

So,  $B$  minus  $A$ , so it is done from here  $B - A$  is  $(0, 5)$  fine, then this is  $2$  minus  $1$  this is  $(1, 5)$ , then this is again  $(1, 5)$ , so this goes here, so there is nothing new is output and this outputs  $(2, 8)$ . So, these are the answers, then now, well you can change a little bit on this, so instead of this thing you could have said  $2B - A$ , then this could have become  $2B - A$  by  $C$ . So, from this, this would have produced  $(1, 5)$ , the second one would have produced  $(3, 5)$ , the third one would have produced  $(4, 5)$  and the fourth one would have produced  $(6, 8)$ .

So, you can see that any operation can be done, now instead of  $2B - A$ , you can do lots of other, other things etcetera. But, this is how the generalized projections works; This is what the generalized part is, because it works on any function. So, generalized projection that is how this works.

(Refer Slide Time: 04:08)



The next one is **aggregation**, so it let us one aggregate, so it can use certain aggregation function, such as *average*, *min*, *max*, *sum*, *count*, so that is what it does. So, it can be applied on tuples or more importantly, it can be applied on groups of tuple. So, I mean this can be applied on the entire relation or on certain groups of tuples and these groups can be defined, The grouping is done. So, the generalized form of this aggregation operation can be written in this manner.

So, there is this generalized  $G$  function and let me try to write it there and you are applying some function  $F_1$  on  $A_1$  and so on, so forth. So, you can apply some  $n$  functions on this and a grouping is defined using some grouping operators, grouping values, which is  $G_1, \dots, G_k$  of course, this is on some expression here. So, what it means is that, this expression is first taken and then, the grouping is done using this  $G_1$  to  $G_k$ .

Then, for each group where each group according to this  $G_1$  to  $G_k$ , each of these functions are applied, then  $F_1(A_1)$ ,  $F_1$  on the attribute  $A_1$  which is part of the  $G_1$  to  $G_k$ ,  $F_2(A_2)$ ,

$F_n(A_n)$ ; that is when applied.

Now, how is this grouping done? This grouping is done, so tuple  $t_1$  and  $t_2$  are in the same group if they agree on all  $G_1$  to  $G_k$ ; They must agree, so for all of them. So, essentially this everything has to be same for  $t_1$  to  $t_2$ .

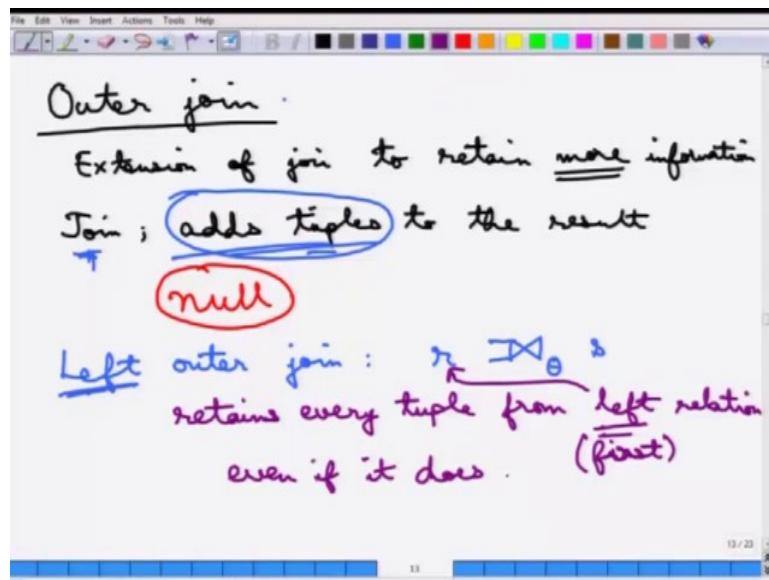
Now, what does that mean? Let us again take an example that is the best way of understanding it. So, let us say this is  $(A, B, C)$ , there is a relation  $r(A, B, C)$ , so now, you are doing this operation, let us say  $G_{sum}(C)$ ; that is all that is being done. First of all the schema that is produced is simply  $sum(C)$ ; that is what is being done and what is being done is that just the sum over. So, let me explain what is it, there is a grouping here defined; that means, that every tuple is in the same group, so everything is in the same group and essentially just the column  $C$  is summed up, so the answer to this is 23, that is it, this is the answer to this.

Instead if you try to define a little more complicated query for us, suppose you say  $G$  is being done  $sum(C)$ ; that is fine, but with an  $A$ . So, by the way I should say that this is of course, some  $r$  and this is also of course on  $r$ , so what is being done is that this is the following,  $A G_{sum}(C)(r)$

So, first of all the schema of this is  $A$  and then,  $sum(C)$ . Why is that? Because, this grouping is done on  $A$ , so all the tuples must agree on the  $A$  value and that  $A$  value is written here and then, the corresponding sum is done. So, how do we do that? So, first of all, all the tuples that agree on the  $A$  value; so that means, these two are in one group and these two are in the second group.

Now, for these two the  $A$  value is 1 and the sum is just this, which is 10 for these two this is two and the sum is 13. Now, suppose there were another tuple, which is what would have happened is this is in a group by itself, so these would have been simply  $(3, 9)$ . So, this is the answer to this query of the aggregate operation with this grouping; that is, what that it has being done.

(Refer Slide Time: 08:24)



Let us see the final operator in this space is that of **outer join**. So, in outer join, what happens is that it is an extension of join to retain, so it is an extension of the normal join that you see, join to retain more information. So, how do you retain more information is that, it first does the join and then adds tuples to the result. Now this seems to be very interesting, that how can it add tuple to the result, which is not defined as part of the join.

So, for the first it computes the join; that is given as part of the condition about that join, it adds all those tuples that are part of the join they need to add certain more tuples. Now, what more tuples? This requires the use of something called a **null** value and we define **null** earlier in some connection that **null** is a part of every domain etcetera. So, null can be considered to be the value for any attribute of any tuple, this is the specialty of **null**.

Now, let us, define one at a time what is it. So suppose we first define, what is called a *left outer join*, so this is called a left outer join. So, this is denoted by  $r$ , this is an interesting operator, this is the join operator with theta, because this is left outer, there are two strokes on the left, so this is  $r \bowtie_{\theta} s$ . So, what it does is that, it retains every tuple from left relation. Now which is the left relation? Of course, this is the left relation, because this is on the left of this, left or; you can say the first relation it retains everything from the first relation.

(Refer Slide Time: 10:52)

even if it does not (first) obey join condition  $\theta$

Right outer join : right (second)

r		s		t		
A	B	A	C	A	B	C
1	5	1	7	1	5	7
2	6	2	8	2	6	8
3	7	x	x	3	null	null

Even if it does not obey join condition, even if it does not obey join condition  $\theta$ . Still probably not clear us to, what is happening, but what is happening is essentially this. So, let us probably take an example; that is the easier way to understand. Now if we do a natural join of  $r \bowtie s$  simply natural join. Now, what happens is if you do a left join of  $s \bowtie r$ , what it does is that, it first completes the join which is  $(1, 5, 7), (2, 6, 8)$ .

And then, it retains every tuple from the left relation, even if it does not obey the join condition. Now which is the tuple that it does not .. So this tuple has already been captured as part of the join condition this tuple has already been captured as part of the join condition, but 3 and 7 is not captured. So, it must retain  $(3, 7)$ . Of course, this is  $(A, B, C)$ .  $(3, 7..)$ . The question, then comes is, what will happen to this value of C? It has got nothing defined, but then the power of *null* comes it essentially says if nothing is defined with this *null*.

So, the answer is  $(1, 5, 7), (2, 6, 8)$  and  $(3, 7, \text{null})$ , so this is a special value that is being used and this is a left join. So, now, that the left outer join has been defined it is probably easier to define the *right outer join*, which is essentially the same thing except the left of the first relation is replaced by the right relation or the second relation and everything else is analogous. So, now, to just complete the example the right outer join for that same example is this way.

So, that first of all the operator is written this way  $\bowtie$ , again and it has once more it has the same schema as  $(A, B, C)$ . Now, once more first of all, the first thing to do is to complete the

join, which is  $(1, 5, 7), (2, 6, 8)$ , then, what is left out. So, this is captured from the right relation this is captured from the right relation this is not captured, so this needs to be copied down. So, 4 and 9 and this is added with null, so that is the *right outer join*.

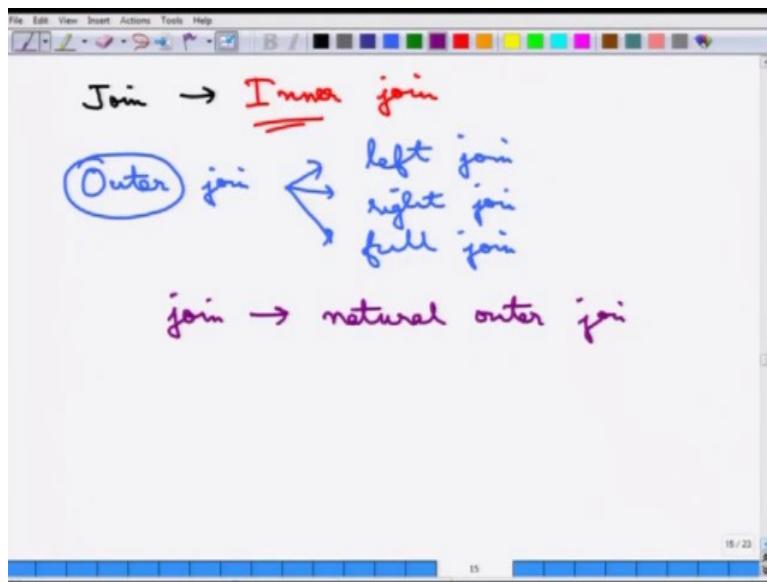
(Refer Slide Time: 13:28)

The slide shows a database diagram with two tables, *r* and *s*, and their full outer join result. Table *r* has columns A and B with rows (1, 5), (2, 6), and (3, 7). Table *s* has columns A and C with rows (1, 7) and (2, 8). The full outer join result has columns A, B, and C, with rows (1, 5, 7), (2, 6, 8), (3, null, null), and (4, null, 9). Handwritten notes explain that a full outer join captures from both left and right relations.

<u><i>r</i></u>		<u><i>s</i></u>		<u><i>r</i> <math>\bowtie</math> <i>s</i></u>		
A	B	A	C	A	B	C
1	5	1	7	1	5	7
2	6	2	8	2	6	8
3	7	4	9	3	7	null
				4	null	9

Now, that right outer join is defined, so there is something called a *full outer join*, which is the combination of both right outer join and left outer join. So, this is the operator for this  $\bowtie$  and it captures from both left and right relations. So, once more going to, that if you go back to that example of  $(1, 5, 2), (6, 3, 7)$ , then the complete outer join  $r \bowtie s$  is defined as  $(A, B, C)$  the first we complete the join, which is  $(1, 5, 7), (2, 6, 8)$  and then, we complete the left and right join. So this is  $(3, 7, \text{null})$  and  $(4, \text{null}, 9)$ , so everything is completed from. So, these were the things that were left out here and left out there, so both find their way in the *full outer join*.

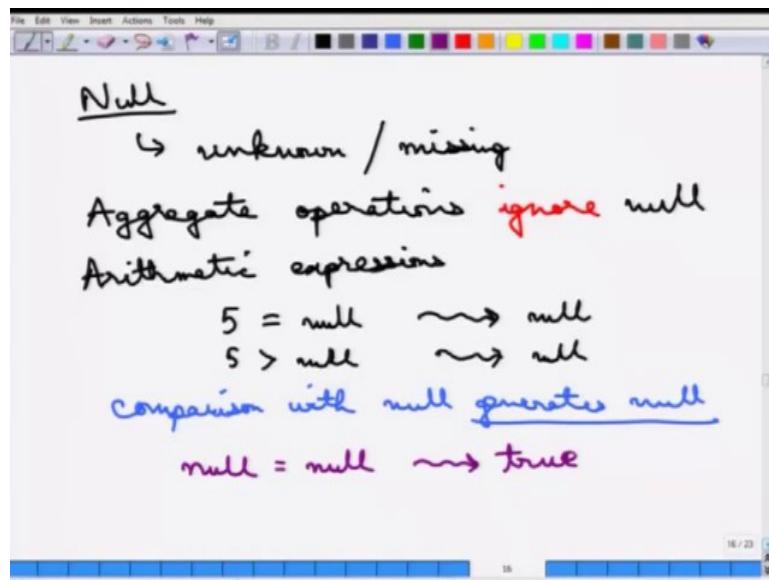
(Refer Slide Time: 14:39)



Now, that these outer joins are defined, so that *normal join* is sometimes called an *inner join*. The word inner comes is now understandable, because there is an outer join. And sometimes the word outer join this outer part outer join the outer part is omitted. So, then what do we essentially get is the *left join*, because there is no left outer join a *right join* and a *full join*. Because, there is no left outer join right outer join and full join.

And, when no theta condition is specified, when it is just the word join is used when simply the word join is used, then it translates, it is generally meant as a *natural outer join*. So, these are just part of the terminology. So, then there is a little bit left about these relational algebra queries, which is the special value about **null**. So, we have already seen, what how **null** are useful etcetera and **null** is a special value.

(Refer Slide Time: 15:51)



So, it essentially denotes an unknown or a missing value. We saw in the examples in the outer join, that when A and B are joined, we do not know the value of C is, so it is a unknown or missing and that is being denoted as *null*. So, now, this null and let us see, what the aggregate operation operators ignore null, so aggregate operations simply ignore null. But, what happens with arithmetic operations arithmetic operations may require comparison such as 5 equal to null, 5 greater than null etcetera. So, what happens is that, so for example, 5 equal to null this evaluates to *null*. Or 5 greater than null etcetera, so all of these things evaluate to null. So, a comparison with a null, generates another null. There is one very interesting exception, which is null equal to null this evaluates to *true*.

(Refer Slide Time: 17:19)

Null → Three-valued logic

0, 1, unknown  
true, false, unknown

<u>OR</u>	u OR t = t	}
	u OR f = u	
	u OR u = u	
<u>AND</u>	u AND t = u	}
	u AND f = f	
	u AND u = u	
<u>NOT</u>	NOT u = u	

So, null equal to null is actually true. So the *three valued logic*. So, using null, null generates to a, what is called a three valued logic. So, we have been all used with binary logic, which is using 0 and 1. There is a three valued logic. The three values are 0, 1 and *unknown*. So, essentially *true, false* and *unknown*. So these are the three values and let me write down the rules for all of this three valued logic.

So, first of all the OR table, so OR is, unknown OR true evaluates to true, because there is a true and it does not matter what the unknown is. unknown OR false evaluates to unknown, because the false does not matter and this is it, unknown; that is it. unknown OR unknown that evaluates to unknown.

AND is similar. So, unknown AND t is unknown. unknown AND false is false. unknown AND unknown is equal to unknown.

The NOT is there. So, NOT of unknown is equal to still unknown.

So, this is the three valued logic table that we require to move forward. The select operation treats *unknown* as *false*. I mean if there is something unknown, then the select operation will not select it; That is what it will be meant.

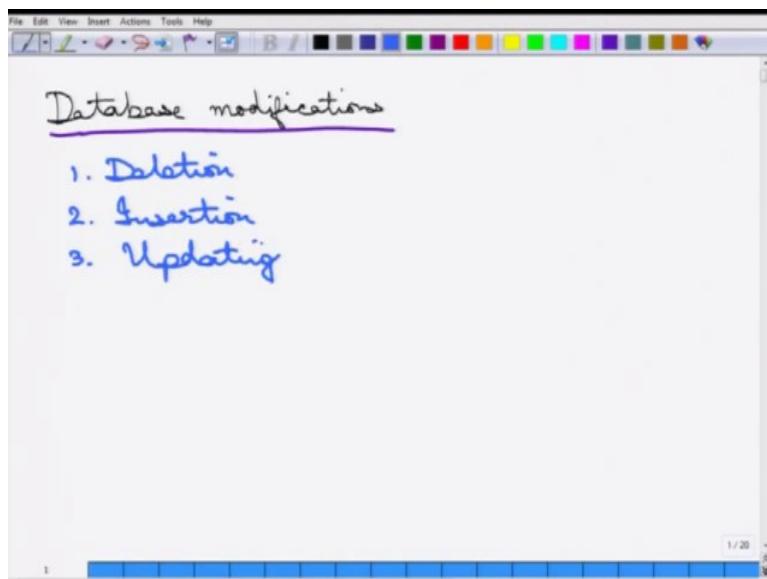
So, that completes the part about the queries of this relational algebra and we will later go over, how to use relational algebra for database modifications.

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

**Lecture - 07**  
**Relational Algebra: Database Modifications**

Our next topic will be Database Modifications on the Relational Algebra.

(Refer Slide Time: 00:15)



Database modifications and there are three types of database modifications that we will be talking about Deletion, Insertion, and Updating.

(Refer Slide Time: 00:36)

So, let us go over them one by one. So the first one is **deletion**. So deletion essentially means one or more tuples deleted from the relation. So, it is simply the relation  $r$  that needs to be deleted, it is assigned in the following manner. So,  $E$  is the expression that says which tuples need to be deleted and the rest is very clear. So, the result of the expression is the set of tuples which are deleted from  $r$ . Now this can be an expression or an actually can it set may be specified and here is one very simple example. Suppose this is your  $(A, B, C)$  with  $r$  with same example that.

So, just one important thing that needs to be remembered is that only tuples can be deleted, the attributes cannot be deleted, only tuples can be deleted from a relation. So, if I now do this  $r$  is equal to an expression which is let us say, something like this  $r \leftarrow r - \sigma_{A=1}(r)$ , then this results in the same schema  $(A, B, C)$ , but then everything where  $A = 1$  is deleted. So, these two are deleted and this results in simply  $(2, 3, 5)$  and  $(2, 4, 8)$ .

And similarly this can be something like this can be also set, you can say delete  $(2, 3, 5)$  which is a set of tuple which is  $(2, 3, 5)$ . So, then this one is deleted, so this will delete only this and the others will be part of this, but in general it is more useful when expression like this can be specified for deletion.

(Refer Slide Time: 02:25)

Insertion

$$r \leftarrow r \cup E$$

$$\underline{r} \quad r \leftarrow r \cup \{(1, 2, 5)\}$$

A	B	C
1	1	5
2	3	5
2	4	8

A	B	C
1	1	5
2	3	5
2	4	8
1	2	5

The next is **insertion** which is similar in the sense that a tuple is added, so this  $r$  gets expression with a union.  $r \leftarrow r \cup E$ . So, again it is a expression and same kind of things can be given. So here is an example, suppose  $r$  is your  $(A, B, C)$  which is  $(1, 1, 5)$ ,  $(2, 3, 5)$ ,  $(2, 4, 8)$  and if  $r$  is unioned with, let us say, specify a tuple  $(1, 2, 5)$  then of course, the same all these three things are copied and a  $(1, 2, 5)$  is added. So, essentially it becomes  $(1, 1, 5)$ ,  $(2, 3, 5)$  and  $(2, 4, 8)$ .

(Refer Slide Time: 03:21)

Updating

$$r \leftarrow \pi_{F_1, \dots, F_n}(r)$$

$F_i:$   $\begin{cases} \text{attribute} \\ \text{expression on attribute} \end{cases}$

$$\underline{r} \quad r \leftarrow \pi_{A, 2*B, C}(r)$$

A	B	C
1	2	5
1	1	5
2	4	8

A	B	C
1	4	5
1	2	5
2	8	8

So, the next one is **updating**. Updating is a little more involved in how to write it. Essentially

the updating is given like an expression.  $r \leftarrow \Pi_{F_1, \dots, F_n}(r)$  So, it is the projection of  $F_1$  up to  $F_n$ , so all the attributes from  $r$  are projected on. So, all the attributes of  $r$  are projected on and each  $F_i$  can be either, the attribute itself that was there or some modification of the attributes, so some expression on the attributes.

So, an example is the same, let us say  $r$  is your  $(A, B, C)$  which is  $(1, 2, 5), (1, 1, 5)$  and  $(2, 4, 8)$ . And let us say this particular modification is being done, which says  $A$ , but then it says  $2 * B$ . So, every  $B$  value is essentially doubled up and this is being done, so this results in  $(A, B, C)$  with the following 1. So, this is doubled up, this 2 is doubled up, this is 5 the others remain the same. So, this is  $(1, 2, 5)$  and this is  $(2, 8, 8)$ , very simply this is what is being done, Then instead of  $r$  what can be done is that a little something more can be done.

(Refer Slide Time: 05:08)

<u><math>F_i</math></u>		
attribute expression on attributes		
$r \leftarrow \pi_{A, 2*B, C}(r)$		
A	B	C
1	2	5 ✓
1	1	5 ✓
2	4	8

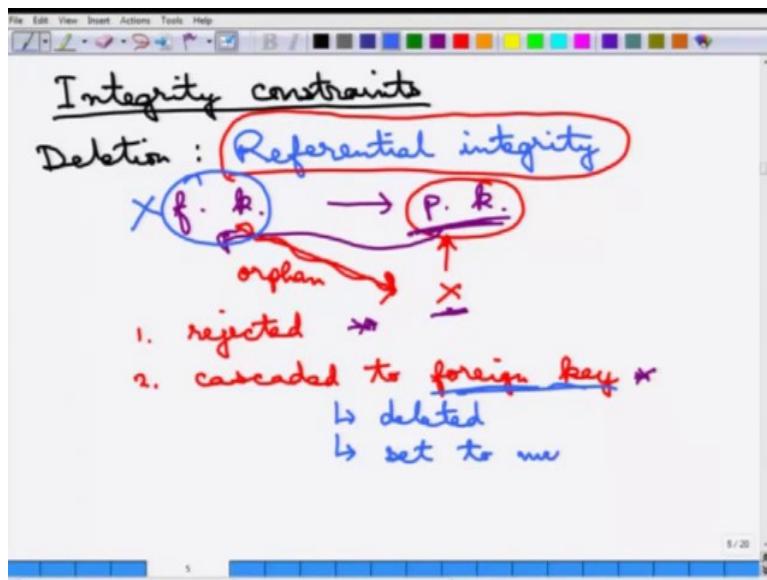
$r \leftarrow \pi_{A, 2*B, C}(\sigma_{A=1}(r))$		
$r \leftarrow \pi_{A, 2*B, C}(\sigma_{A=1}(r))$		
$r \leftarrow \pi_{A, 2*B, C}(\sigma_{A=1}(r))$		
A	B	C
1	4	5
1	2	5
2	8	8

$r \leftarrow \pi_{A, 2*B, C}(\sigma_{A=1}(r))$		
$r \leftarrow \pi_{A, 2*B, C}(\sigma_{A=1}(r))$		
$r \leftarrow \pi_{A, 2*B, C}(\sigma_{A=1}(r))$		
A	B	C
1	4	5
1	2	5

So, for example it can be said that  $r \leftarrow \Pi_{A, 2*B, C}(\sigma_{A=1}(r))$ . So that means, first  $\sigma_{A=1}$  is done. So, only these two attributes are modified and the resulting then  $B$  is doubled up. So, the resulting relation is  $(1, 4, 5)$  and  $(1, 2, 5)$ , so these are the three main database modification techniques, but what is more important for all of this is that all of this can violate some constraints, so these are called **Integrity Constraints**.

(Refer Slide Time: 05:47)

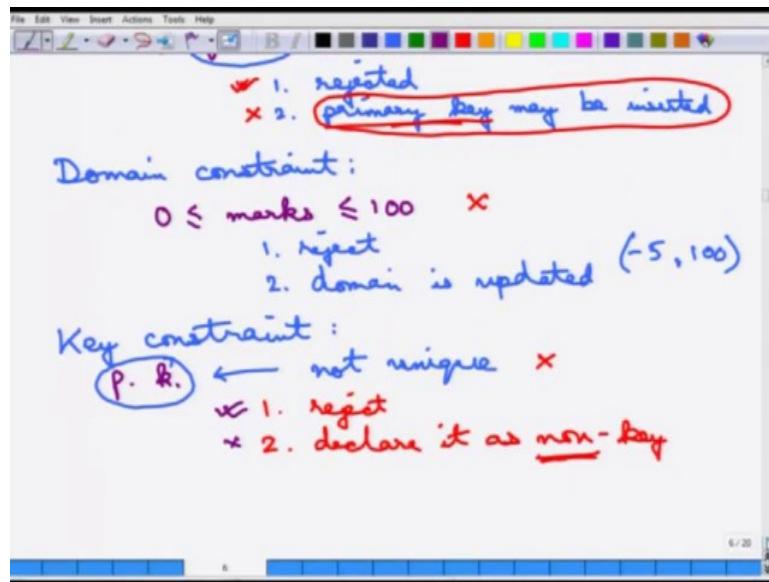


So, all of these three techniques can violate certain integrity constraints. So let us start with deletion. So, what can deletion violate, deletion can violate what is called a referential integrity, this is called a referential integrity. So, what does referential integrity mean? If we recollect what is the foreign key constraint is that every foreign key must be a primary key of some other relation. So, essentially what happens is that if a particular primary key value is deleted from the foreign relation, then a foreign key becomes orphan, because it does not have the corresponding primary key. So, this violates, what is known as, a referential integrity, because this is now referring to something which is not present. So, the referential integrity is violated, so a primary key may not be simply deleted that deletion may not be simply allowed, if there is a foreign key that refers to it. So, what happens if a deletion, or if the database is asked to do such a deletion, there are couples of options first is it is simply disallowed. So, the operation is rejected that is one way of doing it, the other is the deletion is cascaded to the foreign key, there are two ways to handle this. Now, the first one is easy to understand, what do we mean by rejected. So, the deletion does not take place, that simple. Then the second case what happens is that if pk is deleted then the corresponding fk that refers to this primary key also deleted. So, the foreign keys are also deleted from the referencing relations, so these are the two ways of deletion. So, deletion only violates referential integrity and these are the two ways of handling it.

Now of course, as part of this foreign key cascading this can be either deleted totally. So, the entire tuple corresponding to that foreign key is deleted or the foreign key may be set to null,

this is another way of enforcing the deletion.

(Refer Slide Time: 08:08)



So, we next go on what does the **insertion**. What are the constraints that the insertion can violate? An insertion can again violate the referential integrity and this one is probably easier to understand in the context of the previous example. If a foreign key is inserted, then there the corresponding primary key must be present; otherwise, the insertion of this foreign key is not meaningful. So, again there are ways to handle it, so either this can be rejected, so this insertion can be rejected or the corresponding primary key may be inserted.

Now, this primary key may be inserted, but this generally does not make any sense, because if you cannot just insert a primary key, primary key is something very, very important. So, this is not actually not the viable option, so the only viable option is to reject the insertion into a foreign key. So, this is the first kind of integrity that the insertion can violate and insertion can also violate what is called a domain constraint.

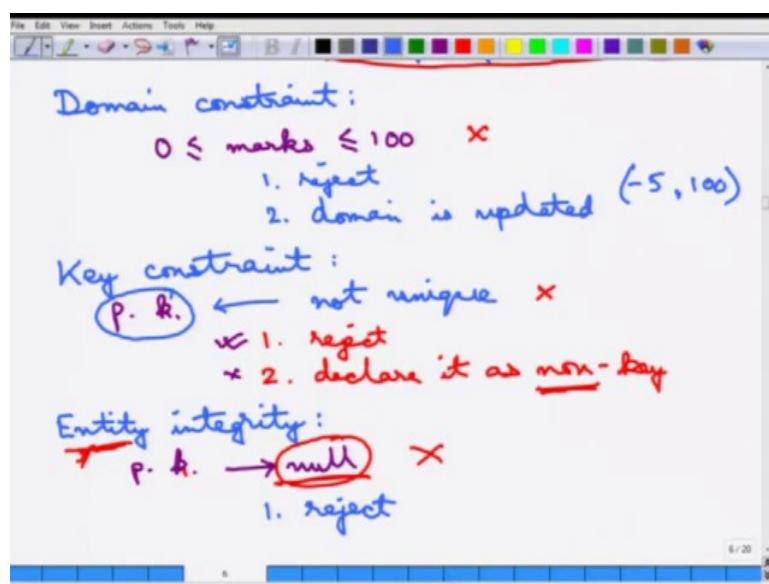
So, domain constraint is probably easier to understand, so there are certain domain values that are specified for example, you can say the marks of a person are between 0 to 100 for a particular exam or whatever. So, if a marks column is being inserted if in tuple is inserted with a marks column outside this range then that is not allowed. So, again there are two ways of handling this, so either you reject the domain is updated.

So, suppose you will later saw that extra marks can be given or marks can be cut for

punishments, etcetera. So, you can the domain can be updated to for example, minus 5 to 100, so there the domain may be updated, so there are two ways of handling this. The next kind of constraint that insertion may violate is called a key constraint. So, suppose there is a primary key or whatever some other kind of key which is there, now if an insertion violates the condition of being a primary key.

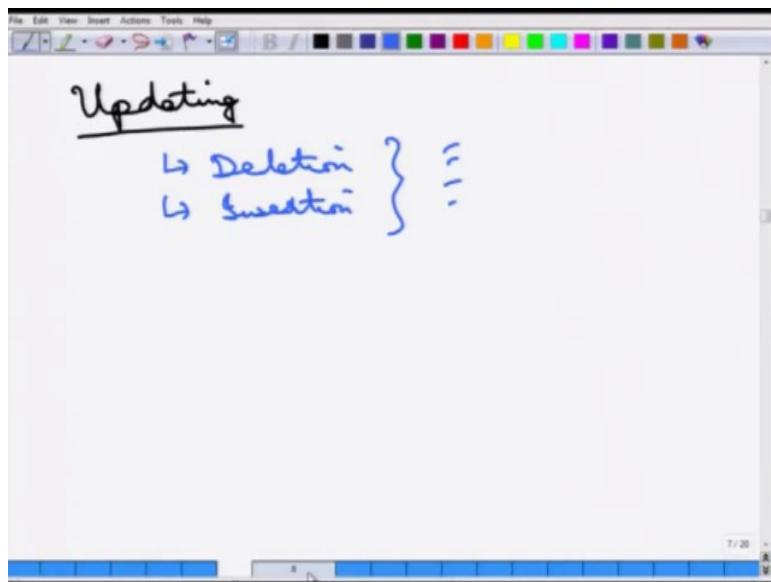
So, how does it violate it becomes not unique, so if another tuple is inserted which has the same primary key or some other tuple then; that means, this is no longer a primary key, no longer a candidate key or super key whatever and this is a problem. So, that must not be allowed, again so this there are two ways of handling this either this is rejected or there is a more extreme option which is declare it as a non key, in which case some other primary key must be done etcetera, etcetera. So, again in general this is not a very viable option and this is the only thing that can be done and that is being done.

(Refer Slide Time: 11:39)



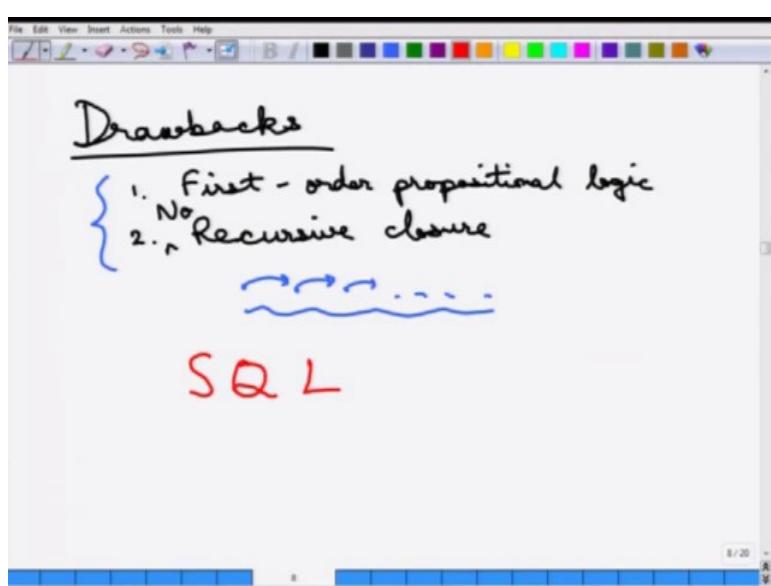
Then there is the fourth type of thing that insertion can violate, which is called an entity integrity. So, entity integrity is that a tuple is inserted, where the primary key is set to null, the primary key is null this again should not happen, because the primary key cannot be null by design. So, because why is this called an entity integrity, what does a primary key mean primary key defines what the entity is about, now if the primary key is null, the entity does not have any meaning. So, again this is not a viable option, so this must be simply rejected.

(Refer Slide Time: 12:27)



So, the insertion violates all these four conditions and if we now go to what the updates updating can violate. Now, essentially updating can be looked upon as a deletion and then an insertion, so any integrity constraints that deletion and insertion violates are also violated by upgrading. So, all those four integrity constraints that we saw are also violated by updating. So, that is what about the relational algebra the all the... So, we looked upon the queries, etcetera and we looked upon the power of relational algebra, the basic relational algebra, the extension of the relational algebra and we looked at all the different, different operations etcetera including the database modifications.

(Refer Slide Time: 13:20)



Now, just to wrap up the relational algebra thing we need to talk about what are the **drawbacks of relational algebra**. So, relational algebra seems to be very powerful, because it can do a lot of queries and insertions and all those things if it can handle, but it has got some drawbacks.

So, the first thing is that this is the *first order propositional logic*, so anything that the first order propositional logic cannot handle, this cannot handle either. Although first order propositional logic is actually very powerful for real life, it may not handle certain other kind of things.

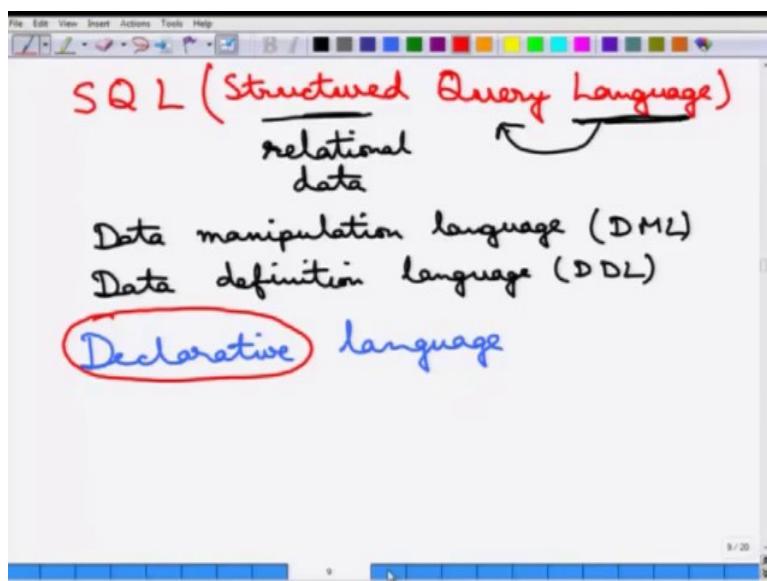
The other very important thing that the database people face almost regularly is that there is no recursion. So, there is *no recursive closure*, so recursive closure that is no recursive closure. So, for example, if you have queries of the form find me supervisors at all levels. So, find me supervisors of this and then supervisor of that person, supervisor of that person up to all the levels it cannot be done, it cannot say at all levels, because there is no recursive closure. So, there is no way to express this in relational algebra, these are the two most important drawbacks of relational algebra, but otherwise relational algebra is very, very powerful. And our next topic which is SQL is based on relational algebra, so this ends the relational algebra module and next we will talk about SQL.

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

**Lecture - 08**  
**SQL: Introduction and Data Definition**

So, let us start on SQL today.

(Refer Slide Time: 00:12)



So, **SQL** stands for **Structured Query Language**. So this is probably the most famous term for databases. Many people by database understands what SQL is. This means, probably a layman understands database as SQL. SQL is almost synonymous to databases and we will go over SQL in some depth. So start up with what is SQL? As you can see it is a language, fine; but it is a language to specify queries in a structured manner.

Now, for this what is a structured manner? This is a relational data model, so this has to be a relational data. So SQL is a language to specify queries in a relational database. So, that is the, what is about SQL. It is also what is called a data manipulation language (DML), because this lets one manipulate it or update values in a database, which is called a data manipulation language, as well as this is also a data definition language. (DDL)

So, it allows to define, how the data is structured in a relational database. So, when this is

both. SQL is based upon relational algebra as we already have said and SQL has many, many, many, many versions currently. So, when it was first invented, after that there have been many, many versions; Many of them are non standard. So, different vendors which provide SQL and so MySQL, there are many other options and PostgreSQL, all those things they are... So, there are differences between them, there are some non standard operators that are provided by some of those vendors or systems, not by the others. What we will try to do is we will stick to the basic versions of the SQL as far as possible and one very, very important thing about SQL which is a big departure from languages such as C or Java is that, this is a *declarative language*.

Now, what does a *declarative language* mean? A declarative language means, this is not a procedural language, this is a declarative language. So, a declarative language essentially means is that, it lets you declare what you want to do. So, for example, in even a relational algebra, we just said select every tuple with the value of the attribute column A is 1, but it does not tell you how to select those. So, should you go over the tuples one by one, should you look at their attribute values or should you hash those attribute values, how it is being done that is not said.

Similarly, in SQL it is not specified how to do it, it is only specified what to do it. So, you just declare the intent, which is declared what the query is supposed to be doing, what it is supposed to be returning and not how it is being done. C, for example, you must specify each and every thing. So, suppose there is a table which is a relation and you are trying to manipulate it in C, so you must say, you go to the first element, go to the first attribute of it, check it with one, if yes you output it, if not you go to the second and so on and so forth.

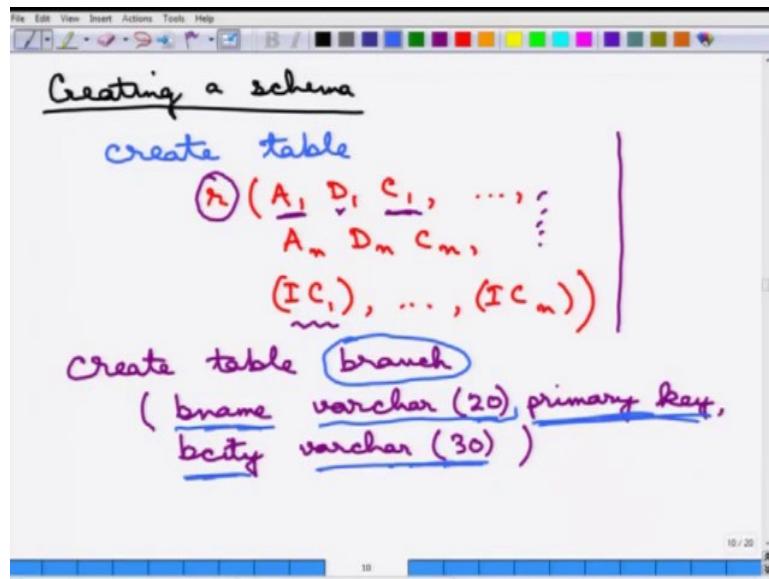
So, it must be specified in very minute detail, in SQL it is not done. So, a question comes is, then how it is being actually performed, so the database engine is very powerful actually. So, when you write an SQL query, it first parses it; then it figures out it has got its internal algorithms and it tries to select an algorithm which will be the best for that particular query. So, then it applies that algorithm, finds the answer and returns it.

So, now, what is the, so this is a very nice feature of SQL, this is an advantage of SQL. Now this has got its side as well. The disadvantage is that, even if you know that a particular algorithm is not good or if you want to apply a particular algorithm for a query, the database engine does not let you do so, it will take over. So, you cannot say I want the answer of this

query in this manner; you are completely at the mercy of the database engine.

Having said that, in most cases the database engine is quite smart and there are lots of research that has gone into it and going into it every day and they are, they do a very, very good job in general. Let us start off with some examples of SQL.

(Refer Slide Time: 05:00)



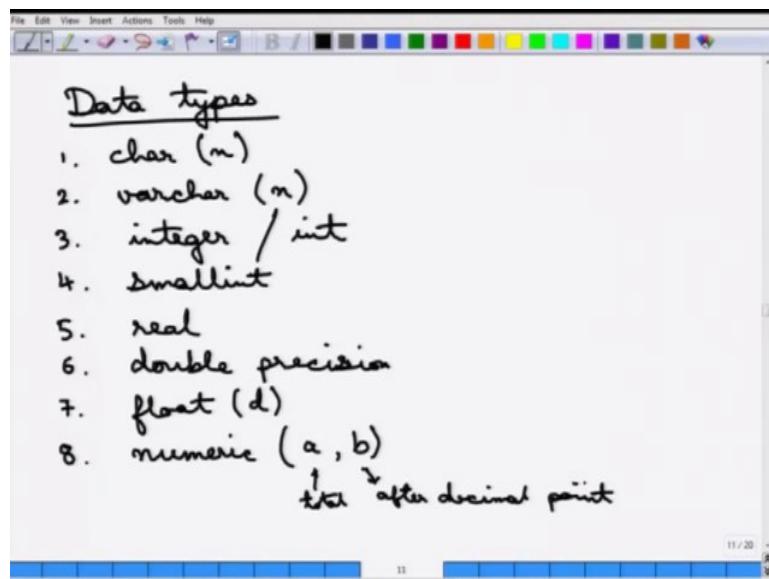
And, so the first thing is how to create a relational schema. So, how do you create, creating a schema? So, the way to create is that, there is a CREATE TABLE construct. So, the create table you say *create table*, then the specify the parameters in this manner A<sub>1</sub> , D<sub>1</sub> , C<sub>1</sub> then some A<sub>n</sub> , D<sub>n</sub> , C<sub>n</sub> , then some constraints. So, let me go over this, what does it mean. So, this construct says that create a table with the name *r*; the first attribute of which is called A<sub>1</sub> whose domain is D<sub>1</sub> and the constraint, if there is any C<sub>1</sub> on that particular attribute and it let us you create n such.

So, you can create whatever number of attributes that you want as part of this *r*, in the end you can also specify certain integrity constraints, some other kinds of constraints if there are any on each of these attributes. So, you can say there is an integrity constraint 1 on this, there is an integrity constraint 2 on this etcetera, etcetera, etcetera. So, this is the way to create a table.

For example, if we just take an example from the earlier banking example that we were doing, what does it mean, it means create a table branch, the first attribute of which is called a

branch name, this is of character up to is can go up to 20 length. And this is the primary key of this table, then you create another attribute which is called branch city which is again a character which can go, I mean which is a string whose length can go up to 30 and that is it, that is about create table. So, now, let us discuss a little bit about the data types that you have.

(Refer Slide Time: 07:00)



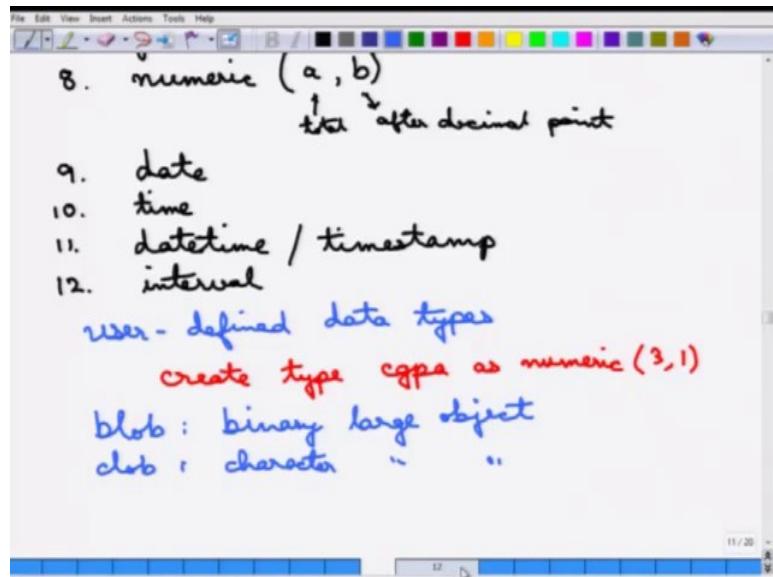
So, the data types in SQL, so the data type is we already saw one example. So, the first is a character n, this is a string whose length is exactly equal to n. As opposed to our varchar, so this is actually you can read it as variable character n. So, this is a string whose length can go up to n, not necessarily exactly equal to n, but it can go up to n. Then, you can say it is an integer or sometimes you can also specify it as an int, so simple this is an integer, then you can specify a small integer.

So, this is a small integer, just like this is a short in C, it says an integer, but it takes lesser amount of space to store. You can say it is a real, so it is a real number, you can say it is a double precision real, so you can say it is a double precision. So, essentially it means it is a real number, but with a higher precision; you can say it is a float with some precision d. So, the float will have at least d digits and finally, you can say it is a numeric.

So, this is a floating point number with a total of a digits of which b is after the decimal point, so this is the total and this is after decimal point. So, these are this and it actually little painful to go over all the syntax and all the data types of SQL. A good standard text book or the SQL manual will be probably a better way to look at this. So, these are the basic data types, but it

does not end here, there are many other data types in the newer versions of SQL, which makes it much more powerful.

(Refer Slide Time: 08:58)

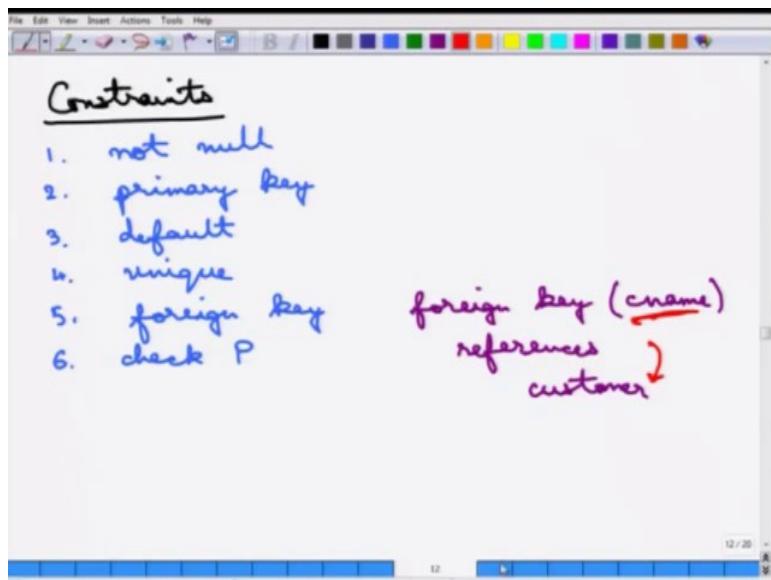


So, the first one is date, so it specifies the date; now this is important because a date can be in the month, year, day format, etcetera, it lets you compare the two dates, take the difference and all of these. Similarly there is a time, then there is a date time which is a combination of these two, so which is actually sometimes called time stamp, so on and so forth. Then, there is you can also specify the time in terms of interval, so this is the time interval.

So, all these are different things that enhances the power of the data types, in addition SQL also lets one create user defined data types. So, one can create their own data types. For example, you can say create type; let us say cgpa as numeric, so it is just like a type definition. So, you are saying well, so every cgpa must be of three digits, at most three digits, so it is one digit after that. This is just a type name for this create, but more importantly it also for storing large objects, it also let us create something called a blob.

So, this stands for **binary large object**. So, for example, if you want to store an image etcetera in an SQL table, you can store it as a blob and similarly there is something called a clob, which is character large object. So, it store a document etcetera, it can be stored as a character. So, essentially these are not stored inside the table, they are pointed to these objects that are stored, so that is about the data types. Now, let us go over a little bit about the constraints that can be done in SQL.

(Refer Slide Time: 11:01)

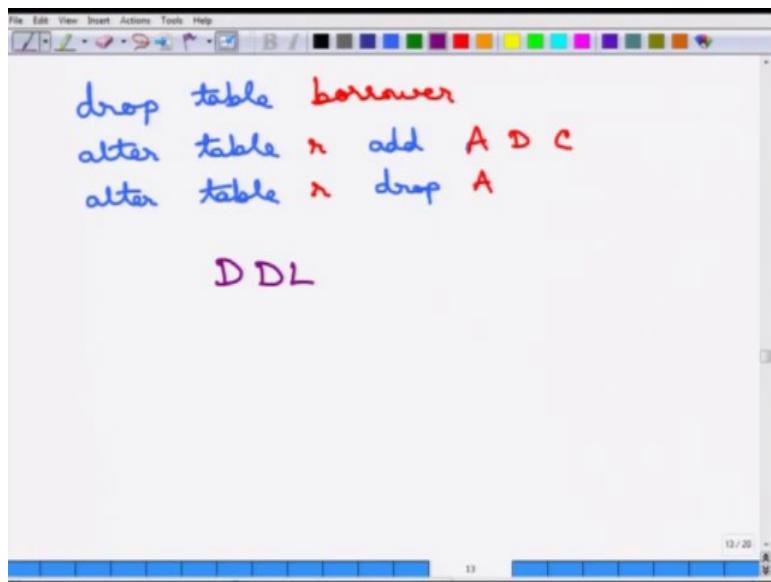


So, you can specify constraints saying something can be set as not null, so a particular attribute can be set as not null. So, whenever a new tuple is inserted, the attribute cannot be null. Similarly, we have already seen an example of primary key, we can specify what the primary key is, we can specify the default value. So, if nothing is specified, the default value takes over, we can specify whether it is unique or not.

So, when you say primary key of course, it is unique, but not the other way around, we can say you must make it unique. The fifth is probably very important, the fifth is important, it is called a foreign key. So, you can specify that this attribute is a foreign key to some other attribute and then, you can also say check p. So, this is like a domain constraint, so you check if the predicate p is satisfied or not.

So, something what we can do is, an example of may be the foreign key constraint, because this is important, is that you can say foreign key for a particular thing, such as let us say cname in some table, you can say this references the customer table. So, if such a definition is inside a table; that means, the attribute cname in this table references the primary key. It references the customer table which means of course, it references the primary key of the customer table, so this can be specified. Question comes is table can be created, can it be modified or can it be deleted of course, it can be deleted and the deletion is simply drop table.

(Refer Slide Time: 12:47)



So, you can simply say **drop table** whatever borrower say, let us say you can drop a particular table. You can also update a table. The syntax is **alter table** and essentially what you can do is, you can say alter table r; then you can say for example, add. So, I want to add the attribute A with domain constraint D and constraint C to this table that can be done. We can also say alter table r, drop a particular attribute A from this and again lots of other things can be done etcetera, etcetera, but it is the way to check up the manual.

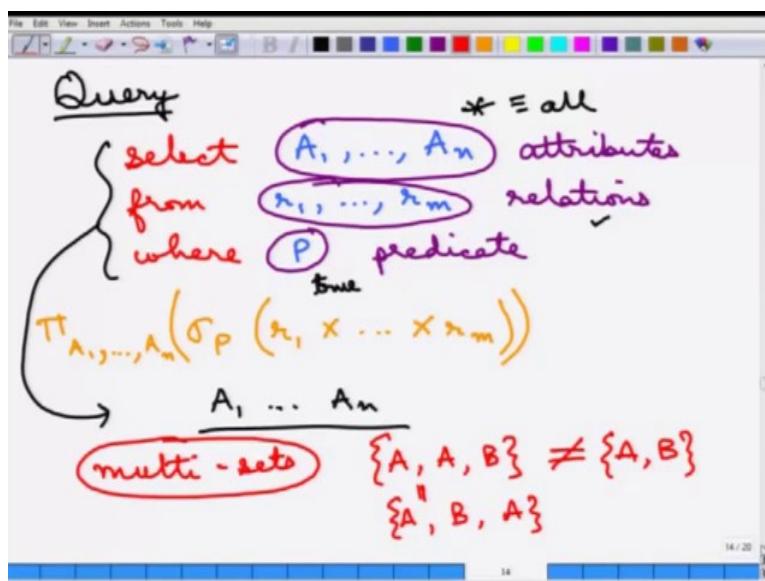
So, this is about the part of the data definition language that we start over; how the table structure is defined. It is essentially just defining the table structure and not doing, not defining about any data etcetera. So, we will see about the insertion, deletion etcetera into the data later, but let us first go to why SQL is famous for, SQL is famous for this basic query structure.

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

**Lecture - 09**  
**SQL: Basic Queries**

The querying in SQL is what it is useful for.

(Refer Slide Time: 00:11)



And the basic query and the typical basic query structure is of this very, very famous SELECT - FROM - WHERE. This is a very, very famous concept of SELECT - FROM - WHERE. And what can you select, you can select a set of attributes and you can select from a set of tables and you can specify a set of, I mean you can specify a predicate. Now, let us go over this one by one, these are attributes, these are relations and this is a predicate.

So, what does this mean, this means that we want to select from the Cartesian product of  $r_1, r_2$  to  $r_m$  which is we first do this  $r_1$  to  $r_m$  and only select those tuples, where this  $P$  is valid and we don't want to output all the attributes of them, we only want to output the attributes  $A_1$  to  $A_n$ , so that is the essential meaning of it. So, if we write down the corresponding relational algebra query, so this is the following thing, the first one that is done is that  $r_1$ .

So, all the cross product is taken, then we apply the condition  $P$  on that and then we select out

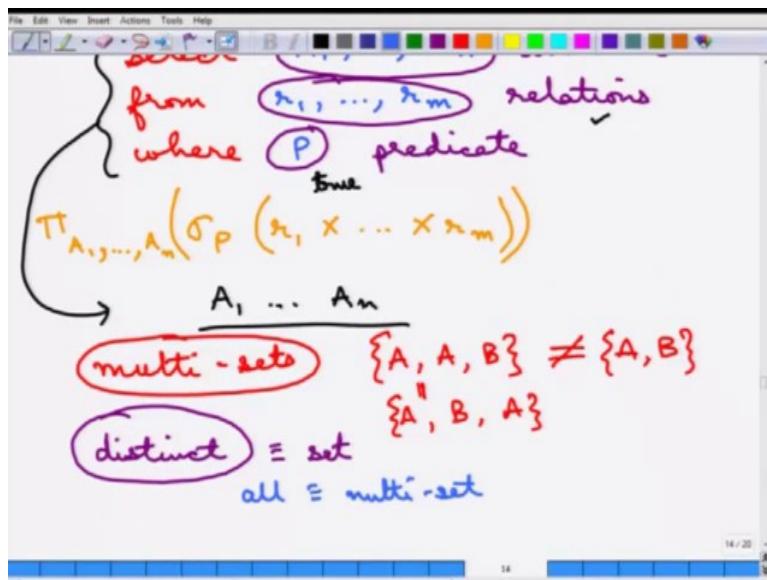
only these particular attributes. So that is the equivalent relational algebra query.  $\Pi_{A_1, \dots, A_n}(\sigma_P(r_1 \times \dots \times r_m))$  So, now, a couple of things is that, so the result of schema of all of these is essentially  $A_1 \dots A_n$ . So, that is the schema it produces, so it produces a new table, where the names of those attributes are  $A_1$  to  $A_n$  number one. Number 2 if P is left out, P is by default true.

So, if P is... So, you can if you leave out WHERE then P is true, essentially everything is selected. And you can of course, SELECT only one relation and then there is a special syntax for SELECT the attribute, you can say SELECT \*, \* means SELECT all the attributes. So, this stands for all the attributes and P is 1 means it is true. Now, we have been saying that SQL is based on relational algebra etc etc, but what is the big difference of SQL from relational algebra is that SQL is a multi-set, SQL's relations are multi-sets. This is a very important thing these are multi-sets.

So, what is a multi-set, it is just like a set, but an element can repeat, so it is a bags of tuples. So, a particular tuple may come may show up more than one time in an SQL query, which is fine. In relational algebra, it is a set, it is strictly a set. So, it does not let you select a tuple more than once, but in SQL by default it is a multi-set, so it can select multiple tuples. So, a multi-set example, just to highlight the point again, so  $\{A, A, B\}$  is a multi set, because although the element A is the same thing it is repeated and this is not equivalent to  $\{A, B\}$  this is not the same as  $\{A, B\}$ .

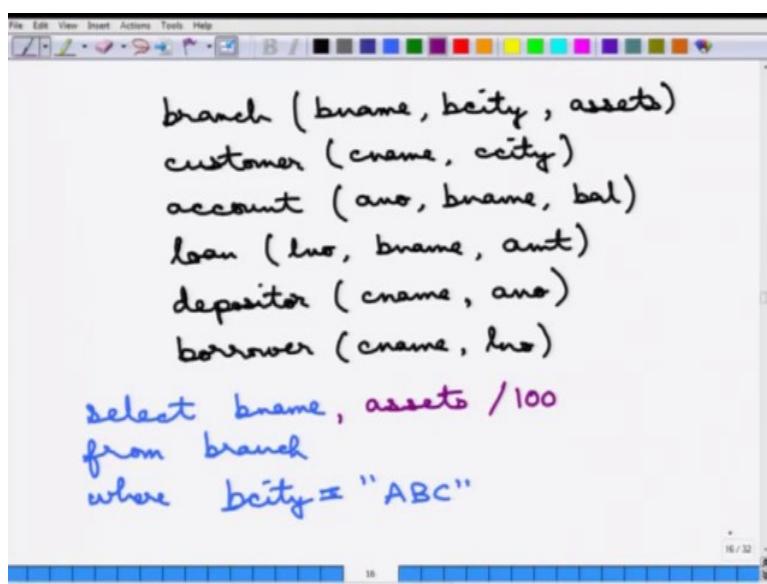
But of course, this is equivalent to  $\{A, B, A\}$  because the set order does not matter, so that is a multi set. So, if however, the SQL can behave like a relational algebra by specifying this keyword DISTINCT.

(Refer Slide Time: 03:54)



So, if one says select distinct, this distinct essentially make it a set and not a multi-set and the opposite of this distinct is all which makes it multi-set. So, all is by default. Fine. So we will go over each of this constructs SELECT - FROM - WHERE in some detail next.

(Refer Slide Time: 04:23)



So, we will use the same banking example as we have been using earlier. So, a branch has a branch name, branch city and an additional attribute called assets, which is the total amount of all the assets in the branch. The customer has a name and a city, account has an account number, which branch it is in and the corresponding balance and the loan has a loan number,

branch name and amount, the depositor is the customer name and account number and the borrower has a customer name and loan number. Alright.

So, let us go over some examples, so first of all suppose we want to select all branches, where ... We want to select all branches from the city ABC. So, how do we do that? So, very simply we want to select all branches ... so we want to *SELECT bname FROM branch WHERE bcity = "ABC"*. So, this selects all the branches from the city ABC. Just to correct it, so this is branch city and this is an equality operator, which says that within a string which is ABC.

Now, suppose we not do not just want the branch name we also want the assets, but we do not want the assets in terms of Paisa or whatever suppose, and we just want it in terms of some other arithmetical operation. So, we can simply say that, we can say simply say assets by 100 or whatever it does not matter, any arithmetic operation can be done on the select thing as well. So, this is one way of doing the select, but ... and the from is essentially from branch etc etc. And let us now go over to some, one or more examples on how to select something from two tables.

(Refer Slide Time: 06:12)

The screenshot shows a presentation slide with a list of database tables and a handwritten SQL query. The tables listed are:

- branch (bname, bcity, assets)
- customer (cname, ccity)
- account (ano, bname, bal)
- loan (lno, bname, amt)
- depositor (cname, ano)
- borrower (cname, lno)

Below the tables, a hand-drawn SQL query is shown:

```
{ select * from depositor, account
  where depositor.ano = account.ano
    aid bal }
```

The word 'aid' is circled in yellow, and 'bal' is underlined in red. The entire query is enclosed in curly braces.

So, suppose if we do this, if we simply write this, *SELECT \* FROM depositor, account* then what happens ... by the way this is one interesting point to note that the select and from are mandatory, but the WHERE can be omitted we have if the where is omitted it essentially means where true; so that means, all the tuples are select. So, if we just say *SELECT \* FROM depositor, account*, this is essentially a Cartesian product of these two. So, this is a Cartesian

product of depositor and account.

And essentially every tuple that is in depositor and account, then all the values are selected. Now, what happens is that depositor and account both have this attribute name *ano*. So, how is it finally shown, how is it actually being represented is that, so the *ano* what the point is that *ano* this is a clash of the name and this is ambiguous. So, if something needs to be done on one of the account numbers that needs to be specified.

So, for example, if we want only the depositor account number, so what we may need to do is instead of this what we will need to do is, you say select *depositor* dot (so, this is the dot operator) dot *ano* (*depositor.ano*) and suppose, we also want to select the, whatever *balance* etc. So, *depositor.ano*; that means, now we are only selecting the *ano* from the depositor column. And by the way, this if we just write this query, as you can see that, this query ... this is a Cartesian product query, this is not very useful we would rather want the join that we saw earlier.

So, how do we specify the join, again the join is an equality join, so we have to simply say *ano* of account is equal to *ano* of depositor. So, we have to make you can simply say *ano* is equal to *ano*. Now, you can see that this does not make any sense, this is ambiguous, so what we need to say is we need to qualify this. Whenever there is an attribute that is repeated you need to qualify it with from which table it is coming.

So, we need to say, *depositor.ano* is equal to *account.ano* and that is the way to select this. Fine. So, that is the way things work, this is important ... this essentially is the way of doing the join and we will see the example of join later. And now, what also happens is that ... so couple of more things is that, this *depositor.ano* this is a very clumsy name to use further. So, the SQL let us one rename operator, so you can say instead of *depositor.ano*, you can add in *depositor.ano* as let us say *aid*.

So, what the ... finally, what will be selected is *aid* and a *balance* table, so this is the table with *aid* and *balance*, because *depositor.ano* is renamed. So, the ‘as’ operator essentially is for renaming, this is the renaming operator *as aid* and even *depositor* can be renamed. So, we can write the above query in the following manner, a relation is renamed it can be used directly here. So, again you can of course, have renamed *depositor* as *d* and just say *d.ano* and *d.ano* here and everywhere else.

So, that can be also done. So this is renaming, renaming is not just a way to reduce the namespace, clash or etc is not just a way of convenience, sometimes it is necessary and here is an example, where it is necessary.

(Refer Slide Time: 10:05)

The slide shows a list of relations:

- customer (cname, city)
- account (ano, bname, bal)
- loan (lno, bname, amt)
- depositor (cname, ano)
- borrower (cname, lno)

Handwritten text: "Find names of all branches that have greater assets than 'DEF'"

Handwritten SQL query:

```

select T.bname
from branch as T, branch as S
where T.assets > S.assets and
      S.bname = "DEF"
  
```

Now, this requires to find branches from *branches* with name DEF, this is a little complicated query, but this can be broken up into the following manner. So, let me write down the answer to this query and then it will be clearer as to how this can be done. So, I am saying *select* you want to find the names of all branches. So, there is some *bname* that we require from ... of course, there is a branch that we require.

Now, the where part, so what do we want. We want the branch ... the *bname* of the branch whose assets is greater than the branch whose name is DEF. So, what do we do? We require actually two branches here, because we need branches from the branch relation twice and now this is a problem. Now, both are branches, so how do we do that, so we cannot use both as branch, we must rename.

So, let us say we rename the first branch as T and the second brand as S, we want branch from T, where *T.assets* this must be greater than *S.assets* and *S.bname* (branch name) is equal to DEF. So, let us analyze this query first of all. So there are ... we are selecting the branch twice here and here, one is renamed as T, one is renamed as S and this is necessary, without renaming it cannot work.

Because, there are ... both are branches and we are saying that we want to select everything from this relation branch, such that its assets is greater than S, but which kind of S? Only those S whose branch name is DEF. So, essentially this to select the assets of the branch whose name is DEF and we want everything else, where that is greater and only selecting that. This is an example of a little complicated query and where renaming is necessary and this also highlights the fact that the same name *branch* is used twice. That is why the renaming is necessary. Okay. Just to ... now that we are inside this, this branch name equal to ... there are certain kinds of string operations that we can define.

(Refer Slide Time: 12:36)

```

loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)
Find names of all branches that have
greater assets than "DEF"
| select T.bname
| from branch as T, branch as S
| where T.assets > S.assets and
|       S.bname = "DEF"
|       = "DEF"
|       like "%DEF%"
|       like ".DEF"
= .

```

So, we have been seeing this equality operator is equal to DEF, this means this matches exactly DEF. You can also say like DEF. So, wherever ... this is like a percentage ("%DEF %"), so the percentage essentially is a substring. So, everywhere where the name DEF is there it will be selected and there are many such things and so you can say a *dot*. So you can say like *dot DEF* ("DEF"). So, *dot* stands for a particular character and there is a regular expression thing that can be done here, so this can be checked up.

So, one important thing that we have been studying about relational algebra and SQL is that these are sets or multi-sets. So, let us stick with SQL, these are multi-sets, so, that means, by default there is no ordering of the tuples. Now, which is true, but sometimes the output needs to be in a particular order. Now once more, this is an important point to remember is that we ... the SQL lets you run a query and lets you see the output of this. So, now, how to actually

visualize the output? The output can be in any order that the database engine wants.

But, suppose you want the output to be in a particular order, so you can actually sort, but do remember the sorting is done after all the tuples have been selected and this is just a .... sorting is done only for the output of this. This is got no actual meaning in the relational algebra model or SQL model. So, how do we do over the ordering?

(Refer Slide Time: 14:15)

The screenshot shows a window titled 'File Edit View Insert Actions Tools Help' with various icons. Below the title bar, there is a list of relations: customer (cname, city), account (ano, bname, bal), loan (lno, bname, amt), depositor (cname, ano), and borrower (cname, lno). A handwritten note in blue ink is overlaid on the screen, starting with 'order by'. It includes three separate SELECT statements: one from customer ordered by cname desc, one from depositor, and one from borrower. The entire note is enclosed in parentheses and followed by a 'union' keyword.

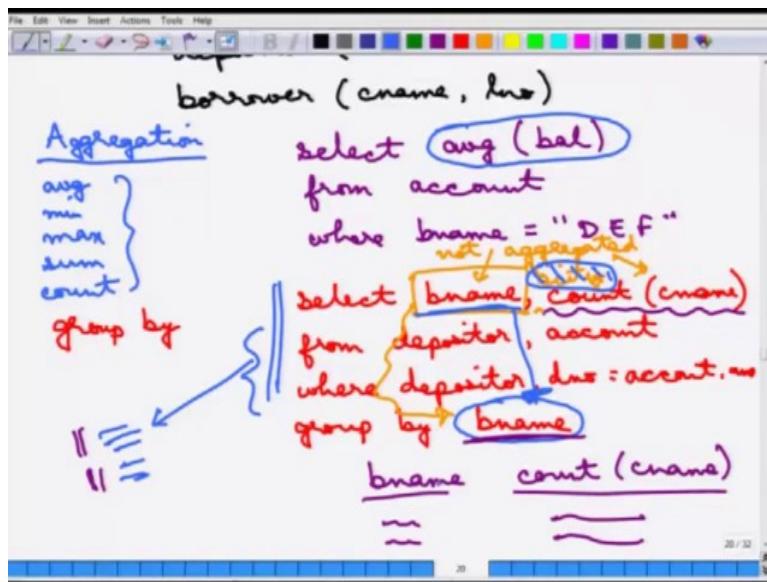
```
customer (cname, city)
account (ano, bname, bal)
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

order by      select cname
              from customer
              order by cname desc
(select cname from depositor)
union
(select cname from borrower)
```

So, you simply use there is a construct called *ORDER BT*, so there is a construct *ORDER BY* which lets you order the tuples. So, you can say *SELECT cname FROM customer ORDER BY cname* whatever. So, this is the *cname* will be now in order, in ascending order by default. If you want to do something else, you can say descending. So, by default this is ascending you can also add *desc* to make it descending order.

So, this is one kind of operation, the other kind of operations is set union etc can be done and you can also say distinct, etc all of these can be done. So this is the set operation. The next important thing that we would want to do is the aggregation, so the aggregation can be done.

(Refer Slide Time: 15:12)



So, again the aggregation is being done, so the aggregation. So five operations are allowed to be used. They are average, min, max, sum and count. So, these are the five aggregation operations that can be done. And a query something like this can be specified. So, *SELECT avg(bal) FROM account WHERE bname = "DEF"*, select average balance from let us say, so say we would select the average balance from account, where let us say branch name is equal to DEF. So, essentially it selects the average balance of all the accounts in the branch DEF, so this is an aggregation operation that is being done on the balance.

Now, as we saw earlier, it is not always useful to apply the aggregation on the complete set of tuples. But, on certain groups of tuples, so how is the grouping done? The SQL offers a way of grouping the tuples, which is essentially using the construct *GROUP BY*. So, there is a construct *GROUP BY*. This let us you group by. So, again we can do the following thing. So we can say account this thing or also ... let us write down another query.

```
SELECT bname, count(cname) FROM depositor, account WHERE depositor.dno = account.ano GROUP BY bname
```

So, what are we doing here? This is important to understand is that.

So, what are we trying to do here is the following, is that we are doing certain query, but we are saying *group by* a branch name. So, whatever is the output up to this point, from depositor account etc. So, we are essentially selecting all the accounts and on the depositor information

around it, we are grouping by *bname*. So, all of those tuples that are result of this are then grouped into different kinds of things, according to how ... whether they have the same branch name or not.

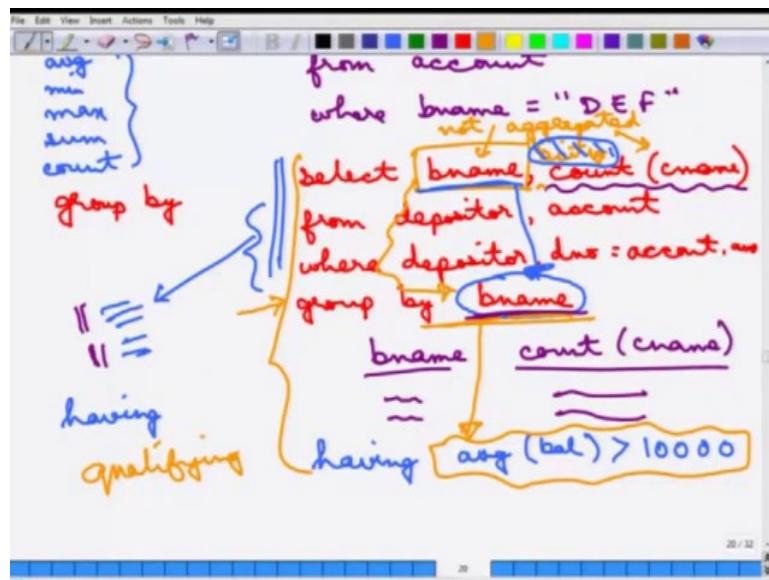
Now, once they have the same branch name for each of this branch name we are applying some aggregate operation, which is a count of *cname*. So, the answer to this query is looks like is of the following form is of the branch name and count of *cname*. So, for each of the branch names, we will have a count of how many customers are there, that is the thing. So, this is the same as the relational algebra that we talked about the relational algebra grouping and aggregation that we talk about.

Now, one important thing is that, the attributes in the select clause that are not aggregated. So, this is our attribute in the select clause that are not aggregated, so this is not aggregated and this is aggregated. So this is not aggregated. They must appear ... they must be grouped by. So, they must appear in the *GROUP BY*; otherwise, which has got no meaning. So, for example, we cannot select branch name and let us see, we cannot select branch name and branch city on the thing.

So, this branch city would have been wrong, because there is no way to get the branch city. Once you have selected by branch name, there is no way to get to the branch city or once you have selected ... So, it does not make any sense, so whatever is not aggregated in the select clause must be present in the group by. So, this is mandatory; otherwise, if you just say select branch name, branch city this thing then, it is wrong. So, this is wrong you cannot do this, this is one important thing to remember. Okay. Fine.

So, then the group by now you can say this is *GROUP BY*, but you can also qualify the groups and how do you qualify the groups, you can qualify the groups. So, you can say I want to select particular groups.

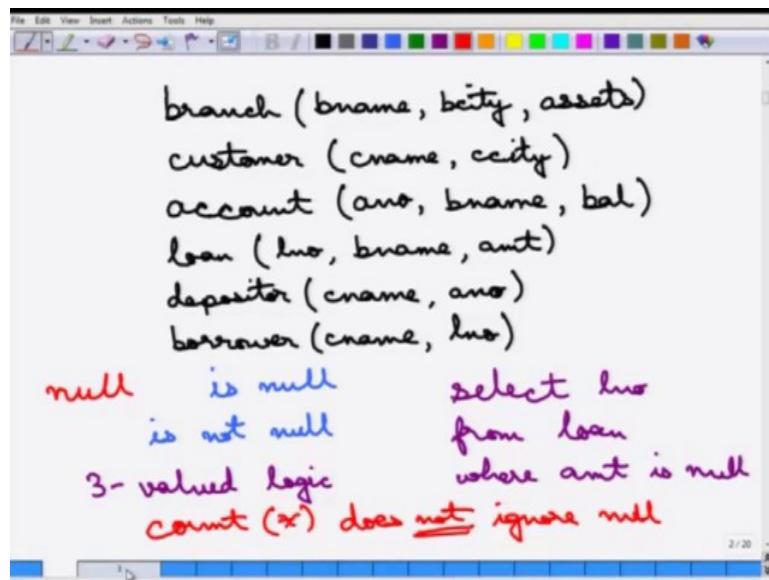
(Refer Slide Time: 18:59)



So, the qualifying is done by the *HAVING* clause. So, instead of saying select branch name and count *cname* from everything etc, etc you can say, I want to select the branches, where the average balance or whatever, the average balance is at least some 1000 rupees or something like that. So, what you will do is, you will write down the same query group by etc and in the end you will add *HAVING* average balance greater than whatever 10,000 rupees or something like that.

So, these four things ... let me highlight ... these four things together ... these five things together is the query. So, *SELECT bname, count(cname) FROM depositor, account WHERE depositor.dno = account.ano, GROUP BY bname HAVING avg(bal) > 10000*. So, you are not going to do the group by on every branch name, you are only going to do the group by on those branch names, where the average balance is greater than 10000. So, this is like qualifying a particular group, so this is useful, because sometimes you want to analyze only certain groups which are useful to your analysis.

(Refer Slide Time: 20:20)



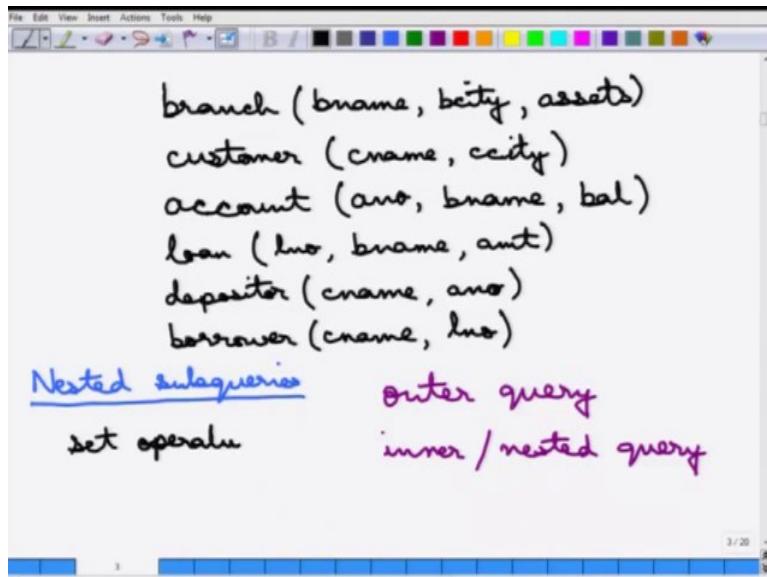
So, we will next consider the issue of *null*. In SQL a value can be checked whether it is null or not by this following construct , is null or is not null. An example query is of the following form, *select lno from loan, where amount is null*. So, that means, it will try to find out the loans, where the amount .... there is some problem with the amount, it has not been updated correctly etc. Now, for null again the same issue as the relational algebra happens, so this does follow that same three valued logic.

And evaluations, so, result of the expressions involving null evaluate to null and comparison with null returns unknown plus the aggregate functions ignore null. So, for example, average etc will ignore null. But the only difference is the *count(\*)* does not ignore null. Every other aggregate operation ignores null. So, we will next move on to a very important part of SQL, which is called a nested sub-query.

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

**Lecture - 10**  
**SQL: Advanced Queries**

(Refer Slide Time: 00:09)



The screenshot shows a presentation slide with a title bar and a main content area. The content area contains a list of database tables and some handwritten notes:

- branch (bname, bcity, assets)
- customer (cname, ccity)
- account (ano, bname, bal)
- loan (lno, bname, amt)
- depositor (cname, ano)
- borrower (cname, lno)

Handwritten notes below the list:

- Nested subqueries
- set operator
- outer query
- inner / nested query

So, this is a nested sub-query. So essentially, what happens is that a query is used in the FROM and WHERE clause of another query. The query on which the FROM and WHERE is used is called the outer query and the query that is used inside is called the inner query. So inner query or sometimes it is also called the nested query. An example I will give immediately, but this is mostly useful for set operations.

So, if there are set operations, then these nested queries are very useful. So, for example, suppose, we want to find out names of all customers having a loan, but not an account. This can be solved in the following manner.

(Refer Slide Time: 00:58)

The slide shows a database schema with tables: account, loan, depositor, and borrower. Handwritten notes explain nested subqueries:

- Nested subqueries**: A section with a blue underline.
- set operations**: A section with a blue underline.
- Find names of customers having loan but no account**: A note describing the query being explained.
- outer query**: A section with a blue underline.
- inner / nested query**: A section with a blue underline.
- select cname from borrower**: A part of the query with a red arrow pointing to it.
- where cname not in (select cname from depositor)**: A part of the query enclosed in a red circle.

This is an inner query, `SELECT cname FROM depositor`. So let us try to understand what is happening ... how the query is being deciphered. So, first of all, this inner query is run `SELECT cname FROM depositor`, so it essentially gives the names of all the customers, who have a depositor account. And then, it tries to select those names from the borrower, that is who has a loan, where cname is not in ... this is a set, so this is essentially equivalent to the not of set membership and this is WHERE nested sub query is very useful.

(Refer Slide Time: 01:46)

The slide shows a database schema with tables: branch, customer, account, loan, depositor, and borrower. Handwritten notes explain correlated queries:

- Scoping of attributes**: A section with a blue underline.
- inherent**: A section with a blue underline.
- correlated queries**: A section with a red underline.

And a related issue for this nested sub-query is the scoping of attributes, so the scoping. So,

this is the same issue as in other programming languages, such as C. When a block of statements is used inside another block of statements. And the same name is used, which scope does it apply to? And the same default rules apply. So, if nothing is qualified it applies to the innermost query, otherwise it can be qualified.

And, a nested query ... so then there is some other terminology that we are going to use. When a nested query refers to something in the outer query, that is called a correlated query. Because essentially these two queries are not independent of each other, but they use the results of each other.

(Refer Slide Time: 02:33)

So, the query is, so this is solved using a correlated query and the answer to this solution can be written in the following manner, so this is the most important part of this. So, this  $B.lno$  is equal to  $D.ano$ , so this essentially this  $D$  refers to the one in the outside query, so this is why it is a correlated query. Let us also go over the query to see how this solves the particular. So how does it solve find names of all customers having same account and loan number.

So, let us first see, what is being done in an inner query. So, the inner query says it selects all customers, who has a loan such that of course, the name is the same, this is just to ensure that the join is meaningful and whose loan number is equal to some other account number  $D$ , where... What is  $D$ , where  $D$  is the depositor, so  $D$  is the depositor which has the same customer name. So, essentially it tries to connect  $B.lno$  with  $D.ano$ . And because of these two clauses  $B.cname$  is equal to  $customer.cname$  and  $D.cname$  is equal to  $customer.cname$ , it

connects them with the same customer. This is a little complicated query, but if this is being understood one at a time, first the inner query with this scoping to the outside and then, it is easier to understand. And it uses different concepts, first is a correlated query and of course, which is a nested query then this renaming and then also this set membership operation *IN*.

(Refer Slide Time: 03:59)

The slide shows the following relations:

- branch (bname, bcity, assets)
- customer (cname, ccity)
- account (ano, bname, bal)
- loan (lno, bname, amt)
- depositor (cname, ano)
- borrower (cname, lno)

Handwritten notes explain the definition of *some*:

$$\text{some } F < \text{comp} > \text{some } r \Leftrightarrow \exists t \in r (F < \text{comp} > t)$$

Below this, two examples are shown:

- $5 < \text{some } \{0, 5, 6\} = \text{true}$
- $5 < \text{some } \{0, 5\} = \text{false}$

So, now, let us go over some of the set operators. The set comparison operators. The first set comparison operator that we will talk about is called *SOME*. So, *some* and here is the definition of *some*, so this is  $F$  compared to some comparator operator with *some r*. This implies that this is equivalent to ... There exists some tuple in the relation  $r$ , such that  $F$  compares to  $t$ .  $F < \text{comp} > \text{some } r \Leftrightarrow \exists t \in r (F < \text{comp} > t)$

So, what does it mean? So, let us take an example, so if we take this following example 5 is less than some of, let us say the set is  $\{0, 5, 6\}$ .

So, what does it mean? That we must find some  $D$  in this relation and if we can find some  $D$  in this relation such that 5 is less than that, then this is true. So, how it is being solved is that 5 is tested with 0, it is not true, that does not matter. 5 is tested with 5, which is not true. But, 5 is less than 6, so that comparison goes correct. So, this evaluates to true. Because there is some element in this, such that 5 is less than ... this comparator is less than one of those elements.

So, there must be at least one element, such that this holds. Then you can say that together if

this holds, this comparator with some r holds. So that's the meaning of some operator. And we can see some examples. For example, if you say 5 less than some  $\{0, 5\}$ . This will, however, evaluate to false, because there is no element such that 5 is less than that. So, this goes through and then, it can be the ... equivalent of this can be defined.

(Refer Slide Time: 05:57)

So, let me just complete this. So, 5 is equal to some  $\{0, 5\}$ . This is true, because there is one element, where 5 is equal to true. Very importantly, and this is a little counterintuitive, 5 not equal to some  $\{0, 5\}$  is also true, because there is some element in the set that is not equal to 5. So this is a little counterintuitive, but this is how the thing works. So, let me go ahead with the next operator, which is similar operator, which is *all*. The *ALL* is the defined in the same manner.

So,  $F$  comparator with *ALL*  $r$  is equivalent to for all  $t$  belonging to  $r$  this must hold.  $F <\text{comp}> \text{all } r \iff \forall t \in r (F <\text{comp}> t)$

So, all the elements must follow this. So, then, if we say  $5 < \text{all } \{0, 5, 6\}$ ; that is false, because there is some element 6 here, where 5 ... there are some element 0 and 5, where this is not *true*, so this is *false*. And, similarly we can go ahead with the other examples, which is with ... if this is 5 less than all of, let us say  $\{6, 9\}$  this is true. Then, 5 is equal to all of  $\{0, 5\}$ , this is *false*, and so on, so forth.

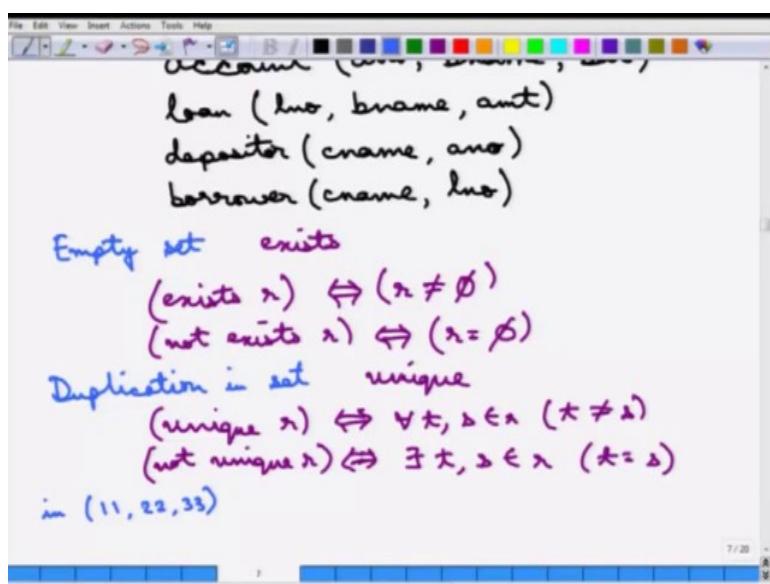
So, essentially, what happens is, now we can decode *some* and *all* in terms of their set

membership. So, *some*, if you say equal to *some*; that is the same as saying ... so this operator equal to *some* is the same as saying whether *in*. So, when you say for example, 5 equal to *some*; that means, is there any element in this set; such that this equality holds, so that is essentially the *in* operator this is the *in* operator.

However, not equal to *some* is not equivalent to not *in*. This is something to be remembered. Because this is not equivalent. Because ... again we see from this example this can be clear that it does not mean that it is not in. On the other hand, if we now look at all, so the all, we can break down *all* in the following manner. Not equal to *all*, if I write not equal to all, that is equivalent to not *in*. So this is essentially this. So, all of the set members must not be equal to this.

So, none of the set members can be equal to this. That means, this is not in that set. However, using the same logic, if you say equal to *all*, that is not equivalent to *in*. So these are just two of the ... two important things to remember. This goes through, this goes through, but these two do not go through. So, this is why these two are equivalents. Okay. Then there is a little bit issue with the empty set.

(Refer Slide Time: 09:09)



And, let me go to the next page, so there is an issue with the empty set. So, what is an empty set, how ... so a set can be empty of course, and there can be tested with the ... So, there is a operator called *exists*, which tests whether the set ... whether there is an element in the set, which is essentially saying whether the set is empty or not.

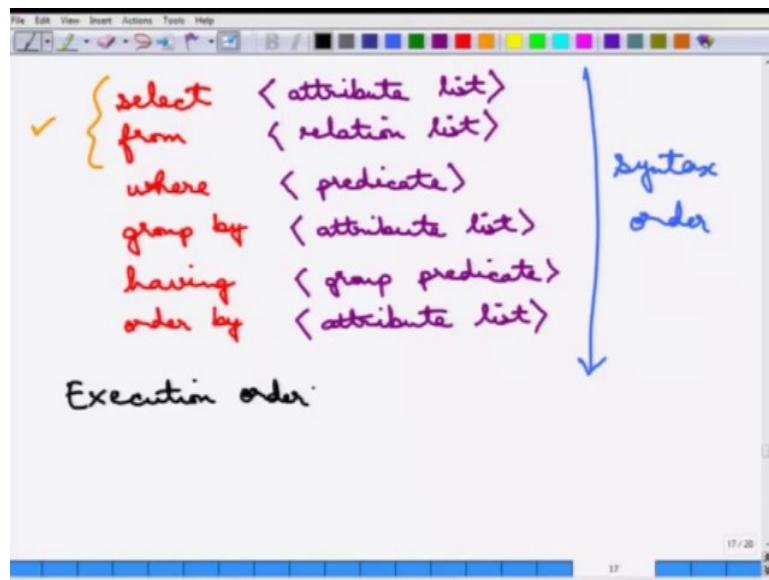
So, exists  $r$ , which essentially is equivalent to saying,  $r$  is not to equal the null set, which is this thing.  $(\exists r) \iff (r \neq \emptyset)$ . And ... the complimentary is not exists  $r$ , which is equivalent to saying  $r$ , whether  $r$  is equivalent to phi or not.  $(\not\exists r) \iff (r = \emptyset)$ . And again, this can be used in different set operations etc. Similar to empty set, there can be duplication in sets. Because remember that SQL uses multi-sets. So duplication in a set can be tested by this operator called *unique*.

So, essentially testing whether there is something unique in that. So if you say *unique*  $r$ , that means you are testing it all the tuples in the set  $r$  is unique or not which is ... it can be written down in the following manner.  $(\text{unique } r) \iff \forall t, s \in r(t \neq s)$ . And then, there is not unique, meaning there is at least one tuple  $t$ , which is duplicated. So, that means, that  $(\not\text{unique } r) \iff \exists t, s \in r(t = s)$ . So, that is how the duplication is done.

So, duplication is useful in the examples such as, find names of customers, who have utmost one account at a particular branch. So, essentially you are testing whether for that customer you find out all the accounts in that particular branch and say if the customer name is duplicated or not. If it is not you return it, otherwise you do not have to do that. So, this is the way to do it. And in SQL, just to complete the story, sets can be explicitly denoted using this kind of operator, so this is a set with this three things  $\{11, 22, 33\}$ .

So, something can be tested, so one can test whether a particular account number is within this or whatever. So, this is the basic idea of a SQL format. So, let me just highlight the format of this SQL, this is essentially ... consists of up to 6 clauses. They may be nested.

(Refer Slide Time: 11:51)



And these are ... 6 clauses are: *SELECT*, *FROM* these two are mandatory. So, these two are mandatory, this must be there. And the others may or may not be there. And it must be specified in this particular order. *SELECT*, *FROM*, *WHERE*, *GROUP BY*, *HAVING*, *ORDER BY*. So, this is ... *SELECT* essentially is on a particular attribute, this is a summary of the attribute list, *FROM* is the relation list, *WHERE* is predicates, *Group By* is again on attribute list, *Having* is again group condition / group predicate, and *Order By* is again on an attribute list.

Note that, this is the syntax order. SQL query must happen in this syntax. However, this is very, very different from the execution order. So, what I mean by the execution order is that, suppose a particular SQL query with these following 6 clauses are being queried. Now, the question is how does the database engine solve it? Does it do the select first? It really cannot. Because unless a *FROM* it knows, which relation the *FROM* has to come, it cannot do it.

So, in these 6 clauses it is ... the syntax order is not the same as the execution order. And I leave it to you as an exercise as to figure out, what can be the execution order of the 6 clauses of the basic SQL query. Alright. So, let us move ahead to this something else called the derived relations.

(Refer Slide Time: 14:09)

loan (lno, bname, amt)  
depositor (cname, lno)  
borrower (cname, lno)

Derived relation      Find avg account bal of branches  
                        where assets > 1000

select bname, avg-bal  
from branch\_avg  
where avg-bal > 1000

select bname, avg(bal)  
from account  
group by bname

branch\_avg (bname, avg-bal)

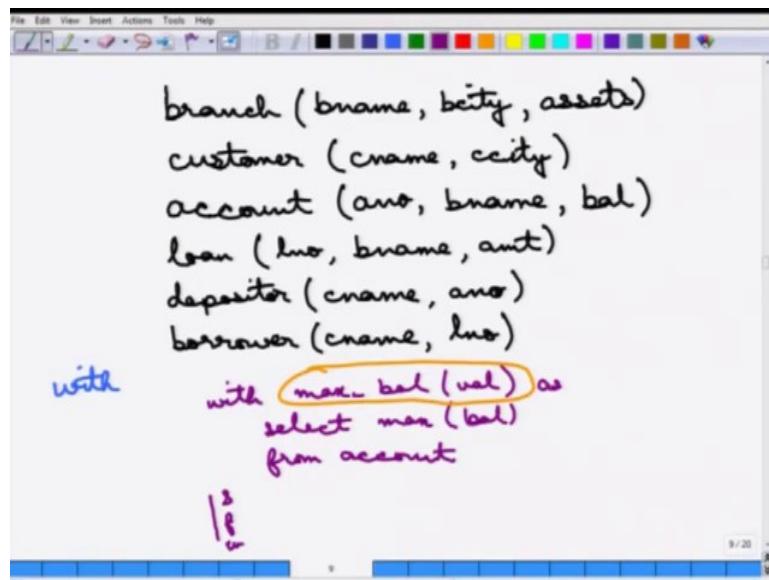
where avg-bal > 1000

So, a derived relation ... something that is used in the middle of a query. So, same like a nested query etc, And ... so, for example, let us take this particular example. Okay. So, how do we solve it? Essentially there is a ...we can do it in the following manner. So, let us see how this is being solved. So, first of all, let us look at the inner query. the inner query essentially just selects the branch name and the average balance of that branch name. So it is grouped by the branch name, for each branch name it finds the average balance.

Now, this as, the answer to this query is branch average is the derived relation. So, this is a new name that is given to the answer to this query, which is the *branch\_avg*. With the attributes name as *bname* and *avg\_bal* So we are essentially renaming that stuff. So, this query, now becomes of the following format, *SELECT bname, avg\_bal FROM branch\_avg WHERE avg\_bal > 1000.*

So, this boils down to *SELECT*, the same thing, branch name, average balance *FROM* this branch average, is what we have been using, where average balance is greater than 1000. So, this boils down to this, because a relation is being derived as part of the query answering. First this relation is derived, then that is being used in an outer query.

(Refer Slide Time: 15:42)



A screenshot of a database management system interface. The menu bar includes File, Edit, View, Insert, Actions, Tools, and Help. Below the menu is a toolbar with various icons. The main area displays a SQL query:

```
branch (bname, bcity, assets)
customer (cname, ccity)
account (ano, bname, bal)
loan (lno, bname, amt)
depositor (cname, ano)
borrower (cname, lno)

with
    with max_bal (val) as
        select max(bal)
        from account
```

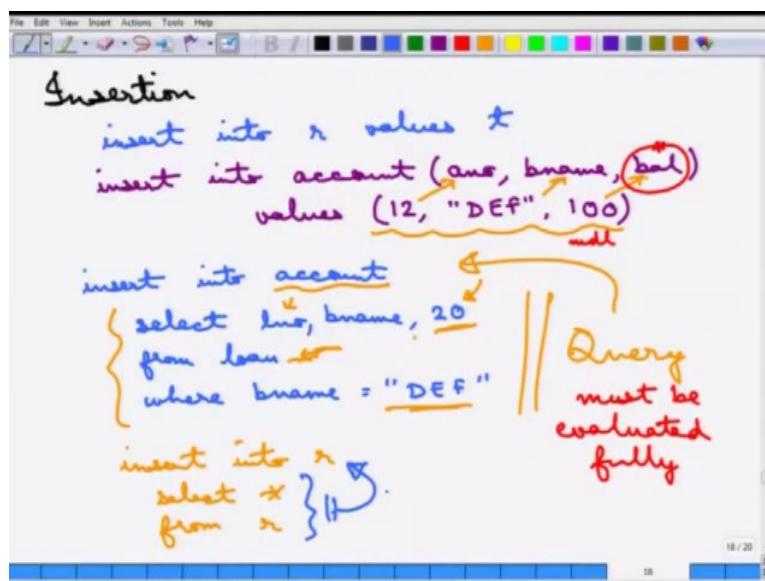
The text "with" is written in blue. The subquery "select max(bal) from account" is enclosed in a yellow oval, and the alias "max\_bal (val) as" is also highlighted with a yellow oval. There is a small handwritten mark "13" at the bottom left.

And, just to go over a little bit more of the syntax of this. There is a *WITH* clause in SQL, which defines a temporary relation. So, you can define *with ...* so you can write other query, some SELECT FROM, WHERE etc. Alright. So, that completes the query part of the SQL. So, we have covered all the query parts. Next, we will cover the updating. So insertion, deletion and updating. How to modify the tuples in SQL?

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

**Lecture - 11**  
**SQL: Updates, Joins, Views and Triggers**

(Refer Slide Time: 00:13)



The first thing that we will cover is the insertion. The insertion, the syntax is simply, *INSERT INTO* the particular relation name, let us say  $r$ , values, which forms of the values, the tuples that is formed. So, an example is,

```
INSERT INTO account (ano, bname, bal) VALUES (12, "DEF", 100)
```

we can say, essentially this means, 12 is an account number, DEF is the branch name and 100 is the balance. So, this tuple gets inserted into the account table.

If the schema is obvious, that means, if the schema of the account is actually  $ano$ ,  $bname$  and  $balance$ , then this can be omitted, but it can be added. And instead of 100 etc, a particular value is not known, a *null* can be also inserted, provided balance is nullable. I mean, if the condition, you remember that while we were setting up the table name, we can define certain attributes as whether they can allow null values or not. If they allow null values, then this is fine otherwise there will be errors. Now, one thing is that, insertion can also be ... so, the value of a query can also be used to insert into a table.

So, for example, this can be done.

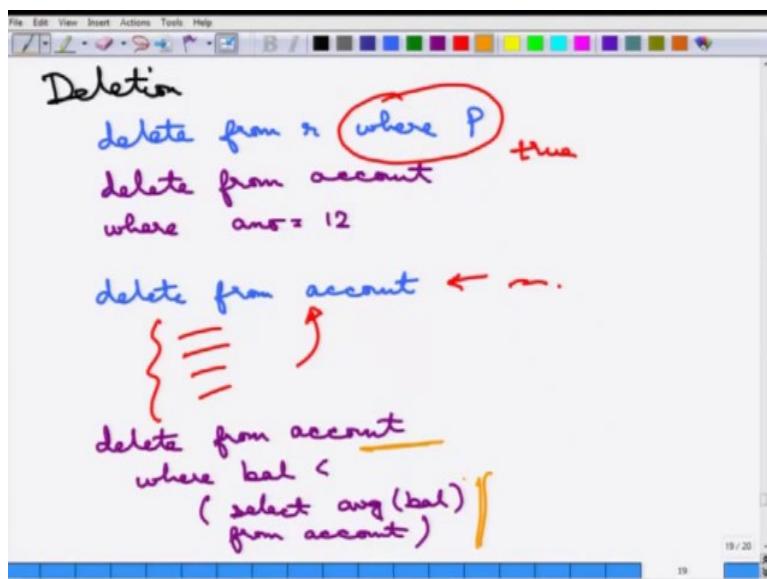
*INSERT INTO account*

*SELECT lno, bname, 20 FROM loan WHERE bname = "DEF"*

What essentially does is that, it creates a new account with the balance 20, so the balance is always set to 20, at this branch DEF. For every loan, wherever there is a loan, it creates an account and the account number is said to be the same as the loan number. So, this is a way of inserting as part of that. So, this is essentially a query, which is first solved, then all the values are inserted into the account.

Now, the one thing is that the query must be evaluated fully before the insertion starts, as otherwise there can happen ... infinite insertions can happen. So, very easy example is the following. So, suppose you want to duplicate a relation. For whatever reasons, its not very clear why. But, suppose you want to do that, so you can write as queries, simply *insert into r select \* from r*. Now, imagine what will happen. If this is not being null, so this must be evaluated first, so ... and then all the results are inserted into it. So, this is only .. it returns 1, so this essentially duplicates the relation. That it, I mean, duplicates the tuples in the relation.

(Refer Slide Time: 02:39)

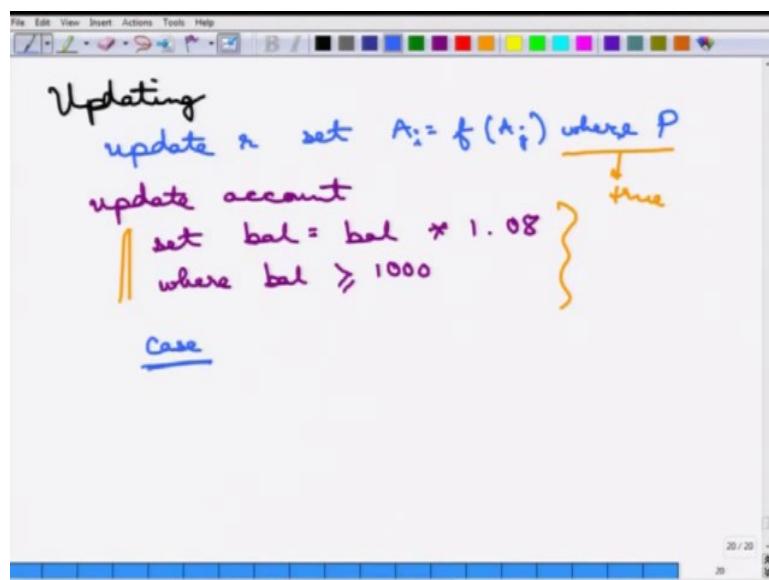


So, then, we can move on to deletion. The syntax is *delete from relation where* there is a predicate. So, you delete from the relation r, where the predicate is correct. So, example is, so essentially the account number 12 is deleted. And if the where is empty, so if we simply say *delete from account*, the same rule is applied, if the where part is empty, then it means true,

which means every tuple will be deleted from account. So, it ... at the end of this, it returns an empty relation nothing else is.

And just like insertion, delete can also ... you can use a query as part of this delete thing. So, only where the query is evaluated and only those tuples that satisfy the query can be ... are deleted. And, interesting example in this space is the following. Once more, so what it tries to do is, it deletes all accounts where the balance is less than the average. Now again, once more, the balance must ... the average must be completed first and then, the deletion happens, otherwise it will keep on deleting. You can see what will happen.

(Refer Slide Time: 03:54)

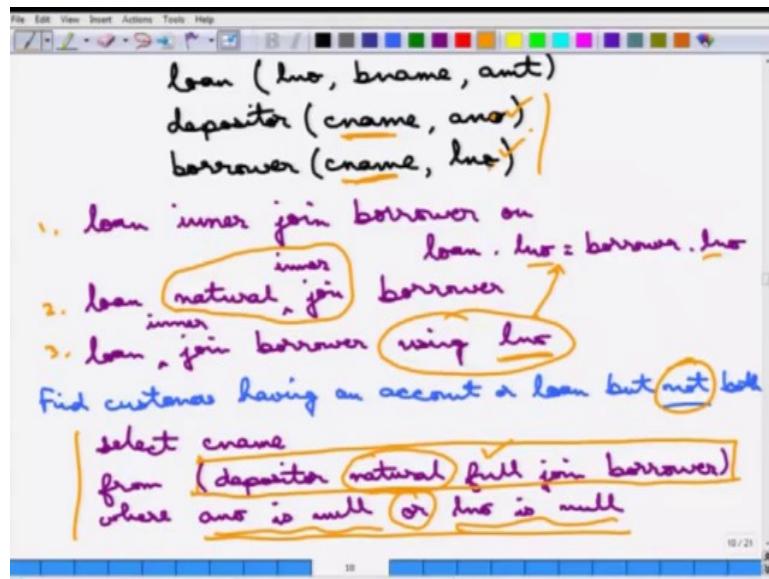


Finally, there is this updating. The syntax is *update relation r set* certain values, set attributes lists etc. So, set attribute lists equal to sum function of whatever, some other attribute list. So, set  $A_i$  is equal to function of  $A_j$ , where  $P$ , so the predicate. An example is, *UPDATE account SET bal = bal \* 1.05 WHERE bal ≥ 1000*. So what this tries to do is, every account, where the balance is greater than some amount 1000, 8 percent interest is given, so the balance is essentially incremented by 8 percent, so this can be done.

Once more, *where*; if the *where* is empty, then it evaluates to true. So, that means, everything will be done and so on and so forth. That's the same issue as past. And again, it depends on how you are doing, so again, this can be a query of ... which is first evaluated and then, this thing is done. And there can be case ... there is a case clause that can be ... which is

essentially equal to the switch case kind of thing and there we can see the syntax later. So, this does ... the basic queries and the database modifications are complete. We will next do another very important topic next.

(Refer Slide Time: 05:21)



So, we will next cover the join in SQL. So join, as we saw there are different types of join. Inner join. Then there is a left join, which essentially means it is a left outer join. Right join. Full join. Natural join. These (first 4) fall in one group. The natural join falls in another group. A particular join can also be set, can be set *on* a particular predicate and then, it can be also set *using* which attribute. So, the *on* is on the predicate and *using* is the attribute. So, these are the different ways one can join.

So, for example, we can say *loan inner join borrower on*, you can specify the condition, which is *loan.lno = borrower.lno*. Now this is equivalent to saying, this is *loan natural join borrower*. This is again equivalent to saying, *loan join borrower*. Okay, I am sorry, this should be *natural inner join* and this is *loan inner join*, using *lno*. So, all these three queries are equivalent and they are different ways of stating.

So, the natural inner join is the same as the inner join on that number, because it is the same attribute value. So, it is essentially, if you say, using real number, it essentially translates to this condition and it reaches the same as writing it down explicitly *on* and there is the natural join as well. So, an example of join may be, so following can be one way of solving it. Now, do remember there are different ways in which a particular query can be solved or there are

different equivalent SQL statements that can be written that will solve the same query.

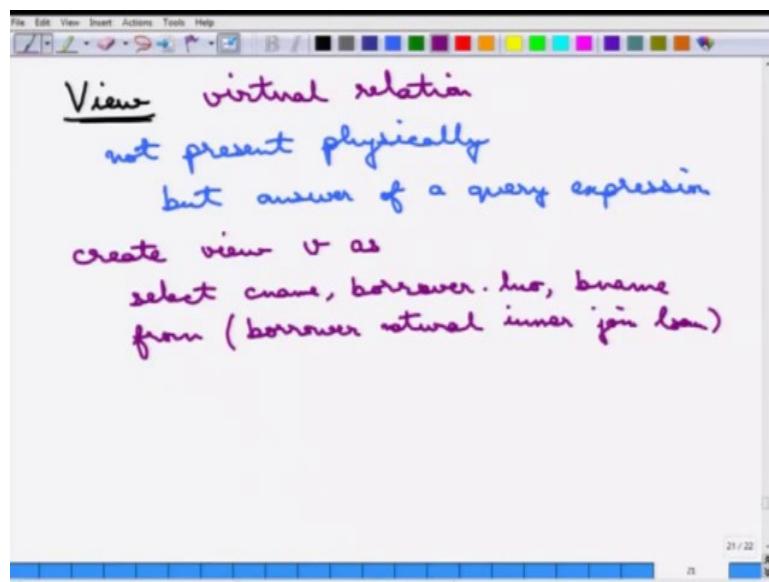
Find customers having an account or load but **not** both.

So, here is one way of doing it. *select cname from*, this is what, I am writing it down. So, *depositor natural full join borrower where ano is null or lno is null*. Now, this requires a little bit of thought. So, what it is being done is the following is that, first of all, this *depositor natural full join borrower*, this will create every possible combination of *depositor* and *borrower*, so it will find out all customers ... now this is a *natural full join*, so, when you say *natural join* the *depositor* and the *borrower* table, they agree on the customer name.

So, it must have that when you are joining, the *depositor* and *borrower*, they must be the same customer name. So it will find out all ... for each customer name the account number and the *lno*. Okay. Now, so, even if there is an account number, but not corresponding *lno*, because it is a *full join* it will output them.

Now, what do we want is that we want those kind of things, where there is either an account number or an *lno*, so at least one of them is null. So, that is why this is an *or* clause, so at least one of them. So, if a customer has both account number and *lno*, then both will be valid. So, this will be false. So, that is not going to be output. So, this is the way to solve find customers having an account or loan, but not both. So, this completes the part about SQL etc and, so there are some more constructs in SQL, some ... a little bit of constructs that we will cover next.

(Refer Slide Time: 09:55)



So, some important other constructs of SQL. The first one is the *view*. So, a view is the answer of a query that is not present physically, so this is not present physically, but answer of a query expression. So ... essentially a view is ... a view can be considered as a virtual relation. An example. So, why are views needed? Views help in query processing. An example may be the following is that, suppose this is being done.

So, *create view*, so this is the syntax to create a view, *v* as ... you say. So, what does this view tries to do is find all the loans of the customers, but, but it will does not bother about the loan amount. So, if we go to this example, so, so this is the view. So, this view can be later used anywhere, where this is use ... useful. So, if we do the following things.

(Refer Slide Time: 11:16)

The screenshot shows a database interface with the following tables listed:

- customer (cname, ccny)
- account (ano, bname, bal)
- loan (lno, bname, amt)
- depositor (cname, ano)
- borrower (cname, lno)

Below the tables, a hand-drawn query is shown:

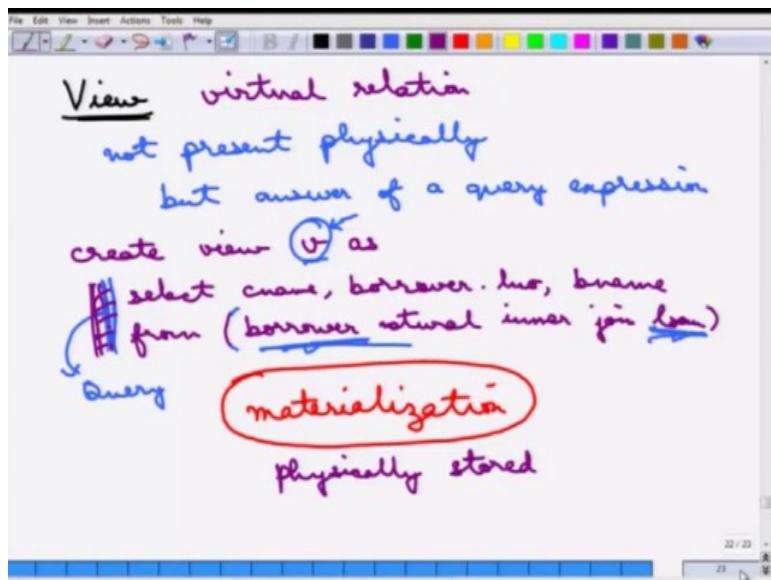
```
create view v as
  select cname, borrower.lno, bname
  from (borrower natural inner join loan)
  select * from v
  where bname = "DEF"
```

The hand-drawn part highlights the subquery and the WHERE clause with red and blue circles and arrows. A yellow arrow points from the WHERE clause to the table name 'v'.

So, suppose *select \* from v where branch name is equal to DEF*. So, you want the names of all customers who have an account at the DEF branch, well that's it. So, this view v is essentially this ... the part of this query. So, this is essentially the same as writing *select \* from*, then this entire thing will go here, this is here, where the same clause is being done. So, important thing to note is that, how are view stored? The view as I said is not stored physically.

So, what is being stored is the query expression is being stored. So, what is stored is this following query expression. So, wherever v is used, it is actually replaced by the query equivalent query expression, so this is the query expression. So, it is actually being replaced by the query expression and at run time, this query is evaluated and the entire answer is resulted. So, why is such a case done?

(Refer Slide Time: 12:34)

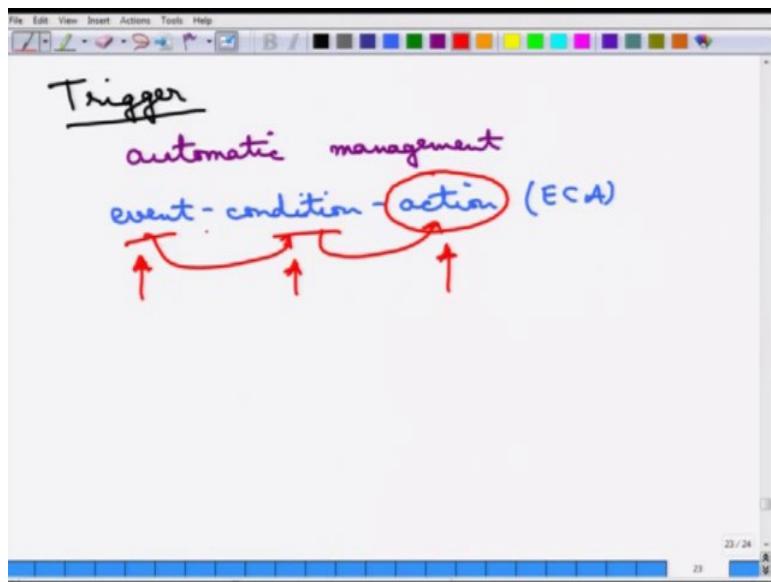


The reason is when the view is used the relation that it uses the view from may have changed. So, for example, if we go back to the query that we see, this create view  $v$ , uses the following tables *borrower* and *loan*. Now, if you store this as a table, if *borrower* and *loan* has changed, there is no way the *view v* will know that change. On the other end, if only the query expression is stored, then it is fine. Because, if *borrower* and *loan* changes it does not matter, this is evaluated again at query time ... at run time, so nothing else is problem.

So, that is why there are restrictions to, what can be done, which kind of views can be done and whether a view can be updated etc. A view may not be updated because updating a view essentially means updating the *borrower* and *loan* tables right, which is not clear how to update or it may get into the problems of null tuples, null values, and all those things.

For some views, there is a term called *materialization*. So some views are materialized. So, this depends on the query engine etc., the database engine. When some views are materialized, it essentially means the view is stored physically, is physically stored. Now, why will that be done when the view is simple enough etc, and when the view is used in multiple queries, then it makes sense to materialize a view, because then, this following query is not going to be evaluated at run time etc. So, that is the view materialization issue. And otherwise, a view is not very much updatable etc. Fine.

(Refer Slide Time: 14:20)



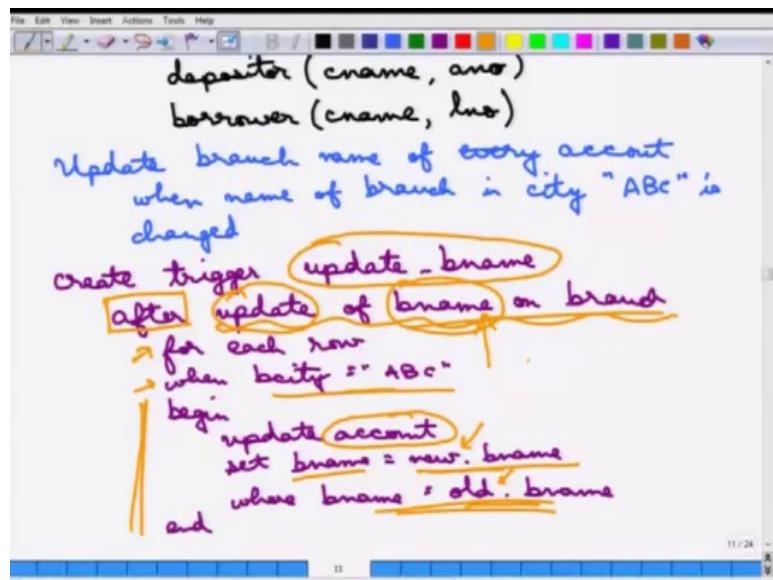
So, the last topic, that we will cover in this SQL is something called a *trigger*. So, a *trigger* ... so let me write it down what a trigger is. A trigger statement allows automatic management of database stuff. So, there is a ... so, it's an automatic ... so, whenever some action takes place in one relation automatically some other statements take place and it triggers ... essentially it triggers a couple of things.

For example, whenever a grade is submitted, for example, in the in the example of student databases. Whenever a grade for a particular course is submitted the CGPI of the student is automatically recalculated and that is being stored. So, that can be written as a trigger. So, whenever a new grade is inserted, there is a trigger that will automatically recompute the CGPI. So, it essentially follows, what is called an **event - condition - action** model. So, this is a ECA model.

So, whenever an event happens it checks for certain conditions, if that is true, the corresponding action is being taken. So, the event is essentially a database modification, so such as an insertion, updates or deletion etc. The condition is a predicate and the action is any other database action or some even external programs can be done etc. And the action can be done either before or after the event. So, this is a modification event.

So, the action can be specified as a 'before action' or an 'after action'. So, generally it is an 'after action', but it can be also specified as a 'before action'. For example, we can think of the following query.

(Refer Slide Time: 16:01)



So, create trigger, so it says ... *create trigger*, you can say update, the name of the trigger, you can give something, let us say, *update\_bname*. This is a trigger. After, so, this says that, okay, after that, this is a after model, so when the event has taken place after that ... *after update of bname on branch*. So, then you can say how this update will be done for ... so what is the action that can be taken.

So, *for each row when bcity = "ABC"*, you say you have the *begin*, there is a *begin* and *end* statement. *Update account*, in the following manner, *set bname = new.bname where bname = old.bname*, this is the *end*. So, this example covers a lot of issues, so let me go over them one by one. So, it first says *create a trigger* the name of trigger is *update\_bname*. Fine. This says *after*, so that means it's an after event. *After* what is the event? Update of branch name on *branch*. So, whenever there is an update of branch name on *branch* this trigger is essentially invoked. Now, how it is done? So that ... the trigger is applied *for each row* only when the branch city is ABC. This is the condition on the trigger. So, it is not for everything on the branch. Then, what is the action that is being taken? Is that, the *account* table, the *account* relation is updated in the following manner. The branch name is set to the new branch name for those things where it was the old branch name.

So, this *new* and *old* are essentially special markers. So, which says in the ... because it's an update event, so this ... the new branch name is got from here and the old branch name is got from whatever was already in the database. So, this is an example of how to work with the

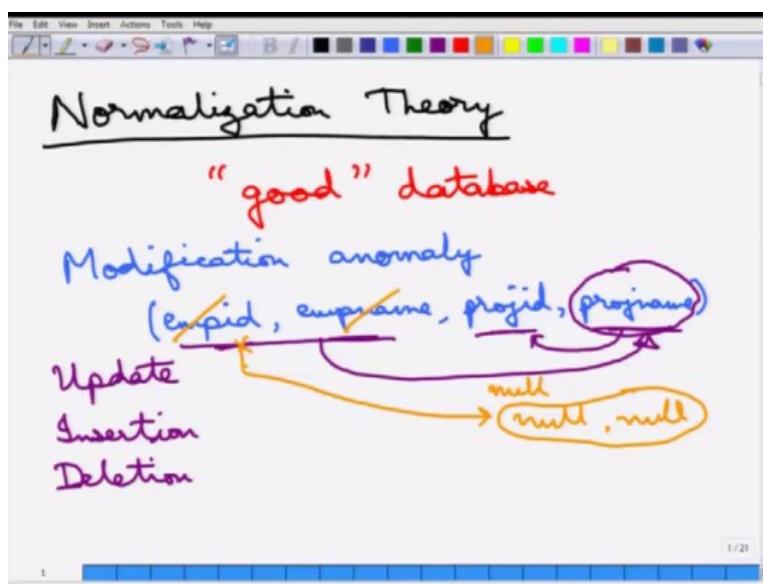
trigger. So, this completes a part of SQL. So, we have covered about all the basic operators. We have covered nested subqueries. We have covered, the updates, the deletion, insertion etc and some special issues in SQL, which are the views and the triggers.

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

**Lecture - 12**  
**Normalization Theory: Motivation**

So, today, we will start on Normalization Theory.

(Refer Slide Time: 00:13)



So, the ... essentially the main question that we will try to answer is, how to design a good database. Now, there can be different meanings of what a good database is and the answer can be given informally or formally. So, informally, we can say the database is good when each relation or schema represents a particular entity. For example, student is one kind of entity. So, there is a relation for student. Faculty member is another kind of entity. So, there is another entity for faculty member. There is a relation for course etc, etc.

Then, we can also say that there are no spurious tuple, that means there is no tuple which has got values that does not mean anything. So, there is very little or almost no redundancy. Then, we can say that the null values, as far as possible, are minimized in the entire database schema.

So, *null* values actually represent missing or unknown values and it does not make sense. So, the lesser number of null values it is the better it is. And then, there should not be any

modification anomalies. So, this is one important part of it. So this is, so we will go over this in a little bit more detail the modification anomaly, but this is the informal way of answering what it is.

So, what we will do is we will try to tackle this modification anomaly part first. So, this is .... modification anomaly ... and, let me explain what it means. So, for example, let us just start with a particular schema, let us say the schema is employee id, then there is an employee name, then there is a project id and a project name. So, essentially the idea of this schema is that there is an employee with a particular name, who works in a particular project id with a name of project name.

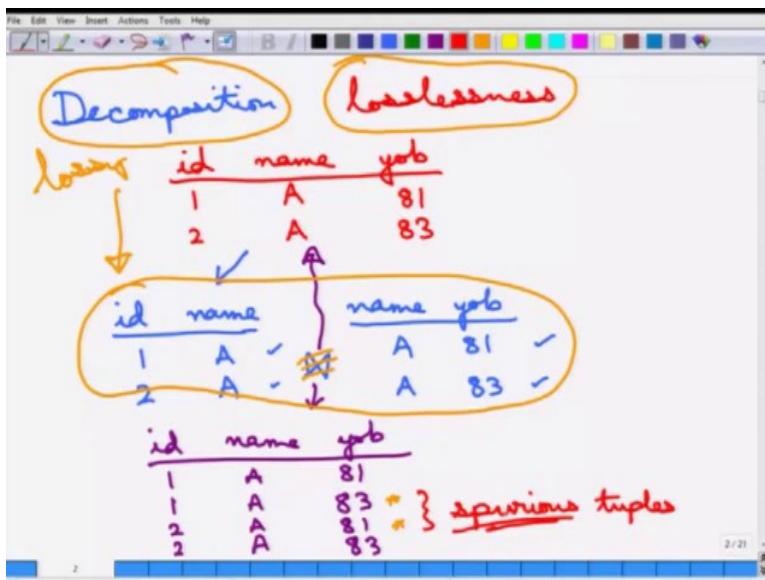
Now, is this a good schema for an employee and project. It is not, because of the following reasons. So, the first is the update anomaly. So, what happens is suppose the project name of a particular project is changed. That means, every employee id, so this is a tuple right, so, every employee id, which was in the project will go and change its corresponding project name attribute, which is a lot of changes. So, although only one piece of information the project name is changed, there are lots of changes in the database. This is not a good way of doing it.

Then, the next one is the insertion anomaly, so the first one was update anomaly, this one is an insertion anomaly. So, now, as soon as an employee is inserted into the relation, the employee must have a corresponding project, otherwise this column becomes null and as we said nulls are not really preferred.

The other way around is also true, whenever a project is introduced, it must have some corresponding employees, otherwise this column becomes null. And, now, it may not be that whenever a project is opened there are employees already assigned or it may not be that when an employee comes to an organization, he or she is already assigned to some project. So, these are some insertion anomalies that will happen.

And then, of course, there is the third one, which is a deletion anomaly. Now, suppose a particular project is being tried to be deleted, now as soon as a project is deleted, this for ... the employee which were in that project, these two corresponding fields become null and it may happen that the deletion algorithm will try to get rid of this particular tuple. So, that even the employee id and employee name may be deleted. So, that, so these are the problems with the modification anomaly.

(Refer Slide Time: 04:14)



Then, the next kind of thing, that we will handle is the decomposition. When a schema is decomposed or when a relation is decomposed, what are the issues that one can face. So, whenever there is decomposition, the important property in the decomposition is something called the losslessness. So, I will explain what is this with an example. Suppose, here is one particular schema, so what does this mean, is that there is a particular employee with a id, suppose id 1, whose name is A and who had born in, let us say, 81.

And then, there is another person id with 2 with the same name apparently and he was born in year 83. Now, what may happen is that the name clash. Now, suppose we decompose this relation into two relations, which is id, name and name, year of birth. So, what will happen is that when we decompose, this is what the decomposition will say. Note that, up to this point it seems that this decomposition is correct, because the information are all correct.

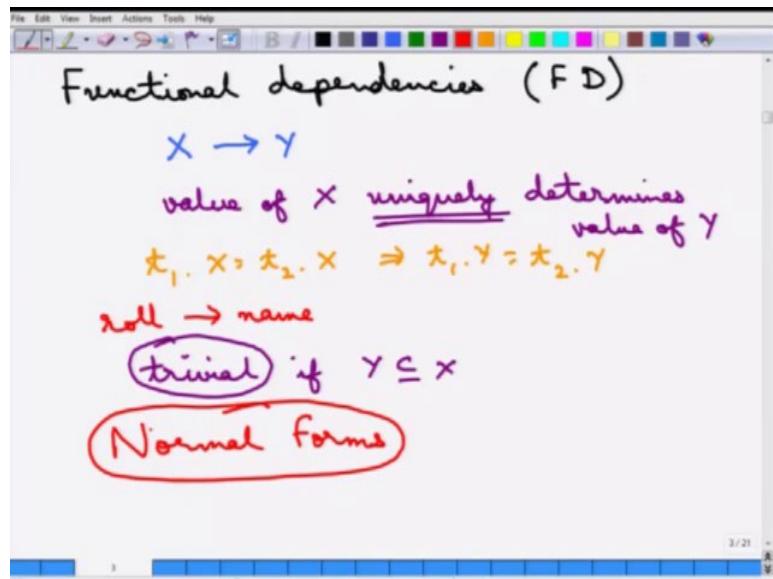
So, 1, A is actually an employee, so is 2, A and for then employee A, whose name is A there is a year of birth 81 as well as year of birth 83. But, the problem is the decomposition must satisfy that when they are joined they should give back the original table. However, the join produced this wrong information in this case. So, the join produces the following thing. This is again id, name and yob. But, it produces, even if you do a natural join, I am assuming a natural join with name, so, it produces (1, A, 81). It also produces (1, A, 83). It similarly produces (2, A, 81) and (2, A, 83).

Now, you can see that there are two spurious tuples. These are spurious tuples, (1, A, 83) are

(2, A, 81); these are spurious tuples. These have resulted because the decomposition into these tuples was not done correctly and, so the join is not correct. So, this decomposition is a lossy decomposition, because this decomposition loses certain information.

So, it essentially violates this very important property of losslessness. So, we would want the database to be such that the decomposition are lossless. So, these two things are called spurious tuples. So, we must try for a design, where there are no spurious tuples. This is important, this is called spurious tuples. So, the normalization theory, it actually tries to say, how to design a good database in a formal manner. So, it tries to handle these problems of modification anomaly, the lossless decomposition etc.

(Refer Slide Time: 07:18)



For that, before we go into the normalization theory, we require the concept of something called *functional dependencies*. So, we will define functional dependencies. That will be used in all the definitions of normalization or we will sometimes use the abbreviation FD. So, a functional dependency is a constraint that can be derived from the relation itself. So, we say  $X$  functionally determines  $Y$ , by the way,  $X$  and  $Y$  are sets of attributes in a particular relation,  $X \rightarrow Y$ , this is the notation. Value of  $X$  uniquely determines the value of  $Y$ .

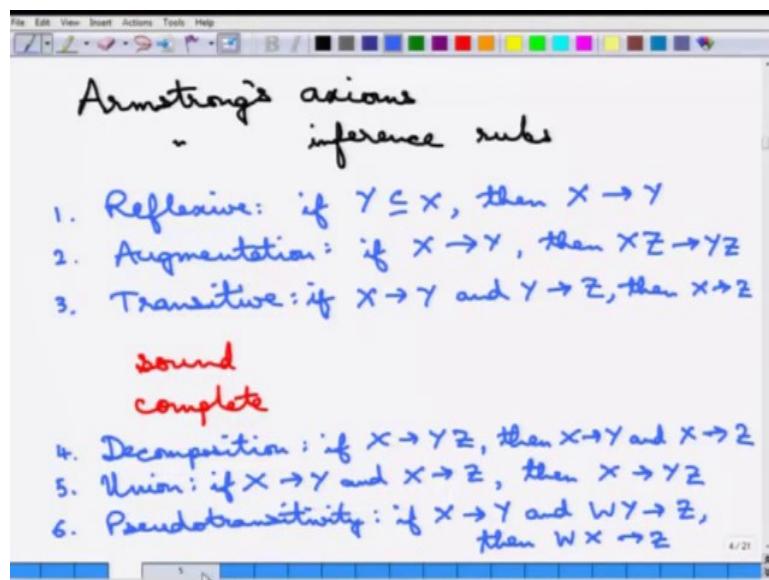
And, you can see where the definition is coming from. This is essentially  $X$  is ... so suppose  $X$  is the candidate key or the super key of the relation, then the unique value of  $X$  determines the entire tuple. So,  $Y$  can be any other attribute. So, here is what it means, once more, the functional decomposition is that, if we say,  $X$  functionally determines  $Y$ , that means, if you

know the value of X the value of Y is fixed and it's a unique value.

So, again the other formal way of saying is that, if for two tuples  $t_1.X = t_2.X$ , because this the X value uniquely determines, this implies the Y value of  $t_1$  should be equal to the Y value of  $t_2$ . Note that, it is of course, not the other way around. An example in a student database may be roll number, so it functionally determines the name. So, if one knows the roll number of a student, the name is uniquely determined, that's the point.

And a functional FD is called trivial, it is called trivial, if Y is a subset of X. Because if X is a unique name of course Y is unique, so it is a subset, this is called a trivial functional dependency. This is an important part. And as we said, a candidate key functionally determines every other attributes of the relation. So, it is the thing. So, the functional dependencies and the keys they together define, what are called normal forms of the database. So, this normal forms are, what we will be going over in more detail next. So, these are normal forms, so this functional dependency and the keys depend the ... functional dependencies.

(Refer Slide Time: 09:40)



So, before that, we go over certain axioms about the functional dependency, These are called *Armstrong's axioms*. So, note that these are axioms, so this cannot be proved etc. These are called ... sometimes called *Armstrong axioms* or *Armstrong's inference rules*. And, there are three such axioms.

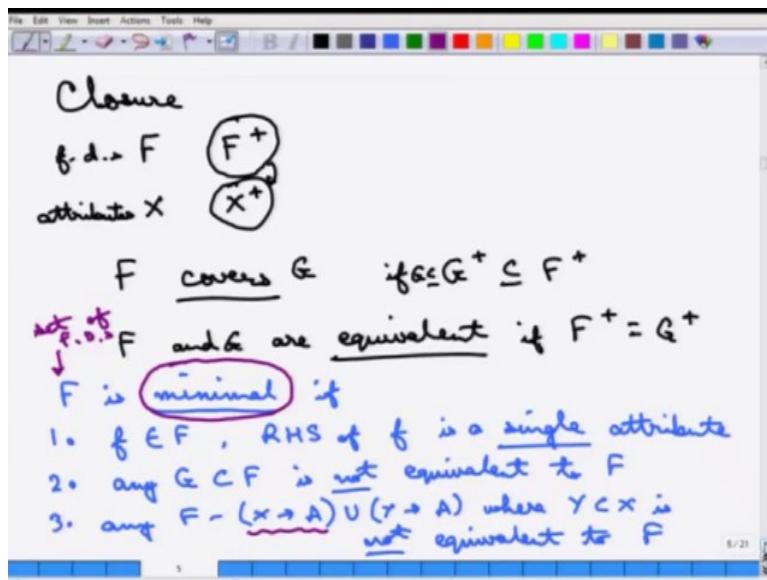
The first one is the reflexive, it essentially says that, the definition of the trivial thing if Y is a

subset of  $X$ , then  $X$  functionally determines  $Y$ . if  $Y \subseteq X$ , then  $X \rightarrow Y$ . The next one is called augmentation. So, if  $X$  functionally determines  $Y$ , then  $XZ$  functionally determines  $YZ$ . if  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$ . So, essentially  $X$  is augmented with another set of attribute  $Z$ , if the left side is augmented, then the right side can be augmented and this is not very difficult to understand why. And, the third one is called transitive. So, if  $X$  determines  $Y$  and  $Y$  determines  $Z$ , then  $X$  determines  $Z$ . if  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$ . Again, this is not very hard to understand. So, if we know the value of  $X$ , the value of  $Y$  is unique. And if you know that unique value of  $Y$ , again the value of  $Z$  is unique. So, you can simply say that knowing  $X$  will determine the unique value of  $Z$ .

Now, these rules are *sound* and *complete*, in the sense that, any other rule that can be derived from it will also hold, and complete meaning no other rule can be outside this. So, every other rule can be derived from all of this from ... one or more applications, so each one or more of this.

There are some other rules, which are useful, but are not actually per se needed, because they can be inferred from the above three rules. But nevertheless, they are very useful. First one is called the decomposition. So, if  $X$  determines  $YZ$ , by the way  $YZ$  means it is an attribute set, which is formed of  $Y$  and  $Z$ , then  $X$  determines  $Y$  and  $X$  determines  $Z$ . if  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$ . So, if  $X$  determines the values of  $Y$  and  $Z$  both then of course,  $X$  determines  $Y$  and of course  $X$  determines  $Z$ , individually. Fifth one is called union, which is, if  $X$  determines  $Y$  and  $X$  determines  $Z$ , then  $X$  determines  $YZ$ . if  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$ . Again this is easy to understand. Knowing  $X$ , if one knows the unique value of  $Y$  and knowing  $X$ , if one knows the unique value of  $Z$ , then knowing  $X$ , one knows the unique value of combination of  $Y$  and  $Z$ . It's essentially a combination, meaning it is just a concatenation of the attributes. And the last one is pseudo transitivity. If  $X$  determines  $Y$  and  $WY$  determines  $Z$ , then one can say  $WX$  determines  $Z$  as well. if  $X \rightarrow Y$  and  $WY \rightarrow Z$ , then  $WX \rightarrow Z$ . Again, it is not very hard to determine, because  $Y$  ... there is a unique value of  $Y$  for each unique value of  $X$  and, so if  $Y$  is replaced by  $X$  the same functional dependency goes through. So, that's the idea about the functional dependency and their rules, the Armstrong axioms etc.

(Refer Slide Time: 13:43)



Then, using this one can define a closure of a set of functional dependencies. So, if  $F$  is a set of functional dependencies, the  $F^+$  is called the closure of it. So, given  $F$ , all the functional dependency rules that can be derived from  $F$ , forms  $F^+$ . So,  $F^+$  is the closure of it. So, similarly closure of a set of attributes of  $X$  with respect to this ... so this is a functional dependencies, and these are attributes, if  $X$  is the set of attributes with respect to  $F$ , then the set  $X^+$  is the closure. If  $X$  is the set of attributes that is derived from  $F$ , then,  $X^+$  is the set of attributes that is derived from the closure of  $F$ , which is  $F^+$ . So, again, this is assuming this.

Then, there is another definition, which is called covers. So,  $F$  covers  $G$ , so, a set of functional dependencies  $F$  covers a set of functional dependencies  $G$ , if everything in  $G$  can be inferred from  $F$ . So, which essentially means  $G^+$  is a subset of  $F^+$ . So, if everything in  $G$  can be inferred from  $F$ , so what is the everything can be inferred from  $F$ , which is  $F^+$  and then  $G$  is of course, part of  $G^+$ , so, if  $G$  is part of  $F^+$  as well, so  $G$  can be determined from this.

And,  $F$  and  $G$  are equivalent, if the closure of them are the same. So, that means, knowing  $F$  is same as knowing  $G$ , because the closure ... so all the functional dependencies, that can be derived from  $F$  is the same as all the functional dependencies derived from  $G$  and it is exactly the same it is not a subset etc. So, knowing  $F$  is essentially knowing  $G$ . So, these are equivalent. It can be also said that,  $F$  and  $G$  are equivalent if  $F$  covers  $G$  and  $G$  covers  $F$ .

Using this, there is a definition called  $F$  is minimal, this is called  $F$  is minimal, so there is a minimal set of things, if, this is the definition is that, every ... so every FD in  $F$  has a single

attribute in this thing. So, for  $F$ , which is a FD in  $F$ , the RHS of  $F$ , so, the right hand side of  $F$  consists of a single attribute, is a single attribute. So, that's a very important condition. This is minimal, because we don't have unnecessary things in the right side.

Then, any  $G$ , which is proper subset of  $F$ , is not equivalent to  $F$ . That is why this is, I mean ... it must minimal meaning you cannot get rid of some functional dependency and get back the same thing. This is  $F$ . And the last one is any  $F$  minus  $X$  union  $Y$ , where  $Y$  is a proper subset of  $X$ ,  $F - (X \rightarrow A) \cup (Y \rightarrow A)$ , where  $Y \subset X$ , is not equivalent to  $F$ . So, what does the third rule mean? So, this is 1 2 3. The third rule means that, let us consider  $F$  and let us check out one functional dependency of the form  $X$  determines  $A$ . Now, let us take that out and add another rule which is  $Y$  determines  $A$ , where  $Y$  is the subset of  $X$ . So, essentially  $X$  is being that this rule  $X$  determines  $A$  is being reduced to  $Y$  determines  $A$ . Then, it is not equivalent any further. So, the left sides are also in some sense minimal. So, because reducing something from the left side of a rule is ... doesn't produce the same set of functional dependencies.

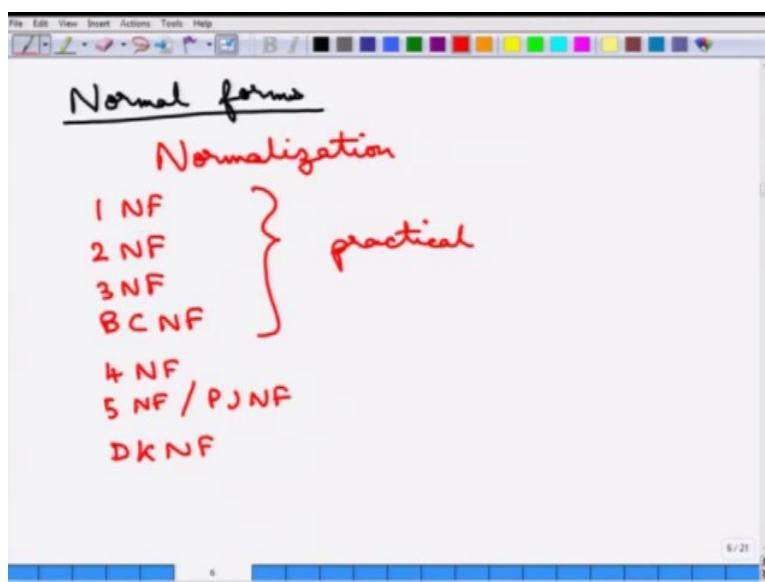
So, this is the definition of when a set of functional dependencies is called minimal. By the way, just remember that this is a set of functional dependencies, not of course, a single functional dependency. Now, using all of these, so we can define the normal forms. And just one important thing is that every set of functional dependencies has at least one minimal set of functional dependencies. So, next we will go over something called the normal forms.

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

**Lecture - 13**  
**Normalization Theory: 1NF and 2NF**

So, we will start off with these normal forms.

(Refer Slide Time: 00:12)

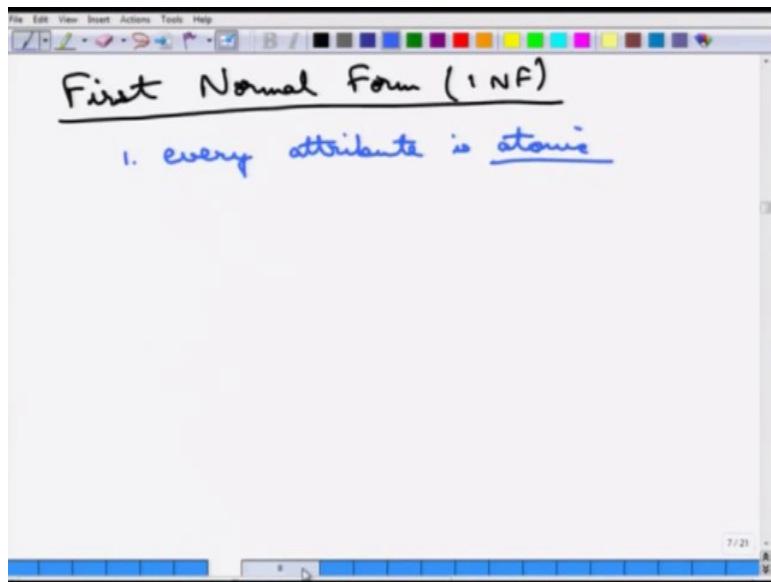


So, the process of ... given a relation, given a schema of a database, so that means multiple relations etc., the process of actually designing what the schemas are and in general taking all the attributes of all the entities of the database together and breaking it up into smaller relations is called normalization. So, that process is called normalization. And this whole theory of how to normalize etc is called the normalization theory. And keys and FDs, as we told earlier, determines which normal form a relation is in.

So, there are different normal forms, there is this 1NF or 1st Normal Form, then 2NF or 2nd normal form, then this is 3NF, third normal form, there is something called a BCNF, Boyce-Codd Normal Form. Up to this, is something which is mostly practical and database designers tries to achieve up to this. Then there are some higher order normal forms, which are also useful in some cases but, mostly they are not attempted which are called 4th normal form, 5th normal form, which is also sometimes called project join normal form. And finally, there is a domain key normal form (DKNF), which is the theoretical maximum that one can reach. So,

we will go over each of these next.

(Refer Slide Time: 01:40)

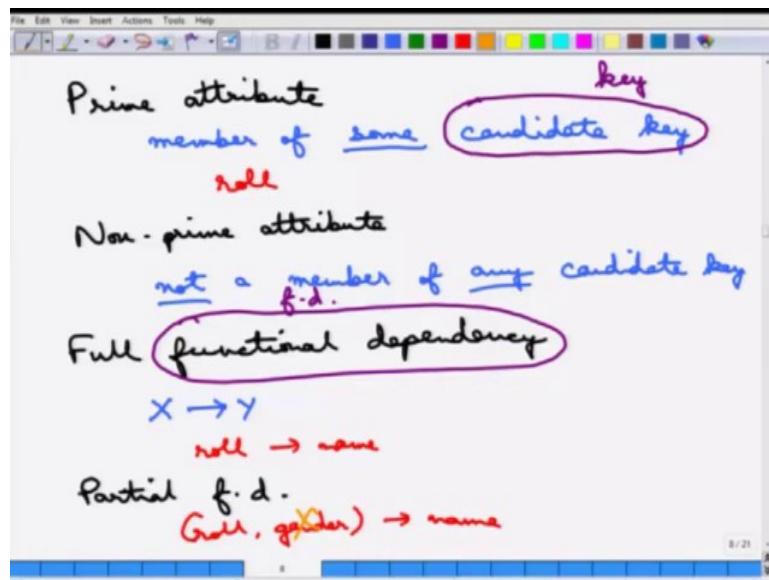


So, we start off with the 1st normal form or the 1NF. The 1st normal form which is the 1NF. So a relation is in 1NF if that following condition holds, is that every attribute is atomic. As soon as this is achieved, then the relation is said to be in the 1st normal form. And this seems to be very trivial but, in some cases we can see that a relation may not be in 1st normal form. For example, we saw that ... an example of a non atomic attribute earlier, which is the name. A name can be, in most cases, can be broken down into a first name and a last name.

So, if a relation contains name, it may not be considered to be in 1NF. Now, the point of 1NF is that there is no way to actually argue about it. Because, one can say that together this make a name and whether it makes sense to break it up for the particular application, it's not clear. So, generally when we study normalization theory, we will just assume that relations to be in normal form. So, again ... we will in the following couple of minutes etc, we will see examples of relations, there we will use the name and we will assume that the name is an atomic attribute.

So, we will just assume that things are in 1st normal form unless so much said. So, this is one thing to remember, that everything is in 1st normal form to start with. If it is not in 1st normal form, you can always break it down into this first name and last name etc. So, it is not a very hard thing to make a relation into 1st normal form. Alright.

(Refer Slide Time: 03:33)



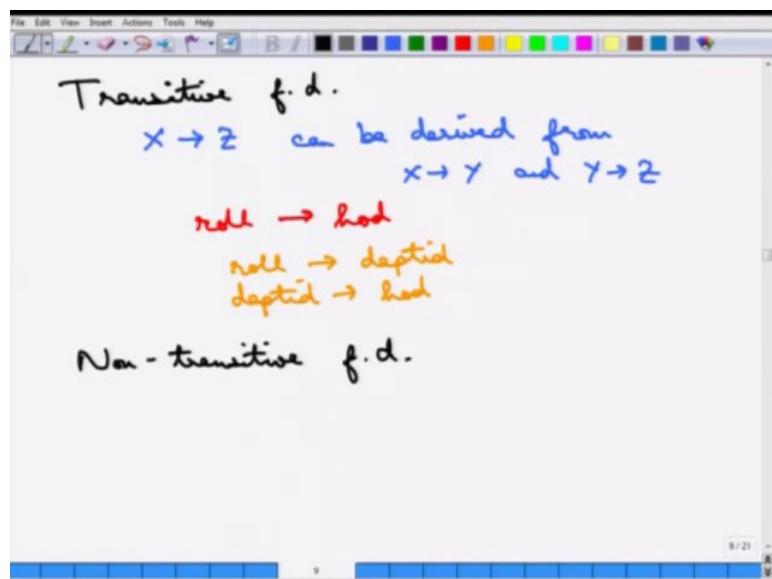
Next, we move on to something that is required for the other normal forms, first is called a prime attribute. So, this is ... a prime attribute is something it so ... an attribute that is a member of some candidate key is called a prime attribute. This is important it is some candidate key. So, an example may be the roll number of a student, so roll is a member of some candidate key. So, it is a prime attribute and analogously one can define the non prime attribute. So, any attribute that is not prime is a non prime attribute.

So, the equivalent definition, it is not a member of any candidate key. So, just to ensure that this is good, not a member of any candidate key. So, let me also use this notation, so, whenever we talk about candidate key from henceforth, I will just use the word key to determine that it's a ... so, if I do not mention the context etc, if I just mention the word key then it actually means a candidate key. So, prime attribute and non prime attribute should be easy to understand. Then the next definition that we will require, is something called the full functional dependency.

So, a full functional dependency will be defined, but again we will use instead of a functional dependency I may write down as FD, which is the same as same functional dependency. A full functional dependency is something if the functional dependency does not hold when any attribute from the X is removed. So, if X goes to Y, if this is the dependency, this FD is called full functional dependency, if X cannot be reduced any further, and it is a partial dependency otherwise.

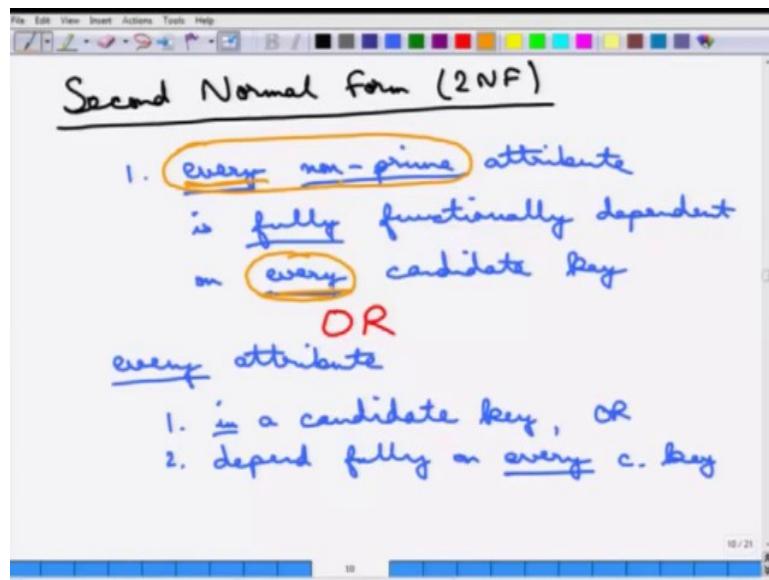
So, an example of full functional dependency may be roll determines name. I mean of course, if the left side is only one then nothing can be done about it, it is always full. But, we will see examples of some more interesting things. And it is a partial functional dependency, if the left side is reduced and it is still a functional dependency. And here is one example, is that if you say roll, gender together determines name, then of course, this is a partial functional dependency. Because, one can get rid of the gender and the still functional dependency holds, so that is a partial functional dependency.

(Refer Slide Time: 06:06)



Then, we will talk about something called a transitive functional dependency. So, suppose X determines Z, if this can be derived from two functional dependencies, can be derived from X goes to Y and Y determines Z. An example of this is, suppose, the roll number of a student to the head of the department of that student. Now, this is a transitive functional dependency, because the roll number essentially can say what is the department of that student and the department id can say, who is the head of the department of ... that department. So this is a transitive functional dependency. And of course, it is not transitive otherwise. So if FD X to Z cannot be broken down, X to Y and Y to Z, then it is called a non transitive functional dependency.

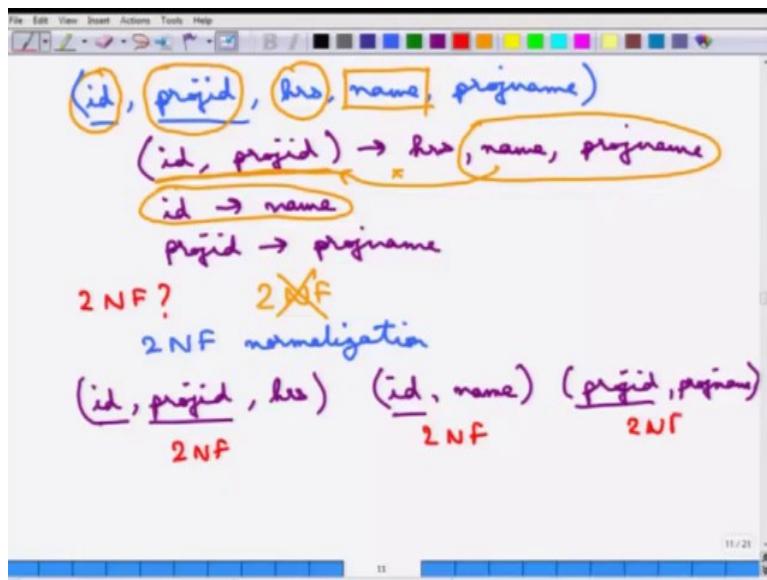
(Refer Slide Time: 07:15)



So, having this definition handy with us, we will next define what is called a 2nd Normal Form or 2NF. So, a relation is said to be in 2nd normal form, if every non prime attribute is fully functionally dependent on every candidate key. Then there is an alternative definition, or, every attribute must be in a candidate key or depend fully on every candidate key. So, these two are equivalent definitions as one can understand, let us take the second definition.

The second definition says that every attribute is in a candidate key, that means, it is a prime attribute. If not, that means, it is a non prime attribute, then it is fully functionally dependent on every candidate key. So, this is an important part, it is that this is every candidate key and also one while testing this has to be every non prime attribute, these two are important things this is every here and every there as well. So, 2nd normal form, what does it mean? Okay, so, here is an example.

(Refer Slide Time: 09:18)



Suppose, we consider the following schema: id, proj-id, hours, name, project-name. So, this is the schema that we will consider and we will try to argue whether this is in 2nd normal form or not. Now, one thing one must remember is that to determine whether a relation is in the 2nd normal form or not, the functional dependencies must be specified. So, this is part of the question. So, one cannot answer whether what type of relation it is in, whether in which normal form it is unless the functional determines .... functional dependencies are given.

So, there are two ways of giving the functional dependencies, first is, generally what is being done is the primary or the candidate keys are underlined. So, this essentially means that this translates to the following functional dependency that id, project id together is the candidate key. So, this determines everything else, so this determines, for example, hours, then this determines name and project name, etc. But in addition there may be other functional dependencies that are specified. For example, one may say that id determines name and project id determines project name.

So, these are the three functional dependencies that is given and now, one can see that essentially this is redundant, because if id determines name of course, id, project id determines the name as well. So, the question is this relation in 2NF? That is the question that we will try to understand. How do we go about answering that question? So, we test each attribute and see whether it satisfies the condition of the 2NF.

So, first of all let us check up this attribute. Now this is in a candidate key, so this is not a non

prime attribute. So, nothing to be done it is already satisfying. Project id, project id is again in a candidate key, so that is fine. Now, let us say take hours, so hours does it depend fully on every candidate key? So, what is the only candidate key? The only candidate key here is id and project id. So, hours dependent completely on id and project id. That means, that neither id by itself nor project id by itself determines hours.

Now, let us consider name. Now what happens with name is, name is a non prime attribute and it is not determined fully by id and project id, because there is this partial id to name. So, that means, name is not fully determined. So, that means, name fails the 2NF test and the answer is that this is the relationship is not in 2NF. Fine. Now, one may argue that, that is fine, but how is this 2NF useful? So, 2NF essentially says that there is some problem with the way the scheme determined.

What is the problem? Now, you see that name is determined only by id. So, which means that the id and project id together, the key is redundant, it does not require that. So, another way of saying is that suppose, the name of a person is changed then ... so, corresponding to the id the name is changed. Now the person is working in many different projects, supposedly and all of those tuples need to be modified. So, that is the problem and why are those ... all of those tuples need to be modified?

Because, just by changing the name, one only has to argue about the id or vice versa. But, the project id is something that is also part of the candidate key of this relation. So, the project id is in a sense redundant when we are arguing about the name. And why is it redundant? Because, name is not fully dependent on id, project id that is why 2NF is useful and that is why it makes sense to determine whether a relationship is in 2NF or not.

Now, suppose we want to make this relationship into 2NF. So, that this process is called a 2NF normalization. So, we understand that above schema is not in 2NF. So, how do we 2NF normalize it? The way to do it, 2NF normalization is that we break it up into the following tables. So, the first table contains id, project id and hours, this is the first table. The second table is ... so this key for the thing is id and project id.

So, second one is id and name. And the third one is, project id and project name. One can intuitively see that this is a better design. The reason is what we have been arguing about. So, name depends only on id, so why not break it up into a separate table. Similarly, project name depends only on project id, so why not break it up into a separate table. And now what

happens is that, if we go about to the previous example, if one changes the name there is only one place that this whole information needs to be changed in the database, the tuple corresponding to that particular id to name.

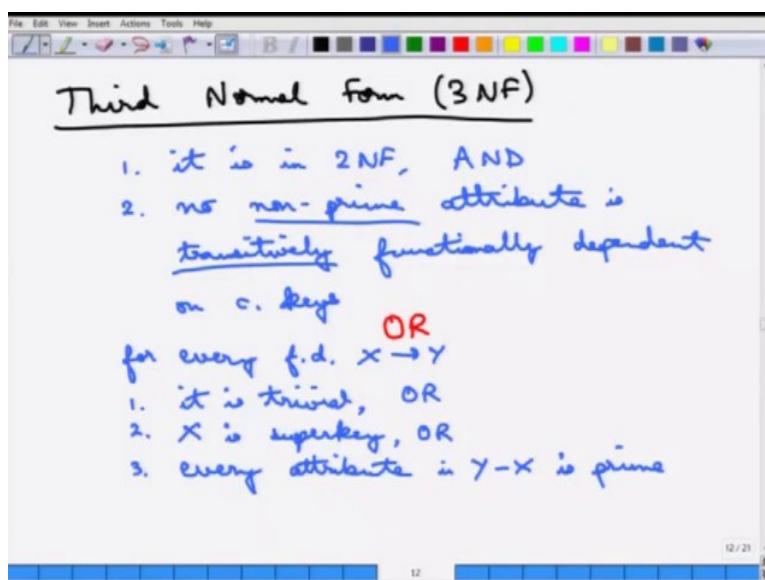
Similarly, if the name of a project is changed there is only one place that it needs to be touched. The other tables are not touched. So, there is no modification anomaly etc. Formally, we can also test if this is a better design than the other one. Because, now we can test whether each of these three relations are in 2NF and if you go about testing it, you will find that all of them are actually in 2NF. So, this is in 2NF, this is also in 2NF and this is also in 2NF. So, we have argued, both informally and formally, that the design where you break this up into these three tables is a better design.

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

**Lecture - 14**  
**Normalization Theory: 3NF**

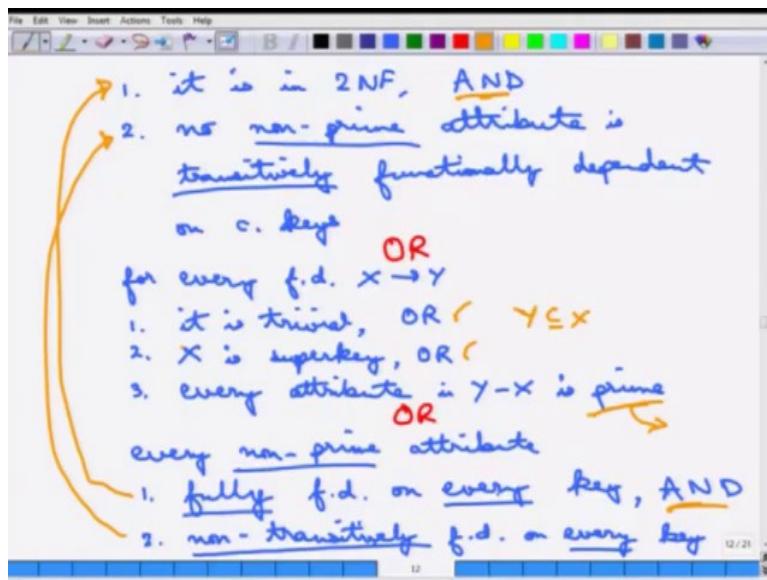
So, we will next go over the 3rd normal form or the 3NF, more popularly known as the 3NF.

(Refer Slide Time: 00:11)



So, a relation is in 3rd normal form, if first of all it is in 2NF and no non-prime attribute is transitively, functionally dependent on candidate keys. This is one way of defining it. There is an alternative definition ...or ... let me write down that alternative definition. For every functional dependency  $X \rightarrow Y$ , either it is trivial, Or, X is super key. So, that means, X functionally determines every Y. Or, every attribute in Y minus X is prime. There is another way of alternatively defining this.

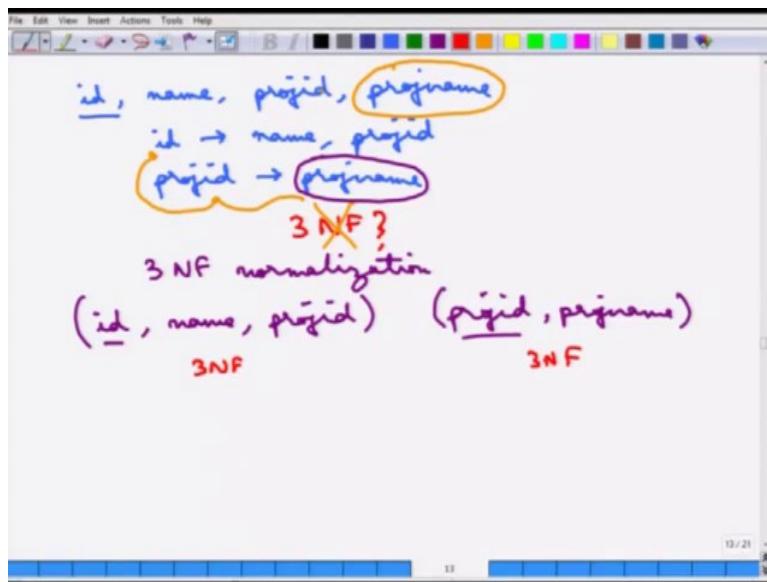
(Refer Slide Time: 02:01)



So, or, every non prime attribute is fully functionally dependent on every key and non-transitively functionally dependent on every key. So, let us start from the third definition, probably that is easiest to understand. So, we take every non prime attribute and it must be both fully functionally dependent and non-transitively functionally dependent on every key. Fine. So, how is this equivalent, for example, to the first definition. This is equivalent to this. So, this is essentially the definition of 2NF, and this is equivalent to this definition.

So, every non prime attribute is functionally transitively dependent on this. So, at least the first and third definitions are equivalent and let us see now the second definition. How is it equivalent to the other definition? So, see for every functionally dependent thing, if it is trivial so, that means, Y is a subset of X, that means, Y is not non prime etc., or X is a super key, that means X is part of this prime attribute. So, the things is ... this part of the prime attribute or every attribute in Y minus X is prime, that means, every attribute in Y minus X also functionally determines everything else. This, remember that these are all or and this is and, this is also and. So, these are the three equivalent definitions of 3rd normal form and let us now take an example.

(Refer Slide Time: 04:03)



So, again we will consider this following kind of definition. So, it's id, name, project id, project name and here we are saying id is the key. So, the functional dependencies are essentially id, of course, determines everything else, but it also determines name and project id and then project id essentially determines project name. Is this in 3NF? So, it is not in 3NF, because of the following reason. So, let us identify one at a time. So, id is a prime attribute, so nothing to be done. Name, name is fully determined on ... by the id.

So, and it is not transitively dependent on any other thing. So, name is also fine. So, is project id. However, project name has a problem. Project name, it is a non prime attribute and it transitively determines on id. So, because project name is determined by project id which in turn is determined by id. So, project name, this fails and the entire question of whether this is 3NF or not is answered in the negative. This is not in 3NF. Now, we would like to again do this process of 3NF normalization and once we do that, so this is 3NF normalization.

And once we do that, we will essentially try to isolate the offending attribute. Offending attribute here is the project name. So, we will try to isolate it. So, we will break this relation into two parts. First is the id, name and project id. So, this is the first relation. And, the second relation is project id with project name. So, this is both are in 3NF. So, let me write down this is the keys and this is the only functional dependencies that are available.

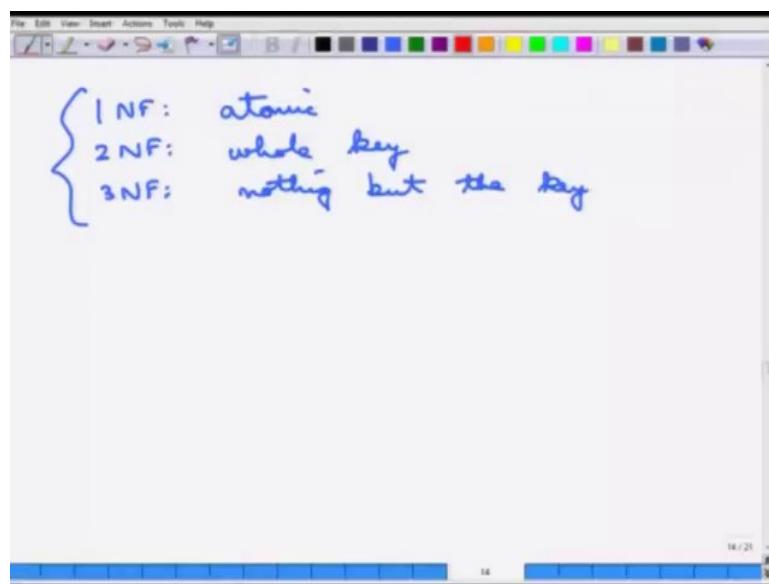
So, this is again in 3NF one can determine. So, this is in 3NF normalization. So, what again is the problem with 3NF? Why it is important to break a relation into 3NF? So, let us consider

this example ... and the project name is determined on project id which in turn is determined by id. So, once more if the project id changes or if the id changes, project name is affected unnecessarily. So, if project name affects this id unnecessarily, it does not make sense.

So, if project name is changed, the effect is carried in a ripple manner to the id. So, corresponding to every name who works in that ... so every employee who works in that project name gets affected. So, all those tuples now need to be modified. On the other hand, when it is broken down into this 3NF normalized forms, the project name is isolated from this.

So, the first table is not touched at all and in the second table there is only one change that needs to be done, which is just the project name is changed. So, this is a much better design as we can see. So, we have looked at these three normalized forms and one can informally summarize them in the following manner.

(Refer Slide Time: 07:15)

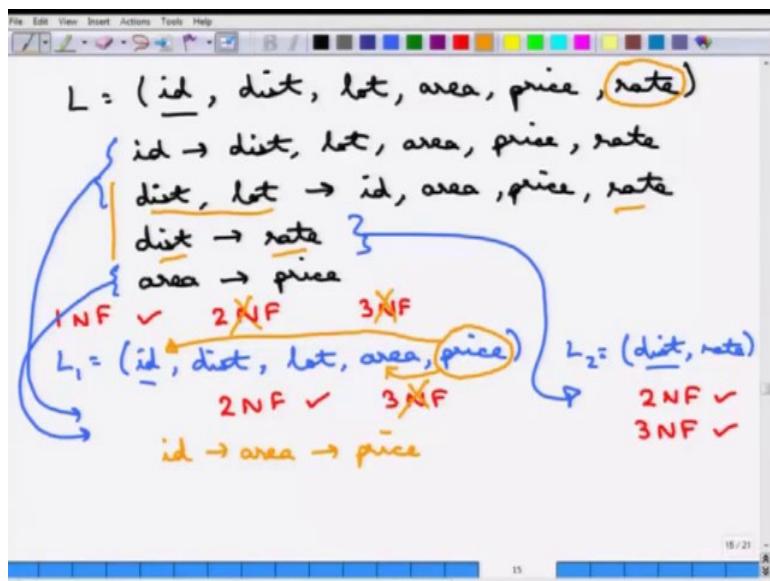


So, 1NF is that all attributes are atomic, correct? So, that is all that can say. In 2NF, all attributes must depend on the whole keys. So, that is why it's called a fully functionally dependent. And 3 NF, all attributes must depend on nothing but, the key or nothing. So, it is of course, in 2NF and nothing but the key. So, that means, there is no transitive dependency it must depend only on that key. So, this is of course ... I mean ... an informal way of remembering what the 1NF, 2NF and 3NF does.

So, how do we determine if a relation is in 1NF? There should not be any multi-valued attributes or nested relations etc. etc. And in the 2NF, it should fully functionally depend and in 3NF it should transitively depend. So, that is the way to do it. And how does one remedy? So, if a particular relation is not in 1NF then what one needs to be ... done? Is that, offending attribute, the multi-valued attribute, the nested attribute needs to be broken down into atomic attribute.

So, what does one do when a relation is not in 2NF? The offending key which does not fully depend on one of the candidate keys is isolated. The same thing is done for the 3 NF. The offending key which is transitively dependent on a key. So, those two functional dependencies are broken into two relations. So, that is the way for the normal forms. And let us do a little bit of example to ensure that the understanding has gone through.

(Refer Slide Time: 08:45)



So, here is an example that we will do. So, this is  $L$ , we will just try to say what is the thing. Here is an id, district, lot number, area, price and rate. And this ( $\underline{id}$ ) is the candidate key. Okay, so, what are the functional dependencies in this? Of course,  $\underline{id}$  determines everything else because  $\underline{id}$  is the key. So,  $\underline{id}$  determines ... district, lot, area and price and rate. Now, there are some other functional dependencies as well. So, district and lot if one knows the district and lot together, it can determine which id it is in, which area it is in, and the price and rate of the part of land that one talks about.

So, if one knows the district then one can know the rate of the land that is being there. And if

one knows the area in which where the land is located, one can know the price of the... know, because this essentially area time is the unique price, etc. So the first question is, is this in 1NF? The answer is yes, because we are just assuming that every attribute is atomic. So, and the next question is, is it in 2NF? The answer is no. The reason why the answer is no is that consider this attribute, let us say, rate.

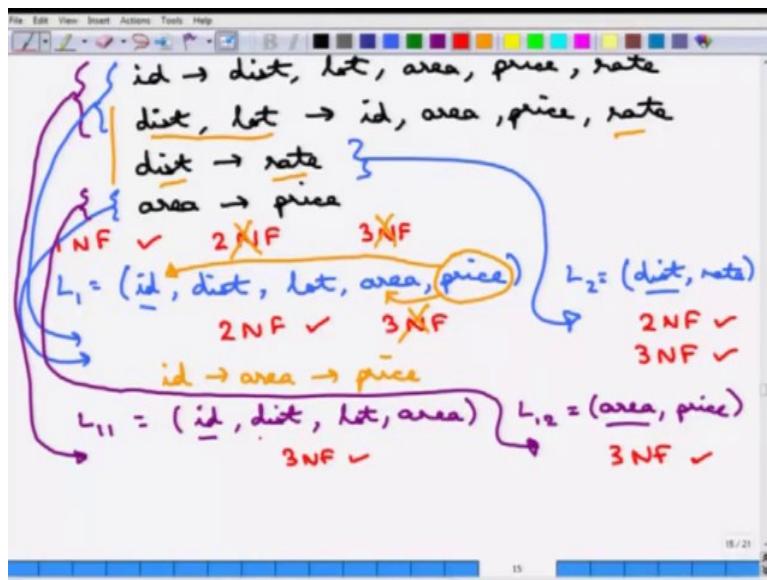
So, rate depends on district and lot and rate also depends on district. So, it is not fully dependent on district and lot. So, it violates this 2NF, so this is not in 2NF. The question then comes is it in 3 NF? Now of course, not without any testing one can say this is not in 3NF, because if something is not in 2NF it cannot be in 3NF. So, if this goes progressively. So, now, let us try to see how to do this 2NF normalization.

So, if one attempts to break it up into 2NF normalization, the following relations can be produced. So, the  $L_1$  ... so, we essentially need to get rid of all the offending keys in the 2NF testing method. So, what one does is the following this  $L_1$  is produced with the following attributes: area and price. This is id and  $L_2$  is simply district to rate. So, that is what one tries to identify and here the key is district. And the same functional dependencies flow through. So, these two flows here, this again flows here and this flows there. These are the functional dependencies.

Now, one has to test whether this is in 2NF or 3NF, etc, so  $L_1$ , is  $L_1$  is in 2NF? By the way, it is probably it's easier to argue about  $L_2$ . Is  $L_2$  in 2 NF? It is yes, I mean this is nothing to be done. This is very simple. It is in 2NF. is  $L_1$  in 2NF? It is in 2NF, because one can check that these all go through. The question is, now is  $L_2$  in 3NF?  $L_2$  is 3NF, because one very simple rule is that if there are only two attributes and one of them is the key and that is the only key, it is in any normal form that one can think of this. This rate is determined only by district, there are no other functional dependencies to start with.

What about  $L_1$  is it in 3NF? It is not, the reason why it is not is that, this attribute price is determined ... it depends on id. Because, id is of course the key. But it is also dependent on area. So, that means, id determines ... so, this is a problem, id determines area which determines price.

(Refer Slide Time: 12:50)



So, this is the offending key. So, this is not in 3NF. Once more, if one wants to break it up into 3NF, then what does one need to do, is to break this up into two more parts and the first thing is  $L_{11}$ , let us say, which is simply id, district, lot and area. And third one is  $L_{12}$  is simply area and price. So, the offending price is broken down into this thing. So, this is area and this again remains as id. So, these two functional dependencies are satisfied here. And this functional dependency is satisfied here.

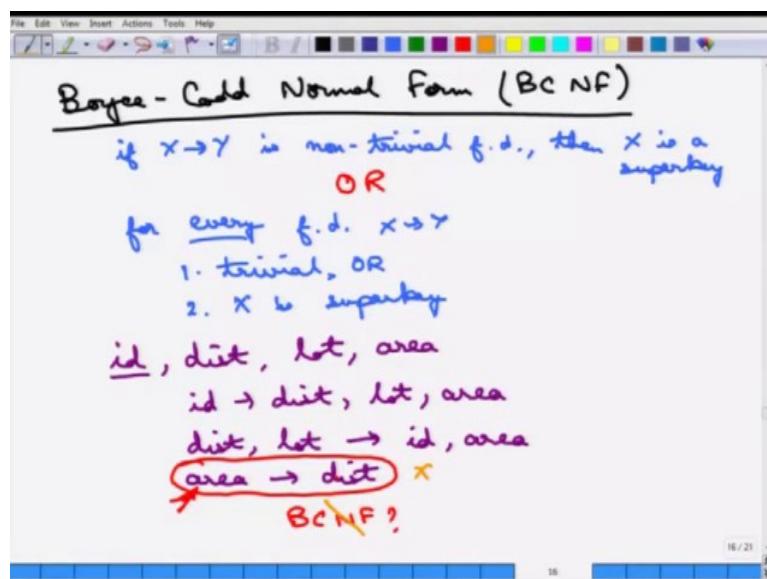
So, then the test comes is this is in 3NF? Of course, as I said, this is easy. Is this  $L_{11}$  is in 3NF? Yes, it is in 3NF as one can test, because there is only one key so far. And the district and lot together determines everything else, but that doesn't matter right now. So, this is all about 3NF, we will start on the next form later.

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

**Lecture - 15**  
**Normalization Theory: BCNF**

So, the next form is called Boyce Codd Normal or BCNF.

(Refer Slide Time: 00:13)



This is the Boyce Codd normal form. A relation is in Boyce Codd normal form if  $X \rightarrow Y$  is non-trivial FD, then X is a super key. So, for every non trivial FD, the left side must be a super key. There is an alternative definition, which is for every FD  $X \rightarrow Y$ , either it is trivial or X is super key. I mean the equivalence is very easy, this is essentially just breaking this down to that non trivial thing ... down.

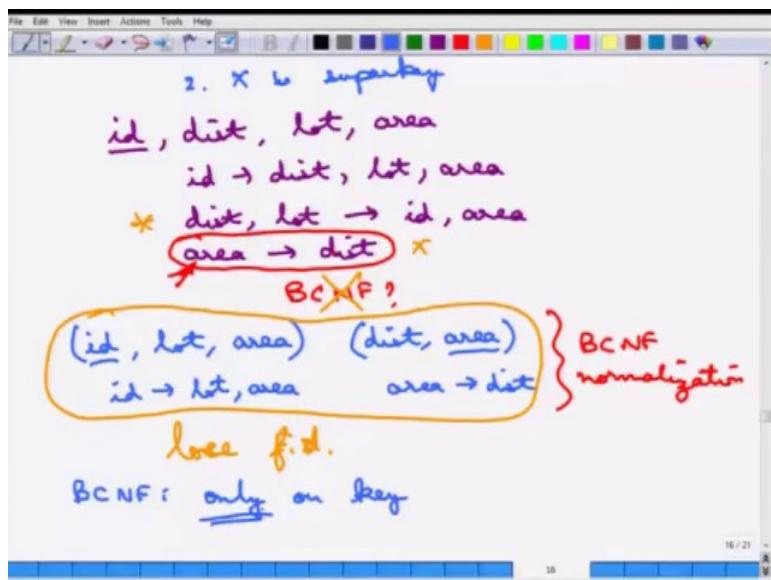
So, what does BCNF tries to do? Is to ... BCNF tries to go to a higher form than 3NF. It tries to say that whenever there is a functional dependency, it is non-trivial then the left side must be on the super key. So, it tries to say that every relation, the functional dependencies must be only on the keys, it cannot be on anything else. So, they are ... so, if there is a functional dependency  $X \rightarrow Y$ , the X must be a super key, so that is what it tries to do.

Now, the process of BCNF normalization may be actually a problem. So, before that let us consider the example that we were looking at earlier. So, this was the relation after that  $L_{11}$  that we looked about earlier. So, this is just to recollect, this was the dependency with id determining everything dist, lot and area. However, there was another dependency which was

saying distance and lot together determines id and area. And there was, let us say, there is ... this was another thing, area determines district.

So, the question is, is this in BCNF? The answer is no, because of this functional dependency. The other two functional dependencies are fine, because of this functional dependency area determines district, but area by itself is not a super key. So, this is where this fails, so this not ... this relation is not in BCNF.

(Refer Slide Time: 02:58)



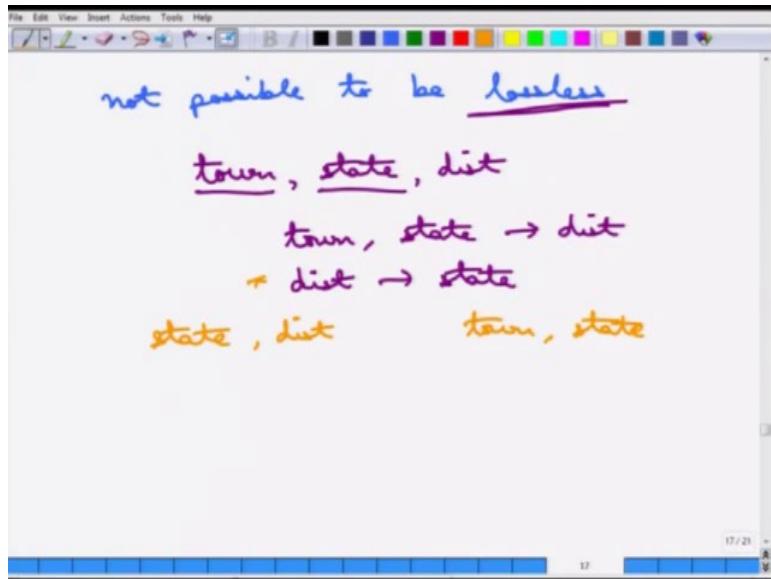
So, if one wants to normalize this BCNF, what ... one can break this down is that id, lot, area, then it can be district to area, that is it. Correct. Now, what are the functional dependencies in this? The first, here the functional dependency is id determines lot, area. And area determines district. So, very importantly after BCNF normalization, this is the process of BCNF normalization, one can see something important here, is that it can lose functional dependencies. So, there are certain functional dependencies that are lost.

So, what are the functional dependency that is lost? This functional dependency is lost. So, after the BCNF normalization is done, it is not always that one can preserve all the functional dependencies. So, this is ... can be problematic may not or whatever it, may or may not be problematic, but that is the thing. So, functional dependencies may be lost. And informally BCNF can be summarized as everything depends on only the key. So, that is the way of saying.

So, why is BCNF normalization important? Is that, if the ... area to district this actually introduces a problem. So, if one knows the area, one knows the district, but area is not a key.

So, in ... it may happen the changing area, area is changed due to something else and the district needs to be also changed. So, again, the problem comes with those anomalies that one attribute is touched and other attributes are touched unnecessarily, when it is not useful to touch it. So, the other problem with BCNF decomposition is that it may not be always possible.

(Refer Slide Time: 04:57)



So, not possible, in the sense that it may not be always possible to be lossless. So, it can lose certain, sometimes it can lose not just functional dependencies, the decomposition may not be lossless. So, for example, let us consider a relationship in the following manner. A town, state and district. So, the functional dependencies are town and state, together determines the district, And the district name determines the state.

So, suppose these are the two functional dependencies. Now of course, this is not in BCNF. So, to break it down into BCNF the rule says that the problem is with this district to state. So, let us break it down into this two relationships state, with district and town and state. So, let us isolate district and state together and this is also town and state.

(Refer Slide Time: 06:08)

town	state	dist
iit	up	east
iit	wb	mdp
prayag	up	east
prayag	wb	dinaj
kanpur	up	center
lucknow	up	west

state	dist	town	state
up	east	iit	up
wb	mdp	iit	wb
wb	dinaj	prayag	up
up	center	prayag	wb
	west		up

Alright. But, here is an example where this will fail. So, town, state and district, this is an example. So, IIT is a town in UP and suppose it is in the district east and so on and so forth. IIT is a town also in the state West Bengal, then Prayag is in UP and WB as well and Kanpur and Lucknow all those things. Now, if one breaks it down to this state, district and town state, so if one breaks it down to state, district and town, state; let us see what does one get.

So, state, district, if one breaks it down, so you get (UP, East), (West Bengal, MDP) then (West Bengal, Dinaj), (UP, center), (UP, West). And town and state if one breaks it down, this is ... and well. So, this is .... all the other six is there. The point is one can check that this is definition ... this decomposition is not lossless, because if one joins this then you do not get back the original relationship. So, this is lossy and this cannot be allowed under any circumstance.

So, this decomposition is wrong. This is not allowed. And one can also say, just to argue, that may be the BCNF decomposition rule was wrong, that there is no way to break this relationship down into any other way. For example, if one tries to break it down into town, state and town, district or state, district and town, district, none of this will actually give back the original form. So, this is this problem with BCNF decomposition. So, it is not always possible to ensure that a relationship is in BCNF. It may happen ... it may be possible it may not be possible.

(Refer Slide Time: 08:11)

course	teacher	book
db	ab	f1b
db	ab	dbm
db	sb	f1b
db	sb	dbm
nt	nm	ntb
nt	nm	nsc
nt	ab	ntb
nt	ab	nsc

There is another problem with BCNF, it is the anomaly. There is a particular anomaly that can happen with BCNF. So, for example, consider the relationship of a course and teacher and books, so what essentially means is that, so, to all these three ... together is the primary key. So, what does it mean? Is that suppose you consider a course database, it can be taught by many teachers in a particular institute and each teacher can take up many of the books.

So, a particular invocation of this thing may be in the following manner. So, let us say there is the database course which can be taught by a teacher say AB and whatever these are the name of the books, and this can be taught by whoever and so on so forth. So, let us just see what happens. So, essentially what one means is that the course can be taught by any teacher and the teacher can adopt any book and similarly, this can happen with lots of ... there can be other courses as well, which this where this can happen.

Now, the problem that will happen is that the functional dependencies, there are essentially no functional dependencies. So, this is determined to be in BCNF. So, what I mean to say is this relation is in BCNF, because there is only one functional dependency and nothing is valid. So, this is all seems to be fine, but modification anomalies are still there. So, there is a modification anomaly there, in this thing.

So, although this relation is in ... seems to be all fine from the point of view of normalization. So, it is in 1NF, 2NF, 3NF, it is in 1NF, 2NF, 3NF and BCNF, all of these things, so it seems to be all fine. There is still a modification anomaly. Why is there a modification anomaly? One can consider that if there is a new teacher that comes who can teach database. So,

suppose introduce the name of a teacher VN who can teach database.

Now, there can be two books adopted by it, so FDB and DBM. So, you require both the tuples to be inserted. So, even though there is only one piece of new information that is inserted into the database, two tuples are added. And similarly, if AB today leaves from this institution and so there is only one teacher which deleted there are two tuples are needs to be deleted and so on so forth. So, you can see that there is a problem with this relationship per se.

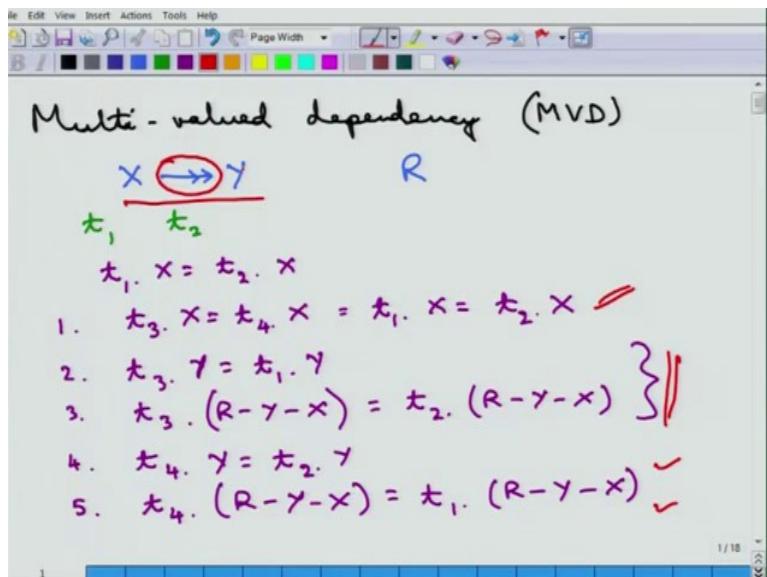
So, what it essentially means is that the functional dependencies and the keys can normalize a relation only up to a certain extent. So, it can make the design of a database good up to a certain extent. It cannot make it completely bereft of any modification anomaly. So, there can still be modification anomalies even if one covers all the normal forms that is possible with ... functional dependencies and keys. So, that will be the start of our next topic which is on multi-value dependencies.

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

**Lecture - 16**  
**Normalization Theory: MVD**

So, as we saw, up to 3NF or BCNF is practical, but it may still have certain problems. For example, in the database course example that we saw, if a teacher teaches the course on database, the books must be common to all the teachers. Now, to handle such anomalies and to go to higher normal forms, we will require the concept of what is called a multi-valued dependency.

(Refer Slide Time: 00:34)



So, a Multi Valued Dependency, it is mostly written as MVD, instead of a functional dependency, which is sometimes written as FD. So, it is a Multi Valued Dependency MVD. It is defined in the following way. So, suppose Y depends, multi value depends on X,  $Y \twoheadrightarrow X$  so this is the symbol ( $\twoheadrightarrow$ ). First of all note the symbol, this is a different from the FD symbol, because there are two arrows. And now this holds for all value or for a relation schema R, this holds if for all legal instances of R.

Now, if there is a pair of tuples  $t_1$  and  $t_2$  such that the following happens. Now suppose these are the conditions. So, suppose  $t_1.X = t_2.X$ , then there are two more other tuples, such that  $t_3.X = t_4.X$ . And those are then equal to  $t_1.X$ , which means these are also equal to  $t_2.X$ . So, all

these four tuples of the same X part. And if  $t_3.Y$  is equal to  $\text{tot}_1.Y$  and  $t_3.(R - Y - X)$ . So,  $(R - Y - X)$  is essentially all the attributes that are in the relation R, but not in Y and not in X. If this happens ... if these two conditions happen then it must be that the following two conditions happen.

So,  $t_4.Y$  must be equal to  $t_2.Y$  and  $t_4.(R - Y - X)$ , so the attribute values for  $R - Y - X$  for  $t_4$  should be equal to that for  $t_2$ . So, this is the formal definition of this. So, if these two conditions happen, when this is the case, then these two must occur. So, this is the definition of the multi-valued dependency. X, so multi-value determines Y.  $X \twoheadrightarrow Y$ . Now, an example will be the better way of understanding it, so here is what I am trying to do.

(Refer Slide Time: 03:00)

$t_1$	a	b	c
$t_2$	a	d	e
$t_3$	a	b	e ✓
$t_4$	a	d	c ✓

$X \rightarrow Y$   
 $\Rightarrow X \rightarrow (R - Y - X)$

course  $\rightarrow$  teacher  $(c, t, book)$

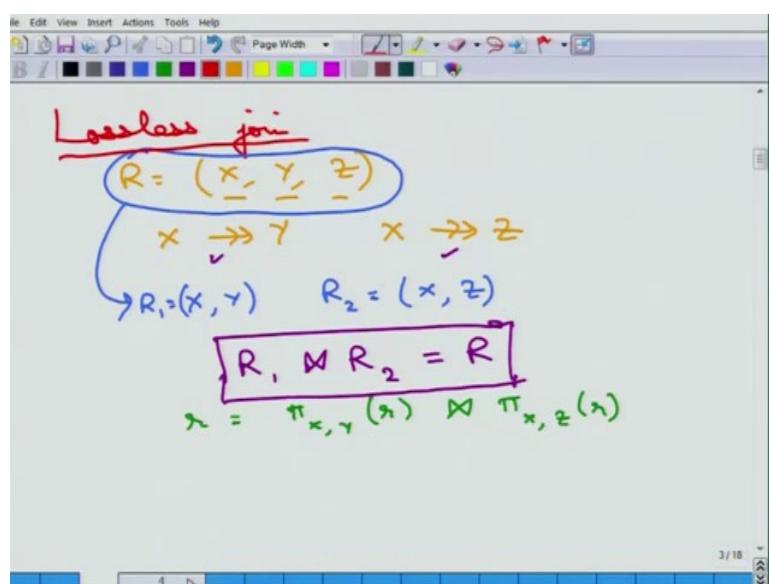
$(db, ab, fdb)$   $(db, xy, dbm)$   
 $(db, ab, dbm)$   $(db, ny, fdb)$

So, suppose this is X, this is Y and this is all the attributes that are not in X and Y. So, this is essentially  $(R - Y - X)$ . And let us now consider first tuple  $t_1$  which has value  $a$ , and this is  $b$ , this is  $c$ . These are the values. Now, the  $t_2$  also has the same X values which is  $a$  and it has got different Y and  $R - Y (- X)$ . So, let us say those are  $d$  and  $e$ . Now, if  $t_3$  has  $e$  and  $t_4$  has  $a$ , then it must happen, that this combination  $b-c$  and  $d-e$  must again take place.

So, it's essentially  $b-e$  and  $d-c$ . This ... they must be happening. So, because this is the combination and this is the combination, both these combinations must be there in the relation. And then it is called that X multi-value determines Y. Now, from the symmetry of this one can say that if  $X \twoheadrightarrow Y$ , so if there is an MVD,  $X \twoheadrightarrow Y$  then there is also the MVD that  $X \twoheadrightarrow (R - Y - X)$ , and that is obvious from the example and the definition of this.

And, in the database course that we saw, the example of the database course that we saw earlier, essentially what is happening is that the course MVD from course to teacher,  $course \twoheadrightarrow teacher$ , exists when the relation is this course, teacher and book ( $c, t, book$ ). Now, what does that mean? Is that, suppose, there is the course is this database. So, that means, that if there is a course database, which is the course and if there is a teacher, let us say,  $AB$  and if that teacher teaches the book  $FDB$ , if this happens and if there is another tuple for the same course database and then there is some other person,  $XY$ , which teaches some other books  $DBM$  book, if these two tuples happen .... So, if the teacher  $AB$  teaches the book  $FDB$  and if the teacher  $XY$  teaches the book  $DBM$ , then it must happen that in that relation the two other tuples must also exist. So,  $AB$  must be able to teach with the book  $DBM$  and  $XY$  must be able to teach with the book  $FDB$ . Now, if this ... why is this required? Intuitively, if this does not happen then there is something special about the book and that teacher. It is not about the course then, the teacher probably prefers the book more, much more and refuses to use the other books and similar things may happen. So, that is not a desirable situation and this is why this multi-valued dependency is useful. So, now, if we use this then we can move ahead and see how multi-valued dependency is actually useful for a lossless join.

(Refer Slide Time: 06:00)

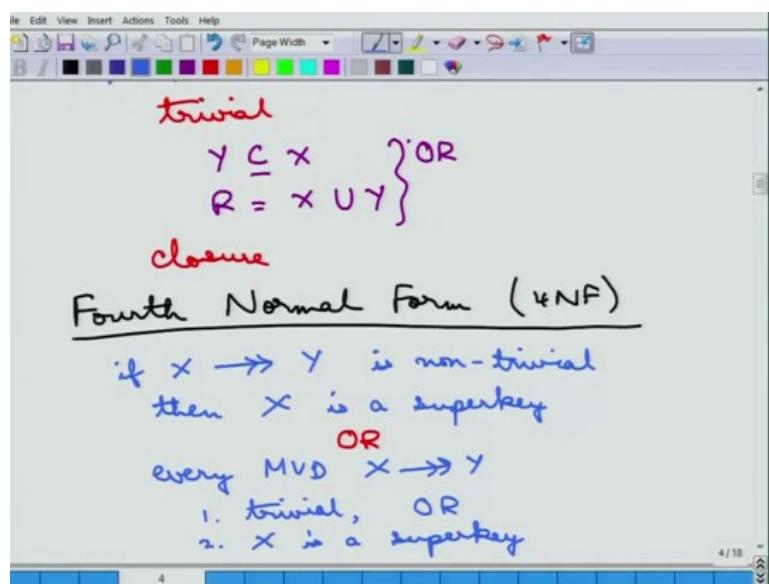


We define this concept of a lossless join and here is a thing. So, suppose there is this relation  $R = (X, Y, Z)$ . And then, if  $X \twoheadrightarrow Y$  then also  $X \twoheadrightarrow Z$  determines. Now, if this relation is now broken up into the following manner. That, this is  $X, Y$ ; so  $R_1$  is equal to your  $X, Y$ ; and  $R_2$  is equal to  $X, Z$ , let us say now. And then, we take the join of  $R_1$ , natural join of  $R_1$  with

$R_2$ , which is of course on X, this must give us back the relation R.  $R_1 \bowtie R_2 = R$ . So, that is the whole point.

So, essentially we can say write it in a different manner that R is equal to your ... So, if you project it only on X and Y, then this is simply equal to if you project it on X and Z on R.  $r = \pi_{x,y}(r) \bowtie \pi_{x,z}(r)$  So, that is the same as saying  $R_1$  must ... so this must hold. And then this satisfies your MVD. And then if this MVD and if this lossless join condition holds, this because this MVD is hold, then and R can be broken down into X, Y and X, Z which is a much better design as we saw, because, we do not need to store all the tuples of this teacher and course. We just need to store the DB is can be taught by this teacher and DB can use this book and that is good. So, this is essentially why MVD is useful, because it defines the concept of lossless join. And now, similar to all the other things, is this MVD can be used to define certain things.

(Refer Slide Time: 07:55)

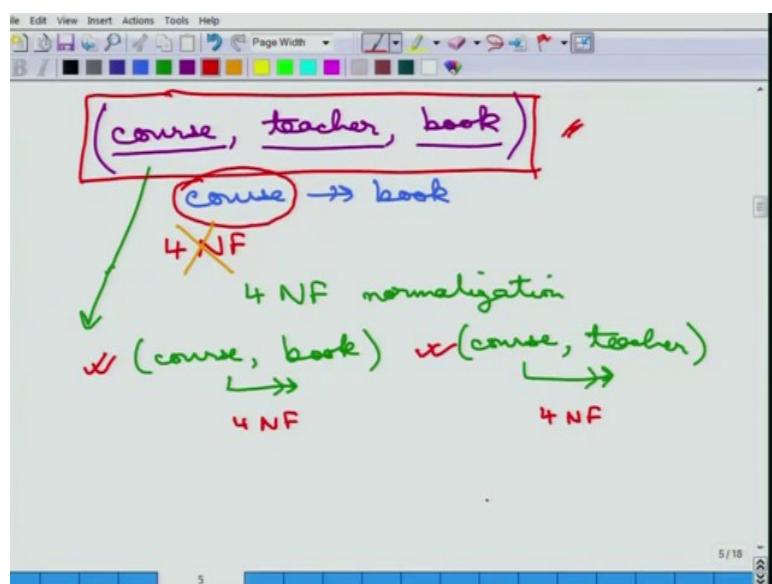


So, before that, just simple things is that  $X \rightarrow\rightarrow Y$  is called trivial, if the following things happen. Either  $Y \subseteq X$  or simply  $R = X \cup Y$ . So, I mean this is an or condition. This ... either this or this happens then this is called a trivial and any other MVD is non-trivial. Okay. Fine.

So, then analogously like the functional dependencies, the closure of a set of MVDs can be defined etc. And then there are different inference rules for MVDs, such as the complementation, augmentation, transitive, replication and coalescence.

But, using that let us define what is called the 4th normal form. So this is the 4th normal form or the 4NF. And 4NF essentially, a relation is said to be in 4NF, if there is a functional dependency .... if  $X \twoheadrightarrow Y$  is nontrivial, if this is non-trivial then, X is a super key. So, for every non-trivial thing, the left side is a superkey. Now there is an alternative definition, just as we saw the definition for other things. There is an alternative definition. And the alternative definition says that, for every MVD  $X \twoheadrightarrow Y$  the two conditions happen, either it is trivial or X is a superkey, that is a same definition and this is what the definition of the 4th normal form is. So, that is how to define the 4th normal form. And the example we have already seen, just to recap it.

(Refer Slide Time: 10:03)



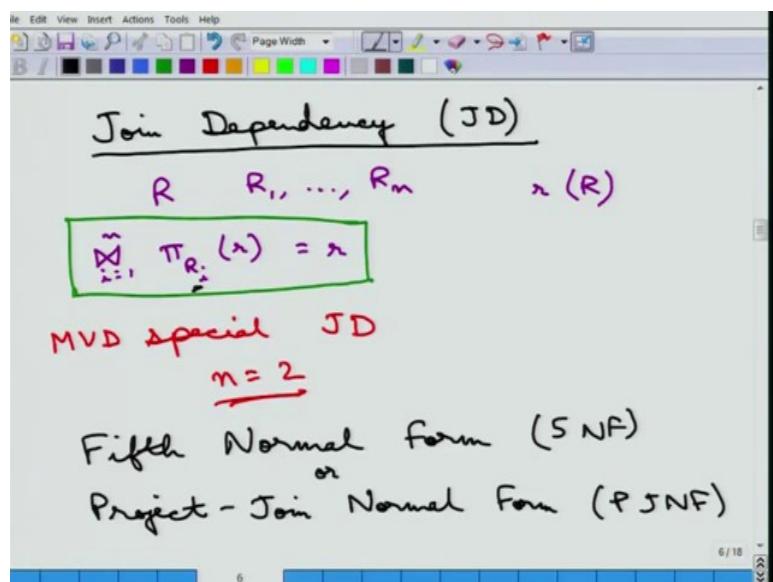
So, this course, teacher, book and the primary key is all of this together is the primary key, and this ... if the MVD course to book exists there, if  $\text{course} \rightarrow \text{book}$  exists then, this relation can be broken down into this, course, teacher and course, book. Now, what happens in this case is, so, if this is the only MVD that is there, the first thing we need to test is whether this is 4NF or not. The question is whether this is in 4NF or not. Now, the thing is course is not a superkey, so this relation is not in 4NF.

So, that is the problem with this and therefore, once we do the 4NF normalization, suppose we do this what is called a 4NF normalization then, the following things happen, is that this is broken down into the following two relations, which is course with book and course with teacher. And the only functional the MVDs are from course to book and from course to

teacher. So, which are both trivial, because X goes to Y and the right side and the (R - Y - X) is empty. So these are trivial, so this is by definition in 4NF.

So, this is a better design than using this entire thing, because, as we argued earlier, this has got modification anomalies. In the sense that if a new book is introduced multiple updates require or if a new teacher is introduced multiple updates are required in this relation, but not in these two. So, these two are in a better design. That's the whole point of the MVD and the 4NF, the 4th normal form.

(Refer Slide Time: 12:07)



Now, the MVD can be extended to a general concept which is called the *Join Dependency* or JD. This is essentially as I have been saying, that from MVD onwards it goes into the realm of more theoretical than practical things, and JD is an even more higher form of theoretical dependency. It essentially says that if there is a relation R for ... so suppose the relation R consists of this. It can be broken up into these things,  $R_1$  to  $R_n$ , then if you take the .. any  $R_i$  from this relation  $r$ ,  $R$  belongs to ... this  $r$  is a relation instance for the relation schema R.

And if you join all of them together, then you get what is known as the original relation  $r$ . This is of course, from R equal to 1 to n. So, if this condition holds, then there says to be join dependency. So, essentially the idea of join dependency is the following. In the MVD what we did is that we broke up the relation R into two parts  $R_1$  and  $R_2$  such that the join of  $R_1$  and  $R_2$ , the natural join of course, gives back the relation, the complete relation R.

Now, here what we are doing is that, we are breaking up the schema R into different set of attributes  $R_1$  to  $R_n$ , such that, the join of all of these together gives back the actual the original relation. So, that is when it happens for any ... for all possible instances or all legal instances of the relation, of the  $r$ . So, then this is called the join dependency. And one can see that multi-valued dependency can be now defined as a special case of join dependency.

So, MVD is a special join ... special case of join dependency, where the  $n = 2$ , that's it essentially. So, it is just broken up into two parts and if you break it up into multiple parts then this is called join dependency. Now, just as we could define a 4th normal form using the multi-valued dependency, we can define what is known as the 5th normal form using this join dependency. This is called the 5th normal form or 5NF. In a similar manner, the 5h normal form is also sometimes called project join normal form, because essentially it says that once you join, once you project it out on this particular attributes and join it you get back the original relation. So, that is called a Project Join Normal Form or PJNF.

So, again these are mostly theoretical things and example can be the following.

(Refer Slide Time: 14:51)

Brand	Product	Salesman	Brand
A	V	J	A
A	B	J	R
R	P	W	A
R	V	W	R
R	B	J	

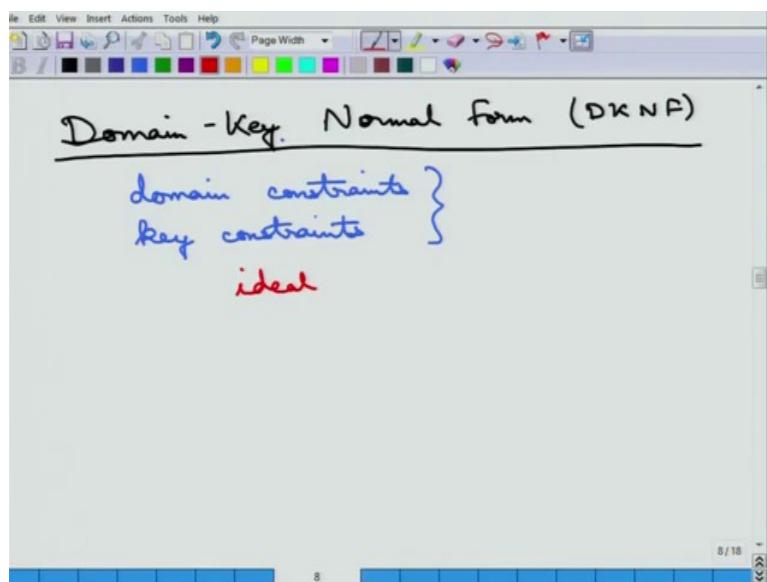
Product	Salesman
V	J
B	J
P	W
V	W
B	W

So, suppose there is a brand and a product and a salesman. Then it can be probably broken up into two parts at a time. So, it can be ... this can be taken or this can be taken or this along with this can be taken. And then example can be found out. And a very quick example may be the following. So, here in this example, if this brand, product, salesman has been broken up into these three relations: brand-products, salesman-brand and product-salesman, then one

can see that the natural join of all of these tables together gives back the original relation, brand, product salesman, such that, the join dependency is satisfied.

So, this is a much better design than keeping this original design of brand, product, salesman. Because, for example, the insertion ... the modification anomalies are minimal here, because now insertion of a particular salesman promotes a particular brand, that can be done by using inserting in only one case. For example, if  $J$  salesman does it in brand  $A$ , it is just needs to be inserted here instead of multiple insertions in the original table, where it is contains all the three attributes.

(Refer Slide Time: 16:29)



So, that is the project join normal form. The final form of this is called the *Domain Key Normal Form* or the DKNF. This is considered the highest form of normalization and this is essentially just theoretical concept. And here it says that, a relation schema is said to be in DKNF, if all the constraints and relations that should hold can be enforced simply by the domain constraints and the key constraints. So, everything that should hold can be enforced by simply the domain constraints and the key constraints, that's it.

So, we do not require any other constraints or any other conditions etc. So, these two essentially define everything that this relation should hold, that's all. So, this is the ideal normal form and this is mostly theoretical. And everything after 3NF or BCNF is mostly theoretical and database practitioner do not attempt to do it. So, this is in, that's it. So, once a relation is DKNF, because this is ideal there is no anomaly, there is absolutely no anomaly

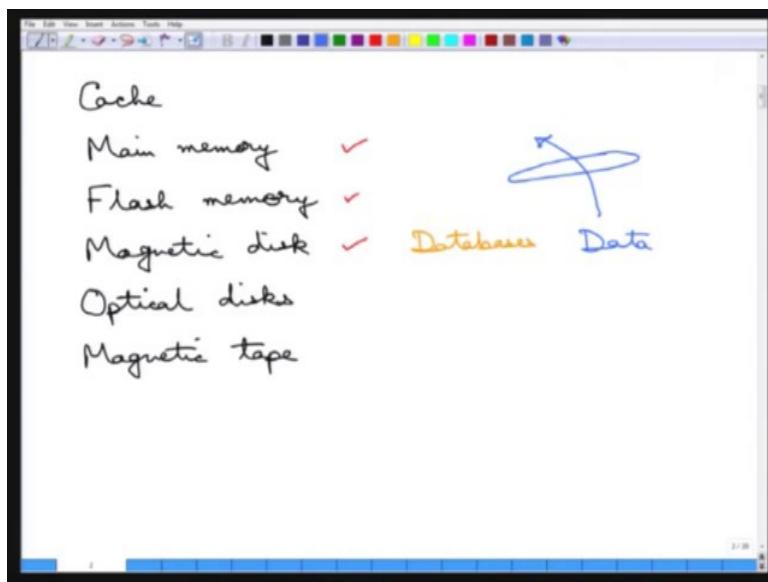
and all the functional dependencies and multi-valued dependencies need not be checked anymore. So, this is the ideal case. And that ends the topic on normalization. So, essentially given a relation you want to try to normalize it up to 3NF or BCNF.

Thank you.

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

**Lecture - 17**  
**Physical Design**

(Refer Slide Time: 00:24)



Welcome, so today we will talk about the physical design, so what do we mean by that is the about we will talk about the physical storage medium, where a database is stored. So, we will start off with the cache, cache as you all probably know is very fast, it is faster than main memory, but it is also very costly. And only CPU registers are more faster than the cache, but cache is very costly, data can be stored in the cache, but the more important problem about the cache is that it is volatile.

So, the contents vanish once the power is off and it cannot be used to persist a database. The next we will talk about the main memory, so once more main memory is quite fast, but also it is again very costly and it is volatile. So, the data cannot be persisted in main memory. And, what generally happens is that databases are quite large and the main memory that is available in a machine may or may not be able to store the entire data.

Even if it is able to store the entire data, the changes that are made in the main memory are not guaranteed to be persisted, because the anything such as power off can happen and the

updates can be lost. The next that we will talk about is the flash memory, flash memory the USB drives that we use is a type of flash memory, it is a solid state drive SSD memory. Flash memory very, very importantly is non volatile, so what do we mean by that is once we write something to a flash memory it remains, even if the flash memory is taken out of power or whatever else happens.

So, it is costlier than the hard drive and we will talk about the hard drive later, but it is less costly than a memory than a main memory or a cache and it is slower than a main memory. More importantly flash memory has certain types of, has a problem about it is the read write cycle. So, the flash memory can support only a limited number of read write cycle. So, after about  $10^6$  read writes,  $10^6$  is about 10 lakhs, the flash memory performance degrades.

So, a particular portion of the flash memory may start showing physical degradation and it may not be able to support anymore read writes and we will talk about flash memories a little bit more detail later. The next important physical storage medium that we will talk about is the magnetic disk, this is the hard disk that you normally understand. Magnetic disks are quite cheap compared to all the other kind of things and it, so it can be used to store a large number large amount of data, it is of course, non volatile.

But, it is quite slow compared to a main memory and it is slower compared to a flash memory as well. So, after magnetic disk, what comes next is the optical disks. So, optical disks are your CD, DVD etcetera and these are also non volatile, these are also quite slow and, but the more important problem about the optical disk is that unless it is a rewritable medium, generally CDs and DVDs are write once and read many times. So, you write it once and persist it and you cannot generally rewrite it unless of course, it is a rewritable medium.

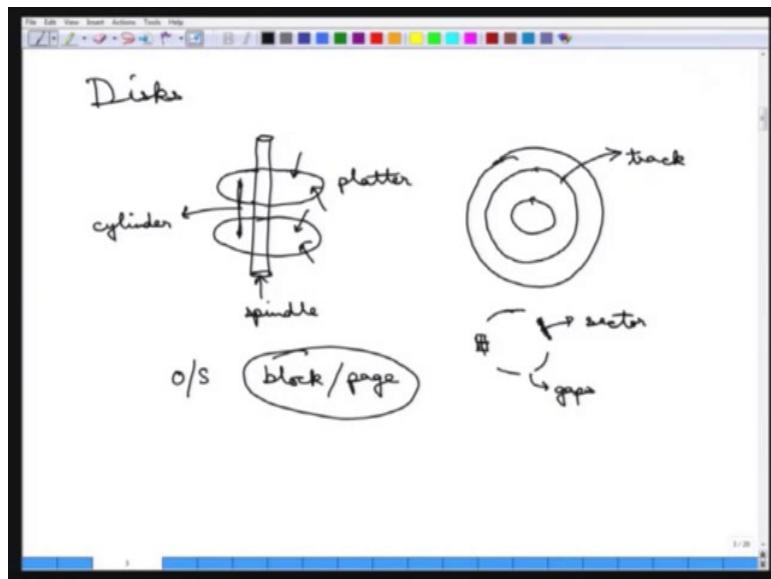
The last one in this type of physical storage medium that we will discuss about is called the magnetic tape. So, magnetic tapes are really, really very cheap and it is extremely slow, because you have to go through a magnetic tape completely to reach a particular byte of data. But, it is quite useful while taking backups of an entire institutional, quite while taking backups of large amounts of data and it is used very sparingly and magnetic disks tapes are generally used only for such back up purposes, it is not for day to day access, because it is extremely slow.

So, the important parts that we will worry about is the main memory and the magnetic disk, which are the hard drive and we will also talk about the flash memory a little bit. But

generally, what happens is that the databases are generally stored in the magnetic disk, so these are where the databases are stored. But, importantly the operating system cannot access data that is stored in a magnetic disk.

So, the operating system must bring the data back to the main memory for any access. So, the data that is stored in a magnetic disk must be brought to main memory and perhaps move to the CPU registers etcetera depending on the architecture of the machine, but it cannot be accessed directly from the magnetic disk. So, there is an overhead involved in bringing this data back from a magnetic disk to the main memory and this is what is called the disk access time.

(Refer Slide Time: 05:45)



Now, before we analyze what the disk access time are, let us first try to understand what the disks are, the magnetic disks that we will be talking about. Magnetic disks generally consist of a spindle and there are certain types of cylinders attached to it, so certain types of plates attached to it, so these are different plates. And data can be stored on each of these surfaces, so this surface and the back surface, so on, so forth.

And this is called a platter, so each such disk is called a platter, this is a spindle around which the platters rotates, so essentially everything rotates and a particular point in each of this platter consists, what is called a cylinder. Now, how is data stored in a platter? So, each platter if you lay it out it looks like this, this is a complete platter of course, with a spindle in between. So, a platter has two surfaces on each surface there are different tracks on which the

data is stored, so this is one track.

So, if this is one track each track is broken up into multiple sectors. So, there are different sectors that are broken up, so each of this is called a sector of data, so sectors are separated by gaps and in each sector consists of a multiple bytes of data, so this is how the entire design is there. So, each sector contains multiple bytes of data, each track contains multiple sectors, each surface contains multiple tracks, each platter contains two surfaces and a complete magnetic disk or a hard drive contains multiple platters and that is how the entire system is there.

Now, what the OS does is that, OS reads a unit of data, which is the block or page of data, this is the unique by which the OS reads it and each sector consists of multiple blocks of data.

(Refer Slide Time: 07:52)

The document contains handwritten notes and calculations:

$$T_{\text{access}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$$

Seek time :  $6/8 \text{ ms}$

Rotation time :  $\frac{1}{2} \times \frac{1}{2200 \text{ rpm}} \times \frac{60 \times 10^3}{1 \text{ min}}$

Transfer time :  $\frac{1}{400 \text{ sectors/track}} \times \frac{60}{2200 \text{ rpm}}$

Rotational speed : 2200 rpm      Seek time : 8 ms  
                                        400 sectors/track

Track = 8 ms

$$T_{\text{rotation}} = \frac{1}{2} \times \frac{1}{2200} \times 60 = 4.17 \text{ ms}$$
$$T_{\text{transfer}} = \frac{1}{400} \times \frac{60}{2200} = \frac{0.02 \text{ ms}}{12.19 \text{ ms}}$$

And having understood this, let us now try to understand, what is a disk access time, so a disk access time. So, the smallest amount of information that can be read or written from a disk is a sector, so one must reach a particular sector to read. Now, what happens is that, if we go back to the previous figure in the cylinder there is a read write head, this is the read write head of the disk that moves back and forth across the surface of a track radially, this movement is only radially.

So, the read write head the arm, the head of the read write arm, this is the read write arm, read write arm, the head of the read write arm can place itself on any track. But, having placed

itself on any track it cannot access any sectors, so for that the track must rotate. So, that the sector is placed under the particular read write arm and these movements of the read write arm and the rotation are all mechanical movements. So, that is why there is a physical limit, a mechanical limit as to how much faster hard drive can be.

So, coming back to disk access time, the disk access time consists of three kinds of things. So, disk access  $T_{access}$  time, consists of three different times, time to seek, time to rotate and time to transfer and we will describe each of this in more detail next. So, the seek time is the time for the arm of the read write head to place itself on the particular track that it wants to. So, it is a radial motion of the read write head, this is the read write head and this is the radial motion of the read write head; that is the seek time.

The rotation time is the time for the disk to rotate; such that the particular sector that is needed is placed under the head of the read write arm. So, it can be a complete rotation or it can be a zero rotation with the sector already under the head of this read write arm, so that is the rotation time. And finally is a transfer time, so a transfer time is the time; such that the complete sector of data is read, so suppose this is the sector of data; that is needed to be read by the OS.

So, this complete time that this rotates; such that that the entire amount of data is passed under the read write arm. So, these are the three kinds of things, the typical seek times are generally of the order of 6 to 8 milliseconds, this is milliseconds mind you. The rotational latency is well we can say it is half the time to rotate on average, because it can be either 0 or it can be a complete rotation. Remember, the disks can rotate only in one way, so if the read write arm has just missed it, so it will take a complete rotation to be back.

So, it is essentially half into  $1/\text{rpm}$ , this is the revolutions per minute. So, to convert it into seconds need to be conversion of 60 minutes to 60 seconds; that is the rotation time. And the transfer time is the number of sectors, so it to read one sector it is a number of sectors per track into that similar  $60/\text{rpm}$ , that we will do. So, this is the number of sectors; that is, where, so if to read one sector you require this amount of time and to convert it into seconds this is what is being done.

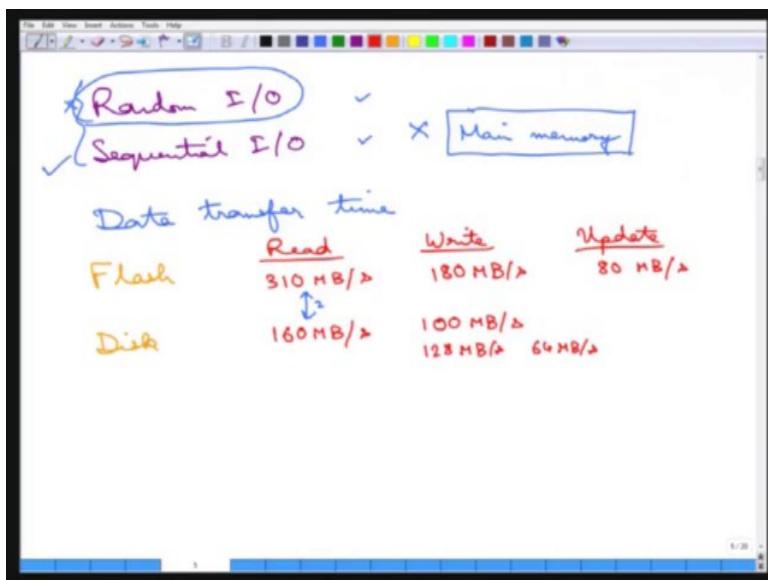
So, what does some typical disk parameters, so this typical disk parameters, let us take an example to do it. So, suppose the rotational speed of a disk is say your 7200 rpm and suppose the seek time given is 8 seconds, these are given by the manufacturer of the disks. Now, what

is the typical time to read one sector? We can compute it in the following manner.

So, is your 8 ms; that is given the will be you can compute it using the formula half into your 1/rpm, so 7200 into 60 that means, seconds, so this comes out to be around 4.17 ms and the we can again compute  $T_{transfer}$  it similarly. So suppose there are 400 sectors per track, so  $T_{seek}$  this is  $1/400$  into  $T_{rotation}$   $60/4200$ , so these are comes out to be 0.02 ms. So, if we add all these up together we get a total time of 12.19 ms; that is the average time to read one sector from the disk.

Now, as you can see the disks are quite slow, because to read one sector this requires 12 ms and this seems to be extremely slow, what happens is that, this is what is called the random IO.

(Refer Slide Time: 13:23)



So, there are two important things that we must understand one is called the random I/O and the other is the sequential I/O. What is called random IO is that, if you remember in the last computation we are trying to read an average time for a sector. So, the sector is essentially assumed to be randomly placed anywhere on the disk and we are trying to find the time to do it, so that is why it is called a random I/O. So, you are just given the address of a particular sector and you want to read that; that is the random I/O.

What happens in the sequential IO is that? You read a particular sector and the next sector to be read is sequentially placed after that the sector that you are already reading. So, there is no

seek time the seek time is 0 there is no rotational time as well, because the read write arm will is already placed on the correct track and just before the sector; that is being read. So, the only time that is required is the transfer time, so there is the big difference between the random I/O and sequential I/O.

So, to highlight the point once more, suppose you are reading more than two sectors it makes sense to place the two sectors one after another. Because, to read the two sectors, then the first sector needs to be written in a random manner, but the second is just after it, so there is no seek time etcetera, so second sector can be read much faster. So, it makes sense to put the two sectors sequentially on the disk and; that is why sequential access are much fast; that is why sequential access is, what is preferred by the database design.

On the other end if the two sectors are placed randomly anywhere on the disk it will take much more time to read the two sectors. So, if there is a big large table or a big large database it makes sense to layout the layout the data in the database physically in sequential sectors, so that is what the typical thing about this is. So, the data transfer rate etcetera are computed using this sequential I/O time and random I/O time and sequential I/O time, it actually is a little bit more complicated, because there are gaps between the two sectors.

But, it is roughly taken to be the, what we just did the computation, the one thing to note is that the sequential I/O is much faster than the random I/O, but much more faster is the main memory time. But, main memory is more costly, so the data is generally stored on the disks only, what is being done is that out of these two as I already said most algorithms will try to avoid the random I/O time as heavily as possible. So, it will rather do a sequential I/O time and not a random I/O time, because sequential I/O is much faster than main memory.

One more important thing that the database designers generally take into account is that the time to compute in the main memory is ignored. Why is it ignored? Because, the time to bring a particular data to the main memory is, so large compared to the computation time in the main memory. So, nowadays the machines are GHz machines, so it is essentially nanosecond or some orders of magnitude larger than nanoseconds time to do any computation on the data; that is brought to the main memory.

However, the time to bring the data is milliseconds, so it is let us say microseconds as compared to milliseconds, microseconds on the CPU computation time on the main memory versus milliseconds time to bring the data. So, the microseconds time can be ignored and it is

generally ignored by the database designers. So, the database designers essentially do not take care of how to reduce the main memory time, because it is only optimizing on the microseconds it rather ties to tackle more on the milliseconds time and more on the random I/O time.

So, the algorithms are designed, so that they reduce the random I/O time this is one very important point that we need to understand from this physical design is that the database algorithms are typically designed; such that the random I/O time is minimized. This was the traditional database story; nowadays as we have already mentioned quickly that there is a flash memory that has come into the picture.

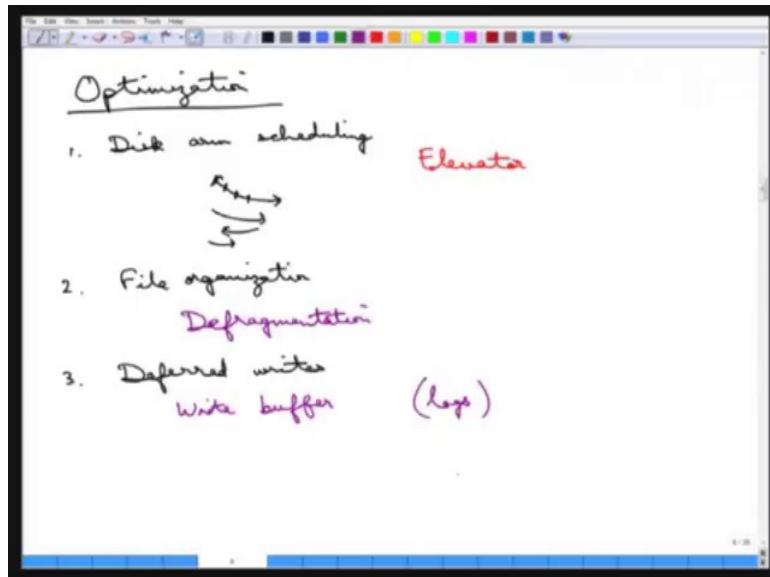
So, the flash memory it is, now becoming cheaper in the sense that some of the databases can be stored on the flash or at least some portions of the database can be stored on the flash and flash memories are typically faster than a hard drives or the magnetic disk that we are talking about. So, let us take some typical numbers, so flash drive read time for flash can be around the read throughput can be around 300 MB per second this is a typical time.

So, I mean different flash drives will do different the typical read write times are 180 MB per second and flash drives can update, but update is much slower it can be only 80 MB per second. As opposed to disks, disks are read times typical read times can be 160 MB per second and write times can be depending on sequential IO or this IO, let us say about 100 MB per second and update times are essentially just twice of this 100 MB per second.

So, you see there is a cost ratio of about two times here and this can be depending on the whether it is a sequential MB or this thing. So, this can be as large as 128 MB per second or 64 MB per second, so there is again a ratio of about 2 to 3. So, flash drives are about twice as fast as magnetic disk, but it is also quite costly. So, the database designer can now optimize as to how much data to put on the flash drive.

So, there is a cost versus access time optimization that, that database designer can do also flash drives has this problems that data needs to be deleted in much larger blocks than magnetic disks and it erodes after  $10^6$  read write cycles.

(Refer Slide Time: 19:40)



So, the disk block access that we are talking about, so there are two kinds of optimizations, that can be done about this disk block access the first is the disk arm scheduling. So, what it does is that this kind of algorithm first tries to see, what are the sectors that are going to be read or written by the database and then, it schedules such read and write unless they are dependent on each other assuming that independent, such that the disk arm is moved from one sector to another from one track to another.

So, it moves surface the first time this data; that is sitting on the first track is read or written, then this is accessed, then this is accessed and so on, so forth. So, that there is only one disk arm movement rather than the back and forth movements. So, that is the disk arm scheduling operation and then, there is an algorithm called the elevator algorithm, which tries to do that.

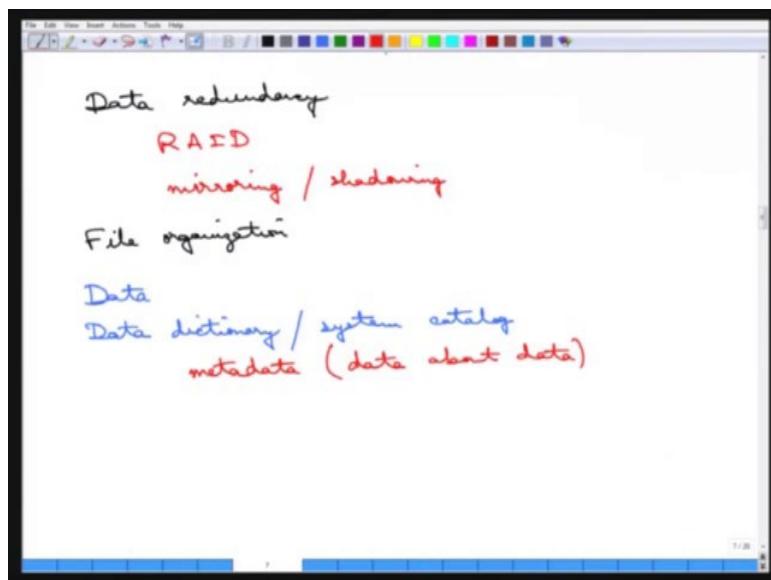
Then, there is of course, the other kind of this is one kind of operation the other kind of optimization is the file organization and we will talk about this more detailed, but the file organization it tries to do this we all have seen these terms, somewhere in the defragmentation. So, what it tries to do is if a file is fragmented if a file is broken into pieces, then it goes into different parts of the disk.

So, then there are more random I/Os that are needed to read the complete file, instead if it is defragmented the entire file is stored together and after one random Io the rest are sequential I/O, so that is faster for a database, so that is what it being done. Then, there are other kinds of optimization that it can do it can use this something called a deferred writes, so essentially

the databases is use this write buffers.

So, as soon as a write is available the database does not go and write it back to the disk it puts it back into the write buffer and only after sufficient writes are being buffered it, then goes ahead and writes them one after another sequentially in that thing. So, this mean times logs etc and we will see how this is done later, but that is the basic idea is that it is the writes are deferred it is not one after another it is not immediate; that is the main point. Fine and then, let us talk about the related issue, which is on data redundancy.

(Refer Slide Time: 22:17)



So, data redundancy is done to improve the reliability. So, instead of storing the data in namely one disk or one place it is stored in multiple places; such that even if one copy is lost due to some mechanical problem etcetera, it can be retrieved from the other things. And there are this famous terms are called raid, so redundant arrays of independent disk. So, this is essentially independent the disks are maintained independently and each one is a redundant copy of the other, then this mirroring is allowed and you have probably heard the term in terms of web mirror.

So, inter website is mirrored or shadowed, so that one can access it from any of those sites. This is more or less all about the data redundancy, then there are this problems of how to store the files the issue of the file organizations. So, the file organization the files can be stored in a sequential manner or it can be hashed depending on the block address etc. So, the database stores the data actually about the data, then it also stores something called the data

dictionary or the system catalog, which stores the data about data, so the term for that is called metadata, this is data about the data.

So, how is the primary data stored, so how are the tables organized, what are the tables mean, what are the schematics of the table, what are the types of the table etc. Now, one thing to notice that, why is this being highlighted, because metadata is accessed much more often than a particular piece of data. So, whenever anything about the table is read the metadata about the table needs to be written.

So, it makes sense to have some algorithms that can read the metadata faster. So, it can be stored in a faster memory for example, a particular. So, it can be brought to main memory and persisted in main memory, because generally metadata does not change much it does change a little bit about the statistics of the data, but the type of the data does not change much. So, the metadata can be brought into main memory, because it is accessed much more often.

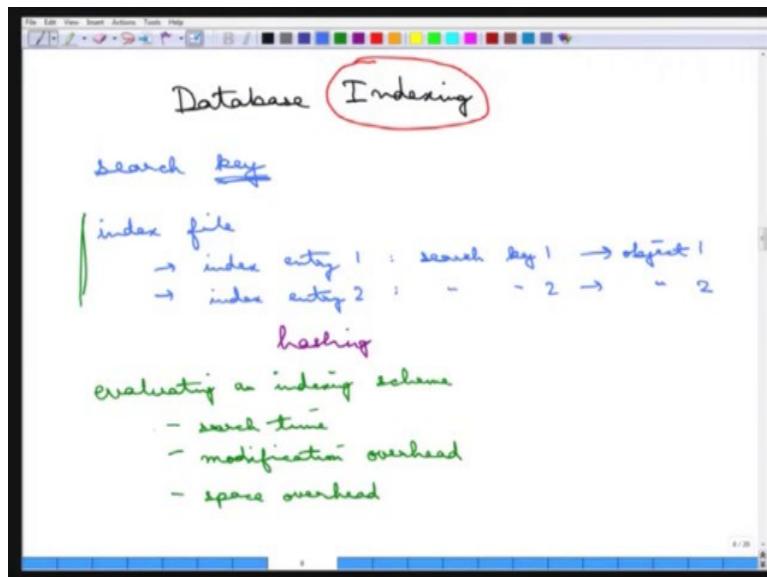
So, this is all about the data, so of course, what I meant to say is that statistics about the data is also maintained. So, and that is all about this storage of physical design and next we will cover the indexing.

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

**Lecture - 18**  
**Database Indexing: Hashing**

We will start on the next topic which is on Database Indexing.

(Refer Slide Time: 00:15)



So, we will first try to understand what does the word indexing mean and you all have probably seen examples of indexing is in a book towards the end of the book, there is a word is written on the page number to where the book is found is written and that is called an index. So, what does indexing help one do is to make the search faster.

So, imagine that a book has no index and you want to find a particular word. What is the way to do it? You have to go through the book probably page by page, word by word to find that particular word. If the index is there, you essentially just go to the index, find the word that you are looking for and just simply go to the page number. Now, also important in the index is that the words are stored in an alphabetical manner. So, that finding the word inside the index is also faster, so that is the whole concept of indexing that is the whole idea of indexing and databases use index very heavily, so that the queries can be fast.

So, what are the kinds of queries that we are talking about? Think of this select query, select all branches or select all loans whose amount is greater than 300, etc. Now, how do you find

out all such loans? The naive way of doing it, that the simplest way of doing it is to go through all the loans and select whichever is larger than 300. But, suppose an index is built and we will see exactly how the indexes are built, suppose the index is built which tells you all loans that are greater than 200 and all loans that are less than 400 very easily, then all one needs to do is to find it inside this bucket of the loans and not the other one, so that makes it much faster, so that is called the indexing.

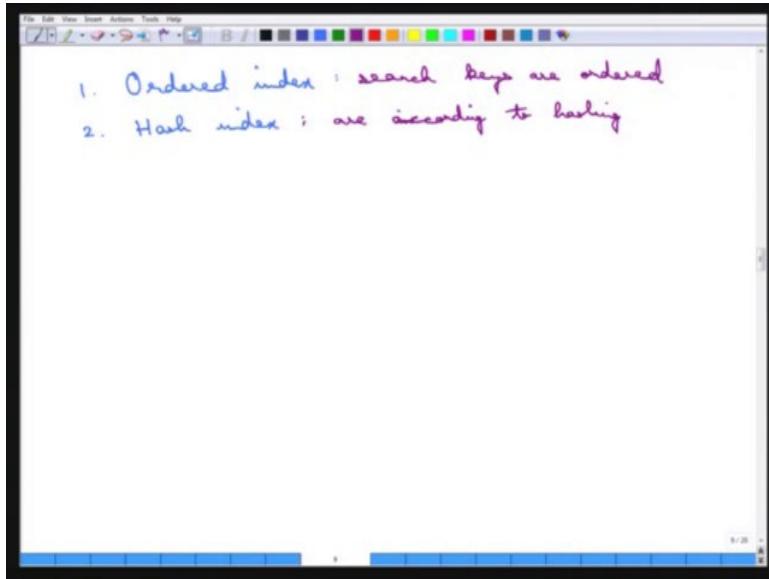
So, what is being done in an indexing is a search key is used. So, we are talking about searching, so there is a search key that is being used. So, we want to find this particular key and how is this done. So, an index file is maintained, there is an index file which contains multiple index entries, so each index entry 1, index entry 2, etc, so the multiple index entries are maintained. So, what is a look index entry? It contains a particular search key and a corresponding object to it.

So, as soon as I make this kind of design, the first thing that comes to one's mind is we have seen this and this is exactly what is being done in hashing, hashing does something similar. There is a search key and then there is a actual object corresponding to the search key, corresponding to the hash key. Before we go into the types of indexing, let us just see that how does one evaluate an index. So, evaluating...

So, there can be different types of indexing schemes, evaluating an indexing scheme requires the following thing is that on what are the ground that we will index, it is the search time of course. So, the whole point of indexing is to reduce the search time, so it must be able to do so, then the modification overhead. So, what is meant by the modification overhead is that, suppose the data in the original database has changed. Now, how much time does it require to change the corresponding index or the entire index file that is the modification overhead and the space overhead.

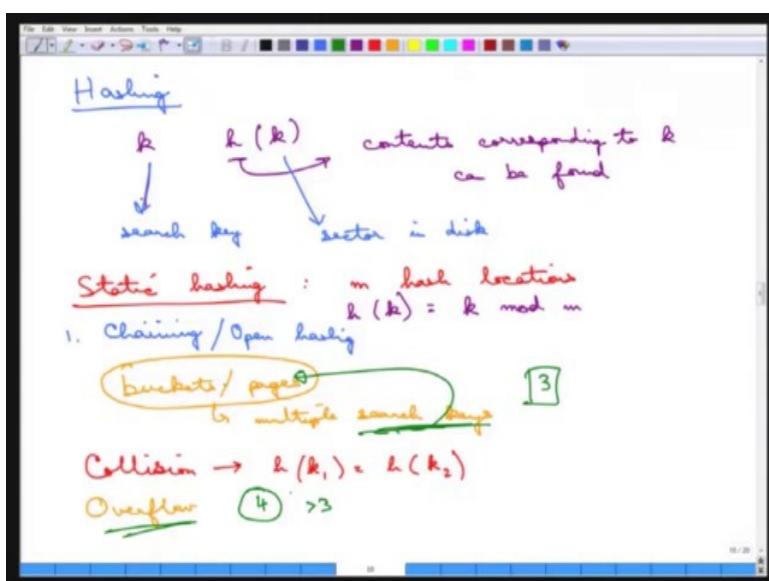
So, suppose the original data is 1GB, so how much extra space does the index needs, because you see the index is different from the original data. So, there will be some extra space, so what is that space overhead that we are talking about. So, these are the three criteria for evaluating an index.

(Refer Slide Time: 04:22)



Then, there are two basic types of index schemes, the first is called an ordered indexing scheme and the second one is a hash index. So, in the ordered indexing schemes, the search keys are ordered, the search keys are stored in an ordered manner and this is what we see in the index of a book. The search keys are stored in an ordered manner in the index and in the hash key, the search keys are in some hash order or according to hashing. So, there is essentially no order or you can, one can say that there is a hashing order, so it depends on the hash function. So, let us go to the indexing that we all know about which is the hashing.

(Refer Slide Time: 05:10)



So, we will talk about hashing, hashing there are two main types of hashing. So, before that what is a hash function does, the hash function does the following is that there is a key  $k$ , then a hash function is applied on that, hash function  $h$  is applied on that. So, that gives a location, where the object is stored or where the contents corresponding to the key, contents corresponding to  $k$ . So, that can be in the object, that can be the actual some other else, whatever if corresponding to key can be found, so that is the whole point of hashing.

Now, what does it mean in the context of a database? The key is the search key and the hash key, there is a sector or in the disk sector in disk, where the contents corresponding to the search key are stored. So, that is the context in the database, so that is the hashing and then there are different kinds of hashing, the first one is called the static hashing. So, what are the ways of static hashing? The first one is chaining.

So, static hashing by the way static hashing essentially says that there are  $m$  hash locations. So, everything that needs to be hashed is one out of this  $m$ , so very common example of hashing is that if there are  $h$  locations, there is  $k \bmod m$ . So, any key gives you a number, once you take a modulus according to  $m$ , gives you a number between 0 to  $m - 1$  which is what, where the key is stored or the contents corresponding to the key are stored.

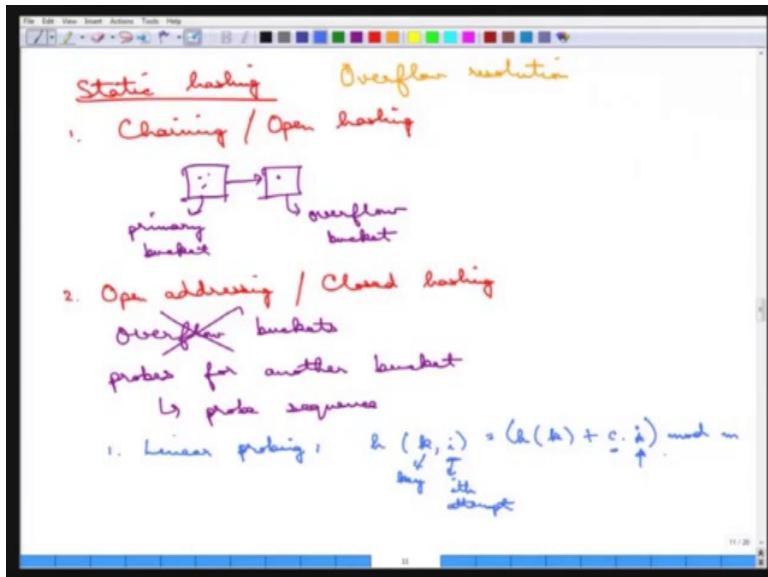
Now, what is in the context of chaining? Chaining is also sometimes called open hashing. What does chaining mean is that, before we go into the chaining, let us go back to the database context and in the context of hashing, what is being done is the following is that in the database, the objects are stored according to buckets or pages in the disk. So, a single page can contain multiple search keys, so this can contain multiple search keys.

So, once a search key is given the point of the hashing is to find the bucket, where it is stored and a bucket or a page can contain multiple search keys. So, in normal hashing, in the OS sense, what we termed as collision, what does a collision mean is that when two hash keys have the same location. So, then  $h(k_1) = h(k_2)$  they collide, because they are both trying to store in the particular position. In the context of database, there is no concept of collision, because multiple search keys can be stored in a bucket.

So, the collision does not make sense what makes sense is overflow, the concept of overflow. The overflow is the following is that when there are so many search keys, so suppose in a particular bucket three search keys can be stored. Now, overflow may happen when more than three, suppose four hash keys are trying to go to the same bucket, then the bucket is said

to overflow. Because, the bucket or the page or the sector or whatever does not have enough space to contain these four things, it can only contain three, so anything greater than three is a problem and that is what this the overflow is about.

(Refer Slide Time: 08:56)



Let us talk about what the static hashing means, so there are static hashing and the first scheme there is called the chaining. So, chaining is that when... So, all these static hashing, etcetera what we are trying to do is something called an overflow resolution mechanism. So, as opposed to a collision resolution mechanism, this is called an overflow resolution mechanism. So, chaining, what does chaining or open hashing does is that when there are multiple keys that are going to the same bucket, so this suppose this is the bucket and then there are multiple keys going to it, another bucket is linked to it, chained to it.

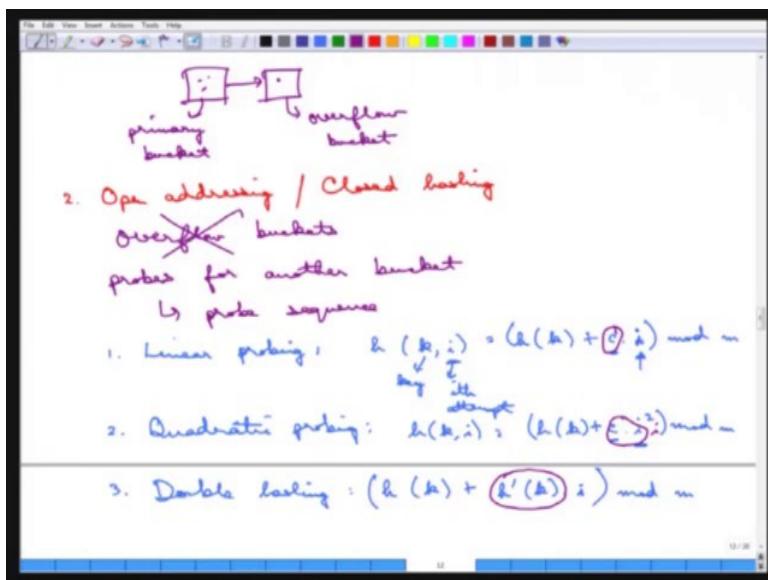
So, the next key that falls to the same bucket is sent to the bucket that is chained to it. So, that is called a chaining. So, these are the overflow buckets, this is the primary bucket, where a particular hash function finds its way and if it is full, it goes to an overflow bucket, so that is chaining. As compared to chaining, the next one is called open addressing or closed hashing, so it does not employ overflow buckets.

So, overflow buckets are not employed, what it does is that it probes, it uses a something called a probe, it probes for another location, another bucket and it probes according to a probe sequence and then there are multiple ways of doing the probe sequence and let us just list them, the first one is called a linear probing. So, the intervals of this probe sequence

remain fixed. So, suppose this is  $h(k, i)$  so what does  $i$  mean, it is the  $i$ 'th attempt to...

So, this is the key and this is the  $i$ 'th attempt, so whenever there is some problem with the bucket. So, whenever the bucket is full, it takes another attempt at hashing it. So, this is the  $i$ 'th attempt to find another hash position, so this  $h(k, i) = (h(k) + c * i) \bmod m$  assuming this is anything, so this is a constant and this is a linear. Why it is called a linear thing? Because, the position corresponding to the original position is changed linearly that is why this is called a linear probing.

(Refer Slide Time: 11:44)



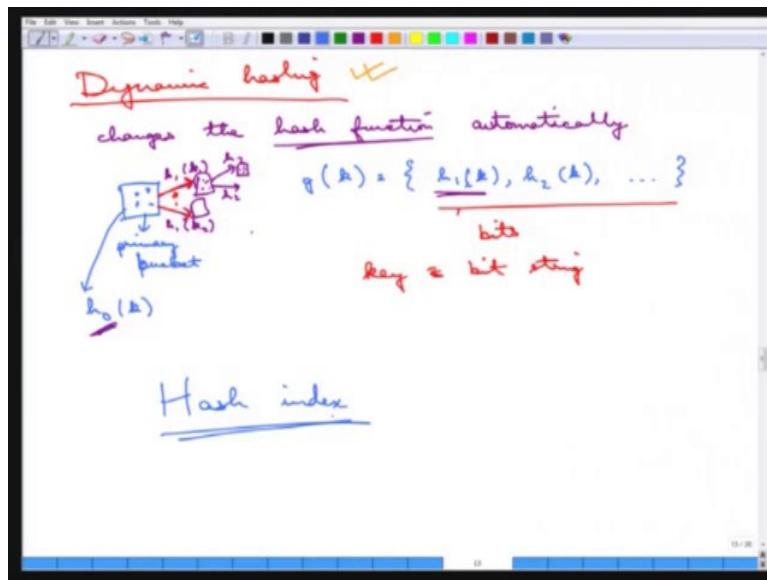
Then one can understand there is a quadratic probing, where this function changes to  $i^2$ . So, it depends on this, this is why it is called the quadratic probing and then the third one is called a double hashing, where instead of using the  $i$ 'th attempt, a completely new function is evaluated. So, it is  $h(k) + h'(k)i$  that is being done, so instead of a constant this is which  $h'(k) * i$  is of.  $\bmod m$

So, one thing to notice that everywhere there is this  $i$  function, but for linear probing this is a constant for quadratic probing this is  $c * i$ , this is essentially you can write this as  $c * i * i$  and for double hashing, this is the completely new function, double that is the double hashing. So, this is the, these are the three different ways of static hashing.

Static hashing; however, has the following problem, is that suppose one has allocated  $m$  buckets to start off with and the data really does not fit into any of these  $m$  buckets. So, the

data for is much more than  $m$  buckets, then what can happen is this static hashing performance degrades. So, the problem of static hashing is that it starts off with the idea of how much data is going to be hashed, but it cannot adopt automatically if the data is much more than what it has guessed or much less than what it has guessed.

(Refer Slide Time: 13:37)



So, that is the problem with static hashing and to handle that there is something called a dynamic hashing. So, the dynamic hashing tries to attempt to rectify the situation by dynamically adopting the hashing function itself. So, it changes the hash function itself, the hash function is not static anymore automatically to adjust to the volume of data that is coming. We will not talk about many methods of dynamic hashing that we will talk about one very simple way of dynamic hashing that is called, it was originally just called dynamic hashing.

Suppose, this is the primary page that one goes to, primary page or primary bucket, whatever one can call. So this goes to and suppose it has overflowed, then what happens is that, this is produced by some function, let us say . So, this is  $h_0(k)$  the 0'th function, then there is a series of function, which is etc,  $g(k) = \{h_1(k), h_2(k), \dots\}$  there is a series of function that can be done whenever the data overflows, so all data that overflows from a primary bucket.

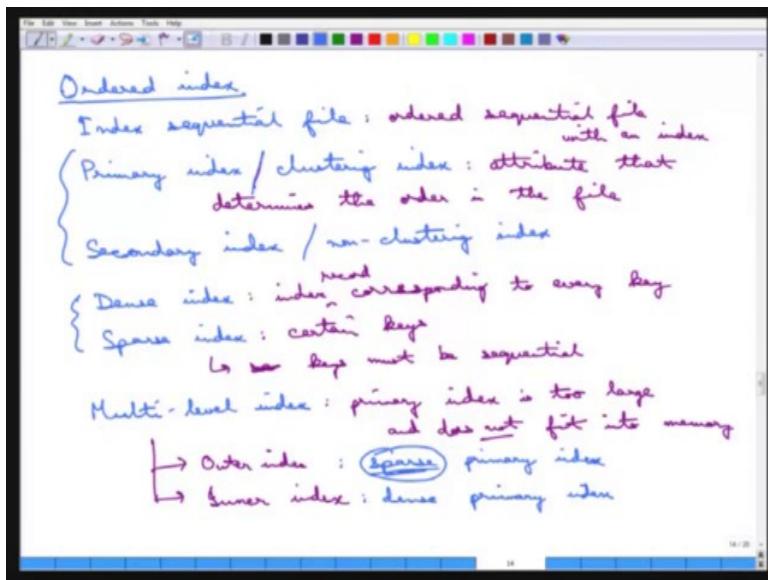
The next function in the series is applied, so this goes to  $h_1(k_1)$  and this is  $h_1(k_2)$ .

So, even though  $h_0$  and  $h_1$  is the same that goes to the same thing, it goes to different bucket after that and that is the dynamic, and even if one of them fails then it utilizes  $h_2$  and so on, so forth. So, that is  $h_0(k_1) \rightarrow h_0(k_2)$  how it keeps on growing and that is how it keeps on automatically adjusting, because you see that between  $h_0$  and  $h_1$  it is  $k_2$  not just that  $h_0$  is the function, it is  $h_1$ ,  $h_2$  because  $h_0$  has overflowed whereas, for other  $k_3$  it is simply  $h_0$ , because that bucket has not overflowed.

Now, one very useful way of what getting this  $h_1$ ,  $h_2$  is just the bits. So, if a key is represented by its bit-string, then one can simply apply the bits in order to get the different branches in this hash tree, one can call this a hash tree. So, if this is the bit, is 0, it can go here if the bit is 1 it can go here and so on, so forth that is a very simple example. But, the whole point is that this does what is called a dynamic hashing, because this adopts dynamically to the situation and if the data is reduced it can collapse back in the same manner.

So, this is all about dynamic hashing and the next important thing that we will cover is called a... So, this is called about this is a hash index, because the data is arranged in a hashing manner, the next in that we will cover about is the ordered index.

(Refer Slide Time: 16:46)



So, little bit of the ordered index that we will do, so the ordered index file, there is an index sequential file corresponding to this and then there is a primary index, this by the way is the ordered sequential file with the index. Then, there is a primary index, this is also called the clustering index, this attribute determines the attribute that determines the order in the file

and then there can be secondary index or the non clustering index which is any other attribute.

So, the file is ordered according to the primary index and any other attribute on which the index is done is called a secondary index, this can be one way of doing it then there is other way called a dense index, this is a different terms that is one needs to know dense index is that there is an index record, index corresponding to every key. So, that is a dense index as opposed to a sparse index, where there are index records corresponding to only certain keys.

Now, how does this sparse index works? If there are index records corresponding to only certain keys, keys must be sequential here; otherwise, it cannot happen one cannot find a particular key, if there is no record corresponding to it and the keys are not sequential then it cannot be happened. So, that is a thing about sparse index, so this is another way of classifying the indexes, then one more important way is called the multi level index.

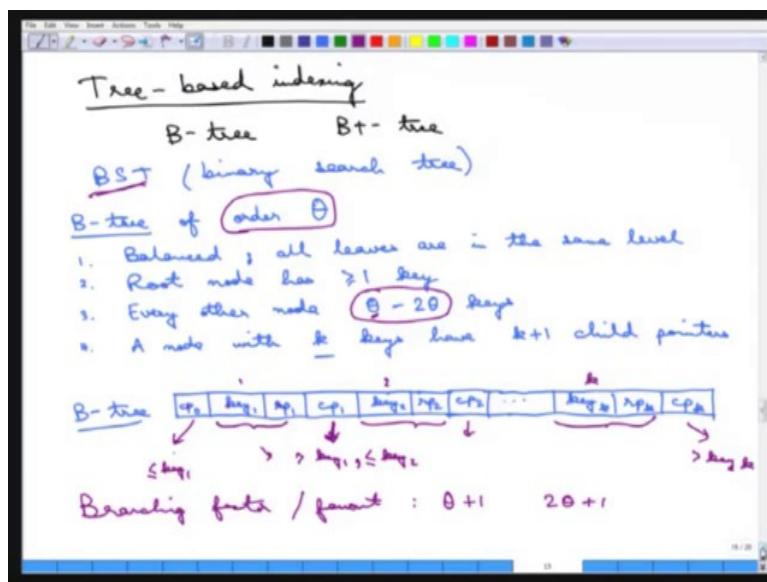
So, multi level index means there are two indexes, so this may happen if the primary index is too large and the entire index does not fit into memory. So, that can happen when a multi level index and then this can be broken into an outer index and an inner index. So, the outer index is the sparse, because it does not fit into any main memory, so not everything can be indexed. So, that is why it uses sparse and this must be a primary index. While the inner index can be a dense primary index, this is the way a multi level index was this is very important that this has to be sparse and this is called the outer index. So, that is about the ordered index. Next we will cover one very important topic in this indexing which is on the tree based indexing.

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

**Lecture - 19**  
**Database Indexing: Tree-Based Indexing**

Next we will start on one very important type of indexing, which is a Tree Based Indexing.

(Refer Slide Time: 00:15)



And we will cover two important structures in this, the B-tree and the B+ tree. So, these are for indexing the data on the disk. So, these are secondary storage or the disk is sometimes called the secondary storage. So, these are disk aware or designs and I am assuming that you are familiar a little bit with a binary search tree. So, this is essentially a binary search tree built for the disk.

So, if you recall the main design of a binary search tree is that corresponding to a node, the data is divided into two parts and the left subtree of that node, the left child gets all the data that is less than the key that is stored in the node and the right subtree or the right child gets all the data that is larger than the key that is stored in the node. So, that is the binary search tree and we will build upon the BST, so that is the Binary Search Tree and the B-tree and the B+ tree builds upon that and how is it stored.

So, the B-tree let us take, the B-tree, there is an order corresponding to a B-tree, B-tree of a particular order we will see what the order is, suppose the order is  $\theta$  has the following

properties. First of all the B-tree is a balanced structure and balanced and all leaves are in the last level and final level, all leaves are in the... So, balanced meaning all leaves are in the same level, so this is the balanced. Then, there is a root node has at least one key it must contain of course, at least one key. Any other node, every other node will have between  $\theta$  to  $2\theta$  keys. So, it will have number of keys which is between  $\theta$  to  $2\theta$ .

And a node with  $k$  keys will have  $k + 1$  child pointers, so the internal node design of a B-tree looks like the following is that there is a key, let us say  $key_1$ . Now, corresponding to  $key_1$  there is a child pointer  $cp_0$  and then there is a child pointer, there is a record pointer  $rp_1$  corresponding to the  $key_1$  and then there is a child pointer  $cp_1$ . So, this goes on, so child pointer  $cp_1$  then there is a  $key_2$ , then there is a record pointer  $rp_2$ , then there is a child pointer  $cp_2$  etc. It goes on like this till there is about  $key_k$  let us say, then there is a record pointer  $rp_k$  and then there is a child pointer  $cp_k$ . So, this is the structure, this is the B-tree internal node with  $k$  keys.

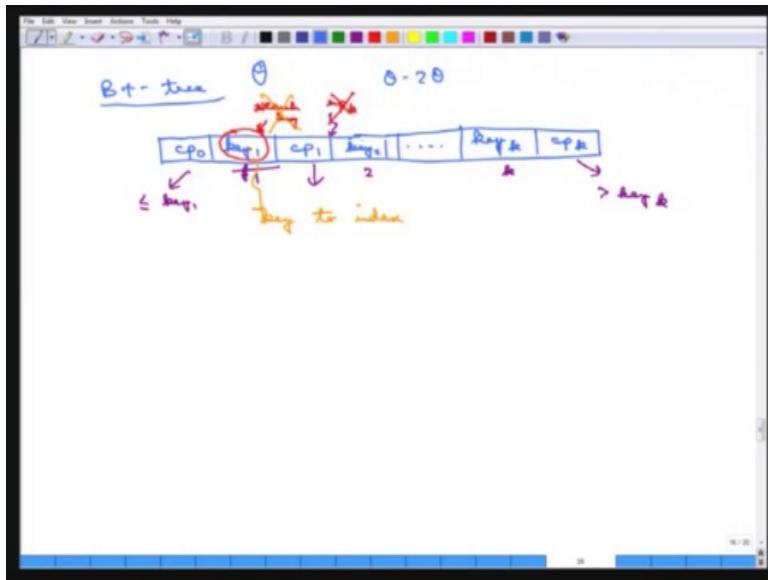
So, let us try to understand this, so there are these  $k$  keys, this is  $key_1$ ,  $key_2$  up to  $key_k$ , corresponding to each key the record pointer. So, this is a pointer that stores the record of the corresponding key is stored next to it. So, there are this  $k$  such record pointer, key pointer and then there are  $k + 1$  child pointers and the child pointers this follows the same idea as the BST. So, this child pointer contains all the keys that is less than this  $key_1$ , this child pointer contains all the keys that is greater than  $key_1$ , but less than  $key_2$  and so on, so forth it goes on all the way.

So, this is everything that is less than  $key_1$  and this is everything that is greater than  $key_k$  technically correct, this has to be less than equal to so on, so forth, but that is the whole idea of a B+ tree and it is organized in that entire manner and then there is a order theta that is mentioned as part of the B-tree. So, every internal node must have between  $\theta$  to  $2\theta$  keys cannot have anything; other than it cannot have less than  $\theta$  things,  $\theta$  key that cannot have more than  $2\theta$  keys. So, there is a term called the branching factor or the fan-out of a B+ tree is therefore, it must have between  $\theta$  to  $2\theta$  keys.

So, the branch out must be at least  $\theta + 1$  and it is at most  $2\theta + 1$ . So, the branching factor

the fan-out is the number of children that one node can have. So, in this case of  $k$  things there are  $k + 1$  children, so that is a B-tree. Now, let us move on to what is a B+ tree, the B+ tree is very similar, but with one important difference.

(Refer Slide Time: 05:49)



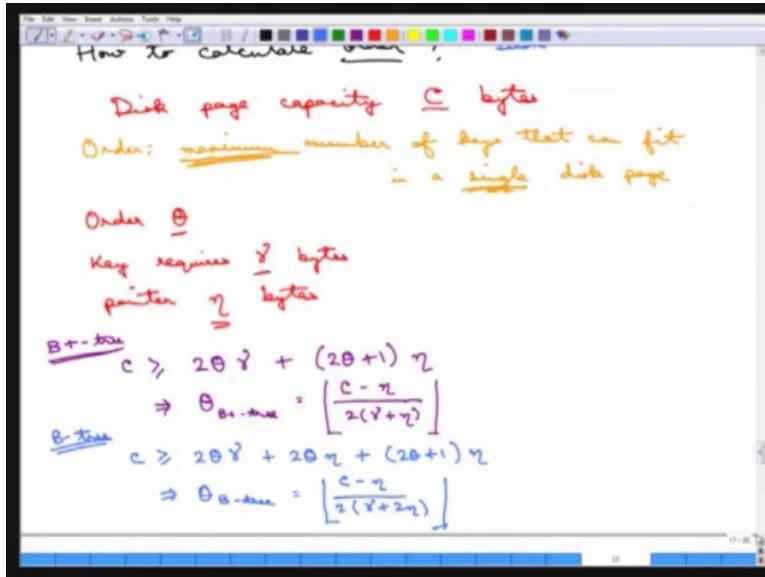
So, the B+ tree is a very similar structure, but there is one important difference with the B-tree. So, if one draws the internal node structure of a B+ tree, one can understand the difference. So, this is a  $key_1$  then corresponding to this there is a  $cp_0$ , then there is no record pointer. So, that is the very important difference, so then this all goes on, then there is  $key_k$  and then there is a child pointer  $cp_k$ , then there is no record pointer corresponding to a key in B+ tree. So, there is just a key and there is no record pointer. So, this is 1, 2 again up to k and then there are this again  $k + 1$  children and this again follows the same idea  $key_1$  etc.

But, it does not have the corresponding record pointer, so this record pointer  $rp_k$  is not maintained inside the internal node. So, what does this mean is that these keys cannot be actual search keys, because if it is an actual search key the record pointer must be present. So, even if it is an actual search key, this actual search keys must be stored only at the leaf level, this cannot be an actual search key. Because, no key is stored, no record pointer is stored, this is not a search key this is just what is called a key to index.

So, these are not real keys that are present in the database, this may not be real keys that is present in the database, these are just keys to index. So, that is a big difference between a B-

tree and a B+ tree. Now, again the, everything else remains the same as the B-tree. So, it is balanced and everything is in the leaf and it is of an order, it can have an order theta so; that means, it can have between  $\theta$  to  $2\theta$  keys etc of the same thing.

(Refer Slide Time: 07:49)



Now, how is the order of a B-tree or a B+ tree, how to calculate the order of a B-tree or a B+ tree. So, I mentioned that these are disk based design. So, where is the disk based design coming? How to calculate order is where the disk based design comes, this depends on the disk that is there. So, if you remember the OS cannot access everything from the disk, it can only access data in terms of a page or block, this is called disk page or a block.

So, suppose the disk page or a block is some 4KB of data, so it cannot access only 1 byte of data, it must access the entire 4KB of data from the disk to the main memory and then, process that 1 byte or how many whatever number of bytes that is required to do that. And if you also remember, the main point of the physical design that we showed is that is to reduce the number of this disk accesses, the number of random IOS or the number of sequential IOS is the main point.

So, it does not matter what is the time spent in the main memory, but the point is to optimize on the number of blocks that is read or written by the algorithm, so the number of blocks that is accessed. So, since a particular page must be accessed from the disk and since accessing the page meaning, accessing all 4KB of data, it makes sense to packing the disk page with as much data as possible. So, as much of means as much of 4KB of data whatever is the size

there, such that once the page is accessed, once the page is brought into memory, the entire data that is stored in the page can be utilized and it does not require another access to bring in another data point.

So, what I mean to say is that, suppose the disk page capacity, so a disk page suppose the disk page has a disk page capacity is of  $C$  bytes. So, then it makes sense to put in as many search keys as possible within those  $C$  bytes, because this... So, one has to access this  $C$  bytes, one has to bring this all this  $C$  bytes anyway even if one search key is accessed. So, it makes sense to put in as many search keys as possible within this  $C$  bytes.

So, now, how to calculate the order? So, order is the maximum number of keys that can fit, maximum number of search keys that can fit in a disk page, the order is calculated using that can fit in a single disk page. So, that is the maximum that disk can fit in a single disk page. Then, how to compute it? Let us say the disk page capacity is  $C$  bytes, let us say the order is  $\theta$  and we will calculate what  $\theta$  is, this is  $\theta$  and suppose key consumes  $\gamma$  bytes to store a pointer that the pointer to where the data is requires some  $\eta$  bytes.

So, for a B+ tree what will happen is that this  $C \geq 2\theta$ . So, if the order is  $\theta$  then there will be  $2\theta$  keys that can be stored, so  $2\theta \times \gamma$  plus there are no record pointers for a B+ tree. So, this we are doing for a B+ tree, there is no record pointer. But, there are  $2\theta + 1$  child pointers, so which is  $\eta$ .

$$C \geq 2\theta\eta + (2\theta + 1)\eta$$

So, this is the definitive equation for a B+ tree, so one if you calculate this comes out to be the following, it is

$$\theta_{B+tree} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

And of course, one has to take the floor function of it is the maximum number that can fit, this is how to calculate the order for a B+ tree.

For a B-tree, the same thing can be done except there is something more. So, there is a  $2\theta \times \gamma$ , this is the number of bytes that is required to store the  $2\theta$  keys, then there are  $2\theta$

record pointers. So, one has to add this,

$$C \geq 2\theta\eta + 2\theta\eta(2\theta + 1)\eta$$

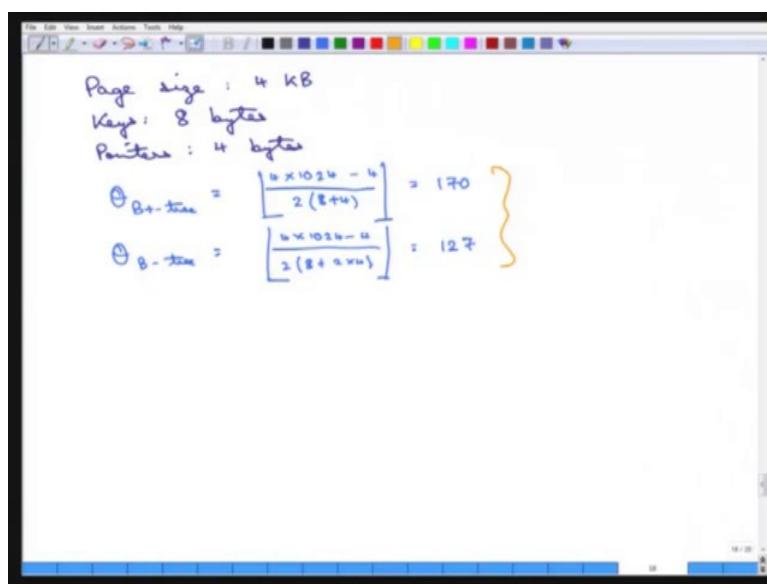
for the child pointers, so this then comes out to be

$$\theta_{B-tree} = \left\lfloor \frac{C - \eta}{2(\gamma + 2\eta)} \right\rfloor$$

So, these are the two important formula that we need to remember on how to compute the order of a B+ tree, how did we come to this conclusion to repeat is that it makes sense to read an entire disk space. So, it makes sense to pack in as much data as possible within a disk space, because that is the least that the OS will read anyway and the disk accesses are the costliest operations. So, it makes sense to reduce the number of disk access operations.

And once a disk, once a block or once a disk space is accessed it makes sense to pack in as much of that C bytes as possible, if theta is the maximum number of keys that can be done then this two equations give a way of how to find out this theta.

(Refer Slide Time: 13:46)



And we can take a simple example of how to do it, let us compute this example there is a page size, typical page size is of the order of 4KB, 8KB, etc, let us say it is 4KB. So, this is 4 KB the page size, let us say the keys take 8 bytes and pointers suppose it takes 4 bytes, let us assume this is the situation. Now, how do we find out? The B+ tree can be used the formula

that is there, so this is the capacity. So, if one computes this, this comes out to be 170 while the  $\theta$  for B-tree., now, I have one to highlight one important point here, you see that the order of a B+ tree is more than the order of a B-tree. So, it is in this particular example it is 170 versus 127, so which means that for a particular node there can be much more data packs for a B+ tree than a B-tree. So, which also means that a height of a B+ tree can be lesser than that of a height of a B-tree, because assuming the same amount of data since more amount of data can be fit within a node, less amounts of nodes are required and the branching factor is also larger for this node, so which means that the height of the B+ tree can be less.

However, what may happen is that in a B-tree the search key is located higher up than the level of the leaf. So, a particular search key can be located much faster when going all the way up to the leaf level, for a B+ tree there is no way, because the internal nodes do not contain the search key, every search key is stored only at the leaf level. So, to find the search key for a B+ tree, the search has to process all the way up to the leaf level.

In general what happens is that the number of search keys that the B+ tree found without going all the way up to the leaf level is very low. And therefore, what the database practitioners actually do is to build up a B+ tree and not the B-tree and even though one will find that the B-tree index has been built it is actually the B+ tree index that is being built for commercial databases and for other implementations of database systems.

So, it is the B+ tree that is the de facto standard and that completes the B-tree and the B+ tree thing and that also completes about the tree based indexing, there are little more left about the indexing and one very interesting kind of indexing that can be done is called a bitmap indexing.

(Refer Slide Time: 16:53)

Bitmap indexing

bitmap / bit vector

Gender	Grade
M	C
F	A
F	C
M	D
M	A

$\{M, F\}$        $\{A, B, C, D\}$

$M = \{10011\}$

$F = \{01100\}$

$A = \{01001\}$

$B = \{00000\}$

$C = \{10100\}$

$D = \{00010\}$

Find male student who got 'D'?

bitmap (M) AND bitmap (D)

Ans

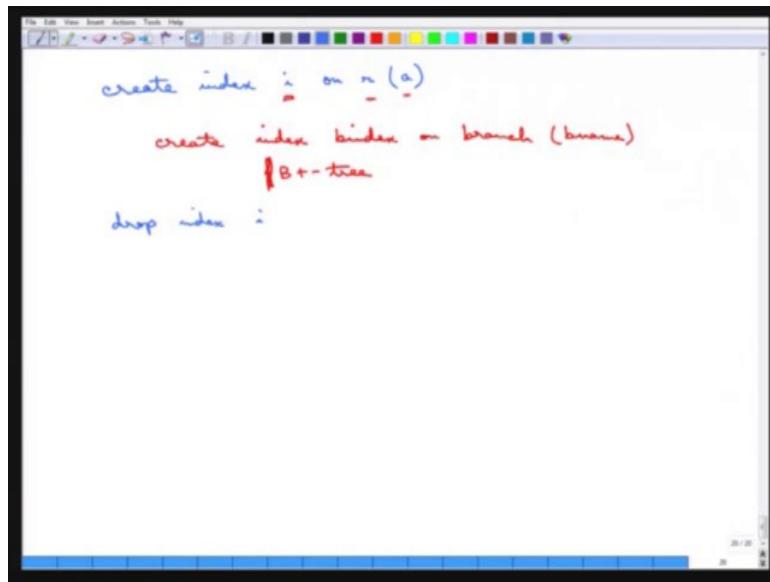
So, what may happen is that when the attribute domain consist of a small number of values. So, a bitmap or a bit vector can be employed to store all such values. So, let us take an example to highlight what I am trying to say, suppose there is a simple table which is gender and grade and suppose this is male, C, female, A, female, C, male, D, male, A. Now, each of this gender as a very small attribute domain, because it contains only male and female. So, is a grade it contains only A, B, C, D and suppose that is the case then one can employ a bit vector to encode all these values.

So, what one can do is to say that the male consists of the following manner, it is 10011 why is it one, because the first one is a male, the second one is not a male, the third one is not a male, fourth one is a male, fifth one is a male. Now, note that one means it is a male, the 0 means it is not a male, it does not say what the value is and for that we require this female coding as well which is 01100 and although this looks like a redundant thing, because it is complimentary, because it has got only two values, it does not it will not happen for A, because A will have this 01001, B none of the values is in B. So, it will have 00000.

Now, note that for example, here it is not A, but it does not say what the value is and one has to go to C to find out that this is actually C and D is simply 00010. So, how is this useful? So, suppose a particular query is find all males, find male students who got D. Now, this is one only need to do the take the bit map of m and do a bitwise AND with bitmap of D. So, that is easy to do and once that is being done, the answer comes out to be...

So, bitmap of M is this one and bitmap of D is this one and once that is being done, this is the one that is highlighted. So, it is the fourth student who is the male, who has got a D. So, bitmap operations are generally much faster and more efficient, because OS packs them in nice byte sized order etc. So, that is why this is more can be faster, to complete the indexing scheme this is how things are being done in an SQL.

(Refer Slide Time: 20:06)



In an SQL, one can say create index i on r(a) so; that means, create the index which is named as i on the attribute a of the relation r. So, for example, one can say create index b index on branch b\_name, generally when only one attribute is being mentioned it is the B+ tree index that is being meant. So, that is the B+ tree index that is being meant and simply one can also say drop index just to delete the index.

That completes the part about indexing. And we studied about hashing, static hashing and dynamic hashing and some tree based indexing.

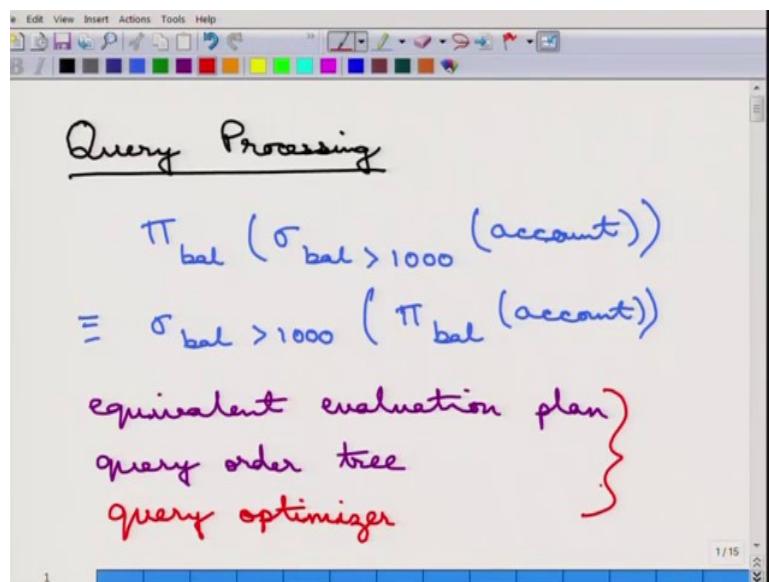
Thank you.

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

**Lecture - 20**  
**Query Processing: Selection**

Welcome, today we will start with Query Processing.

(Refer Slide Time: 00:13)

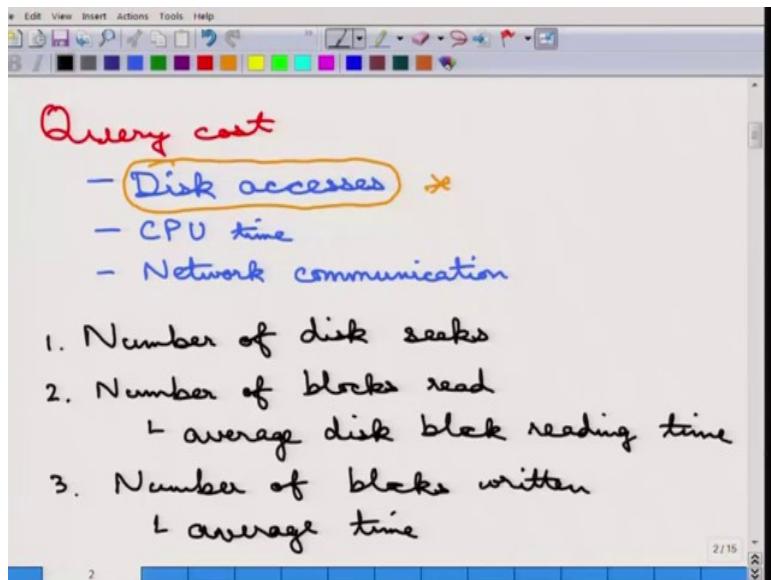


So, what do we mean by query processing is that when a query comes, for example an SQL query comes, how does the database actually answer it. So, the first thing that database system does is to parse the SQL query and it figures out, what the query is trying to do, then it evaluates the different expressions that are equivalent to the SQL query. So, what do we mean by that is that we will go over these examples in the form of a relational algebra, because SQL is finally broken down in the form of relational algebra, so let us for example take this query.

So, we all understand that for this query, the aim of this query to find out the balance of all accounts, where the balance  $> 1000$ . Now, once we write this query this has an equivalent expression of the following manners, so both of these queries are the same and will result in the same output. So, two things are done when a query is first, first of all the query, so all the equivalent action plans are generated.

So, these are called equivalent evaluation plans and for each of this evaluation plan, a query order tree is created and then, the query order tree is evaluated. So, which equivalent action plan will be evaluated which is depending on the query optimizer, so there is a query optimizer, that we will cover in the next chapter. The query optimizer will decide out of the different equivalent action plans, which one is going to be faster. So, based on all of these finally, the query code is generated and that is being executed.

(Refer Slide Time: 02:44)



So, what are the different things that affect the query cost? The cost of running a query can be affected by different things, where of course, as we saw the most important thing is the disk access, as we have been arguing from the last time. The number of random and sequential disk IO is the more important time, also CPU time is a factor and if there is some network communication, then the network communication can also be a factor.

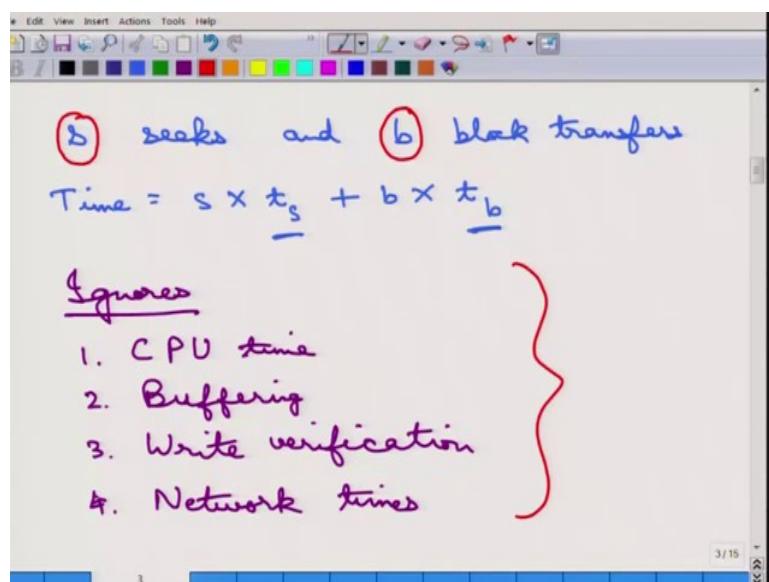
If, however, the database is in the same machine, then there is no network communication, but in general it can be done. As we said disk accesses is the biggest factor, so that is what it is being, so the all the query optimization plans generally optimize only on the disk access. Now, how can the disk access be estimated? Disk accesses can be estimated based on simply the random IO and the sequential IO cost.

So, the number of disk seeks, these include the disk rotation as well and then, the number of, so this number of disk seeks is one parameter. The number of blocks that is read, number of blocks read as a part of this of course, is the average block reading time that is assumed,

average disk block reading time and similar to the number of blocks read, there can be a number of disk blocks written.

Now, generally writing time is more than the reading time, because once the database writes it also checks whether whatever has been written is correct, so there is a reading time hidden into it. And again the average time for each of this block read, so these are the three things that can be done, but very simply it is estimated as the following manner.

(Refer Slide Time: 04:58)



So, suppose there are  $s$  number of disk seeks and there are  $b$  number of block transfers. So, generally we will ignore the reading and writing difference, you will just say  $b$  number of block transfers. So, the query cost, the time is simply estimated as  $s \times t_s + b \times t_b$ , this is the average time for a disk seek, this is the average seek time and this is the average block time.

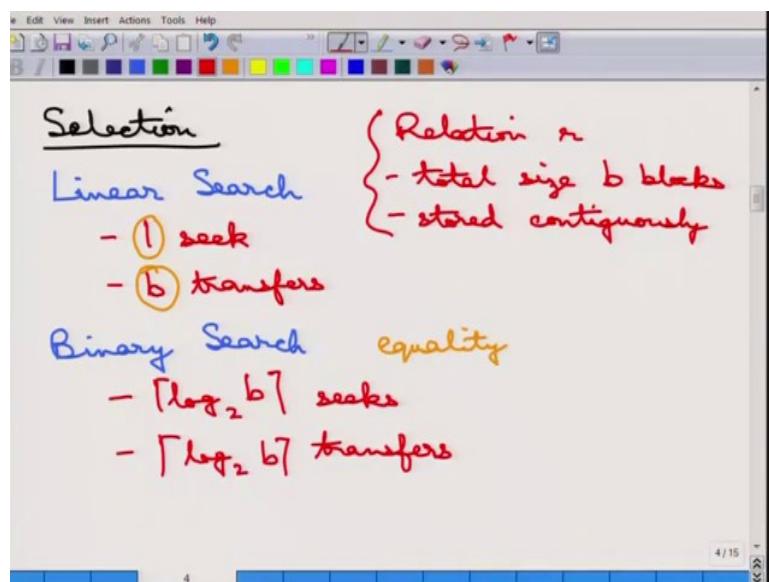
So, the two important parameters that are there is the number of seeks and the number of block transfers. So, all of the query processing algorithms that we will study will try to estimate, what is the number of seeks and what is the number of block transfers, because everything else is a constant and can be figured out there.

So, these does ignores CPU time, so very importantly we should note, what it ignores. In actual system this does play a role, it ignores the CPU time, it ignores the buffering time, as we said the writes or buffer, etc. It ignores the write verification time generally and it assumes

there is no network communication, so network time seeks ignores So, these are the important assumptions that one must understand when while we are analyzing this query cost.

So, once more the only important thing is the number of seeks and the number of block transfers. So, we will go over some of the algorithms in detail and we will try to analyze each one of them.

(Refer Slide Time: 06:43)



So, the first thing is the selection, remember this selection is in the relational algebra sets, not by SQL, so this is the selection as in the relational algebra. So, how can selection be done? So, selection, just to recap, selection for every tuple it tries to asserting whether the tuple can be part of the answer set or not whether the tuple should be selected or not. So, the first thing the very basic algorithm that can be employed to do it is a linear search.

What does a linear search algorithm does? It goes over all the tuples, applies the predicate the selection predicate on it if it passes, then it is fine, it is output; otherwise it is not. So, linear search is therefore, always applicable, no matter what the selection predicate is, it can be always applicable and it scans all the files and test each of these records. So, what is the cost for a relation? So, suppose there is a relation r, whose total size is  $b$  block, so the entire relation takes  $b$  blocks, total size of this thing is  $b$  blocks.

So, what is the cost for linear search? It is essentially one seek, because it needs to go to the beginning of the, where the relation is stored. So, here the assumption is that it is stored

contiguously; the entire relation is stored contiguously. So, these are the assumptions that we will go over, that we will assume for all the algorithms, this is one seek plus  $b$  transfers that is it. So, these are the two important parts about linear search, there is only one seek and there are  $b$  transfers; that is all for the linear search, because it goes over the entire relation and test each record one by one.

The next important algorithm of course, is the binary search. Now, binary search although it sounds more useful, etc and faster, have certain background be needed. So, it is applicable only when the relation is sorted according to the attribute that is being compared. So, the selection it cannot be applied for all kinds of selection predicates, it must be only on the predicate on which the relation is sorted.

Once it is done the cost can be broken down into the following manner, suppose it is just an equality thing, so you might just want to find the equality, so find the record which is equal to a particular thing. Then, the answer to that it is about  $\log_2 b$  seeks and that many transfers, because these are random jumps, so that many seeks for each of these blocks we require a seek and a transfer, so that is the answer.

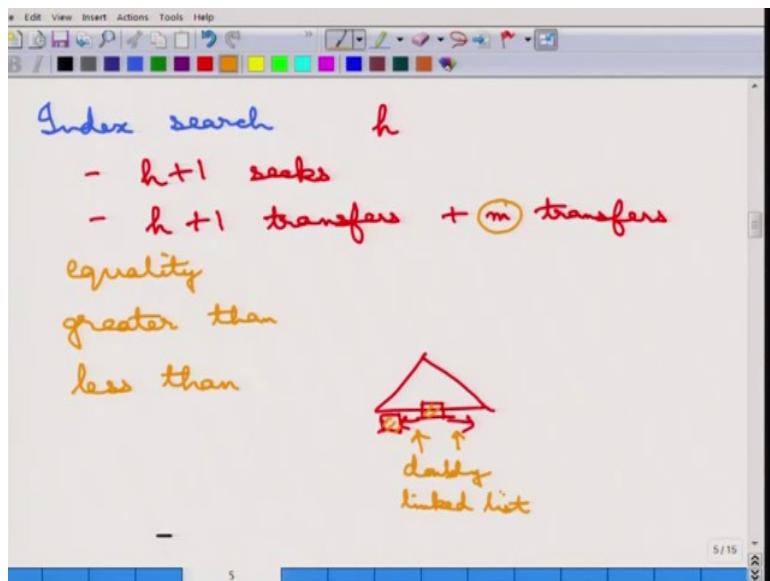
However, different cases need to be handled. So, suppose the equality condition is on a predicate, where all the tuples, where the equality is satisfied does not fit into one block. Then, the next block needs to be searched etc, so this is kind of a conservative estimate, that, that many seeks and transfers are needed this is kind of a average estimate. Moreover, suppose the condition is now changed to not equality, but it is let us say greater than a particular value. Then, what it is being done is that the first block is identified and from then onwards, everything else to the end of the file is being scanned.

So, the first block is identified, then after that everything else is read. So, everything else is read; that means, there were so many block transfers that needs to be done, because these are read sequentially, so that many block transfers need to be added, so that is for the greater than case. For the less than case it is a little bit more interesting, the previous strategy of handling the greater than case cannot be applicable completely on the less than, because the file cannot be scanned in the backward manner.

So, what it is being done is the other way around, the file is scanned from the beginning of it and it goes on the scan goes on till the value for which this is less than is being sort. So, it is

only one seek plus all the block transfers that is needed and these are the two ways that this binary search can handle.

(Refer Slide Time: 11:09)



There is another important kind of search, which is called the index search. So, what exactly is an index search it is that we use that index that way saw earlier the b plus tree and the search is done using the b plus tree index, so it is a searching using that primary index, which is the B+ tree. Now, suppose the height of the B+ tree is  $h$ , so this means there are  $h + 1$  seeks and then, there are  $h + 1$  transfers to go to the correct thing.

So, why you use  $h + 1$  the leaf has to be searched, again this is assuming that the entire answer fits into one block if not there are some  $m$  transfers that are added. So, if the numbers of answers do not fit into one and then, there are  $m$  more transfers needed to be done, but it is essentially equivalent to the height of the B+ tree. So, for these of course, it is assuming the  $m$  is assumed to be 0, so there is no more transfers that are needed.

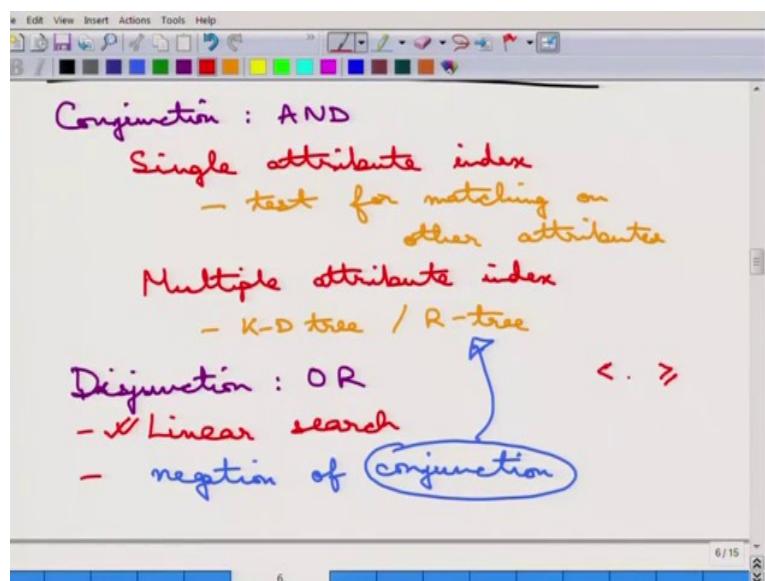
So, this is when the B+ tree is built on the index, for which B+ tree this is for when the B+ tree is build on the attribute, on which the predicate is searched, otherwise it cannot be done. So, this if this is a primary index, then this is all the cost is if this is a secondary index, then there may be more seeks and transfers, because the attributes may be stored in different blocks etc the tuples that match the attributes to be stored in different blocks etc.

So, this is again assuming this is an equality join equality search if it is a greater than search.

So, the predicate is greater than, then what happens is kind of the same as what the binary search did, then the same thing, what the binary search does is being followed. So, the first locating block using the index is located and then, everything from the index is being searched. Remember, the B+ tree generally the leaves are linked as in a doubly link list, so it can go either way and if it is doubly link list the simply the next leaves are followed.

The same kind of a thing is done for a less than if it is a doubly link list the block that matches this particular value is being searched. And since, these are doubly link list it can be traversed in both the ways and that can be done to search the other blocks that match the less than condition. So, this is the sibling leaf pointer this is the property of the B+ tree that these are doubly link list these are between the leaves and these are connected by doubly link list, so this is a doubly link list; that is what the property of a doubly linked list is being used, so this is about the index search. Now, this all we have been assuming that the selection is on a single attribute and then, we have three algorithms the linear search the binary search and the index search.

(Refer Slide Time: 14:25)



Now, suppose the selection is not a single attribute, but on multiple attributes. So, the selection on multiple attributes. So, suppose the condition is a conjunction, so this is an AND condition the conjunction of the AND of selection on two attributes, then there can be different ways of solving it first of all suppose there is a single index. So, there is a index built on a single attribute.

Then everything that matches the first attribute it is being tested for matching on the other attributes that can be done, because this is on and the first one should be matched. So, everything that matches the first test for matching on the other attributes can, then be handled, so that is one way of doing it. The other is an index may be built on a multiple attribute index and this there can be different ways of building it and these are some advanced topics, that we have not covered there can be trees such as K-D tree or R tree etc the index itself is build in multiple attributes and then, that index can be used directly to build that.

So, this is about the conjunction if it is about disjunction or and OR condition, then it is a little the story is a little different, so disjunction OR of attributes, then the story is different. Then, if some of the attributes of the disjunction do not have an index built on it, then there nothing can be done, then the best way of doing it is simply linear scan. So, the first thing is simply linear search; that is the only thing that can be done, if there is no index built on one of these.

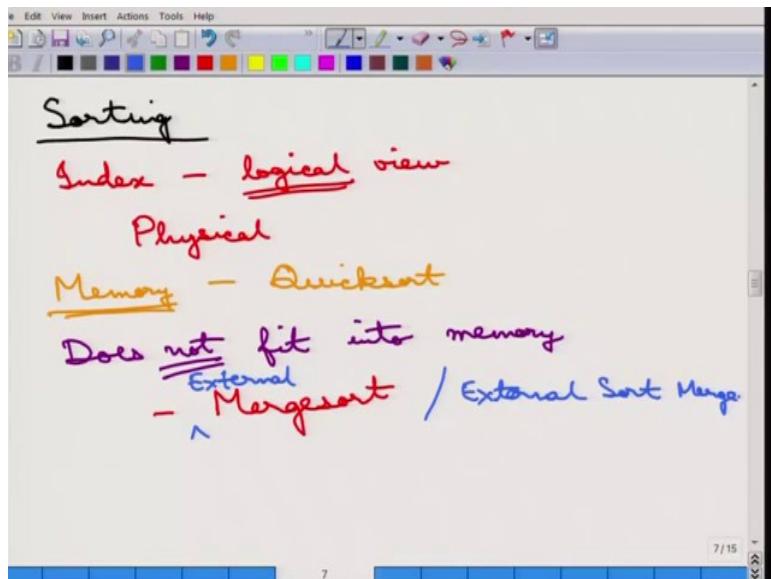
Otherwise, what can be done is a trick that can be done, so disjunction can be written down as a negation of conjunction. So, everything that the conjunction does not pass is part of the disjunction. So, then this conjunction can be done in the manner just like the previous case and everything whenever it is outputting it is the output in the other way around, so the negation of the conjunction can be done. But, mostly it is the linear scan that is the most useful algorithm in place of disjunction.

Because, the negation of the conjunction can be applied sometimes, but not always in the sense that it can be always applied, but it may not be beneficial the linear search may be better. So, there are two ways of doing this, this is one way the other is this way, because negation of the conjunction meaning negating the comparison and negating the comparison is just essentially another comparison for example, if less than is negative then it essentially becomes greater than, so that is about this select.

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

**Lecture – 21**  
**Query Processing: Sorting**

(Refer Slide Time: 00:11)



Next important thing that we will cover is sorting, sorting is one of the most fundamental tasks in Computer Science and databases are no exception. Then, there are different ways of sorting, the sorting is done generally for the display purposes is that, remember that ORDER BY condition in SQL. The relational algebra produces sets, so the sorting has got no valid here, but for display purposes it is being useful.

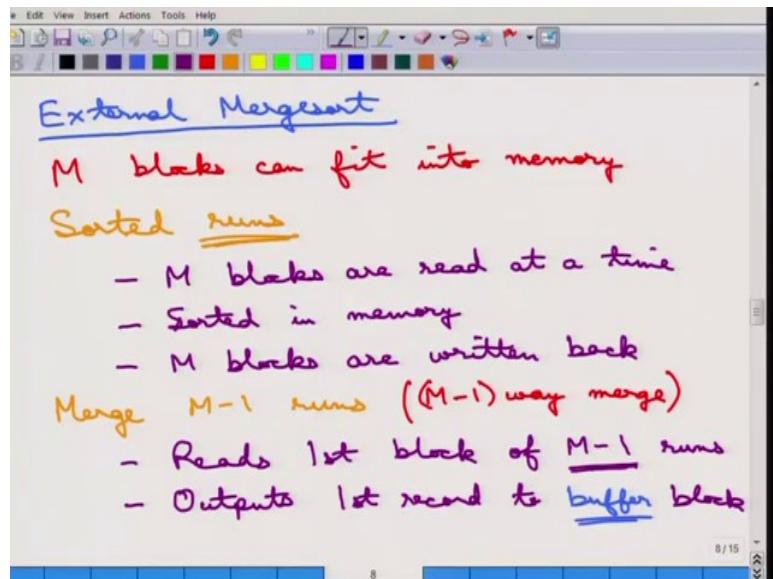
However, it is not the display purpose that is useful, it can be it is also useful, because when the tuples are produced in a sorted order and these tuples are in an intermediate result that can feed to other query expressions later on. If the tuples are produced in a sorted order that may help the query algorithm later. So, sorting is the very, very important operation, that databases also do very regular and what are the different kinds of sorting that we will go over.

First of all understand that, the index, so index generally stores the ripples in a sorted manner, but index only provides the logical view. Because, it has got only pointers to the actual tuples, so it is only a logical view, whereas, sorting here means that we want the physically sorted tuples. So, there is a difference between just outputting the index etc.

So, there are two different ways that sorting can be done, when the tuples fits in memory mean all the tuples fits in memory, then quick sort can be used and I am assuming, all of you know what the quick sort algorithm is. Quick sort can be used when it fits in memory. However, when it does not fit into memory, then quick sort is not a good algorithm and then what is being done is the merge sort algorithm.

But these merge sort is not the basic merge sort that we have studied, this is that merge sort with the twist, this is called an external merge sort algorithm. And we will see what the external merge sort algorithm next, external merge sort is also sometimes called external sort merge, this is the same name that is given to you.

(Refer Slide Time: 02:26)



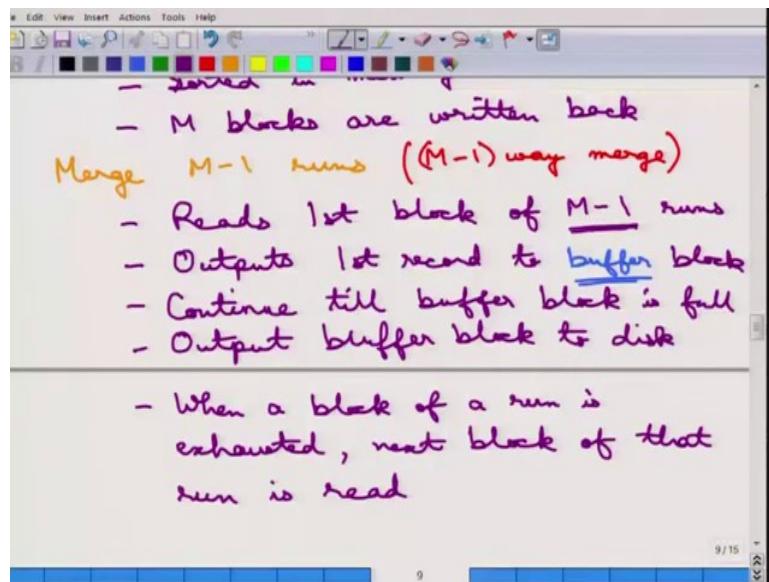
So, we will next study, what the external merge sort algorithm is, so we will study what the external merge sort algorithm is. Now, suppose only  $M$  blocks can fit into memory and the size of the relation is of course, more than  $M$  blocks; that is why this entire relation cannot fit into the memory. So, what is being done is sorted runs are produced sorted runs, now what is a run is the following way it is being done.  $M$  blocks can fit into memory and the size of the relation is of course, more than  $M$  blocks; that is why this entire relation cannot fit into the memory. So, what is being done is sorted runs are produced sorted runs, now what is a run is the following way it is being done.

So, first of all  $M$  blocks are read at a time, then they are sorted in memory using any algorithm, let us say the quick sort or whatever sorted in memory, then these  $M$  blocks are

written back. So, these are the three steps that is done for each run of  $M$  blocks, fine. So, what is being done is  $M$  blocks are read, the first  $M$  blocks are read, then they are again sorted in memory, then they are written back. Then, once that is run, the next set of  $M$  blocks are read, then they are again sorted in memory and then, they are written back and so on, so forth.  $M$  blocks are read at a time, then they are sorted in memory using any algorithm, let us say the quick sort or whatever sorted in memory, then these  $M$  blocks are written back. So, these are the three steps that is done for each run of  $M$  blocks, fine. So, what is being done is  $M$  blocks are read, the first  $M$  blocks are read, then they are again sorted in memory, then they are written back. Then, once that is run, the next set of  $M$  blocks are read, then they are again sorted in memory and then, they are written back and so on, so forth.

So, essentially this one is called a run, this is that so for each run  $M$  blocks is read and sorted, fine. So, this is the first step, after that the merge step comes, which is merge  $M - 1$  runs at a time, this is called  $M - 1$  way merge. So, what this algorithm does is that, it reads first block of  $M - 1$  runs, then it outputs first record that is the best out of the  $M$  blocks to the buffer blocks, so there is a buffer block. So, that is why you see this is  $M - 1$  this is a buffer block; that is being test, so that is the buffer block, which is being done.  $M$  blocks is read and sorted, fine. So, this is the first step, after that the merge step comes, which is merge  $M - 1$  runs at a time, this is called  $M - 1$  way merge. So, what this algorithm does is that, it reads first block of  $M - 1$  runs, then it outputs first record that is the best out of the  $M$  blocks to the buffer blocks, so there is a buffer block. So, that is why you see this is  $M - 1$  this is a buffer block; that is being test, so that is the buffer block, which is being done.

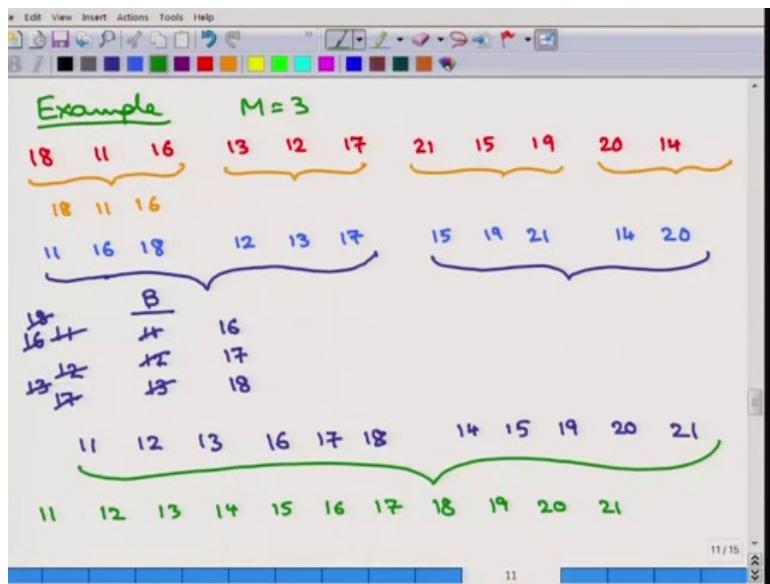
(Refer Slide Time: 05:08)



And once the buffer block is done, continue till buffer block is full, then output buffer block to disk. After this, a new buffer block is done and so on, so forth this exhausted is done. And when a block of a run is exhausted, next block of that run is read and this continues on. So, these may not even finish in the second merge, because  $M - 1$  runs may not be enough, there may be more than  $M - 1$  blocks that are produced in the first stage, so this keeps on going and finally, the output is produced.  $M - 1$  runs may not be enough, there may be more than  $M - 1$  blocks that are produced in the first stage, so this keeps on going and finally, the output is produced.

Now, doing the one important thing to remember is that, this can go on in more than one pass, so this keeps on passing. But, the more important part is that after every pass, some part of the input is processed and the output is sorted up to that point. So, let us take an example to understand the external sort merge algorithm in more detail.

(Refer Slide Time: 06:39)



So, here is an example and suppose here the  $M = 3$ , so only three blocks can fit at a time and this is the data that we need to sort, so 18, 11, 16 etc. So, at the first pass three things will be input to memory, so 18, 11 and 16 will be input to memory and of course, this will be sorted and what will be output is 11, 16, 18. Now, similarly more runs will be created and the outputs will be written back to the disk and these are the outputs that will be created after the first run.  $M = 3$ , so only three blocks can fit at a time and this is the data that we need to sort, so 18, 11, 16 etc. So, at the first pass three things will be input to memory, so 18, 11 and 16 will be input to memory and of course, this will be sorted and what will be output is 11, 16, 18. Now, similarly more runs will be created and the outputs will be written back to the disk and these are the outputs that will be created after the first run.

Now, note that case may happen that in some cases, the last block may be half full, but it does not matter, all it needs to create is to put in a sorted run, so now, note that each of this is a sorted run. Now, since the next state will be trying to do the following is that, since  $M = 3$ . So, the first two blocks first sorted runs will be brought into memory, now not everything will be brought into memory, only the first blocks will be brought into memory.  $M = 3$ . So, the first two blocks first sorted runs will be brought into memory, now not everything will be brought into memory, only the first blocks will be brought into memory.

So, what will be there in a memory is 11 and 12, then there is a buffer block created, the buffer block which essentially output the first out of these to the least out of this 11 and 12.

So, 11 will be output will be read into the buffer block and note that each buffer block can hold up to three of these values. So, 11 will be created, now as soon as 11 is output, the next one from these block which had a 11, next one from these run which had 11, then the next block will be read, which means 16 will be brought into memory.

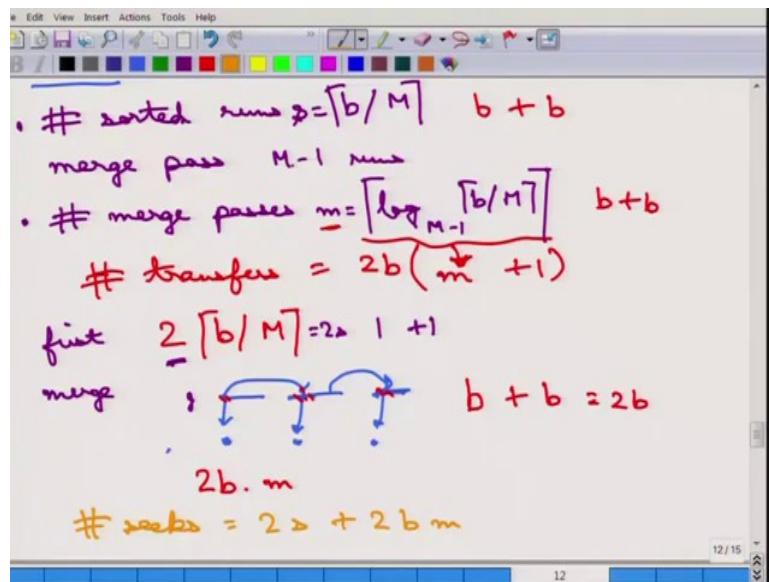
Now, the next comparison will be between 16 and 12 and 12 will be output, once 12 is the output 13 will be brought into memory. Then, out of these 13 will be created and once sorting is done 17 will be brought into memory, more important you what will happen is that at this finding time these buffer block is full, because you can contact three of these things and this is done. So, these will be written back to the disk and the buffer will be emptied out.

So, now, the buffer block can hold from other values, so this is written back to the disk. Then, out of, then the same process is continued, so out of 16 and 17, 16 will be, then put into the buffer block after 16, then 18 will be brought, then 17 will be output, once 17 is erased, there is nothing more left in the blocks. So, nothing else is brought and then, 18 is done, so 18 will be also erase, so then this will be produced as 16, 17 and 18.

Now, in the same manner this two will be sorted and, what will be produced here is the we can simply write it down 14, 15, 19, 20, 21. In the next step this there are only two blocks left know, so these two will be sorted and the total sorted order will be produced 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21; that is being done. So, this is you see the this follow the merge sort procedure, but couple of things taken into account is that only three things at a time is bigger than, because; that is what the memory as the capacity and in the buffer block is used etc, this is the external merge sort algorithm and this is how the things are actually being done.

Now, note in these example two passes when not enough, it requires three passes, because even in the second pass they have us not enough space the first pass is always equal to create the sorted runs and then, after that they may be many passes required.

(Refer Slide Time: 10:31)



Now, one important thing to do is to analyze these algorithms. So, how do we analyze the cost of these external merge sorts, so cost of these external merge sort algorithm is the done in the following manner. Suppose the relation contains  $b$  blocks and memory can contain only  $M$  blocks of course,  $b > M$ . So, the total number of sorted runs; that is produced, the first thing is that number of sorted runs; that is produced is  $\lceil b/M \rceil$ .  $b$  blocks and memory can contain only  $M$  blocks of course,  $b > M$ . So, the total number of sorted runs; that is produced, the first thing is that number of sorted runs; that is produced is  $\lceil b/M \rceil$ .

Now, in each merge pass  $M - 1$  runs are sorted. So, the total number of merge passes, that are required is  $m = \log_{M-1} \lceil b/M \rceil$ , so this is the thing and total number of things; that is required is the following total number of merge passes required is the  $\log_{M-1}$  of these, the total number of sorted runs. Now, during each of these merge passes and the first pass, so these are the two important passes all the blocks are read and all the blocks are written.  $M - 1$  runs are sorted. So, the total number of merge passes, that are required is  $m = \log_{M-1} \lceil b/M \rceil$ , so this is the thing and total number of things; that is required is the following total number of merge passes required is the  $\log_{M-1}$  of these, the total number of sorted runs. Now, during each of these merge passes and the first pass, so these are the two important passes all the blocks are read and all the blocks are written.

So, the total number of block transfers is that, so if all the number of blocks are read; that is  $b$  and all the blocks are written that is  $b$  again, so total number of transfers is simply  $2b(m + 1)$ , this entire number plus 1 for the first pass. So, if we say this is and if we say this is small  $m$ , then this is  $m + 1$ , so; that is the total number of transfers; that is required to understand this. So, now, what is the total number of seeks, so how do we compute the total number of seeks is that  $b$  and all the blocks are written that is  $b$  again, so total number of transfers is simply  $2b(m + 1)$ , this entire number plus 1 for the first pass. So, if we say this is and if we say this is small  $m$ , then this is  $m + 1$ , so; that is the total number of transfers; that is required to understand this. So, now, what is the total number of seeks, so how do we compute the total number of seeks is that.

So, the initial sorted run reads all these  $M$  blocks and there is only one seek, so for the total number of seeks is  $2b/M$ . Because, every time total number of sorted runs is that every time for the first run of the portion of the data is read and then, written. So, for each reading the sorted run requires one seek and then, writing it back requires another seek. So, its twice the number of sorted runs use essentially  $2s$ ; that is, what is the number of things for the first pass this is only for the first pass.  $M$  blocks and there is only one seek, so for the total number of seeks is  $2b/M$ . Because, every time total number of sorted runs is that every time for the first run of the portion of the data is read and then, written. So, for each reading the sorted run requires one seek and then, writing it back requires another seek. So, its twice the number of sorted runs use essentially  $2s$ ; that is, what is the number of things for the first pass this is only for the first pass.

Now, during the merge pass this is the first sorted run think, so each of the merge passes, what is being done is that blocks from different runs may be a read. So, first the one block is read, so suppose there are multiple blocks this is one block, this is one block, where one block. So, first this block is being read, then next block may be read, then next block may be read and so on, so forth. So, the number of this a worst case, is that every time something is output to the buffer block another block is being read.

So, the number of seeks is essentially just copying from one block to another one run to the other, so there may be many, many seeks. So, the number of seeks is essentially all the number of, so for each of these there may be again  $2b$ . So, for each block may be need to be

read and return for each of these blocks there is  $b$  seeks for reading and  $b$  seeks for outputting, because for each of these blocks, again it hopes to some other things and so on, so forth, so each of these  $b$  there is a jump.  $2b$ . So, for each block may be need to be read and return for each of these blocks there is  $b$  seeks for reading and  $b$  seeks for outputting, because for each of these blocks, again it hopes to some other things and so on, so forth, so each of these  $b$  there is a jump.

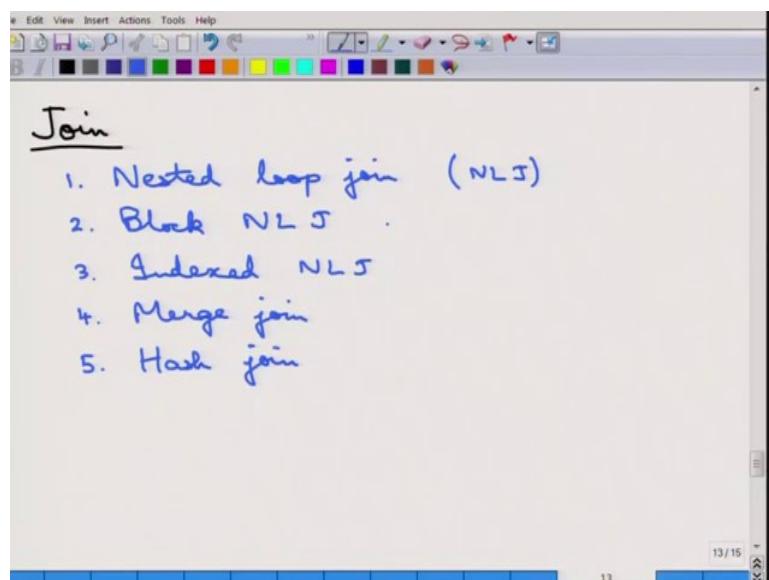
So, there is  $2b$  seeks for merge phase, so the total number of merge things there is that  $2b \times m$  the  $m$  that we use there. So, the total number of seeks therefore, is essentially the total number of seeks in the entire algorithm is simply  $\#seeks = 2s + 2bM$ , so this can be the worst case number of seeks. So, the total number of transfers is  $\#transfer = 2b(m + 1)$  and the total number of seeks is  $\#seeks = 2s + 2bM$  and this is of course, the worst case analysis of the merge sort plan it can be bad, but it generally it is lesser than that, because we have assumed the worst case that every blocks jumps to is the seek.  $2b$  seeks for merge phase, so the total number of merge things there is that  $2b \times m$  the  $m$  that we use there. So, the total number of seeks therefore, is essentially the total number of seeks in the entire algorithm is simply  $seeks = 2s + 2bM$ , so this can be the worst case number of seeks. So, the total number of transfers is  $transfer = 2b(m + 1)$  and the total number of seeks is  $seeks = 2s + 2bM$  and this is of course, the worst case analysis of the merge sort plan it can be bad, but it generally it is lesser than that, because we have assumed the worst case that every blocks jumps to is the seek.

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

**Lecture - 22**  
**Query Processing: Nested - Loop Joins and Merge Join**

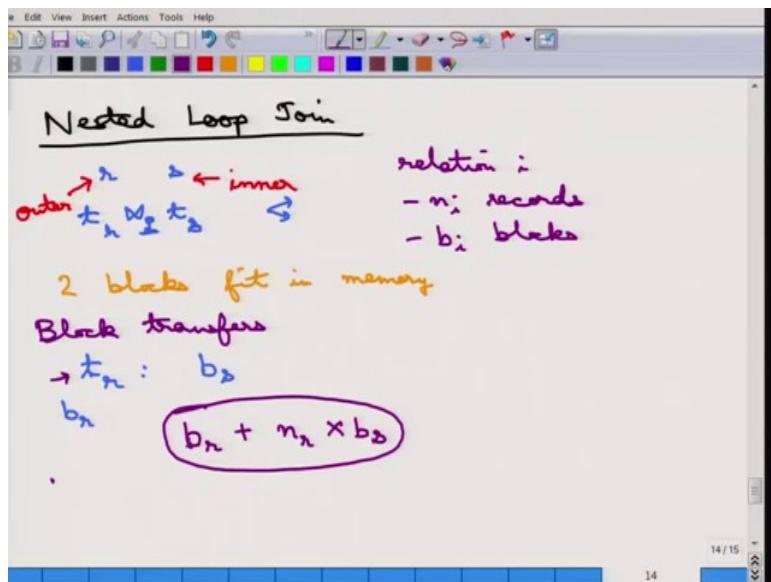
Let us continue with some other kind of algorithms, so we have covered selection we have covered the sorting.

(Refer Slide Time: 00:22)



So, next let us continue with join. Join is one of the very important algorithms in a database and there are different join algorithms, that we will analyze. The first is called the nested loop join, the second one is a version of the nested loop join called the block, so if this is called NLJ, this is the block nested loop join, the third one is called an indexed nested loop join, the fourth one is a merge join and the fifth one is a hash join. So, the second and third are the variations of the first, so we will start off with the first algorithm, which is the nested loop join.

(Refer Slide Time: 01:15)



Now, nested loop join is the simplest algorithm to understand and to apply. What depends, what is a nested loop join does is that it picks up. So, remember that a join is between two relations  $r$  and  $s$ , so what the nested loop join picks up, it picks up tuple from  $r$ , then picks up another tuple from  $t_s$ , performs a join if it satisfies the condition it is the output; otherwise it is not. So, this is a very simple algorithm to understand and very simple algorithm to execute.

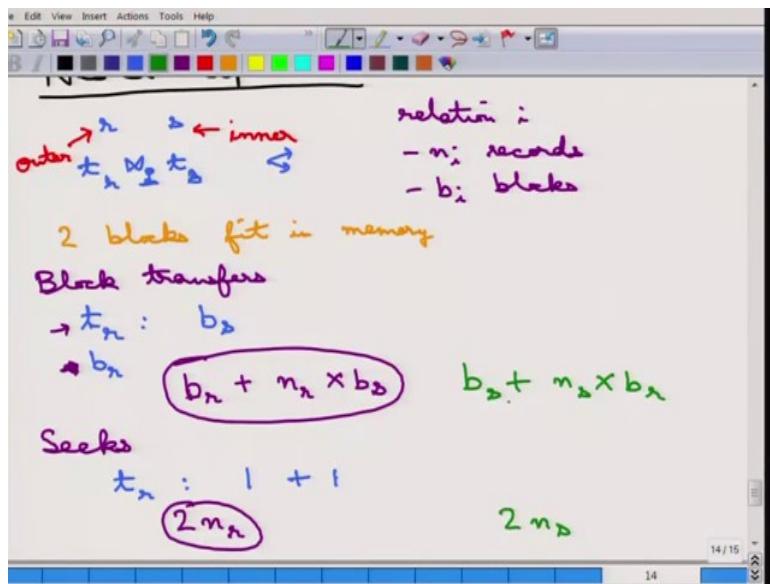
And just like the linear search, this is applicable for any kind of predicate in the join. It does not matter, what the join predicate is. So, what it does is the following, so there is a outer relation, so there is a concept of an outer relation and an inner relation. So, suppose this is the outer relation and suppose this is an inner relation, because this is a nested loop, so this is the outer loop and then this is the inner loop. So,  $r$  and  $s$  and now, suppose let us assume that there are two relations, the relation  $i$  contains  $n_i$  records in  $b_i$  blocks, fine. So, the relation  $i$  has  $b_i$  blocks, which totally contains  $n_i$  records.

And assume, this is an assumption, that no buffering is being done. So, essentially assume that only two blocks fit in memory, but there is of course, the memory has enough space for doing other temporary computational perspectives. So, if two blocks is being done, then what it does is the following, is the number of block transfers, how do we analyze the number of block transfers. The block transfers is done in the following manner, is that every time at record  $t_r$ , so suppose this record is  $t_r$  and this is  $t_s$ .

So, every time a record  $t_r$  is read from  $r$ , all the records from  $s$  are brought. So, that records

every time  $t_r$  is being done, there are  $b_s$  block transfers are being done, because all the blocks from  $s$  are being read. Now, how many transfers are done for  $t_r$ , there are  $b_r$  transfers done, because the records are read one by one. So, the total number of transfers simply is the following, it is  $b_r$  plus this is being done  $n_r$  times. How many times this is being done? This is being done  $n_r$  times. So, this is  $n_r \times b_s$ . That is the total number of block transfers, so  $\#block\ transfers = b_r + n_r \times b_s$ .

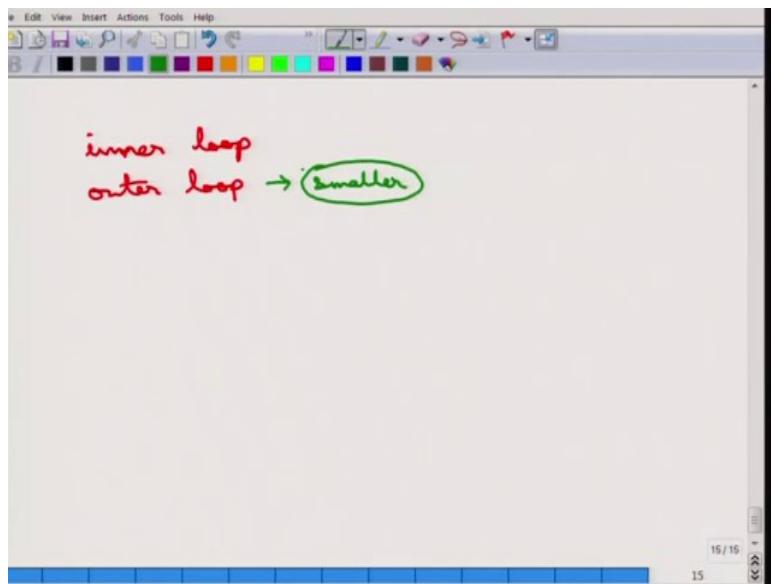
(Refer Slide Time: 04:10)



What is the total number of seeks? How do we analyze the total number of seeks? Again the same principle is applied. So, how do we analyze the number of seeks? So far every time record  $t_r$  is being read, a seek is being done to read the entire relation of  $s$ . But, once that is being read, a seek is again done, such that the next record of  $r$  can be read and this is being done  $n_r$  times, so the total number of seeks is simply  $2n_r$ .

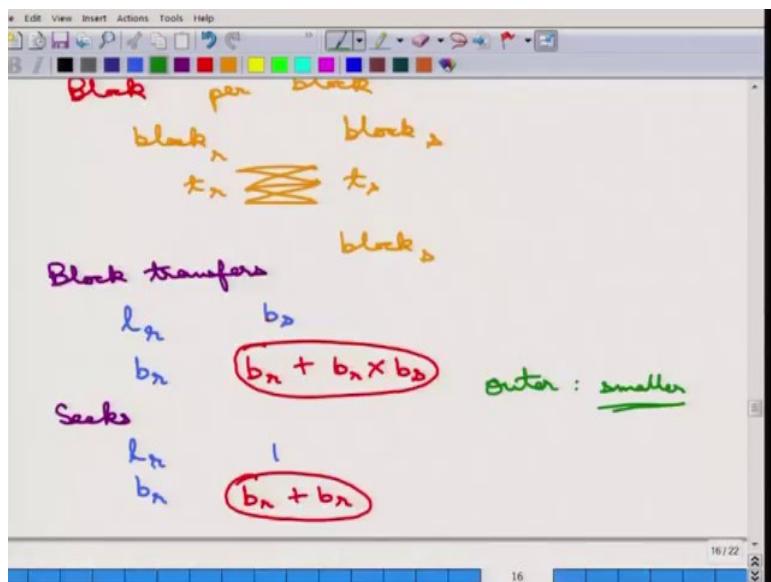
So, this is the nested loop join, let us now move on to the analysis ... There is an inner relation and an outer relation, there is an inner loop and an outer loop. The question is, which one should be the inner loop and which one should be the outer loop. So, if we go back to this analysis ((Refer Time: 04:57)), let us see. So, if  $r$  is being used as a outer loop, then this is the cost, on the other hand if  $s$  is being used as a loop, then the cost is  $b_s + n_s \times b_r$  versus this is  $2n_s$ , so this is the cost if we change the outer loop and the inner loop.

(Refer Slide Time: 04:48)



So, which one is a better cost? Cost wise it is better if the outer relation is smaller. So, the outer loop the outer relation should be smaller, that produces a lesser cost. So, this is the interesting question that which one should be the outer relation, the outer relation should be always smaller, fine.

(Refer Slide Time: 05:40)



So, let us continue with the next algorithm, which is the block nested loop join. Now, one analysis that all of us should be doing is that the nested loop join is extremely slow and it is very much wasteful. Why is it wasteful? Because, even when a block is read from r, it

processes only one tuple and then, throws away the rest of the tuples and it reads s again for that same tuple the next time.

So, that does not make any sense, because once as we have been saying the number of disk seeks is very important and once a disk block is read for all, all the tuples in that block should be utilized, so that is what exactly the block nested loop join does. So, what it does is the following algorithm, is for... So, this is essentially the block version of the nested loop join. So, it does not do it, process it tuple by tuple, it process it per block.

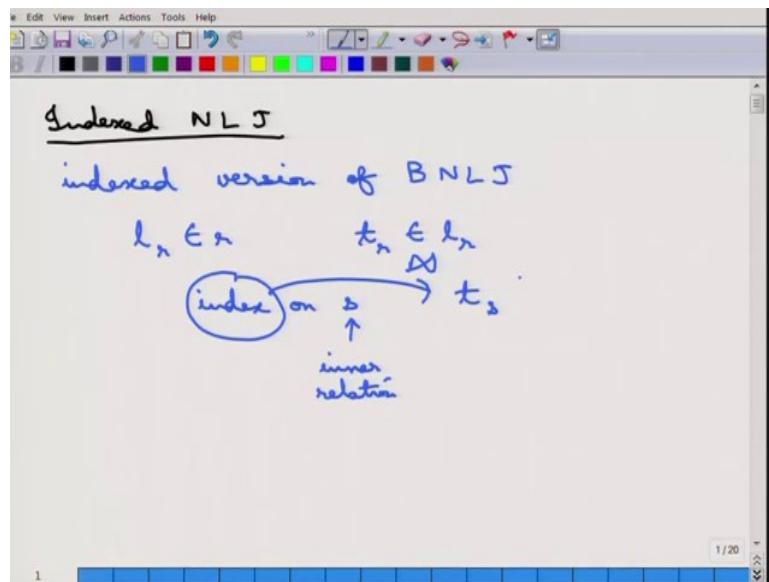
So, it reads a block from r, it reads a block from r, let us say this is block r that is being read, then it reads a block from s. Then, computes every tuple that is part of these two blocks and computes all the joins there and outputs it and then, it is done for the block, then it reads the next block from s and so on, so forth, so that is what the block nested loop join does. So, it is essentially for each block in r and each block in s, for each tuple in r, and then each tuple in s, it does the join predicate and then, it throws away the block r, so it has done completely.

Once more this is applicable for any kind of join and it is just a version of nested loop join, which is optimized for block. Now, what is the cost for this kind of an algorithm? So, number of block transfers, let us try to analyze the number of block transfers that it does. So, every time a block  $l_r$  is being read, so a block  $l_r$  from r is being read, it transfers all the blocks from s, so that it requires  $b_s$  transfers. And, how many times this  $l_r$  is being done? It is being done only  $b_r$  times, not  $n_r$  times as in the last phase you can, it does it block by block.

So, the total number of seeks block transfers, so the total number of block transfers is simply  $\#block\ transfers = b_r + b_r \times b_s$ , that it is being done. Because, this  $b_s$  is being done for each block only and then, the total number of block transfers that is done is  $b_r$ . So, it is simply  $b_r + b_r \times b_s$  and if we analyze the seeks in the following in the same manner, we will see that every time a block  $l_r$  is being read, there are two seeks that is being done. So, every time block is being read there is one seek that is done for s and this is being done for all the blocks in r, so the total number of seeks is essentially  $b_r + b_r \times b_s$ , so that is just  $2b_r$ .

So, this is a much better algorithm cost wise and once more, which what should be the outer relation. The outer relation, should it be smaller or should it be larger, again outer relation should be smaller. So, the smaller relation should be met the outer, this is just like the nested loop join, so this is the block nested loop join.

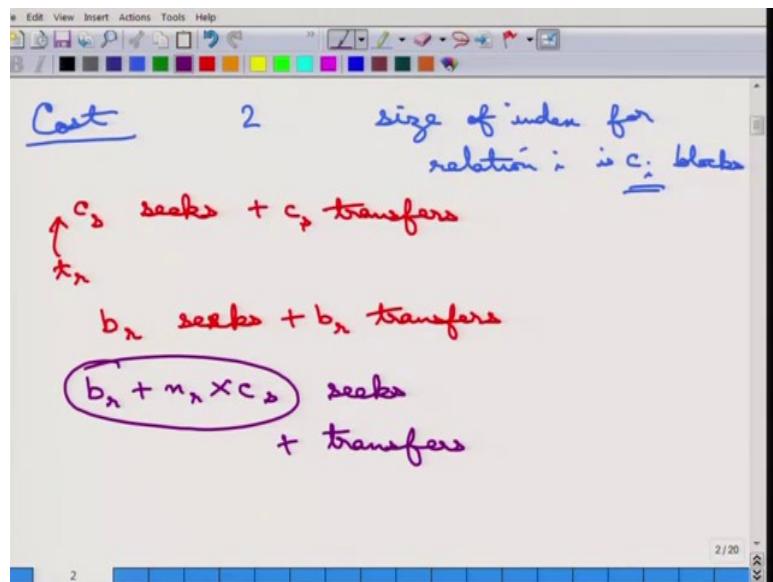
(Refer Slide Time: 09:18)



The next algorithm in this phase is the indexed nested loop join. So, this is just the indexed version of the block nested loop join, indexed version. So, what do we mean by that is block nested loop join, so what do we mean by that is that, for each block  $l_r$  from  $r$  and that for each record  $t_r$  in that block  $l_r$ , there is an index. So, there is an index built on, there is an index on the other relation  $s$ ; that is the inner relation, so this is the inner relation, there must be an index built on  $s$ .

The corresponding, using this index the corresponding tuples are found out from  $s$ , that can join with  $t_r$ . So, using this index the tuples  $t_s$  are found out, that can be joined with  $t_r$ . So, that is the indexed nested loop join algorithm.

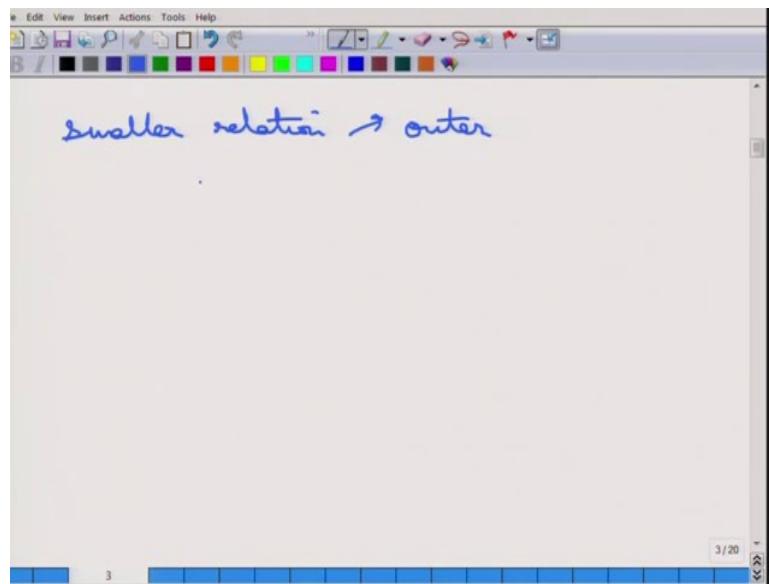
(Refer Slide Time: 10:26)



It is not very difficult to understand, what the algorithm is when let us do the cost analysis of that indexed nested loop join. So, once more the assumption is that only two blocks fit in memory and size of the index for relation i, let us say is  $c_i$  blocks, so this is what a parameter that is. Now, the cost is  $c_s$  seeks and transfers  $c_s$  seeks plus  $c_s$  transfers for selection on an index is for every record in r. This is for every record  $t_r$  in r, this is the cost  $c_s$  seeks and  $c_s$  transfers, because we need to go over all these  $c_i$  blocks etc.

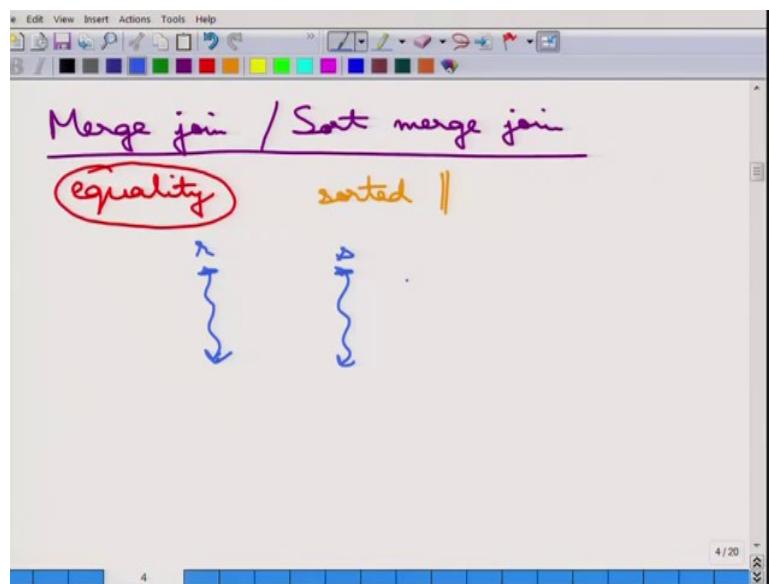
And then, there are  $b_r$  seeks plus  $b_r$  transfers for the blocks in r. So, essentially the total cost therefore, is this is  $b_r + n_r \times c_s$ , the number of seeks and the same number of transfers as well. So,  $b_r + n_r \times c_s$ , so that is the cost of the indexed nested loop join.

(Refer Slide Time: 11:55)



Now, if the index is available for both the relations, this smaller relation should be the outer relation. So,  $r$  should be the outer relation if  $r$  is smaller than  $s$ , because the cost is  $b_r + n_r \times c_s$ . So, that is about the indexed nested loop join.

(Refer Slide Time: 12:13)

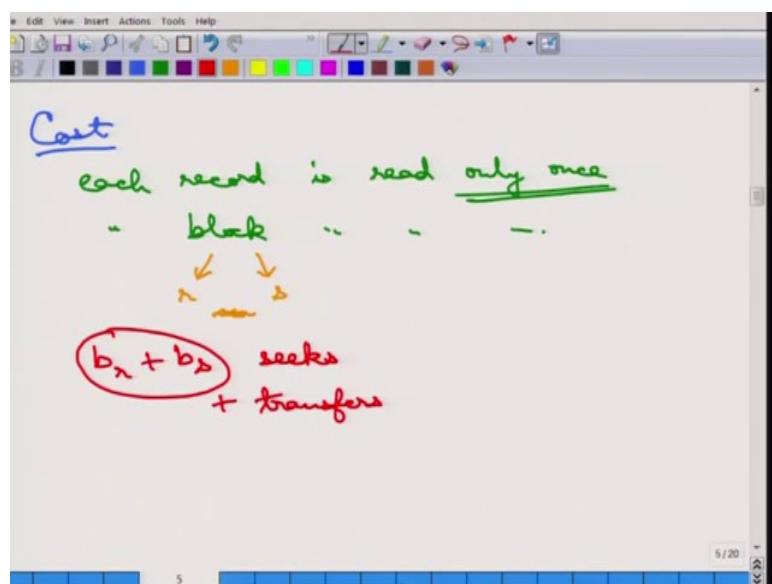


Next we will cover the merge join or the sort merge join this is also known as the sort merge join. So, what does this algorithm do is that this is applicable first of all only when the join condition is equality, so only when this is an equality join, this is very important to understand, otherwise this cannot be applicable. And this is assumed that the relations are

sorted, so if not this sorting cost must be paid. So, this must be first sorted and then, the merge join algorithm needs to be applied.

So, what it does is that there are these two relations, let us say r sorted and s is sorted and just like the merge join in the merge procedure sorting goes it proceeds one block at a time. And then, it tries to see whether it can be joined, if yes, they join it otherwise, they proceed to the next one and so on, so forth, that is all.

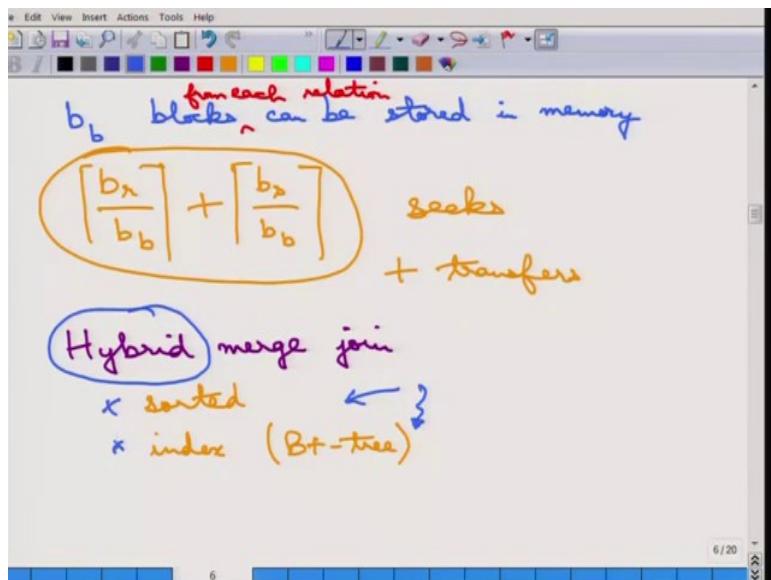
(Refer Slide Time: 13:17)



So, this is essentially just the merge stepping the merge sort that we do. So, what is the cost of this algorithm, the cost of this merge sort algorithm can be analyzed in the following manner is that each record is read only once, because the algorithm proceeds in a sorted manner over the two relations, so that is fine. Therefore, each block is also read only once there is the same thing.

However, the blocks between r and s these blocks can be in r and in s they can be used in any interleave manner. So, essentially there may be that many seeks as the total number of blocks, so the total number of blocks is  $\#blocks = b_r + b_s$ , so that many seeks may be needed. And similarly, that many transfers may also be needed, so  $b_r + b_s$  seeks plus  $b_r + b_s$  transfers may be needed. Because, the blocks can be interleaved and each whenever a block of r is read a block of s it needs to be read and so on, so forth it can be happened, so that is the cost of this merge join.

(Refer Slide Time: 14:28)



Now, one thing that needs to be handled is suppose the memory has space for  $b_b$  blocks, so memory, so  $b_b$  blocks,  $b_b$  number of blocks can be stored in memory at once. So, what does that mean is that instead of only two blocks  $b_b$  blocks from each relation I should say from each relation can be stored in memory. So, then instead of reading one block at a time, what can be done is that  $b_b$  blocks can be read each time.

$$\left\lceil \frac{b_r}{b_b} \right\rceil + \left\lceil \frac{b_s}{b_b} \right\rceil$$

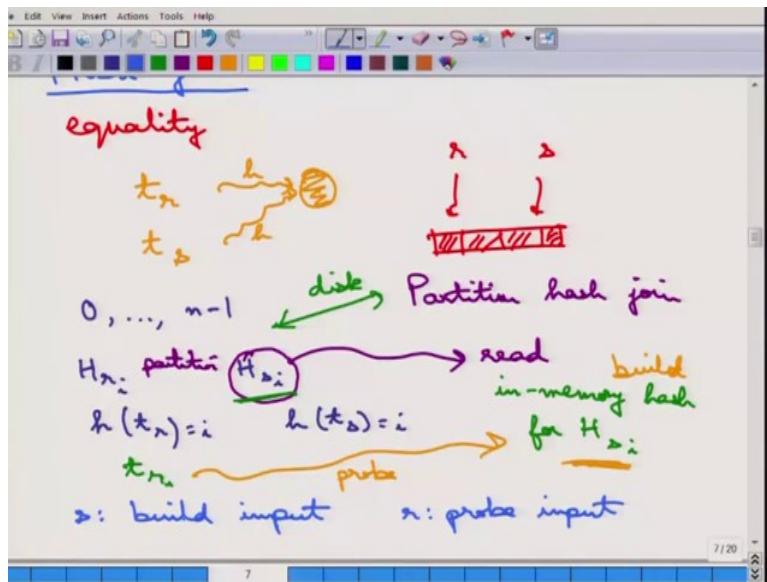
So, the number of passes it requires  $\left\lceil \frac{b_r}{b_b} \right\rceil + \left\lceil \frac{b_s}{b_b} \right\rceil$  is for r and for s and of course, this is the ceiling function. So, these many number of seeks and transfers are needed if  $b_b$  blocks can be stored in the memory. So, this is fine and if the relations are not sorted, then the secondary index can be used or the relations may be need to be sorted. Then, there is another algorithm called the hybrid merge join, where what it does is that the there is one of the relations is sorted, but the other relation has a index built into it the index; that means, it is the B+ tree index that is built into it.

So, what is being done is that, this sorted relation the records the blocks and the records in the blocks are read one by one and then, query is made into the index to retrieve the matching tuples in the other relation. So, that can be done and this algorithm is, then called the hybrid merge join this is called an hybrid, because not all of them are sorted the one is sorted the other is indexed. So, that is why it is called an hybrid merge join.

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

**Lecture - 23**  
**Query Processing: Hash Join and Other Operations**

(Refer Slide Time: 00:21)



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

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

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

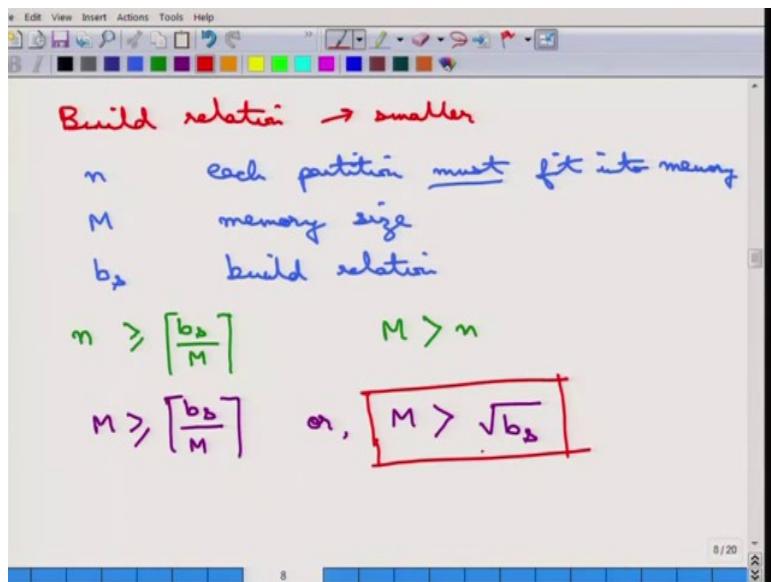
this means that essentially hash function on  $t_r$  is  $i$ , then it goes to the partition  $H_{r_i}$  and hash function of  $t_s$  is  $i$ , then it goes to the hash function of, the hash bucket of  $H_{s_i}$ . Once that is done, the partition  $H_{s_i}$  is read, so these are partitions.

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

So, all the tuples in  $H_{s_i}$ , there is an in memory hash function that is built and then, what happens is that for every record  $t_r$  which is in this hash partition  $H_{r_i}$ . So, this is called the probe, so this is probe to find out the corresponding matching tuples. So, this is a built, because an in memory hash is built and  $t_r$  is probed, so that is why there are some nomenclature that are used.

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

(Refer Slide Time: 04:00)



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

And suppose the build relation has got  $b_s$  blocks for the build relation. Now, what do we want

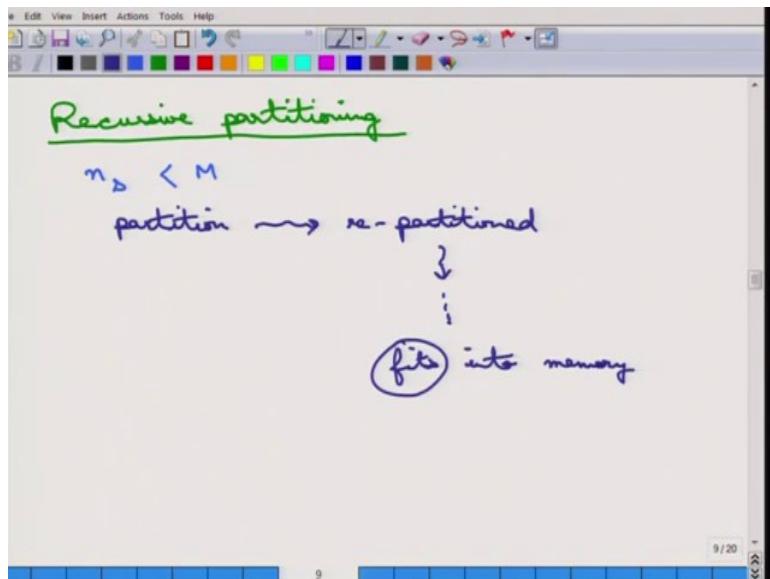
is that, this  $n \geq \left\lceil \frac{b_s}{M} \right\rceil$ . Why is this required? Because, the total number of build relation is M and if each of them fits in M, that gives you the number, that many partitions can be handled and n should be more than that. So, that means, if n is more than that, if the actual number of partitions is more than that, then each partition the size of each partition will be less than what can be fit.

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

$$M \geq \left\lceil \frac{b_s}{M} \right\rceil.$$

So, this is what the condition we were getting or we can write it as  $M \geq \sqrt{b_s}$ . Now, this is one thing that must happen. So, the memory size must be greater than the square root of the size of the build relation.

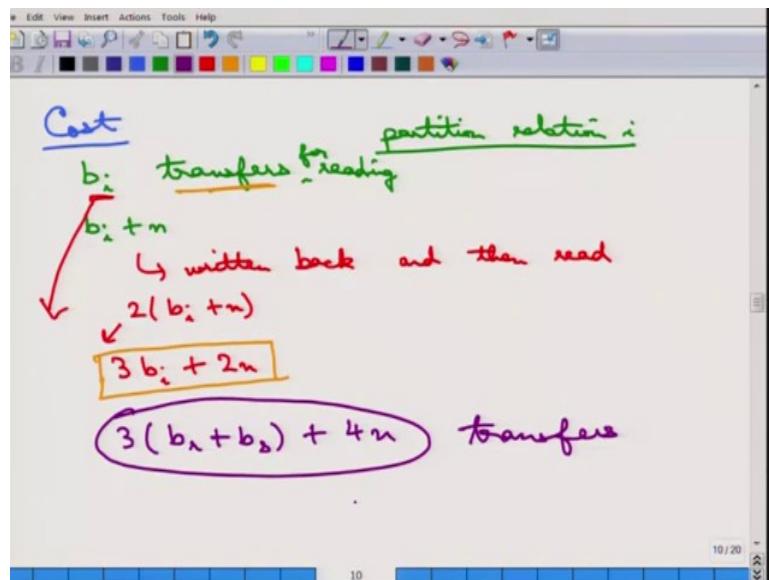
(Refer Slide Time: 06:16)



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

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

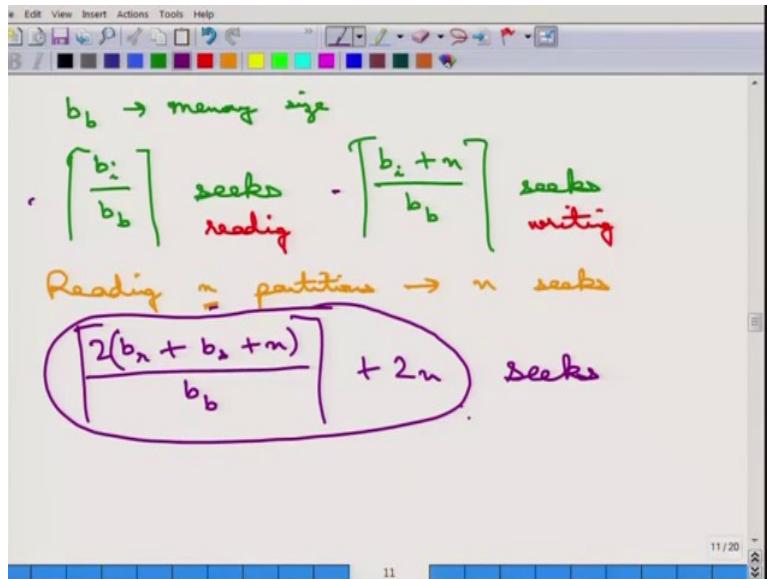
(Refer Slide Time: 07:23)



Now, the important thing is to analyze the cost of this hash join strategy or it may be recursive partitioning etc. So, relation r contains  $b_r$  blocks, so initially it requires  $b_r$  transfers to read relation r. Now, the total number of blocks after the partitioning, so this is trying to partition relation i, so this is what the operation that we are doing, so  $b_i$  transfers read for reading. Now, this may, what may happen is that after the partitioning is done, it may produce  $b_i + n$  blocks, because each of this is just one extra tuple is there, so it fits does not fit into the block etc. So, there can be  $b_i + n$ .

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

(Refer Slide Time: 09:23)



So, what are the number of seeks if we go ahead using the same thing? The number of seeks for reading this, making the partition is that again suppose  $b_b$  blocks fit into memory the memories, so  $b_b$  blocks. So, memories are in the  $b_b$  blocks,  $b_b$  blocks from each relation fits

into memory. So, then there are  $\left\lceil \frac{b_i}{b_b} \right\rceil$  seeks that are required to read all of them, then

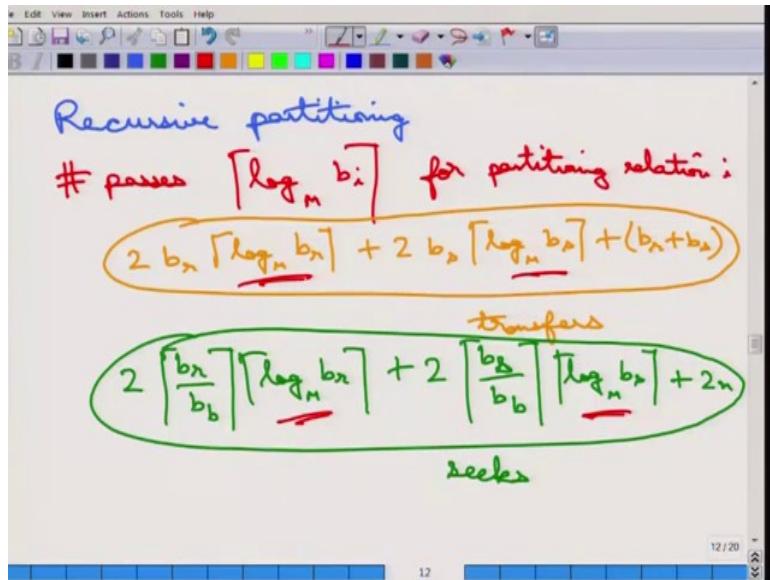
similarly  $b_s$ . So, this is for reading and then,  $\left\lceil \frac{b_i + n}{b_b} \right\rceil$  seeks for writing plus that same kind of thing. So, then reading the  $n$  partitions require  $n$  seeks, because each partition is in a separate thing, reading  $n$  partitions that requires  $n$  seeks.

So, the total number of seeks that is required is your  $\left\lceil \frac{2(b_i + b_s + n)}{b_b} \right\rceil + 2n$ . This is that, these two terms plus of course, this answer plus  $2n$ , this is the total number of seeks that is required for doing this hash join. Now, this is of course, assuming that there is no recursive partitioning, because once you read and then reading this  $n$  partitions is good enough to do the hash join. One important thing also that you should must highlight here is that the output cost is not taken into care of.

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

user in the console or some other manner.

(Refer Slide Time: 11:19)



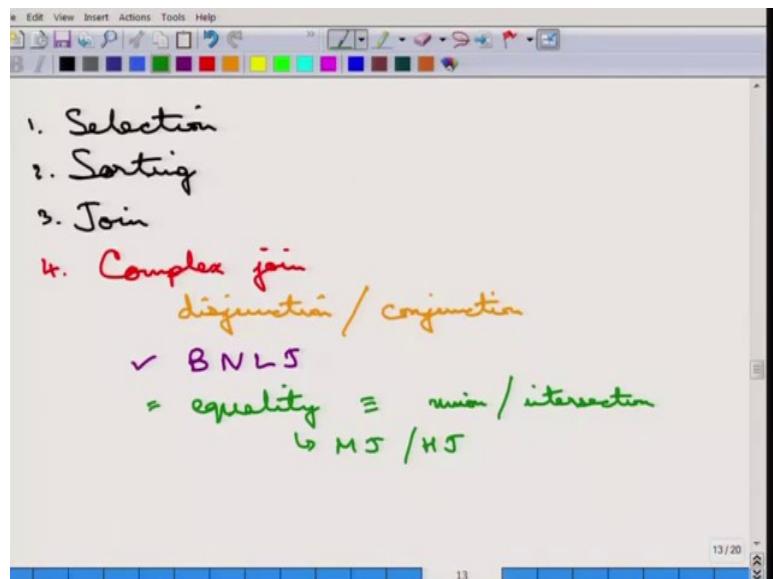
Coming back to the cost of hash join, this the previous case was assuming there is no recursive partitioning. If; however, recursive partitioning is required, then the situation is a little bit different and we can analyze it in the following manner. So, the number of passes, the first thing to enter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

Now, for simplification if we ignore half fill partitions, so total number of transfers therefore, is this has to be multiplied, because in every of this pass the entire relation is read, etc. So, this is of course, and there is a similar term for s and plus your  $b_r$  plus  $b_s$  this is for finally, reading the two partitions all the things. So, there is the total number of this is for transfers, this is the total number of transfers that is required and the total number of seeks that is required will be instead of  $b_r$  this will be  $b_r / b_b$  that many seeks are required and then it is the same formula otherwise. So, plus  $2b_r$ , once more I want to highlight that I am ignoring the half fill partition.

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

If this is the number of passes that cost, for each of these passes everything is read and written, so that is the idea of a recursive partitioning.

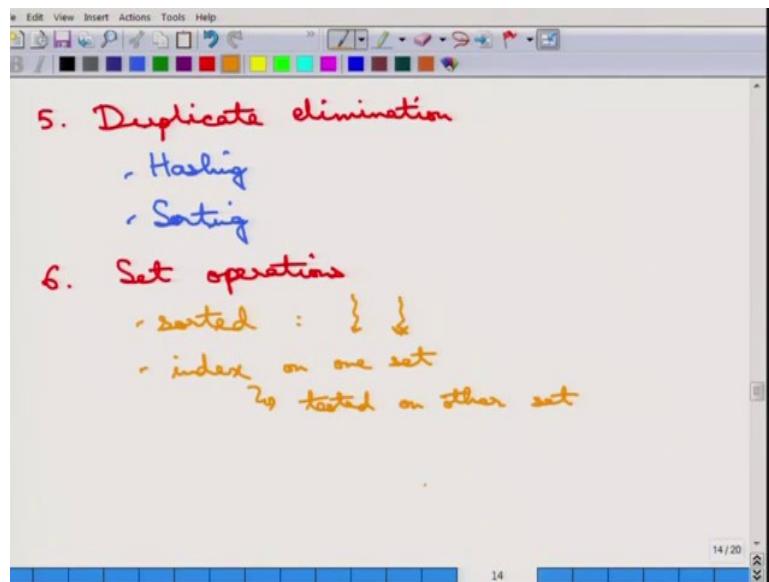
(Refer Slide Time: 13:40)



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

Now, of course, the block nested loop join can be applied, because the block nested loop join does not depend on what the predicate is, it just takes up two tuples and applies the join condition. So, this can be always done if the disjunction and the conjunction is only based on equality, then this hash join or merge join may be done depending on the equality and then those indexes, union and intersection needs to be done. So, depending on whether it is disjunction or conjunction, union or intersection can be done on top of your merge join or hash join, so that can be handled.

(Refer Slide Time: 15:05)



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

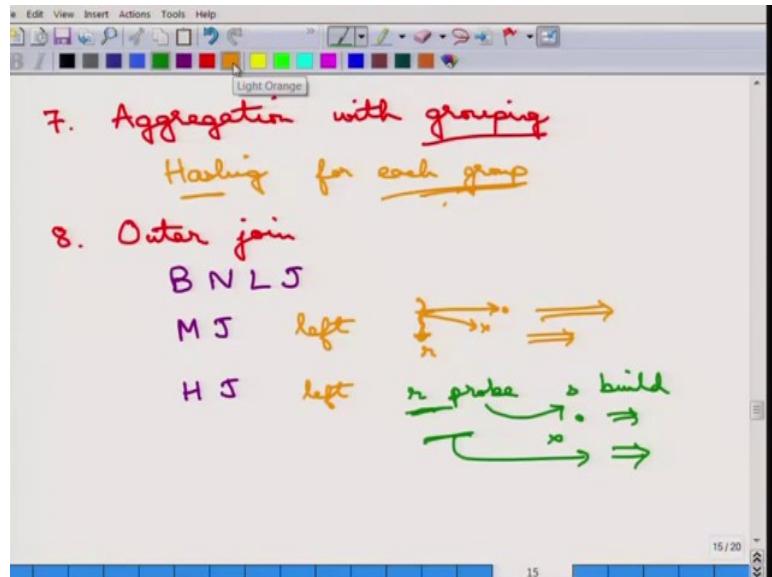
So, how is this duplicate elimination done, well the first thing that comes to mind is hashing of course, hashing can be done. So, all the tuples can be hashed to find out and then whenever the two tuples has to the same location they can be checked attribute by attribute to say if they are done. The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

So, the analysis of duplicate elimination is the same as doing the hashing or the sorting. So, that is the same thing, then there are certain set operations that the relational algebra and SQL needs to handle. So, what are the different set operations that can be done, union intersection difference all those things. Now, if the relations are sorted already then this is easier, because then it just needs to be scanned the two relations or the two sets etc, just needs to be scanned in order and this can be applied.

The other way of doing it is that and index can be built on one of the sets, index on one set and the other can be then tested on the other side. So, you see that this set operations etc is

kind of like the join and the idea is instead of two relations there are two sets, but then the intersection union operations are being done in that almost the same philosophy using the same philosophy, so all of those things can be done.

(Refer Slide Time: 16:59)



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

I mean it can be treated like a relation by itself and then it can be aggregated. So, that is the aggregation with grouping the hashing is required to partition the relation into this group, a very important that we have left out is the outer join. So, whatever we have been talking about so far is on the inner join. So, we have been assuming that the operations are inner join, it is now outer join to recall even if your tr and ts do not agree on the equality, suppose it is a left outer join then the tr must be output as well, so that is the whole use of outer join.

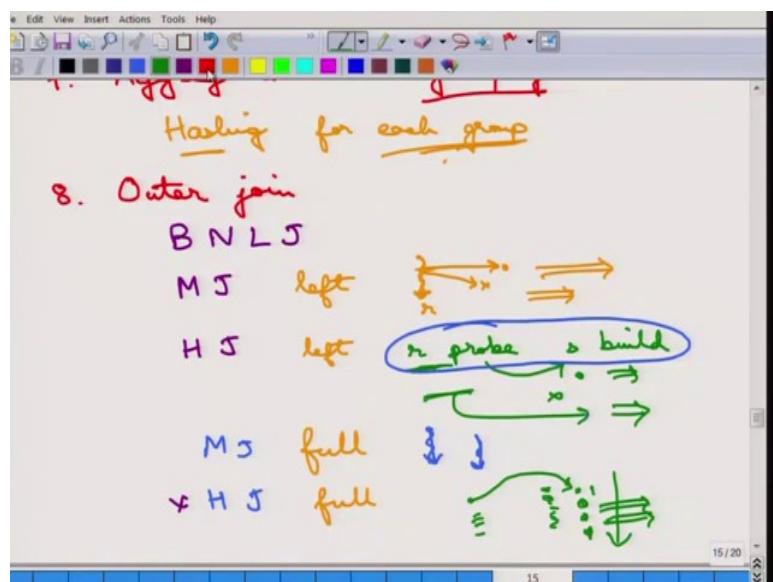
So, how to handle that? So, once more the block nested loop join requires no modification, because this is a most generic algorithm and if the tr and ts match it is output it; otherwise, it is padded with suitable null values etc can be done. For the merge join what can be done is that suppose it is a left outer join then when r is being scanned even if... So, if there is a

matching tuple in  $s$  fine this is output, if there is no matching tuple even then this is output. So, when  $r$  is being scanned it is made sure that everything on the  $r$  side is being output.

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

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

(Refer Slide Time: 20:02)



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

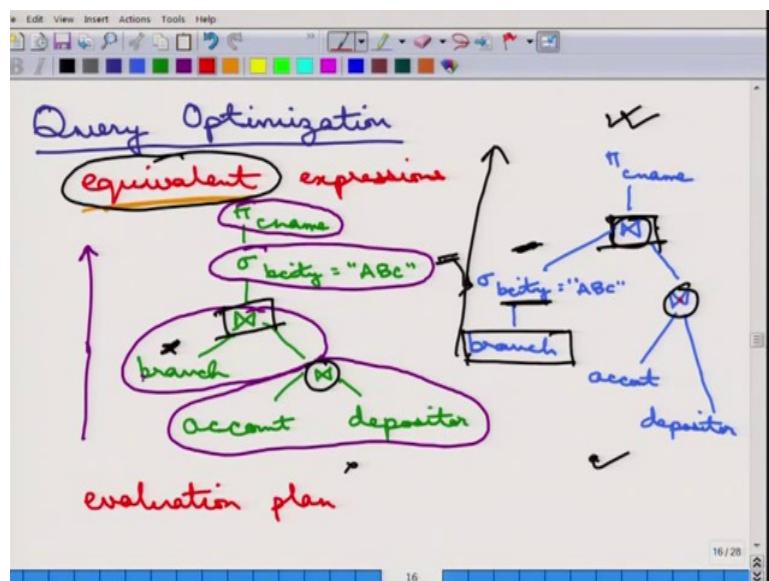
So, if  $s$  is probed one of the tuples in  $r$  probes it then this is marked as 1 and by default it is otherwise 0. At the end of the whole output at pass is taken over this  $s$  an everything that is 0 is output. Why is this? Because, if this is 0 meaning nobody from  $r$  has caught it and  $r$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

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

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

**Lecture - 24**  
**Query Optimization: Equivalent Expressions and**  
**Simple Equivalence Rules**

(Refer Slide Time: 00:15)



Welcome, we will start on the next topic which is on Query Optimization. So, what do you mean by query optimization is that, we have previously seen the different query processing algorithms and for a particular query, different such algorithms can be applied. So, the question is, which algorithm to apply and that is the topic of discussion for this query optimization. Now, first of all we must understand why there can be different ways of evaluating a query, so that is called equivalent expressions.

So, there can be multiple expressions in duration of algebra, which are equivalent. And I am going to give an example right away, but equivalent, the word equivalent means that the moment what the input is, the two expressions that are considered to be equivalently each other. For the same input they must result in the same output, no matter what the input is, it is not for only a specific particular type of input, but for all possible inputs and here is an example of two equivalent expressions, let us see.

So, first of all suppose there is the account; that is joined, a natural join with depositor. Now, this the result of this is then, the natural join with branch, the result of that is then applied

through a  $\sigma$  function, which is say branch city is equal to your ABC it does not some expressions and finally, this is being applied all the customer name. So, what does this expression wants to, what does this expression tries to find, it tries to find out the customer names, names of all customers, who has an account in a branch, which is located in the city ABC.

So, that is what the idea of doing this, now this is one way of evaluating it. Now, what does the equivalent expression mean is that when such an expression is being shown, this is called an expression tree. So, when this is being done, what it says is that first of all an account and depositor, the natural join of an account and depositor is formed. The result of that is then natural join with branch, from the result of that thus the selection of the branch city is being done and finally, this projection is being done.

So, the computation proceeds or the order proceeds from the leaf to the top; that is what is being done. Now, there is an equivalent expression which can be done in the following manner, now let us try to write it down. So, suppose there is the branch, on which the sigma B city is done and that is then, joined with your natural join of account and depositor and finally, the customer name is being projected out. So, first of all, why have this equivalent? The reason is ... So what have it done here?

So, first of all is that this, the branch city expression the selection on the branch city expression that has been brought earlier than the join. So, this is mean, what earlier than the join. So, this evaluation plan goes from here to here, so this is join before the join is join, why is the equivalent is that, the branch city attribute is present only in the branch. So, it does not matter whether the join is joined before or afterwards as far as the correctly is constant.

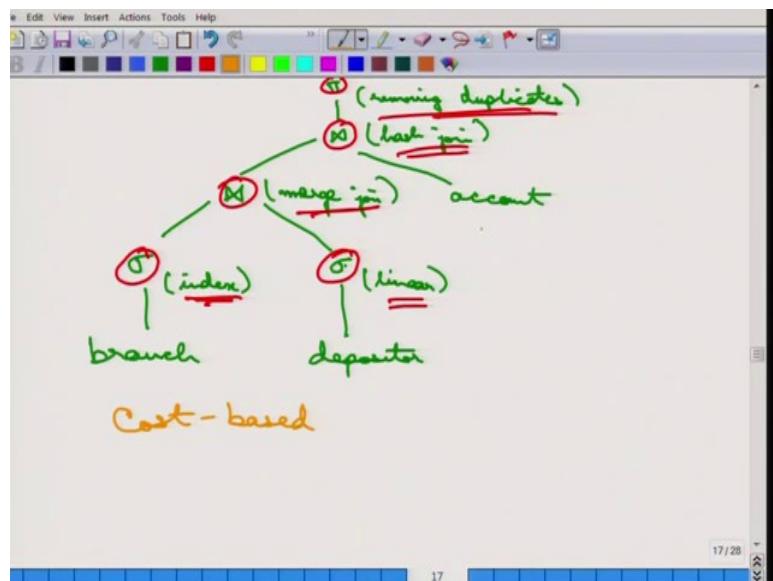
But, one can see, what is the effect of doing this, the output at the end of this selection the output; that is produced here is much lesser, than the output that is produced here. So, this joint is the much faster join much more efficient join than this join, because this is joined with inter branch table this part of course, the same as this, this is joined with the inter branch table while as this is joined with only the branches; that is in the city of ABC, so this is going to be much faster.

So, the whole idea of this query optimization is to figure out all this equivalent expressions and when out of those equivalent expressions once choose the one; that is supposedly faster; that is supposedly better. Now, between these two one can clearly see that this is going to be a

faster or at least as worse as this thing. So, this is going to be the database engine will actually evaluate the query in that this manner and not in this manner, so this is the one that is false.

Now, this is the equivalent expression, then there is something called an evaluation plan the evaluation plan in addition to this equivalent expression also says, which algorithm is going to be used for each other operation. So, for example, this is the natural join and there has been, so there are five join algorithms, that can be applied to do each of these, it can be applied to do this join. So, which algorithm is going to be done, so; that is what is called an evaluation plan. So, I am expressing of the evaluation plan we will be the followings.

(Refer Slide Time: 05:51)



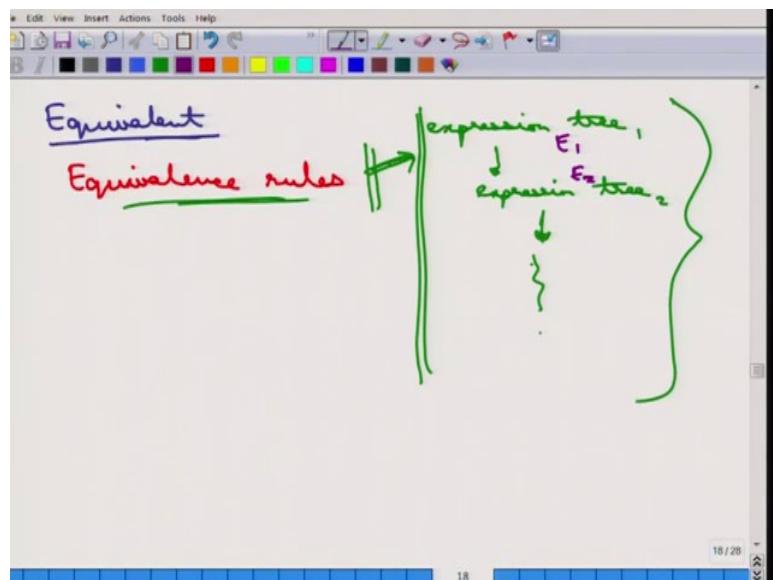
So, let us again write down one example, so suppose this is branch and then, you say this is some  $\sigma$ , which is being done this sigma is done using an index. Then; that is join and this is done with let us say there is another relation depositor from, which some another  $\sigma$  is done this  $\sigma$  is let us say done using a linear scan. Then, this join is let us say the merge join and then, there is another join; that is being done, which is let us say the hash join, because this is done with let us say the account or something.

And finally, there is the projection taken, which is simply the moving duplicates etc, because this new duplicates can be done. So, this is the complete evaluation plan, because in each of these operations the corresponding algorithm is also mentioned. So, that is, what an evaluation plan is it is a little more elaborate than just an equivalent expression. So, the

optimization plan is to do a cost based optimization plan, so each of this evaluation plans the estimate of the cost is being first made and the evaluation plan that results in the least cost estimate is being execute.

Now, note that this is just an estimate, so what may actually happen is there many another evaluation plan, which will result in a better runtime for a particular query, but the estimates were unable to say that and the database engine just follows, what the best estimate is being given.

(Refer Slide Time: 07:43)

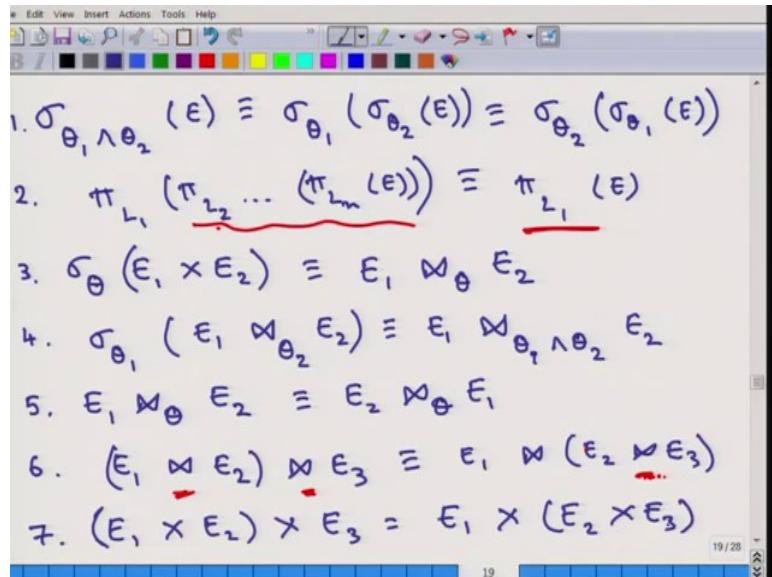


So, coming back to the point of equivalent expressions, let us go over this when can two expressions, we said equivalent as I said two expressions two relational algebra expressions are equivalent, if they generate same set of output for same set of input. And this is been specified by a set of equivalence rules, so we will specify many such rules their and this specifies, which expressions are equivalent. And then, once expression tree is being generated, so from one expression tree, let us say this is expression tree one. each of these equivalence rules is applied.

So, there is some expression inside this expression an equivalence expression rule is being applied to generate another expression tree and this keeps on going till all the expressions and all the equivalent expressions have been exhaust. So, this is the total set of all the expression trees that are generated. And by the generation mechanism these are all going to be equivalent expression tree is, because from one tree to the another only one expression is changed and

the equivalence rules says that the expression one in this tree is equivalent to the expression two in these trees, so; that means these two trees are equivalent, so let us, now go over, what the equivalence rules are.

(Refer Slide Time: 09:12)



The first equivalence rule is the following is that, if this is being done. So, if the  $\sigma$  contains two conditions  $\theta_1$  and  $\theta_2$ ; that is same as applying the conditions that same as doing the two  $\sigma$ 's one after another, this is easy to understand and this is also the same as I mean changing the order of doing these two  $\sigma$ , because and it does not matter, so this is the first equivalence rule. The second equivalence rule is, if there is the series of projections suppose there is series of projections  $\Pi$ . Finally, there is an  $\Pi_{L_n}(E)$ , this is same as just doing the last projection.

Now, of course, for these two happen it must be that  $L_1$  is the subset of  $L_2$  and  $L_2$  is the subset of  $L_n$ , all those things. But, when all these projections it does not make any sense, so one can simply do the final projection. That makes equivalent rule is suppose there is the  $E_1 \times E_2$  and then, there is the sigma place a condition predicate apply to it.

Now, this the essentially the definition of the join, so this is simply the join of  $E_1$  with  $E_2$  and after the cartesian product is done, if the  $\sigma$  if the selection is done; that is the same the joint, using this join something more can be done. So, suppose there is the and there is the joint condition is  $\theta_2$  and then, there is the selection of  $\theta_1$  is being done and; that is same as doing the join itself on  $\theta_1$  and  $\theta_2$ .

Now, whenever I am writing down these equivalent expressions all of them can be proved, but you can at least intuitively follow that these are going to be correct and the proofs if you want you can try. And next one is that the how do you write the join whether even in the left or even it is in the left it does not matter, so it is simply  $a$ , that is a easier one to understand. Then, there is something interesting, so this is the joint and that is finally, joined with  $E_3$  this is equivalent to first doing the join of  $E_2$  and  $E_3$  and then, doing the join and even.

Now, there are some assumption inside this is that  $E_2$  and  $E_3$  can be, so these are natural joins by the way, so there is the there is the assumption  $E_2$  and  $E_3$  do share attributes, so that the natural join can be handled. So, and similarly  $E_1$  and  $E_2$  can also  $E_2$  is essentially the central realization that here attributes with both  $E_1$  and  $E_3$ . So, it does not matter, which natural join is first done.

So,  $E_2$  can go with even one first and then, with  $E_3$  or it can go with  $E_3$  first and then, with  $E_1$ . In a very similar rule is for the cartesian product, which is probably easier to understand, but let me just write it down for completeness, again it is the same idea that it does not matter, which cartesian product is done first.

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

**Lecture - 25**  
**Query Optimization: Complex Equivalence Rules**

(Refer Slide Time: 00:18)

8.  $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3$   
 $\equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$

9.  $\sigma_{\theta_1} (E_1 \bowtie_{\theta_2} E_2)$   
 $\equiv (\sigma_{\theta_1} (E_1)) \bowtie_{\theta_2} E_2$

The previous rules were all unambiguous, it does not matter, what the conditions etc are and now, I am going to write down a little bit more complicated rules. So, this is it,  $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3$ . Note that, these are not natural joins any further, this is equivalent to  $E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$ .

So, couple of things needs to be there is that,  $\theta_2$  must involve attribute from only  $E_2$  and  $E_3$ . So,  $\theta_2$ , it is only between  $E_2$  and  $E_3$ ,  $\theta_2$  involves attributes only from  $E_2$  and  $E_3$ . So, it does not contain any attribute from  $E_1$ , it cannot contain any contribute from  $E_1$ . This is the condition that has to be there, if that is the thing, then  $\theta_2$  can be separated and  $\theta_2$  can be applied first between  $E_2$  and  $E_3$  and then, it can be joined with  $E_1$ , so that is the idea, because  $\theta_2$  does not bother itself about  $E_1$ . So,  $E_1$  can be done later, it can be separated out and a very similar rule is the following.  $\sigma_{\theta_1} (E_1 \bowtie_{\theta_2} E_2)$ , this is equivalent to doing the join. Once

more, here the condition is the  $\theta_1$  involves attribute from only  $E_1$  and it does not involve attribute from  $E_2$ . So, in that case what happens is that, this  $E_1$  and  $E_2$  will go to contain attributes from all  $E_2$ , but since  $\theta_1$  contains attributes from only  $E_1$ , the selection is not going to do any selection on  $E_2$ .

So, the selection can be done that can be done, a pre selection can be done even and then, that can be joined using the condition. So, this is true only when  $\theta_1$  does not contain any attribute from  $E_2$ , because it can be as well applied on  $E_1$  earlier, because it is not going to affect, whether  $E_1$  is joined with something else.

(Refer Slide Time: 02:40)

10.  $\sigma_{\theta_1} \bowtie_{\theta_2} (E_1 \bowtie_{\theta_3} E_2)$   
 $\equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2))$

$\theta_1 \rightarrow^{E_1} E_2$   
 $\theta_2 \rightarrow^{E_2} E_1$

11.  $\pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2)$   
 $\equiv \pi_{L_1}(E_1) \bowtie_{\theta} \pi_{L_2}(E_2)$

$\theta \neq L_1, L_2$  only  $\rightarrow$   
 $L_1 \quad E_1 \quad \times_1$

The next rule is again a little similar rule, this is and suppose, there is a condition  $\theta_3$  and these can be done in the following manner. And I mean waiting you to think for a while, why, when is this going to happen, as you can probably guess, when  $\theta_1$  contains attribute from only  $E_1$  and not from  $E_2$  and when  $\theta_2$  contains attribute only from  $E_2$  and not from  $E_1$ , because as we argued earlier, since  $\theta_1$  does not concern itself about  $E_2$ , so the  $\theta_1$  can be applied on  $E_1$  first.

And similarly,  $\theta_2$  can be applied in  $E_2$  first and those can be then joined. Now, why are these rules important? Because, you see that these are going to produce much faster join, these are going to be smaller relations and in the expression trees that we solve, this was exactly what was being done. So, branch city was brought into  $E_1$ , because the branch city does not bother

itself with anything else. So, this is about these  $\theta$ s, then some projection rules can be written in the following manner.

In this case, what can be done is, one can do the  $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$ , note the similarity of this rule with the previous one. Instead of the selection, there is the projection and again the rule is the same is that, these can be done only when  $\theta$  involves  $L_1$  and  $L_2$  both. So, there are two couple of things here is the  $\theta$  must involve attributes from  $L_1$  and  $L_2$  both.

If the  $\theta$  contains anything else, for example, some other attributes, then that is going to be lost into this. So,  $\theta$  must contain attribute from only  $L_1$  and  $L_2$ , this is only, otherwise it cannot happen, because otherwise those attributes from a lost, when  $\theta$  will be incomplete, it cannot be applied; that is number 1. Number 2 is that,  $L_1$  attributes  $L_1$  concern itself with only  $E_1$  and not  $E_2$ .

(Refer Slide Time: 05:04)

$$\equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_2} (\sigma_{\theta_2}(E_2))$$

$$\theta_1 \rightarrow E_1 \rightarrow E_2$$

$$\theta_2 \rightarrow E_2 \rightarrow E_1$$

$$\text{II. } \Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2)$$

$$\equiv \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$$

↙ θ ⊂ L<sub>1</sub>, L<sub>2</sub> only ✗

L<sub>1</sub> E<sub>1</sub> ✗ E<sub>2</sub>

L<sub>2</sub> E<sub>2</sub> ✗ E<sub>1</sub>

And similarly,  $L_2$  consents itself with only  $E_2$  and not  $E_1$ , this is the same as the earlier, but this is one important condition in addition.  $\theta$  must not be anywhere outside  $L_1$  and  $L_2$ .

(Refer Slide Time: 05:21)

So, a similar equivalence rule can be written, a little bit more complicated equivalence rule can be written in the following manner. We are trying to do the left hand side of this expression with the same as the earlier one and what it does is that, it brings into other attribute. So, there is  $L_1$ , this is  $L_3$ ; note that, there is no  $L_3$  on the left side, but I am going to explain, what  $L_3$  is and then, this can be joined with ((Refer Time: 06:02)).

Once more, there is no  $L_4$  on the other side, but what is being done is the following is that, now  $\theta$  involves  $L_3$  and  $L_4$ , it does not involve only  $L_1$  and  $L_2$ , it rather involves  $L_3$  and  $L_4$ . So, for these  $\theta$  to apply  $E_1$  must contain  $L_3$  and if this expression must contain, so this side must contain  $L_3$  and this side must contain  $L_4$ . So, it must be selected, it must be projected when this projection has been taken.

So, that is what  $L_3$  must be present on this side and  $L_4$  must be present on this side; that is one thing. Other thing of course, since  $L_1$  is separated out from  $E_1$ , it should be the case that  $L_1$  involves only  $E_1$  and not  $E_2$  and  $L_2$  involves only  $E_2$  and not  $E_1$ , number that is 2. Also,  $L_3$ , now you see  $L_3$  is used in only one side here, so that means,  $L_3$  is concerned only about  $E_1$ , it is not concerned about  $E_2$ .

So,  $L_3$  cannot be part of the  $E_2$  and similarly, by similar argument,  $L_4$  is only on these side, which is on  $E_2$ . So, it is concerned only about  $E_2$  and not about  $E_1$ . So, these are the conditions that are required for these two works. Little bit more complicated, but as you can see this is going to be a much faster join, because is a lesser attribute etc, than the original

one, so that is why these complicated rules sometimes used.

(Refer Slide Time: 07:42)

Then, let us go to some of the set operations, the first thing that we are going to say is, suppose  $E_1$ , there is some set operator  $E_2$  is going to be equivalent to  $E_2$  and  $E_1$ . When is this going to happen not for always, this is when this operated is either union or intersection and of course, not set difference. This is probably easier to understand.

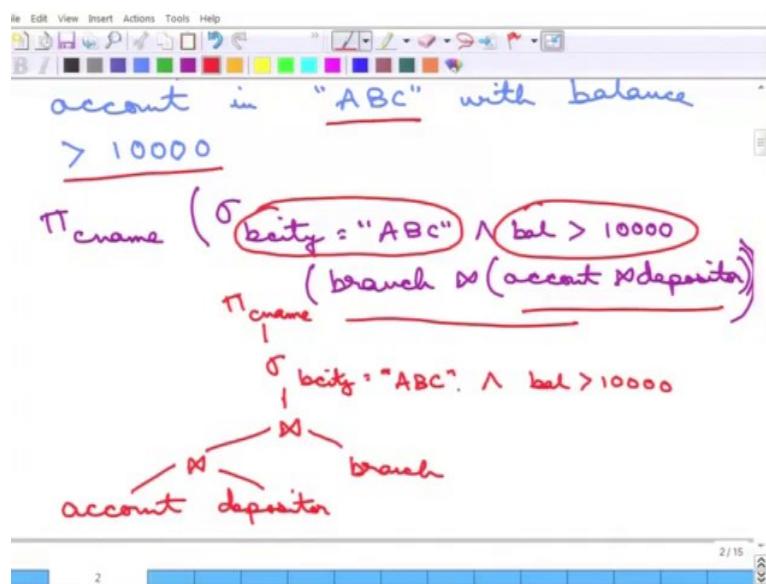
The next one is a more interesting is an order, does the order matter for this operations. So, this is true, when the order is union, this is fine as then the order does not matter is also true, when this is intersection in the order does not matter. But, of course, this is not true, when this is a set difference. The next is on selection using this set of operators. So, if this case, this is equivalent the selection can be separated out and this is true, when the set operated is union or intersection and even set difference, it does not matter.

Because, the selection is going to be on each of them separately; however, an interesting happens, when the following rule is try to be applied in this case, you not doing the selection is just doing the  $E_2$  itself by itself. This is going to work, when the operation is intersection and set difference, but not union and this if you think for a little will, it can understand, why that is the case.

Let us finish off with one last expression is on the projection and if the projection is applied on a set, then it can be separated out only when this operated is union and not set difference

are intersection. Again, the reason is the projection works on the union and the intersection, because the projection may lose some of the values. We will next go over an example of how to use these expressions, equivalent expression rules to evaluate the same query using different evaluation plans.

(Refer Slide Time: 10:19)



So, this is the banking schema that we have been using so far, the branch customer, account, loan, depositor and borrower. And suppose, this is the query that we need to solve, so find names of all customers with we can account in this city ABC and with balance > 10,000. Now, if we want to write it the equivalent relational algebra expression, it can be written in the following manner, I am just trying to solve this.

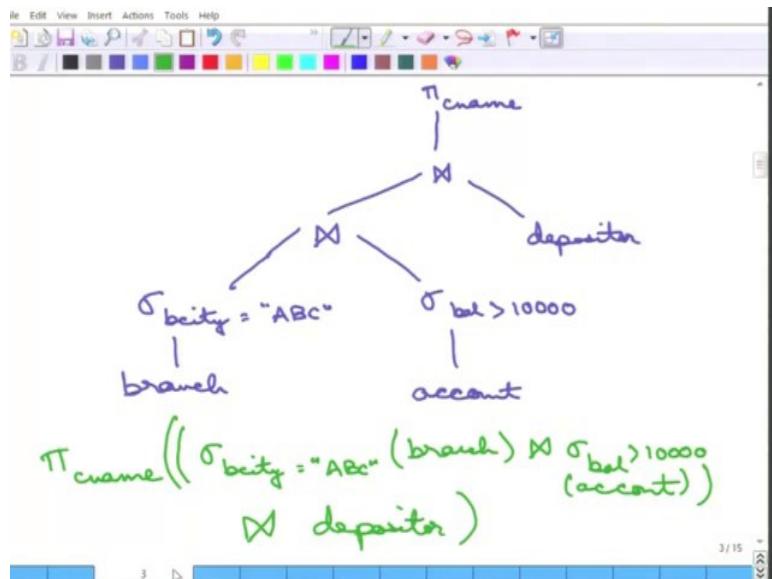
So, this is one way of solving the query, which is first account and deposited are natural join and then, that is taken a natural join with branch. And finally, the two conditions that the account must be in the city ABC and the balance > 10000, these two conditions are applied. Now, if we utilize it, if the relational algebra expression is this way, the corresponding tree can be drawn in the manner that I will show next.

So, the first thing that needs to be done is, the account and depositor are joined, the natural join is done; that is the natural join with branch and then, the selection conditions of these two things the branch city is equal to ABC and balance > 10,000 this is applied and finally, the projection on the cname is taken.

$$\Pi_{cname}(\sigma_{bcity="ABC"} \wedge bal > 10000 (branch \bowtie (account \bowtie depositor))$$

This is the equivalent expression tree for that relational algebra equation. Now, as we have argued earlier, this can be made much faster and for that, we can use this equivalent rules that we saw earlier.

(Refer Slide Time: 12:44)



And after application of different equivalence rules, what will happen is that, the new tree can be written in the following manner. This is the branch on which the selection on the branch city is being done, once more this can be done, because the branch city depends only on the branch. So, the branch city is ABC, then there is an account on which the balance, the selection on the balance is done again this can be done, because the balance comes only from the account table.

This can be then joined, the natural join of balance branch and account can be taken and then, we take the natural join with depositor. And finally, the projection on cname is taken. Now, what is the equivalent relational algebra expression for this, the equivalent relational algebra can be written in the following manner, this is joined with 10000 of account, which is then through these together is then joined with depositor.

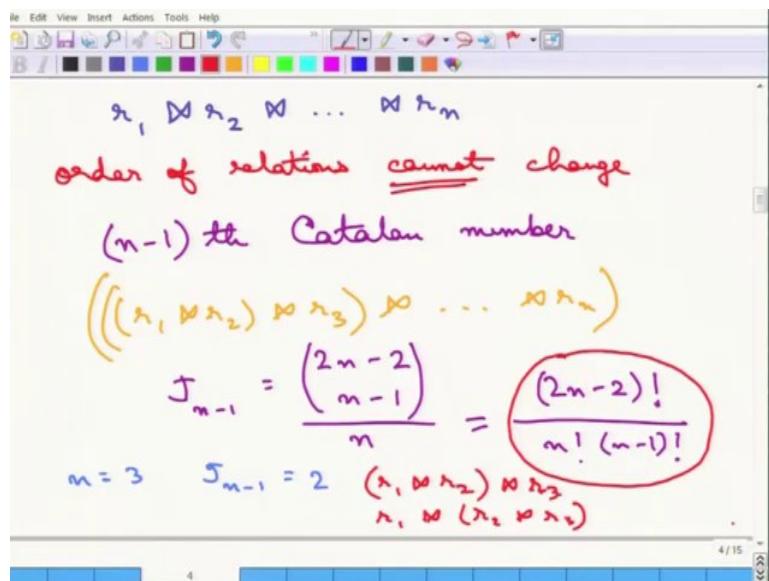
Now, the question is, if we look at this tree, this expression and this tree versus this expression and this tree, why can they be claimed to be equal, because we can successively apply the different equivalence rules that we saw last time to get this tree. So, one tree can be

transform to the other using this. So, that is why, they can be equal and once more, we can argue that the second tree is more efficient than the first.

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

**Lecture – 26**  
**Query Optimization: Join Order**

(Refer Slide Time: 00:18)



Fine, so let us then move to an another important topic of Query Optimization, which is the join order down. What we mean by that join order is the following, suppose the following join, following natural join needs to be computed. So, there are  $n$  relations and the natural join of  $r_1$  joined with  $r_2$  and so on so forth with  $r_n$  needs to be evaluated. The question is which is the best way of doing these joins, clearly  $r_1$  can be first join with  $r_2$ , the result can be then join with  $r_3$  and so on and so forth or  $r_n$  can be first join with  $r_{n-1}$  and that can be join with  $r_{n-2}$  and so on and so forth, so there are many possible ways.

So, what is the best way of doing it? Now before that, the first important question is, what are the number of possible ways. First thing is, if the order of the relations cannot change, so that means, the  $r_1$  and  $r_2$ ; the order of these joins cannot change. So,  $r_1$  cannot be join with  $r_3$  and that cannot join with  $r_2$ , etc. So,  $r_1$  must be join with  $r_2$  or  $r_2$  must be joined with  $r_3$  and that is need to be join with  $r_1$  that is, so if these cannot change, then the total possible ways of competing this join is the Catalan number, this the  $(n-1)$ th Catalan number.

The definition of a Catalan number is precisely this, this is the number of ways of bracketing

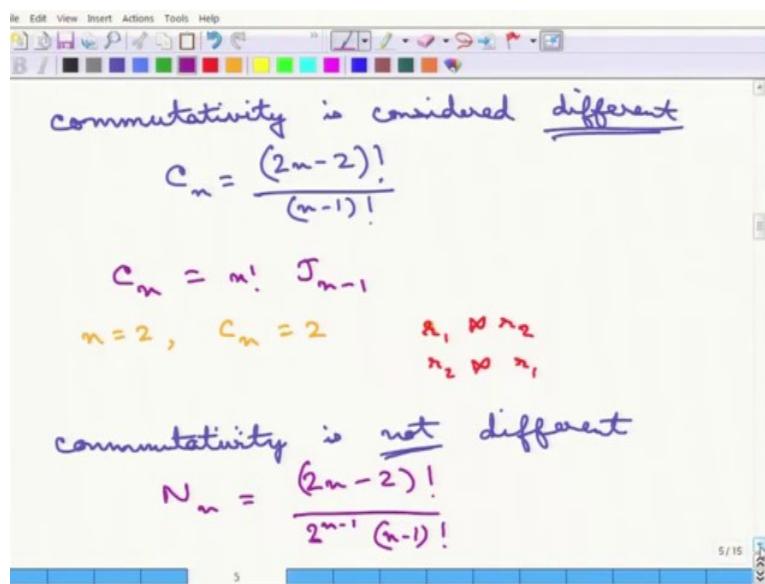
this expression up. So, one way of bracketing this expression up is  $(r_1 \bowtie r_2) \bowtie r_3$ ; that is join  $r_1$  and  $r_2$  and then join with  $r_3$ ; that is then joined and so on and so forth up to  $r_n$ . And one can see that there are multiple ways of bracketing this up and this is precisely what we want to find out and that is precisely the definition of the Catalan number of the order of  $(n-1)$ th, because one of them is already done.

So, there are  $n-1$  ways of doing it, the  $(n-1)$ th Catalan number can be simply written as, so this is  $J_{n-1}$  can be simply written as the following way

$$J_{n-1} = \frac{2^{n-2} C_{n-1}}{n} = \frac{(2n-2)!}{n!(n-1)!}$$

So, that is the number of ways as one can see that, this is quite large for large  $n$ 's, but for example, for small  $n$ , this is not very large. So, for example, if  $n = 3$ , then your  $J_{n-1}$  is only 2. One can evaluate this expression, why is that, because there are only two ways of doing it, either it is  $r_1$  joined with  $r_2$ , that is joined with  $r_3$  or it is  $r_2$  joined with  $r_3$  and that is joined with  $r_1$ . But, in general this is the very, very large number with large  $n$ 's, so it is not very easy to evaluate this expression.

(Refer Slide Time: 03:24)



Now, this is when the order of relations cannot change, another way of doing it is that, when the commutativity is considered different, what does that mean is that, when  $r_1 \bowtie r_2$  is different, is considered different from  $r_2 \bowtie r_1$ ; that way. And  $r_1 \bowtie r_2 \bowtie r_3$  is different

from  $r_3 \bowtie r_2 \bowtie r_1$  is considered different, then the total number of ways of evaluating, this is,

$$C_n = \frac{(2n - 2)!}{(n - 1)!}$$

Now, if you remember, what, that the connection between  $C_n$  and  $J_{n-1}$  or the  $(n-1)$ th Catalan number is that, this is  $C_n = n! J_{n-1}$ . So, what does it mean? Because, it is considered different, so all these  $n!$  combinations are different, so where each of these there are  $n!$  combinations, which are all different, so the  $C_n$  is essentially factorial times the Catalan number.

So, again we can see that the  $n=2$  here,  $C_n=2$ , because and the example is simply  $r_1 \bowtie r_2$  and  $r_2 \bowtie r_1$ , but for large  $n$ , if for  $n$ 's, this is even larger than the previous one, so this is again much more difficult, so that is the other way of thinking about these problem. When commutativity is not considered to be different, so when  $r_1$  joined  $r_2$  is the same as  $r_2$  joined  $r_1$ , is not different, then this number comes down to be, we can write it down simply, this can be written down as

$$N_n = \frac{(2n - 2)!}{2^{n-1}(n - 1)!}$$

So, again the connection between  $N_n$  and  $C_n$  is obvious that  $N_n = \frac{C_n}{2^{n-1}}$ .

(Refer Slide Time: 05:31)

The image shows a digital whiteboard with a toolbar at the top. Handwritten notes are written in purple and red ink. The notes include:

$$C_m = m! / S_{m-1}$$
$$n=2, C_n = 2 \quad r_1 \bowtie r_2$$
$$r_2 \bowtie r_1$$

commutativity is not different

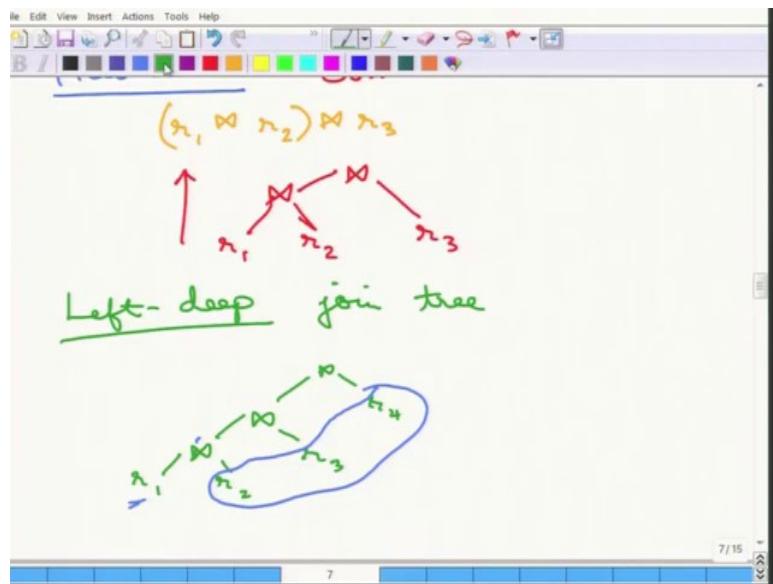
$$N_n = \frac{(2n-2)!}{2^{n-1} (n-1)!}$$
$$n=3, N_n = 3 \quad (r_1 \bowtie r_2) \bowtie r_3$$
$$r_1 \bowtie (r_2 \bowtie r_3)$$
$$(r_1 \bowtie r_3) \bowtie r_2$$

At the bottom right, it says "6 / 15".

The reason is, once you fix the one thing, the other  $n-1$  relations can go anywhere and for  $n = 3$ ,  $N_n = 3$  and examples, the three things are  $r_1$ , for example  $r_1 r_2$  that is joined with  $r_3$ ; that is one. The next one is  $r_1$  can be joined with  $r_2$  and  $r_3$  or  $r_1$  can be joined with  $r_3$ , which can be then joined with  $r_2$ . So, essentially first  $r_1$  and  $r_3$  can be joined or  $r_2$  and  $r_3$  can be joined or  $r_1$  and  $r_3$  can be joined and since commutativity is not different, it does not matter how you write  $r_1, r_2$  or which order ((Refer Time: 06:10)).

So, that is fine; that is the number of join orders, now the question is, why each of these cases the number of join orders to evaluate this too many. And one, the algorithm or whatever system that is trying to evaluate which algorithm to use cannot really look at all of these possibilities. Because, I mean this is exponentially growing with  $n$  and for say  $n = 4$  or  $5$ , this is just the two larger number to evaluate and it does not make sense.

(Refer Slide Time: 06:46)



So, what the database engines actually do is that, they employ certain heuristics. So, in case you are not familiar with the term heuristics, heuristics is a way of approximation of doing an algorithm in an approximate manner. In the sense that, we are not trying to get the correct answer always, it is not the optimal answer that one wants to look for, but it is an efficient way of doing it; that is roughly correct.

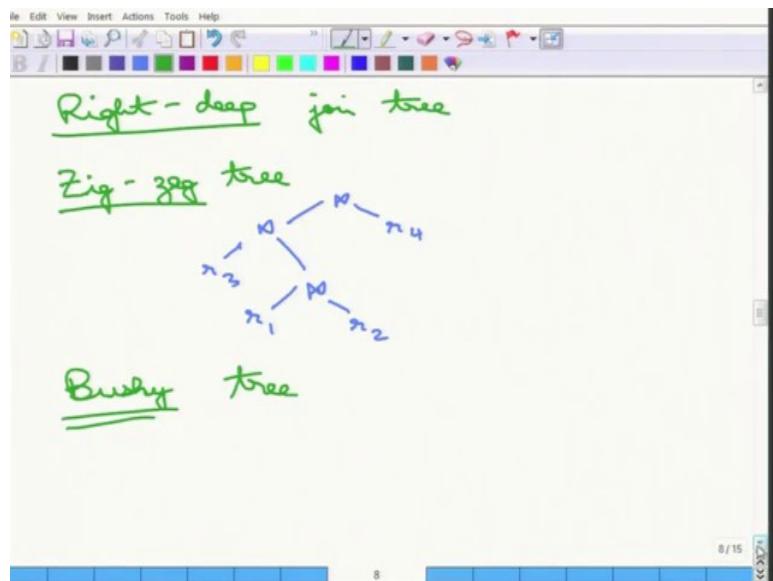
The heuristics is the concept of join tree, now what exactly is the join tree is the way of writing it down, which we order the things are done. So, if considered this example of data three relations  $r_1$ ,  $r_2$ ,  $r_3$  and suppose  $r_1$   $r_2$  is done fast and then, that is joined with  $r_3$ . The corresponding join tree is the following, so  $r_1$  is joined with  $r_2$  and then, that is joined with  $r_3$ .

So, it is obvious on the join tree that how the joins are done in which order, so it gives me this one whatever written. So, the question of evaluating all the join orders is the question of enumerating all the join trees. Now, all the join trees as we have already argued can be too many, so the heuristics that tries to do is to do shorten join trees. It does not try to enumerate joins trees of every manner; but it tries to do only of shorten type join tree.

So, first one is called a left deep join tree. So, a left deep join tree essentially the right side for every internal node, the right side is the single relation and the left tree can be anything, can be a single relation or can be a join. So, essentially the tree looks in the following, it is just the tree looks like, takes the following set, this is  $r_1$ , then  $r_2$ , this is then joined with  $r_3$ , this is then joined with  $r_4$  and so on and so forth. Note that, all these right sides are at the single

relations and the left side is either a single relation or a intermediate join. So, this is a left deep join tree.

(Refer Slide Time: 08:50)

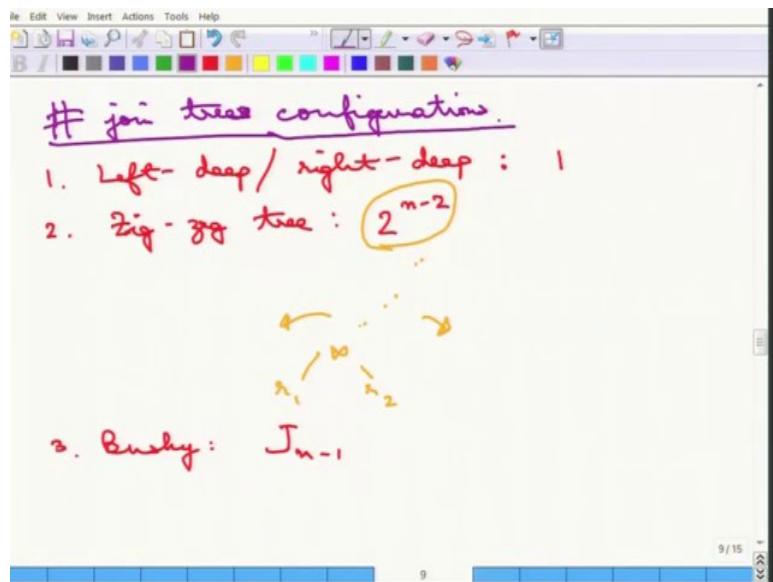


And similar to a left deep join tree, a right deep join tree can also be thought about, where the depth is only on the right side, the left side is only a single relation; that is why it is called a right deep join tree fine. Then, the third one in this space is called a zigzag tree, where it is not specified which side is a single relation, but at least one of the sides must be a single relation.

So, at least one child of an internal node is a single relation; that is called a zigzag tree and an example of a zigzag tree may be the following is that, this is fine and then, let us say, this is on the right side and then, let us say, this is on the left side. So, this is a zigzag tree, now note that, for every join at least one child is a single relation, so that is a zigzag tree.

And finally, there is a bushy tree, it is called a bushy tree, where there is no such restriction and that is essentially the generalized form of a join tree and the number of bushy tree is essentially the all the join orders, so that is the question. So, that is what we have been trying to avoid , so bushy tree is a most general form, but we will not be evaluating.

(Refer Slide Time: 10:03)

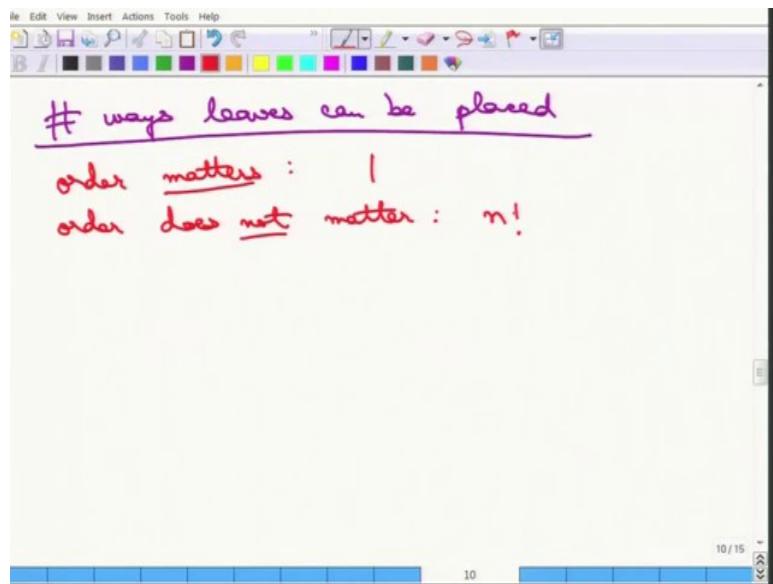


Now, let us try to analyze the number of join trees for each of these kinds of four trees that we are seen. So, number of join trees for the left tree deep or the right deep, it is the same thing, it is not any different. For the left deep and join the right deep, the number of possibilities only is 1, because for left deep, there is only one way, you can do it, only the right side is a single relation and for the right deep, there is only the left side is single relation.

For a zigzag tree, the number of configuration that is possible is  $2^{n-2}$ . Now, let us pause for moment to see, why this is  $2^{n-2}$ , the reason why this is  $2^{n-2}$  is that, so the first two leaves  $r_1$  and  $r_2$  can be placed here and then, for any of the upper joins, the single relation can be placed either on the left side or on the right side.

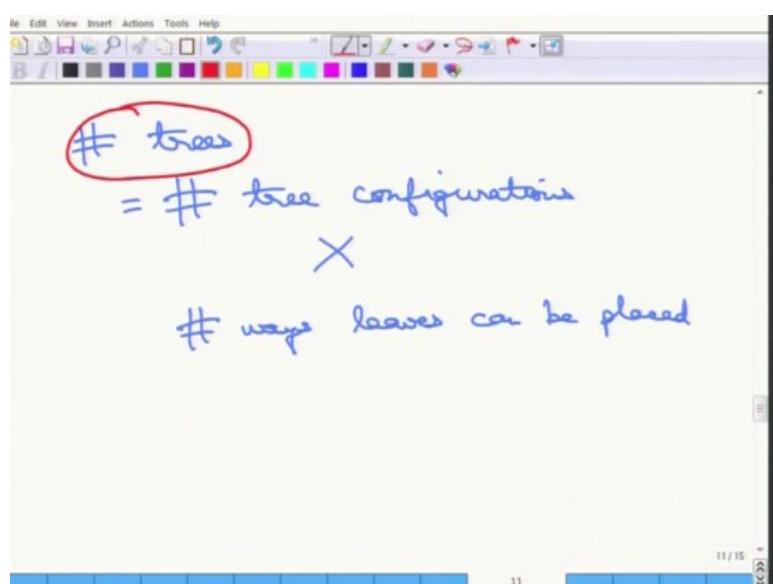
So, for any of these  $n-2$  single relations, it can have two choices left and right. So, that is why, this comes out to be  $2^{n-2}$  and for bushy trees, this way we've already seen and this is essentially the  $(n-1)$ th Catalan number that we have earlier seen. This, the number of join tree configurations, join tree configurations that we can see and the question, then is that, this is just a number of shapes that the join tree can take.

(Refer Slide Time: 11:43)



And the other question, related question is the number of ways leaves can be placed. So, if the order matters, so  $r_1$ , then  $r_2$ , then  $r_3$ , there is suppose the number of ways is only 1, because order matters. If the order does not matter, so if order matter, this is 1, if order does not matter, then need does not matter, vary  $r_1$  and  $r_2$  is placed, etc. So, the  $r_1$  to  $r_n$  can be placed in any permutation; that is why the total number of ways leaves can be placed is  $n!$ .

(Refer Slide Time: 12:22)

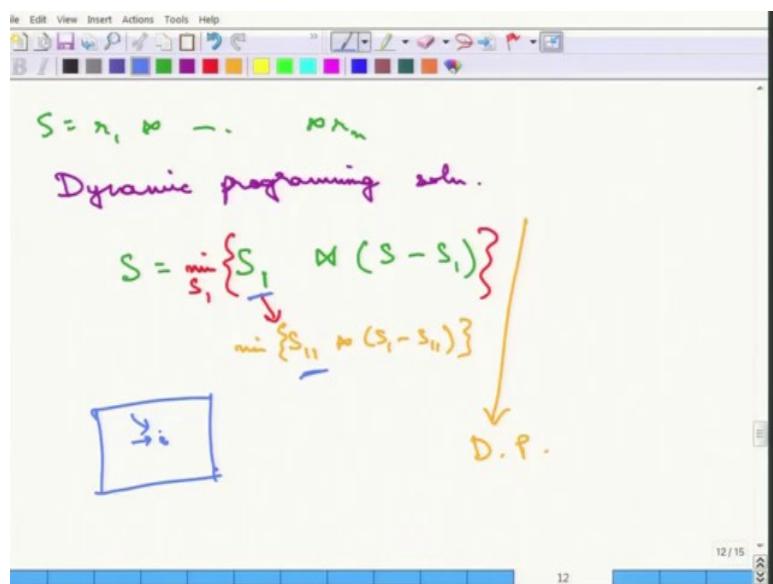


So, if we solve these two, the total number of trees is essentially total number of tree configurations, these are the ways the trees can be generated, times the number of ways,

leaves can be placed. So, now, if we say it is a bushy tree with order matters, we know what is the way, if is a left deep tree with the order does not matter, then we know how many trees can be done etc, etc. So, that is the way of evaluating the join order.

So, now, this is all fine. So, this is just the way of finding the total number of trees and in general, what is being done is that only left deep tree, the practical systems only evaluate left deep trees or right deep trees and may sometimes evaluates zigzag tree is if  $n$  is not very large.

(Refer Slide Time: 13:21)



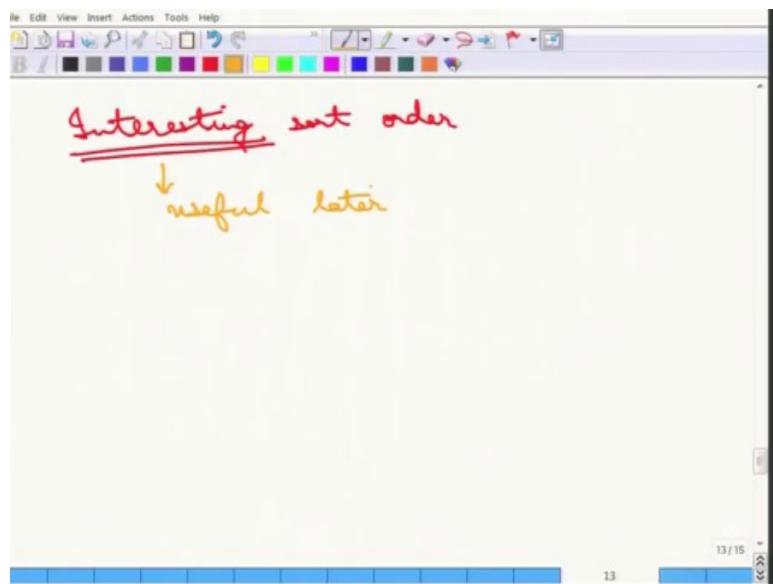
And what is an algorithm for doing it, is that, consider the join of  $n$  things to be done, so  $r_1$  up to  $r_n$ . So, how to actually do it, it has the dynamic programming solution, there is the dynamic programming solution; that is being employed. The dynamic programming solution does the following thing. Consider this is your  $S$  now,  $S$  can be broken up into the following manner,  $S$  can be written down as  $S = S_1 \bowtie (S - S_1)$ .

So, this is the set of all relation is that needs to be join, which is  $S$ ,  $S_1$  is a set of all relation some partition of  $S$ . So, the partition is  $S_1$  and  $(S - S_1)$ . So, these can be written in best way. Now, the question is, what is the best way of evaluating  $S$ , the best way of evaluating  $S$  is to minimized is over the  $S_1$ . So, choose that  $S_1$  that minimizes this configuration.

How is that being done, so how do we find out, what is the best way of doing  $S_1$ ,  $S_1$  can be again broken down into  $S_{11}$  and  $(S_1 - S_{11})$ . Now, what is the best way of finding out  $S_{11}$  is the

minimum way of doing  $S_{11}$  and so on and so forth. So, that is this lend itself to a dynamic programming solution and that is how, this is being done and solutions of each of these subsets essentially  $S_1$  and  $S_{11}$  etc are stored in a table like manner, here is the dynamic programming solution. And then, compute any one of them, it uses everything will surrounded it. So, it uses all the previous computations, fine. So, that is the algorithm for these join order and these as one can see, this is exponential and so on and so forth.

(Refer Slide Time: 15:13)



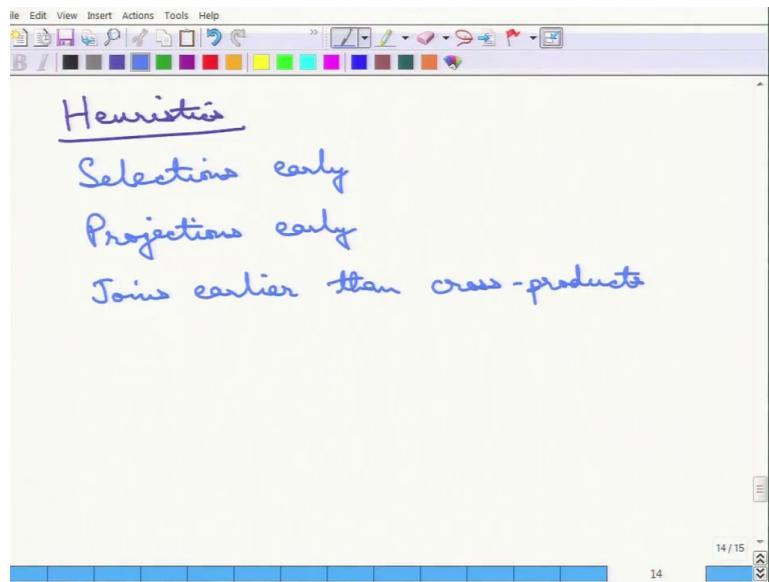
So, the dynamic programming solution, even though, it is faster, but it still there are many, many ways one can do it. There is a related concept called then interesting sort order. So, interesting sort order is that, one can do the certain in any way, but the interesting sort order is that the records are sorted in particular manner; that is going to be useful later.

So, suppose there is an operation after which there is a selection on the particular attribute a. Now, in that old operation, after which the solution is going to be applied, in the old operation, if the output is produced in a manner, where the records are sorted according to a, then the subsequent selection becomes much faster. So, that is an interesting sort order, because it is useful later. And note that the previous operation can be outputting it in any order, but it makes it useful later, if that interesting sort order, where it is sorted by a is produced.

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

**Lecture – 27**  
**Query Optimization: Heuristics and Sizes**

(Refer Slide Time: 00:10)



Heuristics are implied not just in sort order and join tree, it has been used in many such cases. So, some of the heuristics that the database engine performs to do better query optimization is the following is that, first thing, the one of the thing it does is that, selections are performed early. Why is that? Because, once the selection is done, the size of the relation is reduced considerably.

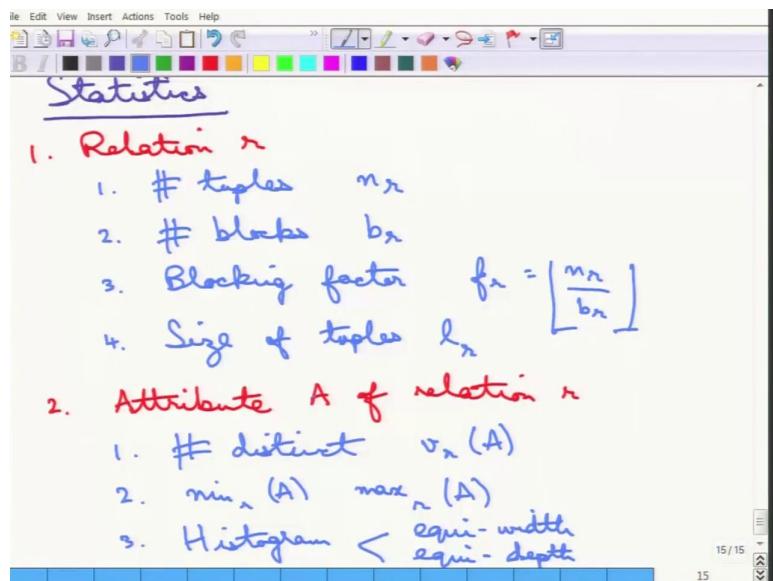
And so, whatever operation that is done on the relation, it can be done in a faster manner; that is one thing. Projections are also, if they can be done early, they are done early, again projections can be done early, because that reduces the number of attribute; that reduces the size of the relation in general and again, that can be beneficial later, again that is mostly beneficial later.

Then, joins are generally done earlier than cross products, if there is a cross product versus a join, then join is generally done earlier. The reason is easy to understand, joins are essentially cross products and then, a selection. So, joins can early produce smaller relations than the cross products. This is one kind of thing, these are all kind of heuristics that do not depend on

the semantics of the query, this is just using the equivalence rules under doing it.

The optimizer can also use certain semantic optimizations. For example, suppose the query is find all employees, who earns more than their manager. Now, a database engine can simply go ahead and try to solve this query by joining the employee salary table with the manager salary table, etc. But, if a semantic knowledge is being built in, that the employee can never get salary more than her manager, then this query uses that semantic rule and simply returns a null set, simply returns an empty set. So, that is an example of a semantic optimization. So, it uses essentially the domain knowledge and the constraints that are built into the database directly and not this equivalence rules.

(Refer Slide Time: 02:27)



So, the query optimizer to do all of these things to choose which algorithm is better, etc, stores a lot of statistics about the relations, about the attributes, etc. So, what are the statistics that is stored? So, for each relation  $r$  it stores the following statistics, so this is about relation  $r$ . First it stores the number of tuples, so number of tuples in  $r$ ; that is being stored, then the number of blocks that  $r$  is required.

So, this we have seen example, this is  $n_r$  and this is  $b_r$ ; that it store, it also store something called a blocking factor. So, the blocking factor is essentially the number of tuples that fit in a block. So, this is the blocking factor, this is the derived statistics, one can say, this is

$$f_r = \left\lfloor \frac{n_r}{b_r} \right\rfloor$$

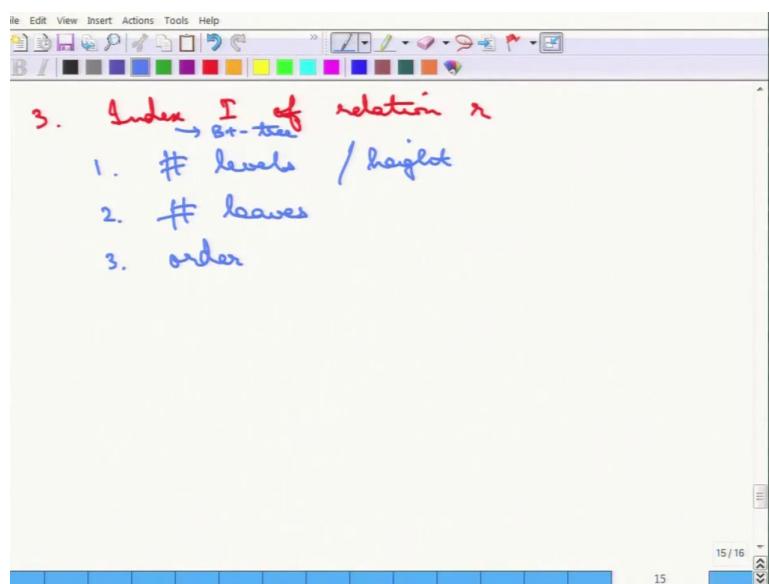
, but it stores it explicitly, because that helps in a certain things.

And the size of a tuple, again the size of a tuple is essentially the block size divided by the blocking factor. So, that can be a, I can derive, but these are the statistics that the query optimizer maintains, this is for each relation. Now, for each attribute A of a relation, this is an attribute column, attribute A of relation r. Again, certain statistics are maintained, the first statistic is the number of distinct values.

So, this is kind of an interesting statistic, this is essentially the size of your... If the projection on A is done on relation r, what is the size? This is the  $v_r$ , this is denoted by  $v_r(A)$ . This is useful for many cases, while duplicate elimination and all those things, this can be useful. Then, the minimum across A and the minimum value and the maximum value and in some cases the standard deviation and mean, etc can also be stored.

Then, genuinely the histogram of values is being maintained. So, how are the values for that attribute A are distributed? So, the histogram can be either equi-width or an equi-depth. So, in an equi-width histogram, the range of value that is put in one histogram is same and in an equi-depth, the number of values that it put in one histogram been from the next is same.

(Refer Slide Time: 05:07)



So, histogram are can be maintained, then for each index, for an index, say index I of a relation r, it also again maintains certain statistics and the statistics are of course, the number

of level. So, this is when I say index, I mean the B+ tree of course. So, this is the number of levels of the tree; that is the height of the tree and then, number of leaves; that is in that, there and some others, it says the order of the tree and the average branching factor and all those things. Again, this helps to decide, whether to use the index for hash, join and etc.

(Refer Slide Time: 05:48)

*Selections*

$$\sigma_{A=x}(r) : n_r / v_r(A)$$

$$\sigma_{A \leq x}(r) : n_r \times \frac{x - \min_r(A)}{\max_r(A) - \min_r(A)}$$

$$\sigma_{A < x}(r) : \sigma_{A \leq x}(r) - \sigma_{A=x}(r)$$

$$\sigma_{A > x}(r) : n_r - \sigma_{A \leq x}(r)$$

$$\sigma_{A \neq x}(r) : n_r - \sigma_{A=x}(r)$$

So, after this, let us see how does the query optimizer estimates the size. So, what we are, the next topic that we are going to study is about estimating size of different operations, estimating size. So, for selections, let us first handle selections, so suppose the selection condition is  $\sigma_{A=x}(r)$ .

Now, what is this? There are couple of ways of solving it. Suppose A is the super key of this relation r, if there is a case, then one knows that the size, the answer said size of this is going to be either 1 or 0. Because, if A is a key, if that value occurs, it occurs, then it is 1, if it does not occur, it is 0. Otherwise, if it is not the key, then what is the answer, then the answer can be got from the previous things, this is roughly  $n_r/v_r(A)$ .

Why is that? Because,  $n_r$  is the total number of tuples and  $v_r$  is the number of distinct values and we are use the number of distinct values. So, roughly on an average for every value, this is  $n_r/v_r$ , if it exist, otherwise it is 0. So, this is for the equality selection. What is on an non-equality selection? Suppose it is a less than query, now once more, if x is less than the minimum value that is maintained, then this the answer to this is 0.

And if again  $x$  is greater than the maximum value that is stored, then the answer is a entire relation, the size of the relation is the answer. Otherwise, this is estimated in a following

$n_r \times \frac{x - \min_r(A)}{\max_r(A) - \min_r(A)}$ , the minimum value that is stored and this is max minus minimum value that is stored. Note that, these are all rough measure; this is not going to be exactly the equal, so this formula assumes that the values are distributed uniformly from the minimum value to the maximum value.

So, wherever  $x$  occurs, the  $x$  minus minimum gives you roughly the percentage of tuples that are going to fall below  $x$  and then, if you multiply that with  $n_r$  that gives the actual number of tuples; that is an estimate. Now, this estimate does not use the histogram actually, this is just uses the min and max  $x$  values and when the histograms are available, the equi-width or the equi-height histograms are available.

Then, the estimates are going to be better, because what the query optimizer needs to do is to simply count the number of histogram means, which below it and the histogram mean that it overlaps, it needs to estimated somehow using this formula. But, otherwise it gets an actual count of the histogram means that are below that are less than  $x$ . So, the next query is this  $A < x$ , the previous one was exactly equal to  $x$  and this is a way of solving it is, if one can solve  $\sigma_{A \leq x}(r)$  and then, if when gets rid of  $A = x$  and that solves the  $A < x$ . So, that is the easy, the next that we will study is  $\sigma_{A \geq x}(r)$  and this is can be again solved very easily.

Note that what is the  $\sigma_{A \geq x}(r)$  is the complement of  $\sigma_{A < x}(r)$ . So, one can use that estimate to solve it.

So, this is simply  $n_r - \sigma_{A < x}(r)$ , the size for that and similarly,  $\sigma_{A > x}(r)$  can be estimated in two ways. So, this is  $n_r - \sigma_{A \leq x}(r)$  and finally, there is only one thing left, which is  $n_r - \sigma_{A \neq x}(r)$  and by this time, we have surely figure out how to solve this, this is essentially  $n_r - \sigma_{A = x}(r)$  So, that completes the estimation of size of selections, simple selections on one attributes.

(Refer Slide Time: 10:05)

$\theta$  selectivity =  $s/n_r$

conjunction :  $\sigma_{\theta_1 \wedge \dots \wedge \theta_k}(r)$

$$\frac{s_1}{n_r} \times \dots \times \frac{s_k}{n_r} \times n_r$$

disjunction :  $\sigma_{\theta_1 \vee \dots \vee \theta_k}(r)$

$$n_r - \left( \frac{n_r - s_1}{n_r} \times \dots \times \frac{n_r - s_k}{n_r} \right) \times n_r$$

negation :  $\sigma_{\neg \theta}(r)$

$$n_r - \sigma_\theta(r)$$

Next, how to do complex selections, so that is not on one attribute, but on multiple attributes; that is what I mean by complex selections. So, the selectivity, so there are multiple conditions. So, something is defined as for the condition theta, the selectivity is defined is that, how many tuples pass only that condition  $\theta$ ; that is called the selectivity of the condition  $\theta$ .

So, it is essentially the probability the tuple in  $r$  satisfies  $\theta$ , now if so the way to estimate is that, if  $s$  number of tuples passed the  $\theta$  condition, then  $s/n_r$  is the selectivity. So, selectivity essentially  $s$  by the total number of tuples which is  $s/n_r$ . Now, if the condition is conjunction of these things, then using this selectivity, so the conjunction the condition let me write on a little more explicitly. The selection is on  $\sigma_{\theta_1 \wedge \dots \wedge \theta_k}(r)$ .

If the conditions are independent, so that means,  $\theta_1, \theta_2$ , etc independent of each other, then the estimate is very simply, so for  $\theta_1$   $s_1$  is the selectivity, for  $\theta_2$ ,  $s_2$  is the selectivity so on and so forth. One can write down these selectivity conditions. So, the number of tuples that is selected for  $\theta_1$  conditions is your

$$\frac{s_1}{n_r} \times \frac{s_2}{n_r} \times \dots \times \frac{s_k}{n_r} \times n_r$$

This is the total selectivity of the conjunction condition that when it is multiplied by  $n_r$ , gives

the total number of tuples that are going to be selected. So, that is the, if these are independent, we are roughly going to assume, these are independent. Again, if the independence is violated, then this does not work out, but that is fine.

The next one is disjunction. So, the what disjunction condition is essentially  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_k}(r)$ . And, how can one solve this, in the following manner. So, what is disjunction? So, what exactly is the  $\theta_1$  and  $\theta_k$  is that is that, when is the tuple selected on the  $\theta_1$  and  $\theta_k$  is that, let us negate this condition, disjunction can be think of as the negation of conjunction. And then, negation of those are the  $\theta_1$  values, so a tuple is selected for this disjunction, if it is not selected under any of the negative conjunction.

So, other way of saying that at tuple is not selected for this disjunction, if it is selected for the negation of those conjunction. So, using this strategy, one can write down the formula in the

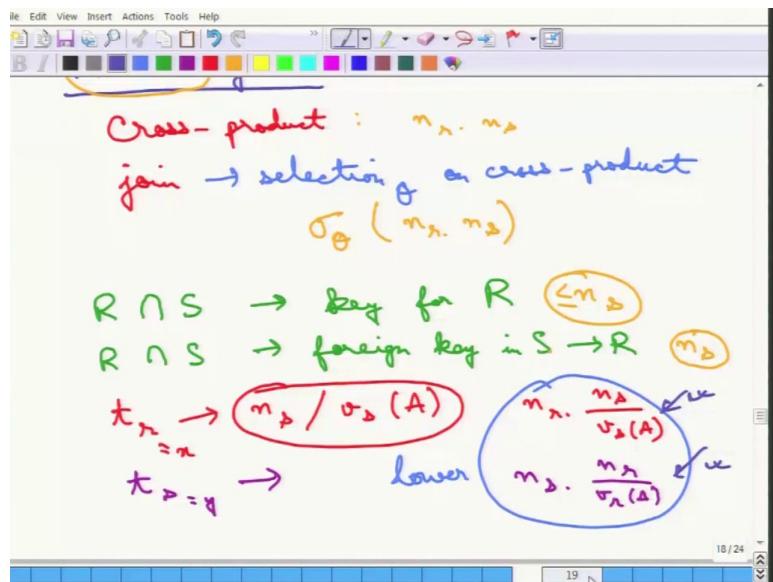
$$\frac{n_r - s_i}{n_r}$$

following manner is that, so  $\frac{n_r - s_i}{n_r}$ , this is the selectivity of the negation of the condition of  $\theta_1$  and so one can write down that and we are again assuming the all  $\theta_1$  to  $\theta_k$  are independent.

So, their negations are also independent. This gives the, on the selectivity of the negation of this conjunction that times  $n_r$  gives the number of tuples, where the conjunction is not correct and one needs to find the negation of that, because is the disjunction. This is  $n_r$  minus this, so that gives the formula for the disjunction. And if one goes ahead and tries to solve it for the next thing, which is negation, just a negation.

So, the negation of the condition is simply, this is  $\sigma_{-\theta}(r)$ ; that is very easy and that we have already handled. Although, one can say this is the complex selection, but this is essentially as the  $n_r$  minus the estimate for  $\sigma_{\theta}(r)$ , this we have already seen. So, that is the estimating the size for selection. So, we have seen simple selection and we have seen complex selections.

(Refer Slide Time: 14:10)



Next we will estimate the size of joins. So, let us first start off with natural join you estimate the sizes for natural joins. Now, the simplest case is size for cross product and you will come to natural in status. So, cross product, what is the size of cross product, it is simply  $n_r \times n_s$ . So, we are assuming it is r and s are two relations. So, just double join, join with only two relation, it is a single join operation that we have done. So, cross product is simply  $n_r \times n_s$ .

So, what is a join? So, if you do not worry join, so for normal join, join is essentially just a selection of  $\theta$  on cross product. So, essentially the estimate of the size of these join is the estimate of  $\theta$  on the cross product size. So, that is simply whatever is the estimate on this  $n_r \cdot n_s$ , so that is a way to do it, essentially just assuming the track.

Now, for we come back to the case of natural joins. Now the following is to going, if  $r \cap s$  is not there, then the size of this natural join is essentially just the size of the cross product itself. Now, if  $r \cap s$  is a key for r, then each tuple of s will join with at most one tuple of r. So, let us worry about the schemas of r and s, and this is the schema of r and the schema of s.

Suppose, is schema of r and s is a key for r. So, which means that at most one tuple of r will join with at most one tuple of s. So, the size of this can be at most number of tuples in s, so that is  $n_s$ . On the other hand, if  $r \cap s$  is a foreign key in s that references r, then what happens is that, this is exactly equal to be  $n_s$ , because in a foreign key in s; that means, all of this s is, so all the values in s must one have a corresponding thing in r.

So, all the tuples in  $s$  must join and so all in  $s$  is a result. This is at most  $n_s$ . This is less than equal to  $n_s$ , because if it is not a foreign key it is not clear whether everything will sure everything  $s$  will have a finding tuples in  $r$ ,  $r$  not. And otherwise general case is following, so every tuples in  $r$  suppose the tuples  $t_r$  in  $r$  can join with suppose  $n_s/v_s(A)$ .

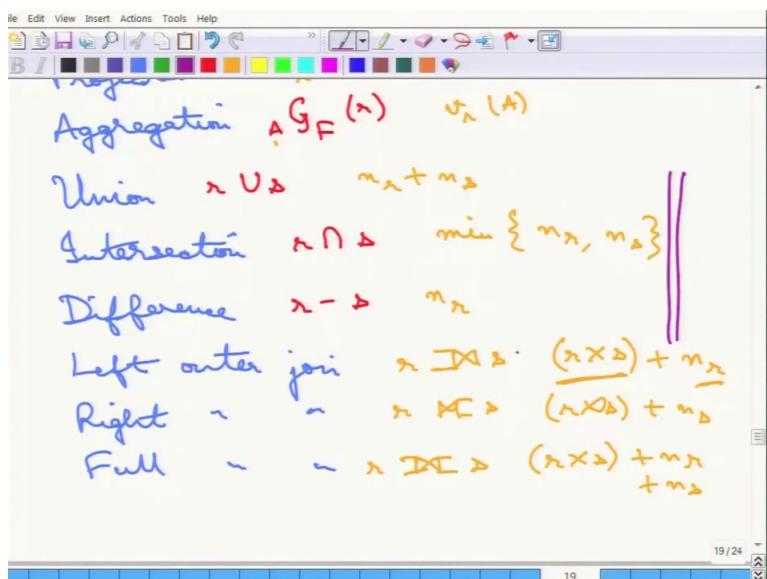
So, why is that, because tuple in  $r$  has a particular value of  $x$ , and the estimate for the selection of  $x$  in  $s$  is  $n_s/v_s(A)$  every tuples in  $r$  can join with  $n_s/v_s(A)$ . So, the total

number of join tuples that can found is  $n_r \times \frac{n_s}{v_s(A)}$ . This is if one thinks for point of view of  $r$ . However, if one thinks with the point of view of  $s$ , then the same thing can be done and

the estimate that one gets is  $n_s \times \frac{n_r}{v_r(A)}$ .

Now, the question is which one is better etc. So, this is the two estimate generally that the find out that the lower size is a better estimate. So, out of these two, the lower is the better estimate, this has been observed. And once more, this is, if there are histograms, if one uses histograms then the estimate can be of course, improved. So, if histograms are used in both of these, both of these estimates can be improved. So, the actual estimate is also improved. So, this is the sizes of joins etc. Now size of some other operations, we can see, so suppose size of projection, this we have already seen the answer.

(Refer Slide Time: 18:17)



This is the number of distinct values that we have seen. So, this simply  $v_r(A)$ . Then, we can see this, estimate the size of aggregation is estimating the size of aggregation, see suppose this is the aggregation function and on this... So, this is the function that is right, this is essentially again just the  $v_r(A)$ , why is that, because aggregation assumes on a particular attribute. So, which essentially means, it is a projection on that attributes. So, this aggregation function is projecting, so this aggregation is being done for each distinct value of A, which is this essentially saying that the number of  $v_r(A)$ ; that is the number of distinct value is of A that is being maintained.

Union of r and s,  $r \cup s$ , the estimate is simply  $n_r + n_s$ , this is of course, upper bound and it can be less than that. Similarly, intersection then be found out, so intersection of  $r \cap s$ , the estimate is the  $\min\{n_r, n_s\}$ , again this is a conservative estimate. The next one is set difference, which is  $r - s$ , the estimate for this is simply  $n_r$ , this is again upper bound, because is assuming that none of the tuples in s is common with r, so all  $n_r$  may be returned.

Then, this all of these are essentially, all of these are upper bounds, fine. So, the next thing is comes is that left outer join. So, the estimate of left outer join is the following, is that the, so the estimate of  $r \bowtie s$  is the size for r joined s and this then does not cover everything in r. So, this plus the size of r. That is so, all the tuples that are join in  $r \times s$  plus all the tuples that are in r, because the worst case is that all tuples in r have been left out.

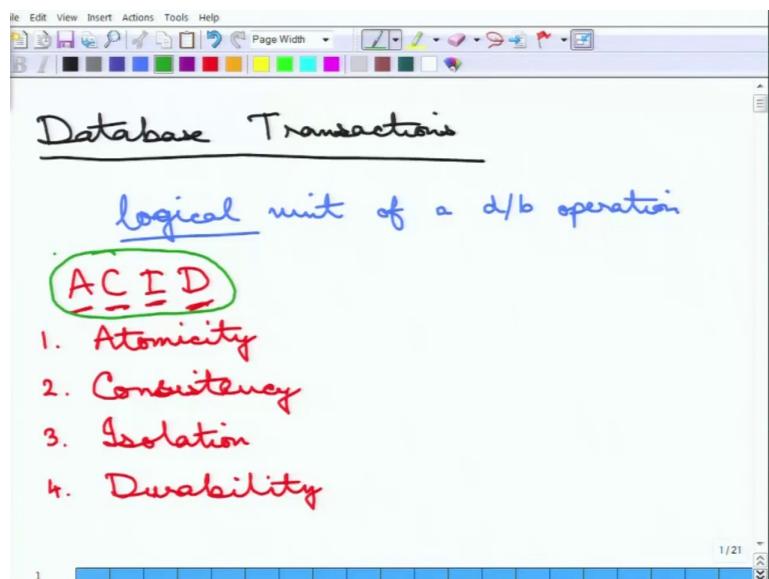
Right outer join is similar and let me just write it down for the correctness. So, r joined with s is r joined with s plus  $n_s$ . The more interesting case is the full outer join of r and s, is that size of r joined with s plus  $n_r$  plus  $n_s$ , because where the worst case is that r and s none of the can be joined. So, again these are upper bound. So, that finishes the topic on query optimization. Next we will move on to one very important topic about databases, which are the database transactions.

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

**Lecture – 28**  
**Database Transactions: Properties and Failures**

Welcome, today we will be talking about Database Transactions. So, transactions if you remember are the one logical units of programs, that let us one transfer money from one bank account to the other, let us one buys rail tickets, etc. So, this is about database transactions.

(Refer Slide Time: 00:28)



So, the concept of transactions we will cover it in multiple sessions, but one thing to remember is that a transaction is not a program or a part of the database, it is a logical unit of a program, it is a logical unit of a database function. So, logical unit of a database operation, so some particular set of operations can be set to be constitute in a transaction. So, the complete operation of for example, the complete operation of moving money from, transferring money from one bank account to the other is the, can be called a transaction.

So, only that part may be declared as a transaction within in the database operations. So, transactions have four very, very important properties, these are called the acid properties, this is the very famous term. So, that probably have heard in the context of databases, this is called the acid properties. So, what are the acid properties? The first property in that acid thing is A, A stands for Atomicity. So, the atomicity of a transaction essentially says that

either the transaction has completely happened or it has not happened at all.

So; that means, in this banking transaction scenario, either the money has been transferred from A's account to B's account completely or it has not happened at all. Now, what do I mean by completely? See the transfer of money from A to B consists of two operations, two basic operations, debiting money from A's account and crediting to B's account. Now, it cannot happen that only the debit part has gone through or only the credit part has gone through, either both have succeeded or both have failed. So, that is the atomicity of transactions.

The next property is the C part, which is called the consistency. So, the transaction should not change the consistent order of a database. So, if the database was consistent to start with before the transaction started, it should be consistent after the transaction has entered. So, the consistency property for example, for this bank account scenario may be that the total amount of money in A's account and B's account is something which was constant, now after the transaction that should not be changing.

So, suppose A is started of with a 1000 rupees and B had 500 rupees and then 100 rupees was transferred. So, the initial amount in A plus B's was 1500 and it should still remain 1500. So, the database should be in the consistent state, this is just an example and then there are other ways of actually defining what consistency is, but that is the property called consistency. The next property is the isolation and this we discussed a little bit towards the beginning of the course.

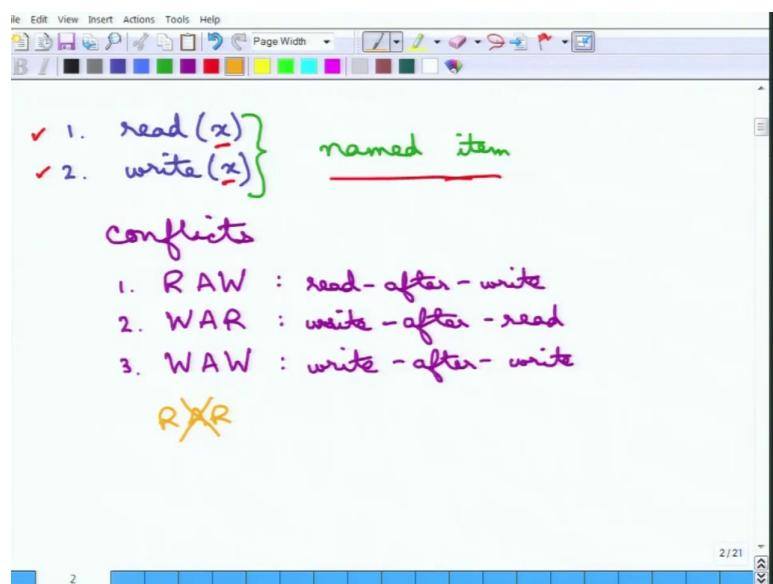
But, essentially this means that multiple transactions may be executing concurrently, but the effect on each one of them is achieved, this is obvious of any other transaction. So, each transaction is apparently happening in isolation of the other transaction. So, the example is let suppose A is transferring money to B and C is transferring money to D. Now, this should not interfere with each other and the effect of whether C is transferring money to D has succeeded or failed, has got no effect on the transfer of money from A to B and vice versa, so that is called an isolation.

And the fourth property is the durability. So, the durability part is that, if a transaction finishes successfully, then the effects of the transaction must be durable or must be permanent. So, after this transfer of money has been happened and if the transaction is succeeded successfully, that is even if the database crashes or there are other problems in

other transactions and there are other issues, this transfer of money is deemed to be permanent.

So, if A's account has been debited and B's has been credited, after the database again comes up or again the database is being operated, the new state should reflect that A's money has been debited and B's has been credited. So, this is the durability, once it happens, it remains, it is permanent. So, these are the four important properties of transaction and together these are called the acid properties. And, this is very, very important to understand what the acid property is. Now, to define a little bit more of the transaction, as I say this is the logical unit of a program.

(Refer Slide Time: 05:17)



And the basic two operations of transaction is read and write. Now, what does the transaction read or write? A transaction is said to read or write, something called a named item. Now, what is a named item? A named item is essentially any logical unit of data can be a named item. So, for example, a bank account with the particular account number, the account may be a named item. So, and in other examples a complete ticket in a ticketing system that may be a named item or etc, etc.

So, when we are starting database transactions we will forget whether the database is in the relational mode or any other mode or relational for model or any other mode, all we will see is that the database is consisting of some named item that is all. Now, the named item as I said is one logical unit of the data and there may be named items that are covering. So, that

the named item may be subsets of each other, etc, but that we will not consider here.

So, for the transaction model that we will study, we will just say that there are named items. So, the database is just simply a collection of named items and the transaction either reads a named item or writes the named item. So, if the named item is suppose  $x$ , then there are two operations, either it reads  $x$  or it writes to  $x$ , that is it. Now, the granularity of this named item may be a block, may be a page, may be the entire database, may be a table, etc, but that does not matter all we are concerned about is the named item.

And transactions are independent of how you define the named item. It is essentially independent of the granularity in which a named item is defined. So, these are the two basic of operations of read and write and using this... So, just to note that  $x$  is the named item variable, so  $x$  denotes the variable for a named item and that is all, that is a and now with this read and write there may be conflicts for read and write and we are know what the three basic types of conflicts are. The first one is read after write, then this is write after read and the third one is write after write.

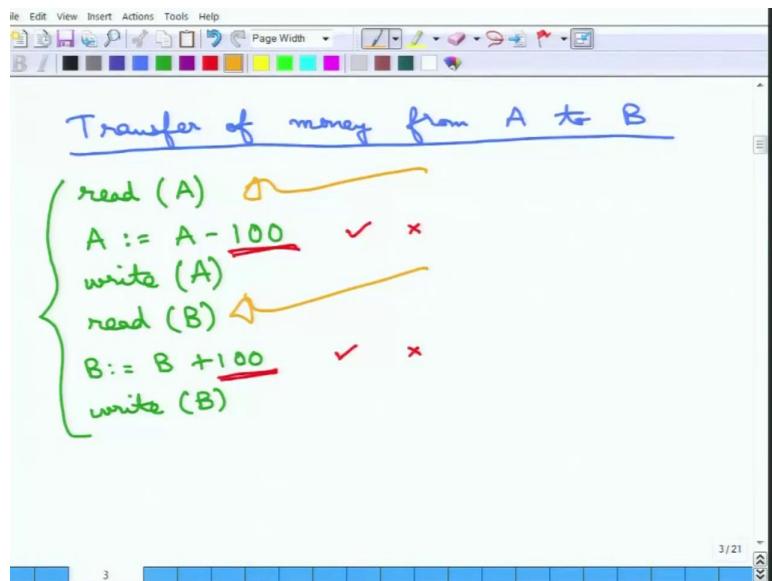
So, this just to complete, this is read after write. So, when there is read after write, conflict happens is that, when there are two or more transactions that reads. So, there are two transactions, two or more transactions, one of them reads and the other writes and then whatever the reading, whenever the first transaction tries to read, it should read the value before the second transaction has written . So, that is the read after write conflict and then similarly there is a write after read conflict and we will go over this conflict in much more detail later, but this is basically these three conflicts are essentially saying which operations can conflict.

Now, note that very importantly there is no read after read conflict, there is no read after read conflict, because if there are two transactions that are reading the same item, the  $x$ , it does not matter in which order they read, both of them read the same value, none of them changes the value of the item. So, it does not matter in which order they read, so it is not a conflict, so the transactions one and transaction two even if they read the same item, it is not a conflict.

But, otherwise whenever there is a write operation involved in it, which is this read after write, write after read or write after write, there may be conflicts. Because, the reading and writing has to go in the correct order; otherwise, the read value or the written value may be wrong and we will go over this in much more detail now. But, this is essentially the concept

of what the conflicts are. Before we go over the conflicts, let us complete to one example of the transaction to understand this acid properties.

(Refer Slide Time: 09:12)



So, suppose this transfer of money from A's account to B's account. So, transfer of money from A to B, this is the example, this is the transaction that we are doing. Now, what are the properties that matter is that, the first thing is the how do we denote these transaction, the way to denote this transaction is that what will happen is that, the amount... So, whatever amount is in the A's account that will be read, *read (A)*, so that is the first operation called read.

Then, suppose A's account will be debited by let us say 100 or this thing is being debited from the A's account,  $A := A - 100$ . So, that is, the new value of A becomes this thing, now this new value needs to be written. So, this is the *write (A)*, then similarly this read, the B's account will be read, *read (B)*, that will be enhanced with that amount 100,  $B := B + 100$ , and then there will be a write to this B, *write (B)*. So, this is the complete description of the transaction in this manner and now we see, let us say, what is the idea of atomicity.

The first thing is, how do we define atomicity? Very simply, if this happens then this must happen or if this does not happen, then this does not happen. So, atomicity as I am saying that either A's account is debited and B's account is credited or none of them has happened, it cannot happen that only one of them has happened, that is the atomicity. Otherwise, you see

what the problem is, the bank can either take away money indefinitely or generate money indefinitely, so in both of which problem occurs. So, the atomicity ensures that those problems do not happen. So, that is the part about atomicity.

Now, consistency, consistency is something about the semantic, is that now you see 100 rupees is taken from A and that the same 100 rupees is deposited to B. So, that is why the sum of A plus B remains constant after the transaction is happening. If 100 rupees was taken from A and only 90 rupees or 110 rupees were credited to B, then that the consistency may suffered.

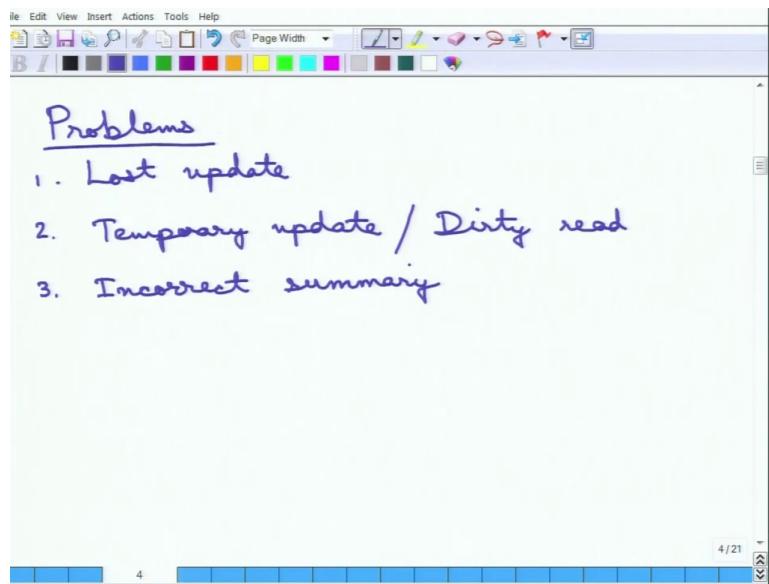
So, the consistency has to be defined. What is exactly the consistency criterion? Here the criterion is that the total amount of money in A's and B's account is constant. So, that is being maintained, so that is the consistency part. Next comes the isolation. So, the isolation is essentially, is that if there is another transaction that either reads A or reads B, then to that transaction, now this is a little bit different concept from what we have been describing isolation.

So, if there is another transaction that either reads A or reads B, then that transaction should be oblivious of what is happening in this thing. So, either it must read the value before any of these two operations have taken places or it must read the value after both of these operations have taken places. So, that is the effect, that is the way to ensure isolation, but what isolation essentially means is that, that the other transaction does not need to know that there is another transaction going on here and it does not care whether this transaction, what stage this transaction is whether it has succeeded or failed. So, that is the isolation property. So, that is the thing about isolation.

And finally, the thing is about durability, the last thing is about durability. Now, if B's account has been credited and B has been sent of a notice by the authority, whatever it is and then the database crashes etc, after the database recovers it should not be that B's seeing less money, B should see the new money that the new balance should be reflected in B's account no matter what, after the transaction has been completed.

So, after the transaction is completed, there is a, generally, a signal given, there is generally a state which is declared. So, in the transaction is completed successfully and after that it should be durable it should not, it can be reversed. So, that is the point of transactions.

(Refer Slide Time: 13:01)



Now, the transactions may end up in certain problems, there may be certain problems in transactions. So, the first problem is called the problem of last update. So what essentially is last update, is that, suppose there are two transactions that are both writing to the same data item. So, what may happen is the update by the first transaction may be over-written by the update of the second transaction. So, essentially the update that is done by the first transaction is lost, because it has been proceeded by some another transaction, so that the last update problem.

The second problem is the temporary update. This is also sometimes called the dirty read. So, what the dirty read problem is, that suppose a transaction writes to a particular data item. So, suppose in the previous example, the transaction writes the new value to A and then it fails before B is being updated. So, then, as we have been saying, the atomicity should guarantee that A should roll back to the original amount. Now, before this rollback happens, suppose there is another transaction that reads the value of A. So, that is a dirty read.

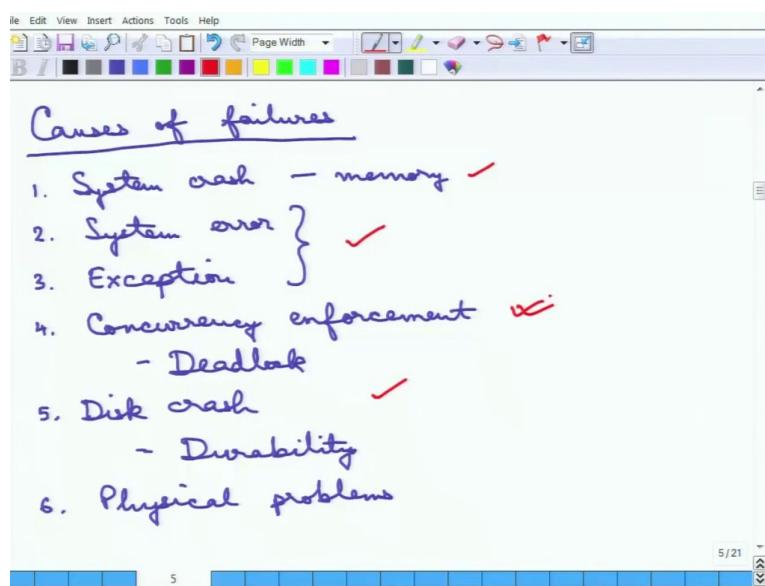
Why is it a dirty read? Because, the value that is read is the new value which is dirty, because the transaction that updated the value of A has not yet committed. So, there is a concept of committing. Committing meaning essentially completing successfully, that has not yet done correctly, that has not yet completed correctly, but another transaction has come and read that value. So, the update in the first case is temporary, because the transaction has not yet said that this update is correct and it has not gone through completely.

So, it is the update has been temporary, but another transaction read this. So, that is why it is called a temporary update or the dirty read problem. And the third problem is called an incorrect summary. So, the incorrect summary problem is that suppose one of the transactions is just trying to find out the aggregate or some aggregate operation on all the bank accounts etc. And what may happen is that it reads the A and then A is changed.

So, the statistics or the aggregate that the first transaction completely wrong, because A has changed. So, by the time the first transaction finishes the aggregate, A has changed, but the value that it shows across the let us say the some of the accounts of A and B etc or the average etc will be wrong, because it has read something wrongly. So, that is an incorrect summary, so the summarization that it provides the essentially the aggregation etc is wrong.

And this happens, because of the temporary update of the dirty read problem, but this is another very common problem that the database practitioners faced earlier when they were looking at transactions. Okay. So, that is all that is fine, but makes the question comes is why transactions fails?

(Refer Slide Time: 16:06)



So, what are the causes of failures, so why would transactions fail? Now of course, the first very simple way to understand is that there is a crash. So, there is the system crash, that is it. So, either the power of the database machine has gone or there is some error, some exception happen, etc. So, the memory is lost, all those things happen, so there is the system crash that is fine. Then there is a system error.

So, how can error happen is that suppose A has 1000 rupees and it is trying to transfer 2000 rupees. It cannot right? That is an error that is a logical error, though there is nothing wrong in that in the sense of read-write operations from the transaction. So, but that is an error and the transaction cannot succeed in any way. So, because there is no way that 2000 rupees can be transferred out of 1000 rupees. And they can be other program error such as division by 0 etc, but that is a system error.

And again, I mean this division by 0 etc can also be called which is called an exception. So, that is may be many other kinds of exception, etc, so insufficient account balances. So, these two are mostly the same, system error or exception. Then the fourth one is that there is concurrency. So, we will define concurrency and all those things carefully, but, so the error that happens, because of trying to enforce the concurrency and this is happening, because of something called a deadlock.

We will see this definition of deadlock etc, definition concurrency in much more detail later. But, the idea is the following is that transaction one requested to read some item B which has been updated by transaction 2 and transaction 2 reads to needs to read transaction A, some data item A that is been written by transaction. So, very simply transaction 1 depends on transaction 2 and transaction 2 depends on transaction 1. So, there is a deadlock meaning, nobody can proceed either without the other completing, but nobody can completely either.

So, the system essentially stops and none of the transactions can finish and none of the transactions can therefore, finish successfully. So, that is a deadlock. And we will define this in much more detail later, but that is one. Then of course, system crash then I mean the system crash can be memory failure, this is generally a memory failure and of course, there can be a disk crash, the entire database has crashed. That is a this thing, that must take care of the durability issue.

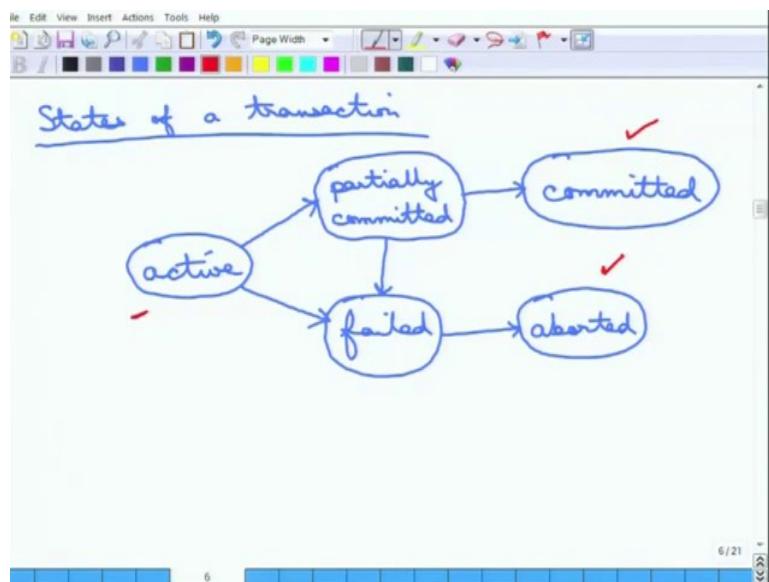
So, if the database crashes the durability can be hampered, so there may be problem with the transaction, so that's the thing. And then there are maybe I mean other physical problems, I mean fire or theft, etc whatever physical, this is essentially why the transaction may be failed. But, the important things is the disk crash or the system crash and these two and then this is we will study much more later.

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

**Lecture - 29**  
**Database Transactions: States and Systems**

The states of a transaction, a transaction can be in a various state.

(Refer Slide Time: 00:12)



Now, what do you mean by state is that the status of the transaction, whether it has completed, whether it has started, etc. This is called the states of a transaction and the state of a transaction can be essentially explained using this diagram. So, a transaction can be active; that means, the transaction is running, is executing somehow. So, this can be in active state, fine. Then there is something called a partially committed state.

So, what is a partially committed state? So, the partially committed state, a transaction is said to be in a partially committed state, if the last statement of the transaction has been executed successfully, but the transaction has not itself declared to be completing successfully. So, the last statement has been done, but it is still not declared to be correct that is a partially committed.

Then of course, there is a committed, which means after the partially committed thing, the transaction essentially completes everything that needs to be done and then, declares itself to be successfully completed. So, committed essentially means successfully completed, that is

the committed state. Then of course, there is a failed state, the transaction has failed because of different reasons, power system, crash, etc whatever, all those things and the system errors etc, that is a failed state.

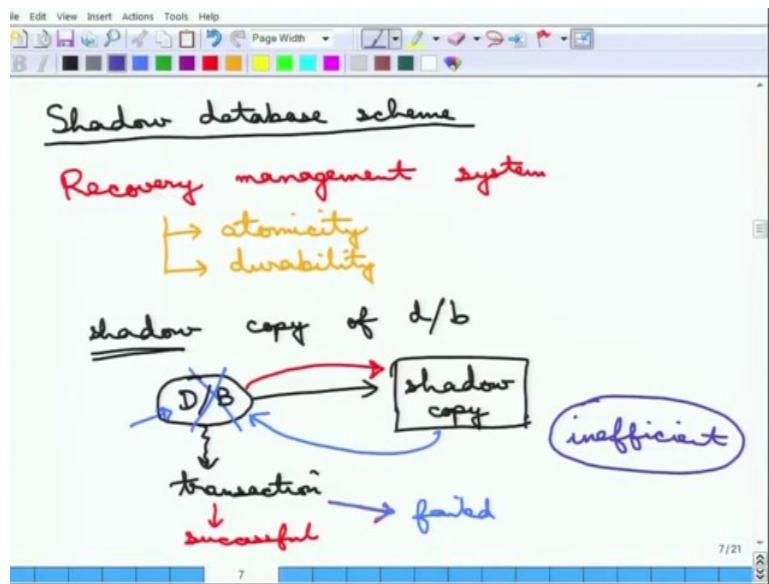
And then, after a transaction fails, it has no other way then to abort. So, abort meaning a transaction must undo all the things that it has done and should rollback to the state before it started. So, if a transaction that transfer of money aborts, what will happen is that the transaction should rollback to the original among that A and B had, so that is the abort.

Now, of course, we can draw this state diagram is that, a transaction starts from active and it can go to partially committed state or it can go to failed. From a partially committed state the transaction can go to the committed state, but unfortunately it can also go to the failed state. So, after the last statement has, of the transaction is successfully done, there may be still some failures.

And we will see why these failures can happen. There is a log needs to be written, etc some more operation needs to be done by the database and during any of those operations, it may fail and of course, from a failed state there is no other way, then to abort. So, now, you see essentially once the transaction starts from active, it ends up in either committed or aborted state, there is no other way, so that is the thing.

So, either a transaction commits, which makes it is completed everything successfully or it aborts; that means, it is failed somewhere and it must rollback to the database state, where before it started. So, that is the thing about states of a transaction. So, the next we will go over this, certain schemes of this transactions and how to manage transactions etc.

(Refer Slide Time: 03:14)



The first one is called a shadow database scheme. So, shadow database, okay, let us explain the term, what is it. So, essentially these are all, form part of recovery management. So, why does transaction needs to recover? So, when a transaction aborts it needs to recover, which means it needs to ensure that the database recovers to the original state. So, if A's money has been debited without crediting B and then, the transaction fails, then A's money must be credited back, the same amount, etc.

So, recovery is the one that is ensures that atomicity is maintained; it also ensures the durability has maintained. So, recovery has some more role, so if the transaction said it has been committed, then the durability essentially says that, well even it is for some reasons the B's account has not be reflected with the correct thing, the recovery system will ensure that B will have the actual, the B's account will reflect the correct balance, if necessary by running the transaction again.

But, making sure of course, that A's money is not debited twice, but we will discuss all those things later, but this essentially that the point of recovery management is these two things. So, the shadow database scheme is the recovery management scheme, so recovery management, this is called a recovery management system and the shadow database scheme is one of them. And what does the shadow database scheme does is that, it enforces atomicity by maintaining a shadow copy of the database.

Now, what is a shadow copy? As we said earlier, there is a mirror or shadow copy which is

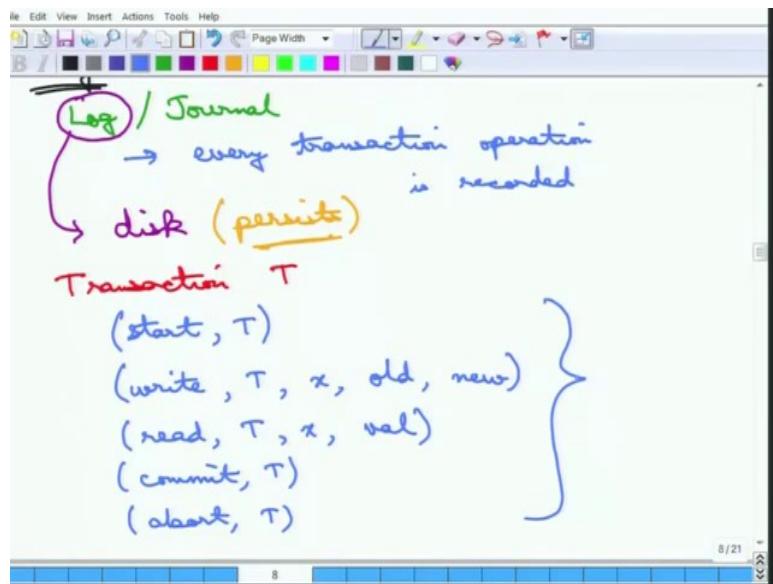
essentially an entire replica of the database and before any transaction starts. So, this is the idea, so this is a database, this is the shadow copy is being made, this is the shadow copy. So, this means that entire database is the exact replica of this thing and now a transaction is run on this database, this is the transaction is being run.

Now, if some problem happens during the transaction, then what happens is that, so now, there are two cases. If it is successful, if everything is successful, then essentially the shadow copy is being updated with the new thing. If, on the other hand, it fails, so let me use a different color for this, if from the other hand it fails, then this database is essentially removed and the shadow copy is now taken as the correct database, because this is where the transaction started.

So, the transaction meaning, were the operations are being done on the database, so there are two ways, either it succeeds or it fails. If it succeeds; that means, the new database the current database that it is working upon is the correct copy, so the shadow copy is updated. If it fails on the other hand, then it means that the shadow copy on which it was started is the correct copy. So, then the shadow copy is copied to the actual database, so this is the shadow database scheme.

Now, as you can see this is highly inefficient. Why is this highly inefficient? Because, you are copying the entire database every time a transaction starts, essentially it is not being done. There are some database pointers etc, but it is you can see that this is very, very inefficient, this can be extremely inefficient. And the other important, more important problem is that when there are concurrent transactions happening, so when there are more than one transactions happening, it really cannot efficiently handle those more than one transactions in a concurrent manner. So, that is the problem with shadow database scheme, for the recovery management.

(Refer Slide Time: 06:58)



So, the important thing that the recovery management then uses is something called a log, this is log, log is sometimes also called, this is log or sometimes also called a journal by a database. So, this essentially keeps track of every operation that is done by a transaction. So, every transaction operation, every operation in a transaction is recorded here, in the log. So, this is like saying whenever you do anything, you maintain a copy of that or you write it down in the logs.

So, if you say that, okay, transaction one reads A, so that is being written in the log, the transaction one has read A. Then transaction one writes B, so, that is also being maintained in the log, etc. So, the log, if one reads the log, then the one can of the entire understanding of which transaction operated and which data item and then in which order, very importantly, so that is the purpose of the log. So, log, very important is, that log is maintained on the disk so; that means, log essentially what is the idea is a log persists.

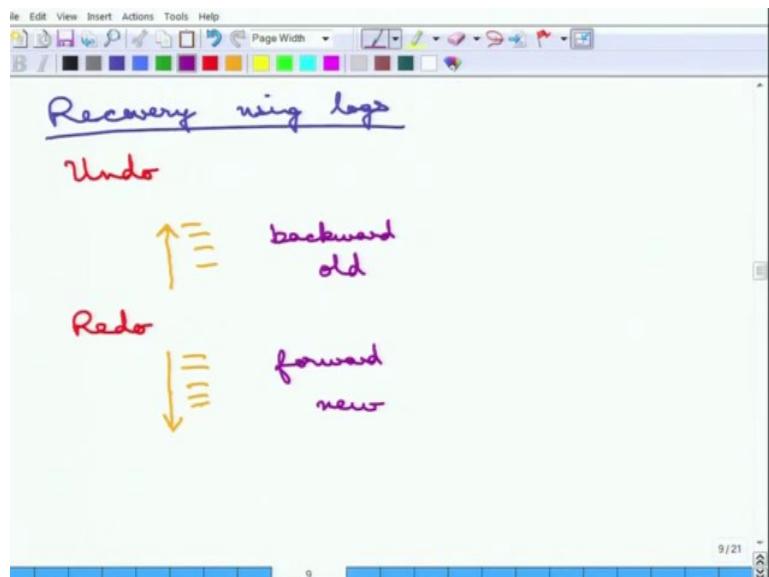
So, the log can be read again and again, has been written as it, because it persists. That is the one thing. And, then log is periodically backed up to archival storage. So, that it really, really persists, and you can go back to any log, any position in the log if it wants. So, that is because of this persistent issue, it is being written on the disk. Now, for a particular transaction T, suppose there is the transaction T, the couple of things are maintained in the log.

The first operation, I mean, it maintains is that is say start, so start is the operation in which transaction started, so it is a  $(start, T)$ , so that is maintained. So, this is one kind of operation

that is maintained, then what it is maintained is that, this is a write operation for the transaction T for the named item x, but the old value is this and then, new value is this. So, this is also maintained in the log.

Now, you see how detailed this is, so it essentially says that transaction T has written the value on x and it has replaced the old value with the new. Then, it also says read, so transaction T has read the value of x and the value that it is read is this value. So, it has read, essentially the when the transaction T read x, the value was *val*. Then, of course, there are this (*commit, T*) and (*abort, T*). So, these are the five operations that the transaction writes on the log. Okay. So, these are the five operations that the transaction writes on the logs and how do we recover using logs? So, the recovery using logs is done in the following manner.

(Refer Slide Time: 09:46)



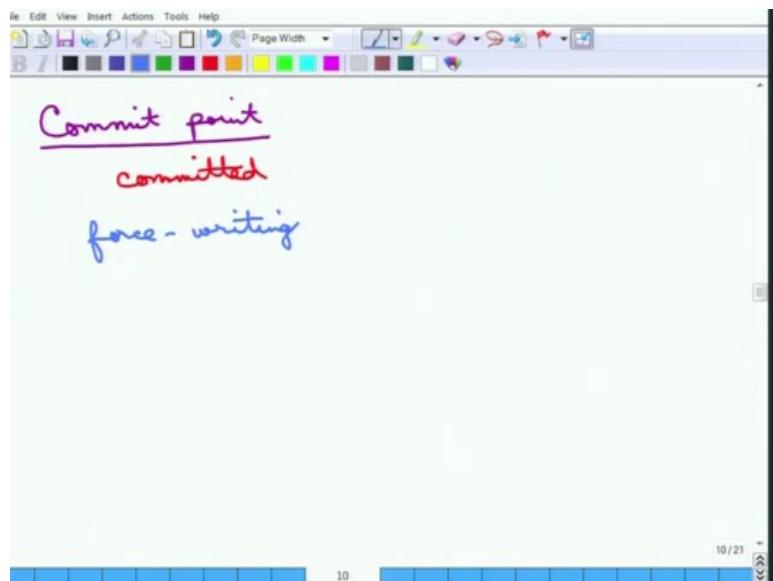
Now, we will try to build a recovery system using logs, recovery using this logs that we just explained. So, the first thing is that, there are two important operations, the first operation is called undo. What is the undo essentially? Suppose A's account has been debited, so how does one undo? Essentially the log will say that A was read, this was the value that was read, then transaction T wrote A and this was the old value, this was the new value, etc.

So, if these, all those things are maintained in the log, then to undo the log can be traversed in a reverse manner, in chronological order; that means, this was the latest in time point and this was the earlier in time point. It can be read in the reverse manner and each of this can be undone, so the old value might be written back etc; that is the way of doing the undo.

And then, there is another operation, which is the redo, which is essentially just the mirroring of that. So, the transaction was supposed to do something, but it could not and for durability etc, it has to make sure that these values are there. So, it is read in the correct order, the forward order, this is the called backward order, this is the forward order and then, those things are set to the new values.

So, once more, so in the undo the log is read in a backward order and the write values are going to their old ones and in the redo, it is read in the forward order and the new writes are updated to the new value. So, this is essentially how the logs are used to recover from a problem in the transaction.

(Refer Slide Time: 11:30)



And then, a transaction is said to reach its commit point, now this is not commit, this is the commit point. So, the definition of that commit point is essentially, when all the transactions, all the operations have been done correctly and they have been recorded in the logs. So, both the things must happen. All the transaction all the operations and the transactions have been executed successfully plus all of them have been recorded in the log.

So, if the recording in the log has not taken place, then it does not said to reach the commit point, the transaction does not reach the commit point, unless all of those are recorded in the log. So, now, beyond this commit point, so once the transaction reaches the commit point and after that, it is said to be committed; that means, once this is written on the log and everything has been done successfully, then the transaction is said to be committed.

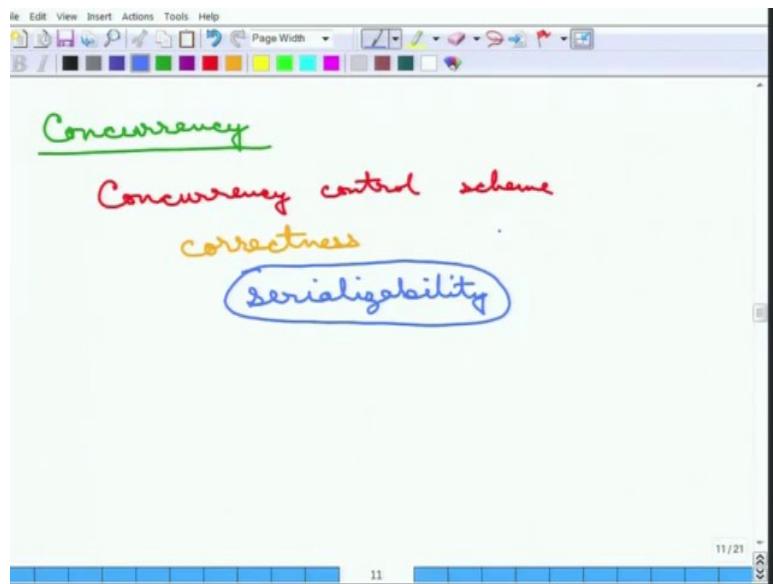
That means, that transaction now cannot say that, I will undo etc it has do only redo and make sure that, the effects of the transaction are permanent in the database. So, that is the commit point and then, once that is done the  $(commit, T)$  entry is made in the log. So, this is the  $(commit, T)$  and on the other hand, if there is an  $(abort, T)$ , then that means, that undo operations have to be done and that is the easier to understand.

So, if once a transaction reaches the commit point, the logs etc must be written on the disk, the logs must be persist in the disk. All the transactions that are done, must also be persistent on the disk. Now, here is the point is that, how does transactions happen, so the other database commits this transaction is that, it brings the necessary data item from the disk to the main memory and changes the value in the main memory, right, because that is the only way it can do.

So, when it is says to reach the commit point, it has to be made sure that all of these writes, which is suppose to take place on the disk has taken place on the disk. So, if necessary, there is a disk flush operation; that means, all the things that are in the disk buffer etc is actually gone to the disk. So, the actually the disk contains the new copy; that is called a force-writing.

So, if necessary, to reach the commit point everything that is changed on the main memory must be force-written on the disk. So, his is to make sure that the disk contains the correct versions, no matter what is there. So, this ensure that the redo operations can be done successfully. So, this is a point of recovery of transactions.

(Refer Slide Time: 13:54)



Now, there is another thing, which is called a concurrency issue of transactions. Concurrency essentially means that more than one transactions are being executed in the database and they may be accessing the same data item, otherwise there is no problem. Anyway, why does concurrency is useful is that they of course, increase the time to do. So, this is much more efficient etc and it increases the utilization of the processor and the disk and CPU and all those things and the average response time, the average completion time for transactions is reduced, so that is the thing.

But, there may be problems in the sense is that, one is trying to read the data item, while the other transaction is trying to write. So, the concurrency control schemes, so this is called a concurrency, just like we had recovery systems, this is the concurrency control schemes, must ensure the correctness of this. Correctness meaning, if one is trying to read a value that it should read, concurrency control schemes ensure that it reads that correct value. So, this is called a concurrency control scheme.

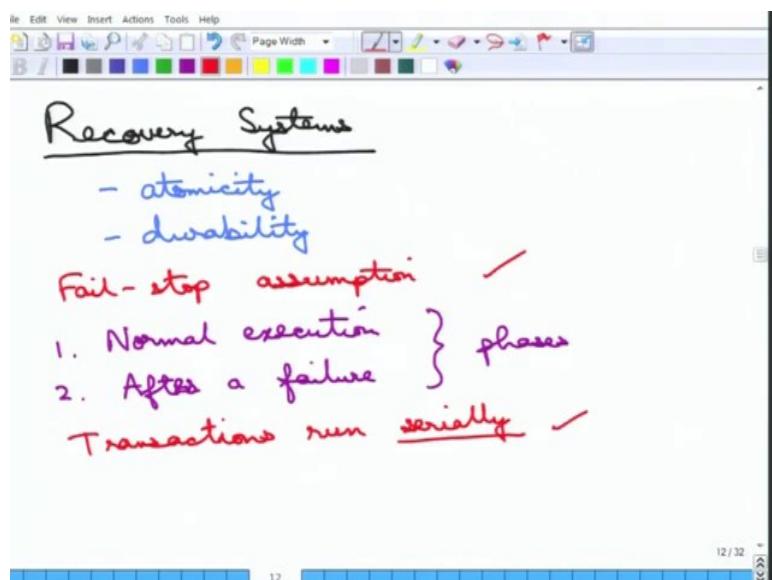
So, the important operation of this is the correctness, it must ensure the correctness of the two transactions; that is the important thing, And to do that, we define those notion of serializability, serializability is the formal way of studying whether concurrent transactions are correct or not. So, serializability is the actual formal way of doing this. So, that ends the introduction of transaction and what transactions are and a little bit of logs etc. Next, we will go over the recovery management systems in much more detail.

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

**Lecture - 30**  
**Recovery Systems: Deferred Database Modification**

So, welcome, we will continue with the Database Transactions. Last time, we saw an introduction to what the database transactions are, what are their properties and what they are used for. Today, we will go over one area of the database transactions in much more detail, which is the recovery systems.

(Refer Slide Time: 00:29)



So, today's topic is on recovery systems. So, essentially, if a transaction fails or something, a miss happens, how does the database recover? So, the recovery management system, essentially, the two properties that it tries to maintain is the atomicity and the durability. So, these are the two important things that the recovery management system targets. And there is, so atomicity and durability will be covered, but there is an assumption, there is an important assumption, which is called a Fail-stop assumption. This is, the recovery management systems make this thing, this is called a Fail-stop assumption, because what it is being assumed, in this Fail-stop assumption is that data, that is on a non-volatile storage, is never lost.

So, what is a non-volatile storage? The magnetic disks, etc, it is never lost. Now, one may

argue that how can it be, that it is never lost, because the magnetic disk may be corrupted, may be broken, may be harmed, lost, etc, what happens. The assumption for recovery management systems is that, there will be adequate number of backups of the non-volatile storage, data in the non-volatile storage. So, that can be always retrieved when necessary.

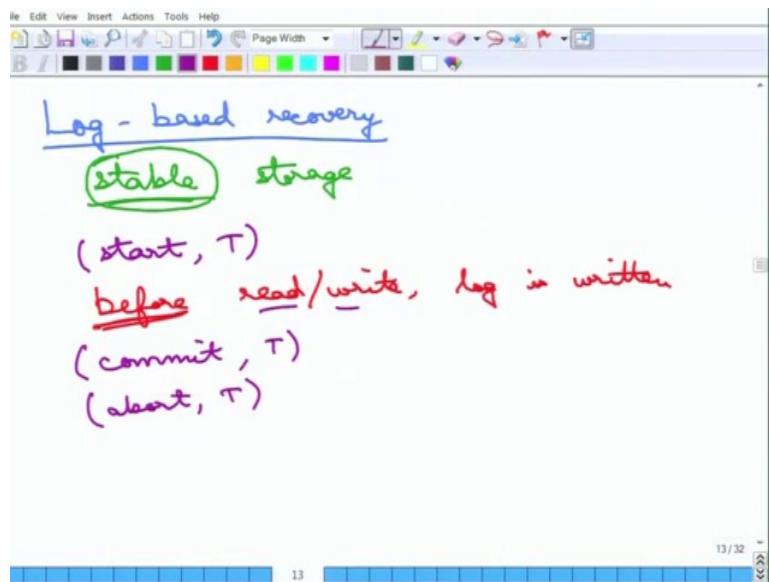
So, that is the Fail-stop assumption, is that the data on the non-volatile storage is not lost due to any problems, system crash, power failure, etc, nothing can be lost there. So, that is the thing and the recovery management systems have two main parts. The first one is action taken during the normal execution, when a transaction is running normally; that is during the thing and then, action taken after a failure.

Now, note that, why is the second part important is that recovery systems, so that word recovery means that, something has gone wrong. So, that is why you need to recover from that fault. So, that is why, there is a failure, failure is assumed and then, that needs to be recover from that. If there is no failure, then nothing needs to be done, every transaction runs successfully and then, the recovery management system is not invoked at all.

But, the point to study is that, when there is a failure that happens. So, these are the two important phases of the recovery management systems. So, these can be consider the two phases of a recovery management system and one other assumption that we do when we study this recovery management system is that the transactions run serially, that is there are no concurrent transactions.

Again, this may sound a little non-ideal that how can this happen, but as we will see that the argument about the correctness or the algorithmic views of the recovery systems, do not depend on how the transactions are concurrent, etc. So, for simplicity, we just assume that the transactions run serially. So, these two are the assumptions of a recovery management system and this is essentially a recovery management system will try to argue about a single transaction and what happens, when it fails.

(Refer Slide Time: 03:40)

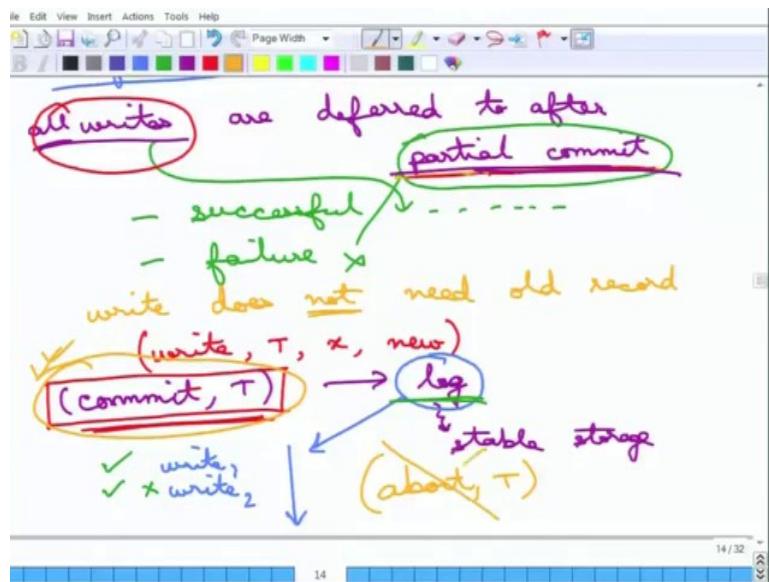


So, we do this Log based recovery first, the logs or the records that are written and as we saw last time, there are five types of entries that are made for a log, for a transaction and so the log is maintained on a stable storage. So, log is maintained on a stable storage, this is... What is a stable storage? This is essentially the secondary storage or the hard drive, etc and again, the stable storage the idea is that, the log is never lost.

So, the stable storage meaning once something is stored into that stable storage, because of it is log stable; that means, it is never lost. So, lost can be always retrieved; that is the idea, so that is the Fail-stop assumption idea. Okay. And then, there are log records as we showed last time. So, whenever, just to recap, whenever a transaction starts, there is a  $(start, T)$ ; that log record entries made before a read or write is done, the corresponding entries made in the log.

So, this is very importantly before the actual read or write, before the read or write is done, log is updated. So, log is written before an actual read or write on the data item is done((Refer Time: 04:51)). So, this is before that is very important and then, if the transaction commits, then successfully, then there is a  $(commit, T)$ , otherwise there is an  $(abort, T)$ . So, these are the five operations that are done. So, now, there are two types of approaches based on these log based.

(Refer Slide Time: 05:17)



So, there are two ways of a recovering. The first one is called a deferred database modification, so this is called a deferred database modification. This is a recovery management system based on logs. So, what is a deferred database modification is that, important point here is that, all writes are deferred to after the partial commits. So, writes, all writes, I should say, all writes are deferred till the point of partial commits.

So, no write is actually made to the database. So, the transaction thus notes, what writes are needed to be done, but these are not actually written to the database and only when the transaction partially commits, then these writes happen. Now, you may remember what a partial commit is, a partial commit for a transaction happens when the transaction has finished all the operations in it successfully.

So, now, suppose what may happen either, two cases may happen, is that, all the things are successful, everything is being successful in the transaction. Then, these writes may then go through one after another, first ((Refer Time: 06:31)) all those things may go through or there is some problem, there is a failure. Now, if there is a failure, then this partial commits stage is not reached at all, but now you see the state of the database if there is a failure.

Because, it has not reach the partial commit, none of the writes are actually gone to the database. So, the database is in the state that before the transaction was started. So, nothing has been changed into the database. So, nothing needs to be done actually, because the transaction just aborts none of the database... So, all the writes are on to some temporary

storage, may be in the main memory or somewhere else, nothing in the databases actually being updated, because the transaction has not reached the partial commit, because there may be some failures, so nothing needs to be done.

So, essentially what all these means is the something very interesting is that, the write record does not need the old value, write does not need old value. Why is it? Because, it will never need to undo, because only the new values need to be recorded, because when there is a transaction write, when it is committed partially; that means, all the transactions in all the operations you need are successful.

Otherwise, there is no need to go back to the old value, because the database already contains the old value. So, it is not needed, so the entries for this deferred database modification, the write entries are of this one, it is simply write the transaction, this says what is the data item and the new value, that is it. So, the old value is not needed. Now, what happens is that, after the transaction partially commits, there (*commit, T*) entry is written to the log.

So, (*commit, T*) is written to the log and this log is then stored to stable storage, this log is then flushed to stable storage. So, every record in the log is then stored to the stable storage and then, there are two things. So, now, what may happen is again, there are two cases. So, what will, a thing is that, so once the (*commit, T*) has been written; that means that the transaction is supposed to finish correctly.

So, that means, now what is happening now, if the transaction actually finishes correctly, the database should have the effect of all the writes that the transaction did. And now how does one get all the effects from the log? So, the database actually collects all those new write values from the log and just keeps on doing the writes. So, this is the write, the first write operation there, then the second write operation there and so on and so forth, it just keeps doing it.

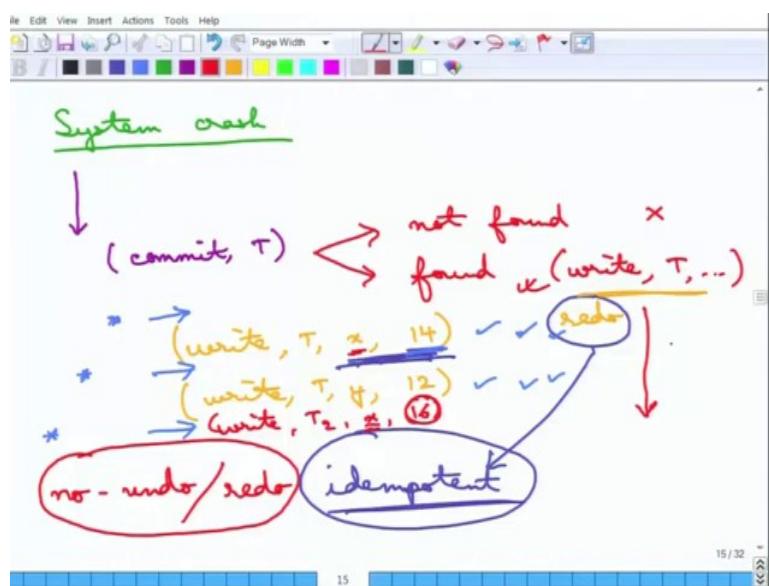
And again, so if all these writes are successful, then that is it, nothing needs to be done; the databases make sure that every write has gone through. Otherwise, what may happen is that, during this write may be there is a problem, there is a problem, because there is a system crash, there is a failure, etc, etc. Now, what needs to be done then? Nothing really, because the next time the system comes up, the system will check this log and will attempt these writes again.

So, it will keep attempting this log and till, it is essentially successful. Now, one important thing here is that, the couple of things. First of all, this  $(commit, T)$  is written to the log after the partial commit is being done successfully. Now, the  $(abort, T)$  entry is again not needed, just like the old value is not needed in a deferred database modification, the abort entry for the transaction T is also not needed.

Why is that? The abort is not needed because when does an abort happen, when a transaction fails, there is some failure. Now, when the transaction fails, it simply stops doing anything else, it simply restarts; that is it. Why is that correct? Because, the transaction has not yet changed anything into the database and when does the transaction actually start changing anything to the database, only after the  $(commit, T)$  entry has been made.

Now, this means that, everything in the transaction has gone through successfully. So, the transaction does not need to abort and therefore, the abort T entry, so this is the point, the  $(abort, T)$  entry is never used, it is because, it never written.

(Refer Slide Time: 10:51)



Now, whenever suppose there is a system crash that is happened, so if everything goes on successfully, then there fine. So, suppose there is the system crash happens, what does the database do, then the database of recovery management system, it reads to the log and finds out all the log records and finds out all the transactions for which there is a commit entry,  $(commit, T)$  entry.

Now, so these  $(commit, T)$  entry, if this is not found, so there are two cases, if this is not found; that means, none of the writes into this transaction have started. So, nothing needs to be done essentially. So, there is nothing to be done. On the other end, if it is found that means, if the transaction has said that, it is committed that means everything has to be successful.

Then, all the writes pertaining to the transaction needs to be done, these needs to be applied and these needs to be applied again and again, so this is called a redo operation, so those writes are needed to be done on the database. Now, the question is the following is that, suppose as part of this transaction there are two write entries,  $(write, T, x, 14)$  and then, there is a  $(write, T, y, 12)$ .

The issue is the transaction has written  $(commit, T)$  and there is a system crash. Now, the system crash may happen at this stage or this stage or this stage. Again we are assuming these write entries are atomic. So, either this complete write has happened or this. Now, let us take all of these cases one by one. Suppose a system crash happened at this stage, now we are doing this redo operation or whatever we are doing, so we are now writing all these two things together, so then that is correct.

Now, the second case is, this has happened at this stage, so that means the first write has gone through and the second write has not gone through. Now, the transaction or the log or the database or any system has no way of knowing whether it has gone through, the  $x$  has been written successfully or not. So, it does not care whether it has written successfully or not, it still goes ahead and writes the new value of  $x$  to it.

Now, what may happen is that,  $x$  may have been the old value or it may have the new value 14. Now, it does not matter because the new value 14 will be again written to  $x$ , which is doing it once more, which is not probably efficient in the sense of doing that operation again and again, but it is correct. So, that is more important, so this is correct and then of course,  $T$  is written.

And finally, the third case is both have been written, but the database did not know this, again it does not matter, it is just written again and again. The important point here is the following, is that, for all of these this redo operation may be done multiple times, even if  $x$  has been written, it may be written again and again and again. So, this redo must follow, what is called a, the redo must be, what is called an idempotent, this is an important term.

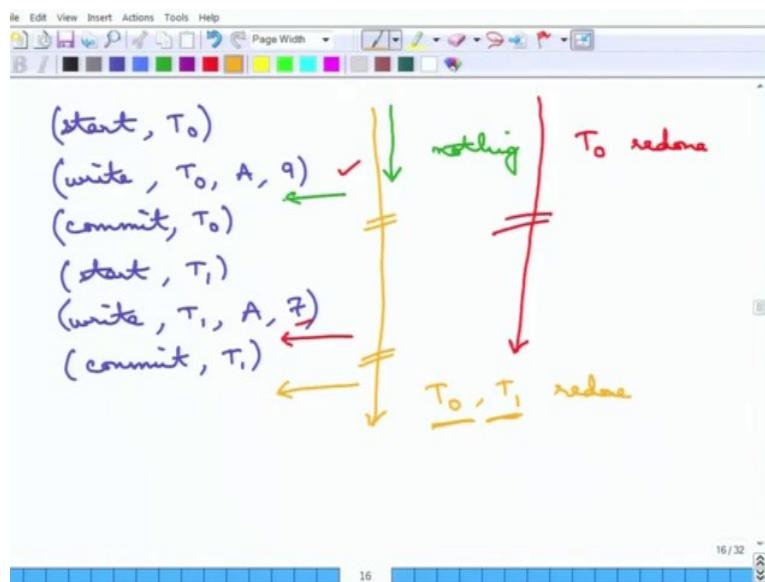
The idempotent means the following, is that, the multiple operations has the same effect as a single operation. So, redoing and operation redoing a write operation multiple times has the same effect is doing it once. So, in other words, if the value of 14 is written to x, in the example, multiple times it has the same effect of writing 14 to x once and it does not a matter.

So, redo must be idempotent, it must be idempotent, because it may happen that the value 14 is written again and again and again but that is all right, so that is called a redo. Now, this redo is needed, but undo is not needed. So, this deferred database modification scheme is sometimes called a no-undo/ redo operation. Okay. So, this called a no-undo/ redo recovery scheme. So, this is the turn for the deferred database things.

So, no-undo because undoing is not needed, but redoing is needed. So, this is called no-undo/redo recovery scheme. And just to complete this, redoa must be done in the order that they appear in the lock; otherwise there may be wrong things. And a very small example to do that is that, suppose there is another write by some other transaction (*write, T<sub>1</sub>, x, 16*).

Now, it must happen that the final value is noted, because the x is a common. So, it must be done in this manner. So, 16 should be the final value; that is mean, so redoing must be done in the operation that they appear in the log. Before moving on to the next scheme, let us see an example.

(Refer Slide Time: 15:31)



So, here is a complete example of these things. So, there is a (*start, T<sub>0</sub>*); okay, that is the first

operation, then a write of  $T_0$  is writing a value to some data item A, which is 9, then there is a  $(commit, T_0)$ . So, then  $T_0$  commits, then  $T_1$  starts  $(start, T_1)$ , we are assuming this is the serial, then there is a write of  $T_1$  to the same A with value 7 and then, there is a commit  $(commit, T_1)$ . Suppose this first what was intended to follow.

And now, suppose there is a crash after the  $(write, T_0, A, 9)$  statement, so there is a crash at this point. Okay. Now, what happens is that the following things, so when the log is searched and there is no transaction that is found with the commit statement. So, nothing needs to be done, because there is nothing committed. So, nothing is being written by any of this transaction to the database and nothing needs to be changed.

So, that is done. Okay, fine. The next thing is that, suppose there is a crash after this  $(write, T_1, A, 7)$  statement. Then what needs to be done is that, again the log is searched and here this  $(commit, T_0)$  is being found. So, the operations of  $T_0$  need to be redone, so  $T_0$  is redone. So, essentially this  $(write, T_0, A, 9)$  this operation is being redone, but nothing on  $T_1$  needs to be done, because  $T_1$  has not committed.

So, that means, the value of A should not be 7 and it is not 7, because it has not committed. So, that A value the new value 7 has not gone through A. So, that is fine and finally, it may happen that the crash takes after  $(commit, T_1)$  is done. So, then, again this entire thing searched and both  $(commit, T_0)$  and  $(commit, T_1)$  is found. So, both  $T_0$  and  $T_1$  are redone and very importantly in the order of  $T_0$  first and then  $T_1$  on the order of the log records. So, redo of  $T_0$  must be done in the first and then,  $T_1$ . So, that is the example for this deferred database scheme.

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

**Lecture - 31**  
**Recovery Systems: Immediate Database Modification**

(Refer Slide Time: 00:16)



Let us now move on to another database modification scheme, recovery scheme, which is called an immediate database modification, as opposed to the deferred that we saw earlier. So, this allows uncommitted writes, so previously all writes were committed, so this allows uncommitted writes. So, that means even if the transaction has not committed, the write of it can go to the database. Okay. Fine. And the write record reads both the old value and the new value; both of them are needed, otherwise this is wrong. Okay.

Now, once more remember that log is written before the actual thing is being done; that is the log is written before the actual operation. So, this is very important and we will see why this is important. Okay. So, first the log is written, and then the actual write operation is, if needed, is done to the database. So, just like the same thing, when a transaction crashes, so again the log is searched and everything for this  $(commit, T)$  is searched. All the records, all the transaction with  $(commit, T)$  is been searched.

And if  $(commit, T)$  is also these two cases if found, so this  $(commit, T)$  record is searched, if it

is not found, but the transaction is started, so that means, there is some  $(start, T)$  somewhere. The transaction has started, but it has not committed. So, that means, that what may happen is that, there may be some writes in between which has gone to the database, but the transaction has not been committed, so there may be some errors in this.

So, what the transaction is do is to undo. What does undo essentially does is that, the transaction, this is the fail transaction, but it has, may have already written some values to the database, assuming it is going to be correct, now it is not correct. So, it must be undone. So, what is an example, if we see that banking example back, a failure transaction may be that A as money as been debited, but before B's been credited; it has failed.

Now; that means, the  $(commit, T)$  is missing. So, that debit operation from A's account must be undone; that means, A's must be restored to the previous value. Now, for undoing of course, you read the old values, this is undone. If, on the other hand, the  $(commit, T)$  record is found, then what may happen is that, not all the records, not all the writes in that may have actually traversed to the database. This means traverse to the database, right, not all of them may have actually gone to the database. So, this needs to be redone.

So, all the transaction needs to be redone. So, now, the point is the same as the previous scheme is that the transaction is said it as committed; that means, that is successfully completed all the operations, but all these operations may not gone to the database. So, the database values may not have been updated and there is no guarantee, it does been updated, it is only been the  $(commit, T)$  is only being written to the log; that is the log is being checked. So, it is needs to be redone.

So, it is just to ensure that if the transaction said it has successfully completed all its operations are actually being done on the database. So, that is why both undo and redo is needed. So, this is why, these scheme is also sometimes called an undo/ redo recovery scheme, because both undo and redoing is needed. Okay. One important thing is that, both of these, as I said earlier, it is redo and undo both are idempotent operations, because there may be multiple crashes etc and undoing and redoing may be needed to be done again and again and again.

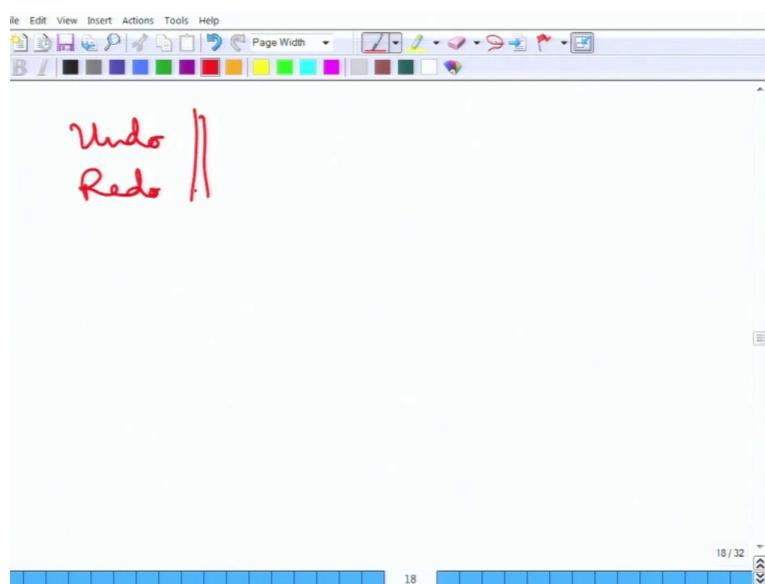
So, this must be an idempotent operation. Okay. More importantly how is this undoing and redoing done? What that is the order in which these things are being done, so this is the way. So, undoing must been done in the order of the reverse of the log. So, undoing is done in the

order reverse of the log and redoing is in the forward order of the log. So, undoing is in the reverse order of log records, which is intuitive and easy to understand, while the redoing is in the forward order of log records.

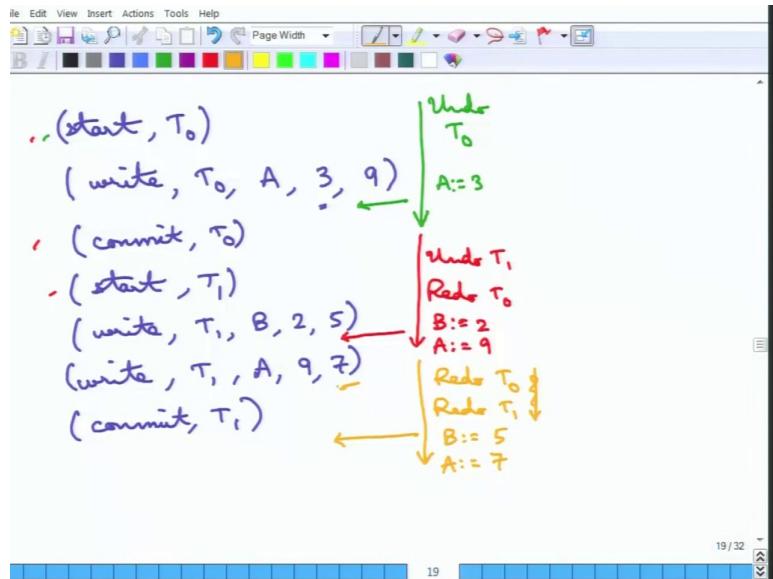
Okay. But, even then one more important question comes is that, suppose there are transactions, one transaction needs to be undone and other needs to be redone, which one should be done earlier? Should the undo be done first and then, the redone or should the redone be done first and then the undo. The answer is the undoing must be done before the redoing. So, first everything that a failed transaction has done must be undone and then, the transaction that has correctly completed will be redone.

The reason is, suppose consider a particular data  $x$ , now suppose transaction  $T_0$  and  $x$  has failed while transaction  $T_1$  on  $x$  has passed. Now, that means, the write value of  $x$  should be reflected as whatever has been written by  $T_1$ . So, if  $T_1$ , the redoing of  $T_1$  is done first and then, undoing of  $T_0$  is done,  $T_0$  may then rollback the value to what  $T_1$  has written.

So, as what  $T_1$  has written, so that is not the correct thing. So, first  $T_0$  should undo the value whatever it has written to  $x$  and then  $T_1$  is redoing it. So, that because  $T_1$  is correct, has finished correctly (Refer Slide Time: 06:13) so, the value that after  $T_1$  has done, is now reflected into the database. So, this is a very, very important point, that undoing must be done before redoing. So, undo and then redo, this is very, very important. Okay. Now, let us see an example to understand this completely.



(Refer Slide Time: 06:26)



So, here is the example, so  $(start, T_0)$ , and this is the same example as we saw earlier, but with one important difference is that the old values are now recorded. So, the 3 is the old value, the last time we did not require the old value, it was not there. By the way one important thing is that, the abort statement is required here, because the transaction may just say, I mean this is it has failed and it will write the abort statement.

Anyway coming back to the example, so there are two transactions is  $T_0$  and  $T_1$ , and  $T_0$  has written the value of A, just change it from 3 to 9 and  $T_1$  is changing B from 2 to 5. And then  $T_1$  also changing the value of A from 9 to 7 and then, there is a  $(commit, T_1)$ . So, now, suppose the crash happens after this statement the  $(write, T_0, A, 3, 9)$  statement, now this is being searched, the log is being searched, what is found is that transaction  $T_0$  has started, but it has not committed.

So, what is being done is that  $T_0$  is undone. So, undo  $T_0$ , because  $T_0$  has started, but has not no commit thing. So,  $T_0$  may have already written this value of A to 9, which is wrong, because this  $T_0$  wrong, it should rollback the value to 3. So, undoing of  $T_0$  is done. So, the value is now set back to 3. Okay. Now, suppose there is the crash after the  $(write, T_1, B, 2, 5)$ , but only the after the B statement, this is the thing.

So, again this, log is being searched etc and what is being done is that  $T_1$  has started, but not finished. So,  $T_1$  is being undone and  $T_0$  has started and finished so  $T_0$  needs to be redone and the order must be the undoing of  $T_1$  is first and redo of  $T_0$  is then. So, essentially undo, once

more, undo  $T_1$  will be done earlier than redo of  $T_0$ . So, then, this will, undoing of  $T_1$  will restore the value of B back to 2 and redoing of  $T_0$  will restore the value of A to 9, because this is a, is the correct new value; that is needs to be redone. Okay.

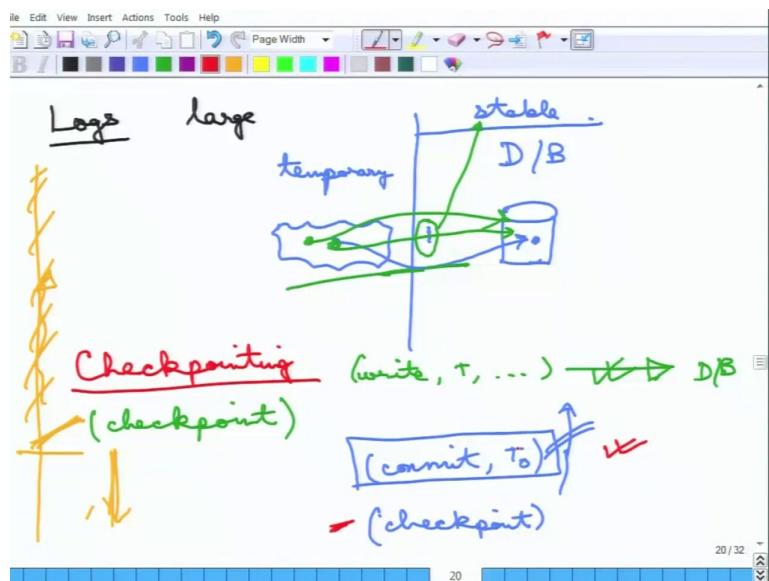
And finally, suppose there is the crash after the (*commit, T<sub>1</sub>*) statement , so again everything is being searched and then, both  $T_1$  and  $T_0$  needs to be redone and again the redoing operation should be done in the order in which they have committed. So, redo  $T_0$  and then, redo  $T_1$ , because otherwise you see the value of A will be wrong. So, again after this has been done, B has the new value of B is written and A is the new value, which is the 7, this value.

If this is not done in the order, then the A will be wrong. This is just like the previous scheme. So, that is the point of these two recovery management systems, the deferred database modification scheme and the immediate database modification. Just to summarize the deferred meaning, the writes are deferred till it commits and immediate meaning, the writes can go immediately, after it has been written on the log, but the log record must be written first.

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

**Lecture - 32**  
**Recovery Systems: Checkpointing and Shadow Paging**

(Refer Slide Time: 00:11)



Now, one problem that happens is that, so, the logs can becomes very large, so they are very large logs because every ... all the transactions are being written in logs etc and suddenly one transaction has finishes. Now, it does not make sense to go over all the logs for some very old transactions, but to maintain the correctness there has to be a way of saying that if the log is written for a particular transaction, all the writes in the transaction have actually gone to the database, the database has actually seen the effect of this, so why is it?

Now, just to highlight my point, what is being done is that the following, so there is the database which has contained the actual values of this data items and then there is a temporary copy of the database, so, all the writes are temporarily changed in this, so, there is a temporary copy. And the log records only ... the log ... there is an entry in the log it only says that it has been changed here. Now, that does not mean that this has actually propagated to the database and has been done and this is the stable storage. So, this part is the stable storage, so this will never be lost.

So, the question is if there is many, many, many, many, log entries then there are many, many,

many, many changes in the temporary thing, but it is not sure which temporary things have gone to the database, that is the whole idea. So, the logs when go very, very large and searching in the logs and finding on which transactions have started but not committed, which has all those things can become very much time consuming. Now, to prevent all of that, an idea is used which is called a checkpointing; I am sure many of you have heard this idea of checkpointing.

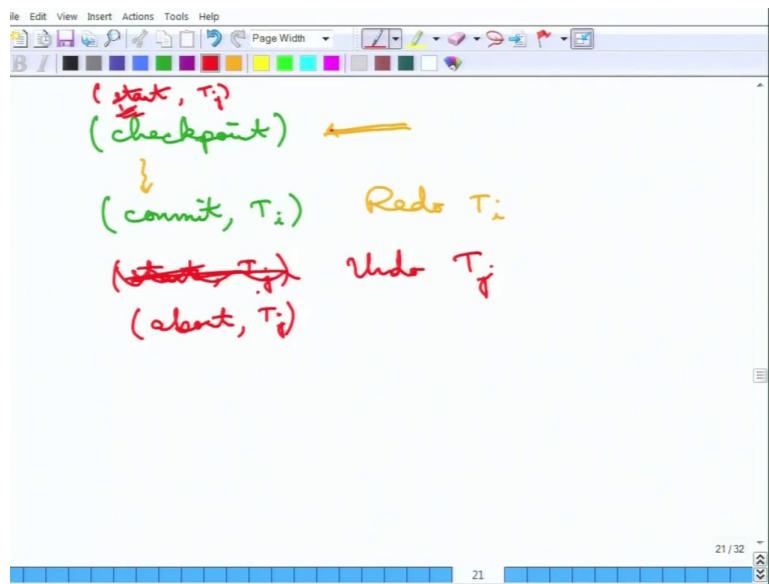
Checkpointing essentially is trying to make sure that whenever a checkpoint is being taken on the log, everything after that checkpoint is correct. Now, correct meaning the following thing is that, so there is a very large log record and a checkpoint is taken, now, this means that everything that is on the log is being put to the stable storage. So, the log upto that point is being put to the stable storage. More importantly all the pending writes up to the checkpoint, all the pending writes are also flushed to the database.

So; that means, if I say there is an entry which says (*checkpoint*) in the log, so if checkpoint is being done, so if there is a write entry for some transactions before the checkpoint, this effect has actually go on through to the database, this is made sure. How is it being made sure? It is being actually forced on the database. So, before the checkpoint is being done, it is actually forced on the database, the log etc. The log is also maintain in the stable storage, the log record also goes, the log is also written on the stable storage and all the write records in the log has gone to the database, it is made sure before the check point entry is being made.

Now, here is the point, why is it that a checkpoint entry will now help is that, now if the system crashes all that the log needs to do is to find the last checkpoint, it is sure that everything before that is correct. So, it needs to worry only after .... of things ... after the checkpoint. So, even if the log is very large, everything up to this larger part of this thing, where the check point has been taken is correct, so it does not need to bother about it, so that is the whole point.

And just to take it a little bit further, so, suppose there is the transaction (*commit*,  $T_0$ ) after which there is a (*checkpoint*) is being written. So, that means, ... so, that fact that there is checkpoint meaning that all the write operations for this ... below this has been correctly return that means all the operations of  $T_0$  have been correctly written. So,  $T_0$  does not need to be bothered about any more, this is all done, just because there is a checkpoint entry. So,  $T_0$  is ... nothing needs to be done, so no redoing needs to be done for  $T_0$ , that is fine.

(Refer Slide Time: 04:19)

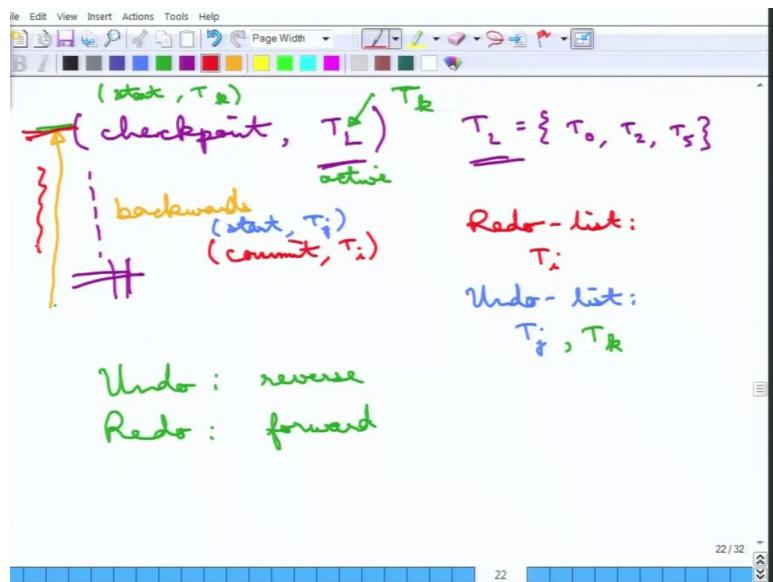


But, there may be other problems, in the sense that something there may be a (*checkpoint*) entry and then after some point in time, there is a commit to some transaction say  $T_i$ , (*commit*,  $T_i$ ). So, now, what it is being guaranteed is that every write up to checkpoint is being done. But, there may be write entries after the checkpoint and before the (*commit*,  $T_i$ ) has been done. So,  $T_i$  is not guaranteed that all the entries, all the writes of  $T_i$  is being done correctly. So,  $T_i$  must be redone this cannot be avoided, so  $T_i$  must be redone that is the thing.

And then, anything, suppose there is another thing where there is a (*start*,  $T_j$ ) and of course, there is no, the start of  $T_j$  may be here or the start of  $T_j$  may be here. And, but there is no commit entry of  $T_j$  that must be undone. Because, again if the check point only guarantees that everything has been written correctly, but. So, there may be some write operations before the check point and then the  $T_j$  may about here.

So, this instead of this thing this may be aborted here or it may not even write abort. So,  $T_j$  must be undone, because the checkpoint only guarantees that the writes are gone through there. So, it must be undone, so that is the thing, so checkpoints... So, any transaction, just to summarize, any transaction that has committed before the checkpoint entries being made, does not need to be bothered about it because, all its writes have gone through correct. Anything that has committed after the checkpoint needs to be redone and any other transaction that has started, but does not have a commit entry needs to be undone; even if it is before the check point, that is the whole idea.

(Refer Slide Time: 06:06)



Now, just to make it even more efficient, this checkpoint can be augmented by the list of transactions that are active at that point. So, instead of just writing checkpoint, so it can also say there is a list of entries,  $(checkpoint, T_L)$ , so  $T_L$  is the list of entries. So,  $T_L$  essentially is a list,  $\{T_0, T_2, T_5\}$ , so it can say  $T_0$  is active,  $T_2$  is active,  $T_5$  is active and so on and so forth. So, this is the list of the entries, so this list is written as part of the checkpoint entry. Now, after that there are some more log entries etc and suppose there is a crash here.

So, once the crash happens the log is read backwards up to the checkpoint and the ... So, this is read backwards this is very importantly, this is read backwards up to the checkpoint and the following thing is done , if there is a  $(commit, T_i)$  is found in this backward scan then there is a redo list, where  $T_i$  is added to the redo list. Now, on this scanning backward if suppose on the other hand somebody finds the  $(start, T_j)$  then there is an undo list, where  $T_j$  is added.

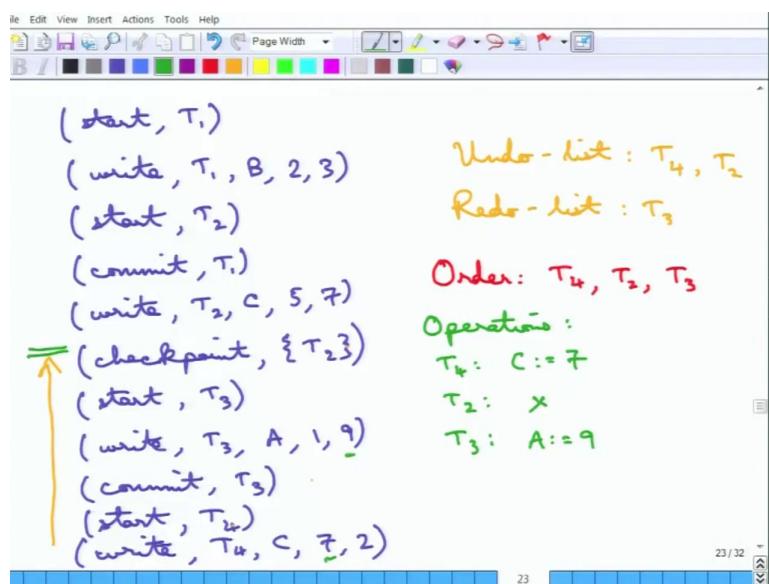
And suppose there is some other  $T_k$  which started earlier ... Now, note that scanning of the log goes only up to the checkpoint, however, what will happen is that since  $T_k$  started before the checkpoint and has not committed,  $T_k$  must appear somewhere in this  $T_L$ . So, this is, must be in the active, in the list of active transactions, so this is the list of active transactions. So,  $T_k$  must be appearing somewhere in that. So,  $T_k$  is not found using this scanning, but  $T_k$  is in the active list, then  $T_k$  is also added to the undoing list.

Of course, if  $T_k$  has not been in the redo list; that means,  $T_k$  there is no commit  $T_k$  entry of this, if there is  $(commit, T_k)$  entry, then  $T_k$  would have been added to the redo list then nothing

needs to be done. But, otherwise  $T_k$  needs to be added to the undo list. So, this is the way how the redo list and the undo list are being made and then we follow the same thing. So, undos are first done in a reverse order and then the redos are being done in a forward order.

So, the transactions in the undo list are first revert it back in the reverse order as they appear in the log and the transactions in the redo list are then done in the forward order as they appear in the log. And one more thing only the operations after the checkpoint, so, this is read only up to the checkpoint, so, only the operations after the checkpoint needs to be either undone or redone, that is it, because before that everything has been either undone or redone correctly.

(Refer Slide Time: 09:14)



And here is an example to see, suppose there is a  $(start, T_1)$  then there is  $(write, T_1, B, 2, 3)$ . There is a starting of  $T_2$ ,  $(start, T_2)$  and then  $T_1$  commits,  $(commit, T_1)$ . Then there is a write by transaction  $T_2$  to another named item  $C$  the value  $7$ ,  $(write, T_2, C, 5, 7)$  then there is a checkpoint. Now, this checkpoint will essentially say the list of active transaction which is only  $T_2$  at this point, so,  $(checkpoint, \{T\})$ . So, the checkpointing is a heavy operation, when checkpointing is done lots of things needs to be checked.

So,  $(start, T_3)$  then there is a  $(write, T_3, A, 1, 9)$ , then there is a  $(commit, T_3)$ . Then, there is a  $(start, T_4)$  and finally, there a  $(write, T_4, C, 7, 2)$ . So, there is a entire list of log record that we see. So what happens is that, ... so, the first thing to find out is that ... so, then there is a crash, at this point. So, now if the first thing to find out is what is the list of undo and redo

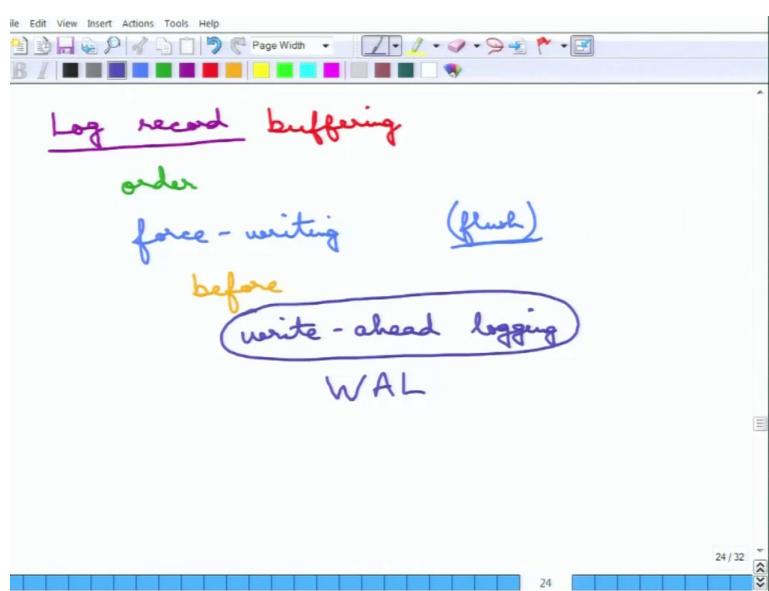
list, so the undo list is the following.

So, undo list and the redo list. So, the undo list is, it is scan backward there is a  $(start, T_4)$  that is being noted. So, that is added to the undo list, correct? And then it goes to the checkpoint and sees that there is an active transaction  $T_2$ , so that is also added to the undo list, because that is not in the redo list. So, the redo list, it finds commit  $T_3$ , so the redo list is at the list is only  $T_3$ .

Now, note that  $T_1$  is not appearing at all, because  $T_1$  committed before the check point that is it. So, then the order of operations will be the undo list first and then the redo list. So, there are two things in the undo list how will be done, it is based on the when it started etc. So,  $T_4$  is first undone then  $T_2$  is undone and finally,  $T_3$  is undone. So, this is the order in which this things will be done and the order of operations will be the following.

So, for  $T_4$  the operation if the C is written back to the old value which is 7, it is written back to the old value and there is a only operations for  $T_4$ . For  $T_2$ , nothing appears here, so, no operations is being done, even though it has written something back, because you see the point it goes only up to the checkpoint and does the redoing and undoing of only those operations that appear up to the checkpoint. And for  $T_3$  the redoing of this thing needs to be done. So, A is written the new value of 9. So, that is the way this whole scheme with checkpointing takes place. We will have two more small things to cover.

(Refer Slide Time: 12:49)



So, the first thing is the very small issue of log records. So the logs are also written to the stable storage etc. And .. so, every time a database does a write operation or a read operation, something is being written to the log. Now, the question is the log is maintained as a temporary list in the main memory and the log needs to be written to the database, or not to the database, the log needs to be written back to a ... some stable storage.

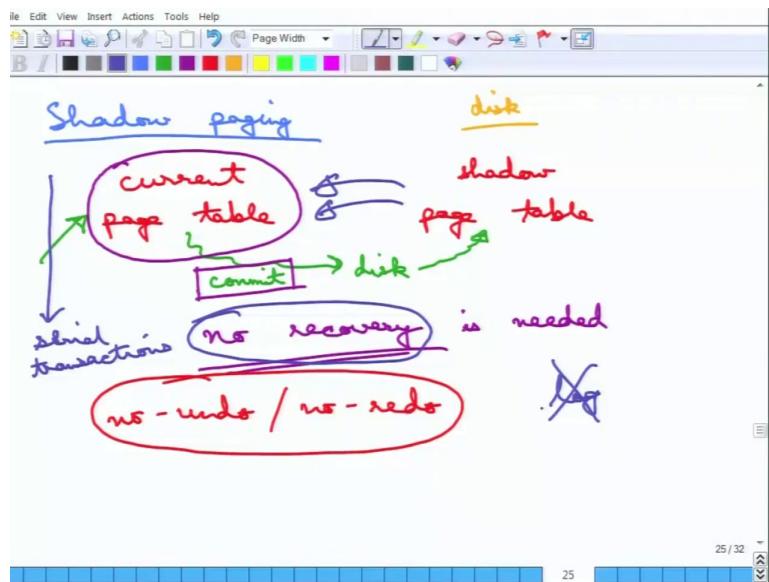
So, how many times will the log be written? So, there is a concept of log record buffering, which means that not every time a log record is being done, the log is written back to the stable storage. A couple of records are buffered together and then this is output to the stable storage. The reason is the same reason that we saw all those, why the inter disk page is accessed together. So, all the log records that fit into one disk block are first filled up and then that disk block is written back to the disk or the stable storage, that is the way.

And the records are, of course, flushed in the order in which they appear in the log. So, this is in the order of which they appear in the log. And then there is the concept of force writing is used, force writing meaning everything that is written, that needs to be written is forcefully written back to the disk. So, this is in the, if you know this programming languages, C etc, this what is called as a flush operations. So, flush means that it is surely gone to the stable storage, so that is the (*flush*).

So, this force writing of the log record is being used and even, so, if there are some commit entries etc, that is also flushed and all the log records. So, if there is a block of data that is written to the database, all the log records before to it must be done before. So, this log must be flushed before the actual, the writes are gone through, so that is the thing. So, the log records are written before the actual thing, that is why this is sometimes called a write-ahead logging.

That means, before the logging, before the actual write is done, the log is being written ahead. So, this is the write-ahead logging or this is sometimes known as the WAL rule, W A L, because it is the write-ahead logging, so that is all the thing about this log.

(Refer Slide Time: 15:23)



Then there is one more type of database recovery scheme that can be followed, which is called the shadow paging and although we talked about it very briefly last time, a little bit more details on this is as follows. So, the shadow paging is essentially there are two copies of the database are maintained. Now, we will assume that there are two page tables in the OS type. So, essentially forgetting about page tables the details of it there are two copies of the database are maintained and the database is broken up into disk blocks or pages.

So, these pages, the list of this pages are maintained and then there are two copies. The first one is the current page table, page meaning the disk block and the other one is the shadow page table. So, there are the two copies that are maintained. The shadow page table is maintained on the disk, this is maintain on the disk and no changes are done to it. So, no changes are done on the shadow page table. So, whenever something needs to be changed the corresponding page in the current page table is what is being changed.

Now, so all the update etc goes to the current page table. Now, whenever a transaction commits, all the changes that are being made to the current page table, all those things are flushed to the disk. So, this is all flushed to the disk. So, what does flushing to the disk mean? The flushing to the disk means these are written to the corresponding pages in the shadow page table, which is on the disk. So, that is being flushed to the shadow pages, the shadow page table is actually modified only when there is the commit, when a transaction commits, so that is the commit operation that has been done.

And if this is followed then actually what can be shown is that there is no recovery is needed, that is the simple scheme no recovery is needed. What it means to say no recovery is needed? Because you see what happens is when a transaction fails what happens all the copies or all these things are in the current page table which is simply lost that is it, there is no ... the shadow page table contains all the effects of only the committed transactions.

So, that means, nothing needs to be changed for the shadow page table and only the current page table is what is being changed. So, that is why this is also called a no-undo/ no-redo scheme, because nothing needs to be undone, nothing needs to be redone. However, at the end of all of these things the shadow page table is the new correct thing. So, that is the thing. So this works ... the this seems to be very useful, because no recovery needed, but there are some practical problems that happen with this. First of all, there are too many pages that need to be copied.

So, the efficiency is not much. The commit overhead also may be too high, because lots of things may be needed to the shadow page table, whenever a transaction commits. And the thing is serial, it really only happens for serial transactions. So, the transaction must come one after another, if the transactions have been interleaved, then this scheme may not work. So, this only works for serial transactions, the advantage, of course, is that no recovery is needed.

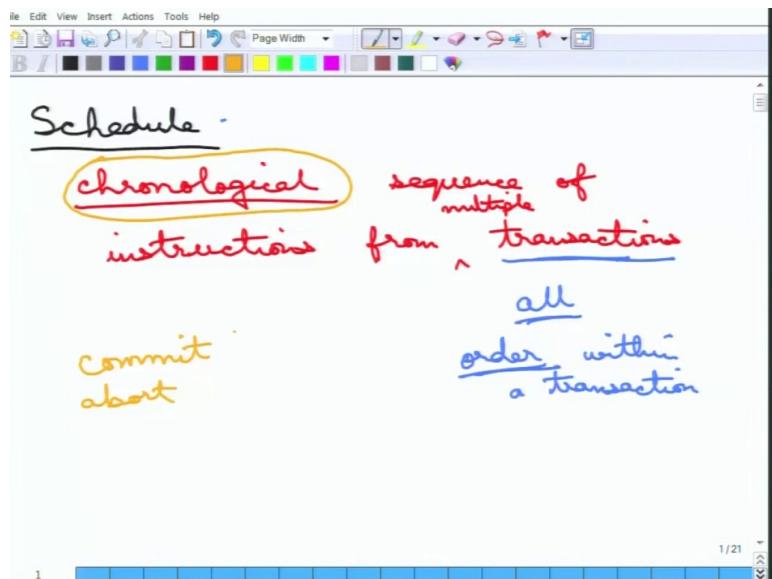
And so there are no logs, no overhead of writing the logs, etc, so logs are not read. So, there is no overhead of writing the log. So, that ends the topic on database recovery management systems and we studied some couple of schemes, one using logs and with checkpointing etc and finally, the shadow paging. We will next covered the important issue of schedules.

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

**Lecture - 33**  
**Schedules: Introduction**

Welcome, we will continue with our study on the database transactions. We have seen the basics of what the database transaction is and what are its properties and we have also studied the recovery management systems. So, today we will start on something which is called Schedules.

(Refer Slide Time: 00:29)



So, schedule, so what is first of all a schedule, a schedule is essentially a chronological sequence of instructions from multiple transactions. So, the important point is the chronological sequence of instructions from transactions, so that the number of transactions may be one or more and generally, so it may be multiple transactions. Now, the important part is that, this is chronological; that means all the instructions from multiple transactions are written one after another.

So, it may be saying that a schedule may be in the form of transaction one writes to A, then transaction two writes to B, then transaction three reads from A and so on and so forth. It is a chronological sequence, so it is a time ordered sequence. Now, there are certain rules of what can constitute a schedule, if a transaction appears, if one instruction of transaction appears in a

schedule, all the instructions must appear from the transaction.

So, all instructions of the transaction must appear to the schedule or none of the instructions of the transaction appear, so that is the first property. The other is the order of instructions within a transaction, so order within a transaction must be maintained. So, if the particular transaction let us concentrate the transaction 1, if transaction 1, first writes to A and then, reads A.

Then, in that schedule that contains transaction 1, it must be the transaction 1 first writes to A and then, reads from A, it cannot be swapped. So, the order of instructions within a transaction must be respected in the schedule that contains the transaction. Then, there is a same rule as earlier, so if the transaction commits, if a transaction finishes successfully, then it writes the commit.

So, that is also part of the schedule or otherwise, it will say abort and sometimes in some schedules, when we study the schedule commit and abort may be omitted, because it may not be needed for that particular study.

(Refer Slide Time: 02:53)

Example

$T_1$  transfers 50 from A to B, and  
 $T_2$  " 10% from A to B then

$\pi_1(A) \vdash T_1 \text{ reads } A$

$\begin{cases} \pi_1(A); A := A - 50; \omega_1(A); \pi_1(B); \\ B := B + 50; \omega_1(B); \pi_2(A); t := A / 10; \\ A := A - t; \omega_2(A); \pi_2(B); B := B + t; \\ \omega_2(B) \end{cases}$

But anyway, let us go to an example to understand, what exactly is a schedule? So, here is an example. Suppose there are two transactions. So,  $T_1$  transaction  $T_1$  transfers, let us say some 50 rupees from account of A to that of B and then, transaction  $T_2$  transfer 10 percent of money from A to B. So, these are the two transactions and both of them can be represented in

a single schedule and this is how, it can be represented.

So, the first one transaction  $T_1$  transfer 50 from A to B, what can be done, it is a  $r_1(A)$ .

Now, please note the notation here  $r_1(A)$ , this is a notation that will be using again and again for schedules. So, that first thing is that, this is the read operation; that is why, this is an r, then this is 1; that means, this is what transaction  $T_1$  is doing. So, this read is done by transaction  $T_1$  and A is being read. So, this essentially means that the form meaning is that transaction  $T_1$  reads A. So, that is the form; that is the notation.

And if it is read it is r, otherwise read it is W, which is for write, anyway coming back the point of this schedule, what it can be written in the following manner is that,  $r_1(A)$  and then, we will write it sequentially. So, in a chronological order and we will write it from left to right as a natural reading. So,  $A = A - 50$ , then it writes to A, then what happens it reads B, it is still its everything is still going on for transaction 1.

Then, plus 50, then there is transaction 1, then writes to B, this completes transaction 1 and if you remember this first transaction  $T_1$  transfer and then, this transaction happens. So, this will then write at as the following manner read now transaction two starts reading and there is a temporary variable that is made use of. Let us say which is 10 percent of A, which is

essentially  $\frac{A}{10}$ .

Then,  $A = A - t$ , then write  $W_2(A)$ , then it is similar to what the first transaction has been doing. So, this money is being transferred to B, this entire thing is the schedule, which is consist of two transactions  $T_1$  and  $T_2$ , so  $T_1$  and then  $T_2$  correct? So, now, this is the one schedule, now there may be another schedule; that is written in the following manner.

So, there may be little bit of swapping or transposition of the instructions. Now, one important thing is that, this transaction is called a serial schedule. Why it is called as serial, because you see transaction one first completes all it is operations, this finishes everything here and then, transaction 2 starts. So, this is why this is called a serial. Now, also note that, we have not mentioned here the commit and abort statements, because what we are going to study here is about serializability.

(Refer Slide Time: 06:33)

The image shows a digital whiteboard interface with a toolbar at the top. A handwritten note is written in yellow ink. The title 'Serializability' is written in yellow. Below it, a sequence of database operations is enclosed in a red box:

$$\begin{aligned} & \text{r}_1(A); A := A - 50; w_1(A); r_2(A); t := A/10; \\ & A := A - t; w_2(A); r_1(B); B := B + 50; \\ & w_1(B); r_2(B); B := B + t; w_2(B) \end{aligned}$$

Below the box, the word 'serial' is crossed out with a large red X. Underneath the X, the word 'equivalent' is written in green and enclosed in a green rectangular box.

So, the study that we will be doing, the first thing that we will be studying about this is called the serializability and that is that will not require us to note whether the transaction has aborted or committed. Now, coming back to this example, so this is the serial schedule. Now, if quantise the entire schedule and do certain transpositions as I have been saying, then an equivalent schedule can be produced. Now, this is the equivalent schedule, so let me just write down the equivalent schedule.

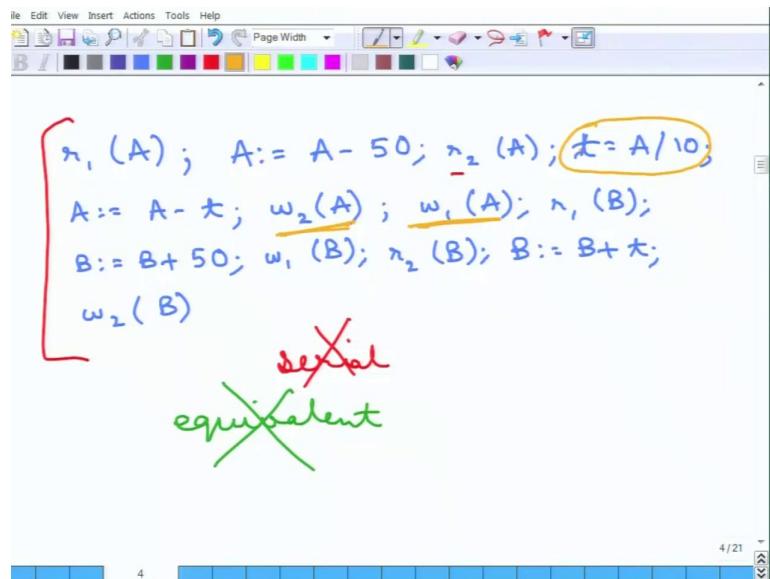
Note, very importantly, what I am doing here is before transaction 1, has written to the B etcetera, transaction 2 starts reading A, which is fine, it should not be any problem. And we will argue our whether there is the problem or not etcetera, but this is essentially, what is being done. So, now, one can do this, so this is not serial, because the instructions of 1 and 2 are inter-leaved.

So, 1, then 2, then again 1, then and again this 2, etcetera, so this is not serial. So, this is not serial; however, as one can very clearly see from the semantics of the transaction that, this is equivalent with the serial schedule that we saw earlier. So, this is equivalent, this is the very important term, why is it call equivalent is the following idea is that equivalent, because if instead of the serial schedule, if this schedule was run.

So, the transactions are inter-leaved and transaction 1 and 2 are taking place concurrently, the effect on the database finally, assuming both the transactions of course, we are assuming both the transactions finish successfully. Then, the there is this schedule and the serial schedule,

the effect of both of the schedules is exactly the same, no matter, what the state of the database is. Thus, if they start from the same state of the database, then they end up in the same state of the database. So, that is why, these are called equivalent.

(Refer Slide Time: 09:19)



Now, we will study on this a little bit more, but before that here is another schedule and which is the following is that. So, we saw one serial schedule and we saw one schedule, which is equivalent to it. Now, let us see another schedule, which is the following, this is another schedule. This is again not serial as we can very clearly see that this is not serial because again this is interleaved, so  $r_2$ , then again  $r_1$ , then  $r_2$  and so on and so forth, so this is not serial.

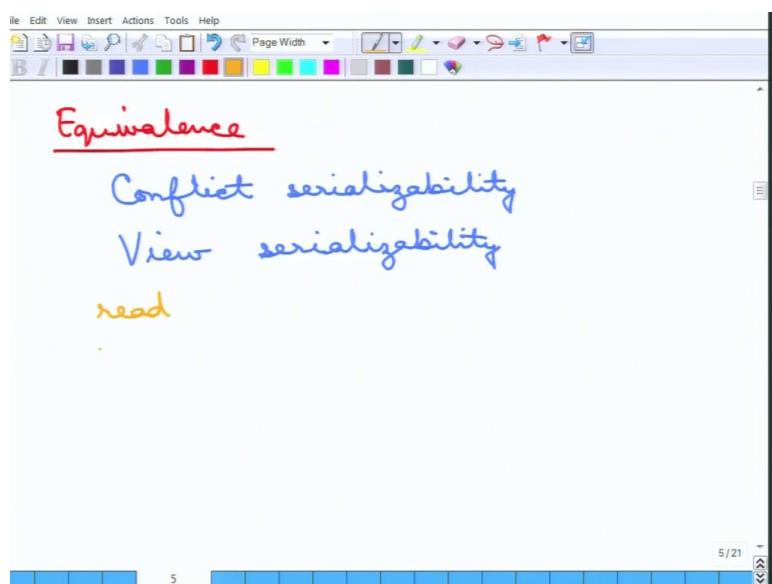
But, also this is not equivalent to the serial schedule that we earlier saw, this is not equivalent, why this is not equivalent, a little bit of examination is show that, this is not equivalent, because look at what T is here. So, this is what T is reading is this temporary value, this is the 10 percent before transaction 1 has written to the A. So, this is the 10 percent of this original A.

And so this is not correct because this T, when it is reads, it is not going to be correct, this  $W_2(A)$  is first written and then  $W_1(A)$  is written. So, this is not equivalent, this will not have the same effect on the database as the serial schedule. So, essentially the study of serializability of schedules is the following is that, suppose as schedule is given with two or more transactions and suppose, there is a serial order of the transactions.

So, serial order meaning, so suppose there is a two transactions either  $T_1$  all the operation of  $T_1$  happen and  $T_2$  starts and complete all its operations or  $T_2$  first finishes all its operations and then,  $T_1$  starts. So, these are the two serial schedules. Now, an another schedule is said to be equivalent to this serial schedule, if the effect of that schedule is the same as the serial schedule as one of the serial schedules.

So, that is what the equivalent and we will try to formally show how equivalences can be shown So, this is the study of serializability that we saw. So, this is essentially just once more, this is this study is called the serializability.

(Refer Slide Time: 12:12)



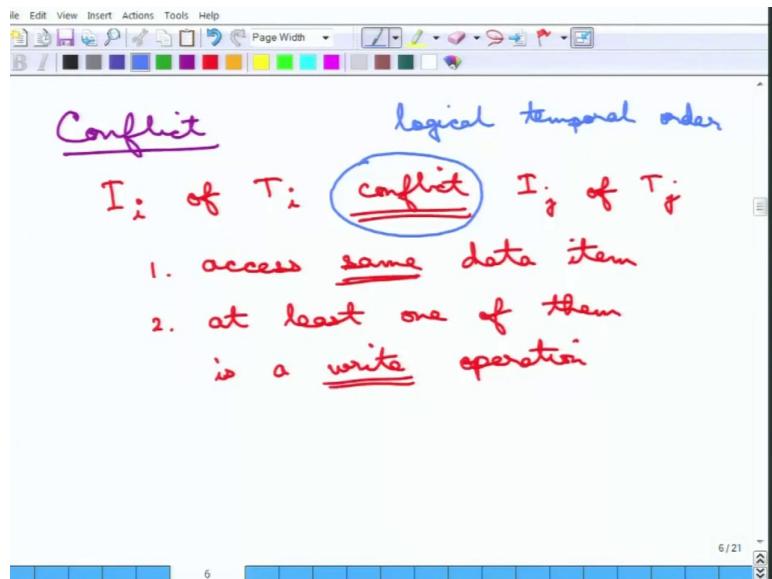
So, there are different forms of this equivalence, so the equivalence. So, essentially we are trying to say when are two schedules equivalent. So, for that there are two notions of equivalence first is the conflict serializability and the second is the view serializability. So, these are the two notions of equivalence of serializability that we will study. Now, what essentially is this, so we will see this?

But, there are some assumptions, first of all the transaction ignores every other operations other than read and write. So, read and write are the only two operations that are consider by a transaction. For the purposes of equivalence, note that, this is only for the purposes of equivalence and serializability, whatever you want to say.

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

**Lecture - 34**  
**Schedules: Conflict Serializability**

(Refer Slide Time: 00:09)



A conflict, so an operation, so an instruction  $I_i$  of transaction  $T_i$  is said to conflict with another instruction  $I_j$  of another transaction  $T_j$ , if the following thing happens is that, they both access the same data item, this is very, very important, same data item and at least one of them is a write operation. So, now, one can intuitively see, why are they said to be conflicting, if you remember, there are this RAW, read after write, then write after write WAW and write after read WAR conflict.

Now, what is the problem, why are they set to be conflicting, this instruction  $I_i$  and instruction  $I_j$  of two different transactions, there must be of different transactions, number 1. Then, there must be accessing the same data item and one of them must be write. Now, what is why is this conflict is very simply the following thing, is that suppose  $I_i$  should have taken place before  $I_j$ .

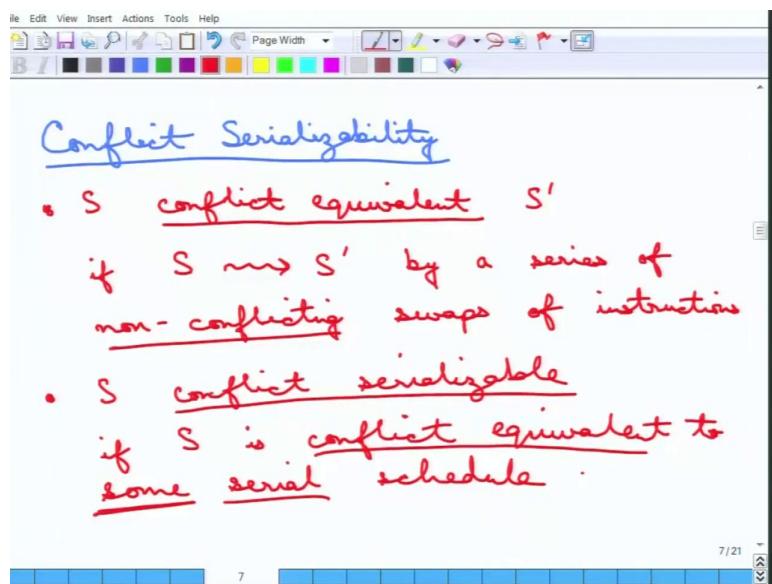
Now, they are conflicting, because in another schedule which is equivalent, this order of  $I_i$  and  $I_j$  must be preserved, it cannot happen that in the another schedule,  $I_j$  is before  $I_i$  and

then, it is not equivalent. Why it is not equivalent? Because, you see that this read write conflicts write, write or write read conflict will happen. So, that is why, they are said to be conflicting.

This is the essentially the definition of why they are set to be conflicting. So, intuitively these are conflict is this conflict, this enforces a logical temporal order on this instruction. This is the logical temporal order; they cannot be violated; that logical temporal order cannot be violated. And now, that is one thing, now on the other hand, the other side of it is that, if there are two instructions, which do not conflict.

Then, they can be interchanged and they can be placed anywhere before any other one and then, that is not a problem. So, that is the other side of it and that is what is we are going to study in much more detail.

(Refer Slide Time: 02:49)



So, conflict serializability, so this is the conflict and we have studied serializability. So, the next thing what we will do is we will study, what is called the conflict serializability. So, first of all a schedule is conflict equivalent is called conflict equivalent to another schedule  $S'$ , if  $S$  can be transformed to  $S'$  by a series of non-conflicting swaps of instructions.

So, non-conflicting swaps of instructions. So, what does it mean? So, suppose there is a schedule  $S$  and then, there are some instructions in it of course and then, let us pick any two instructions within  $S$ . And if that do not conflict and they cannot be inter change. So, let us

inter change it and let us keep on doing this and suppose of by the process of this, one gets to  $S'$  from  $S$ .

Then,  $S$  and  $S'$  are conflict equivalent, because what they have essentially done is that, they have ensured that if two instructions do not conflict, they have been swapped. So, on the other hand which also means that, if two instructions conflict, their logical order has been maintained, so these two schedules are equivalent in the sense of the conflicting instructions, the conflicting instructions maintain the same order, so that is why, they are called conflict equivalent.

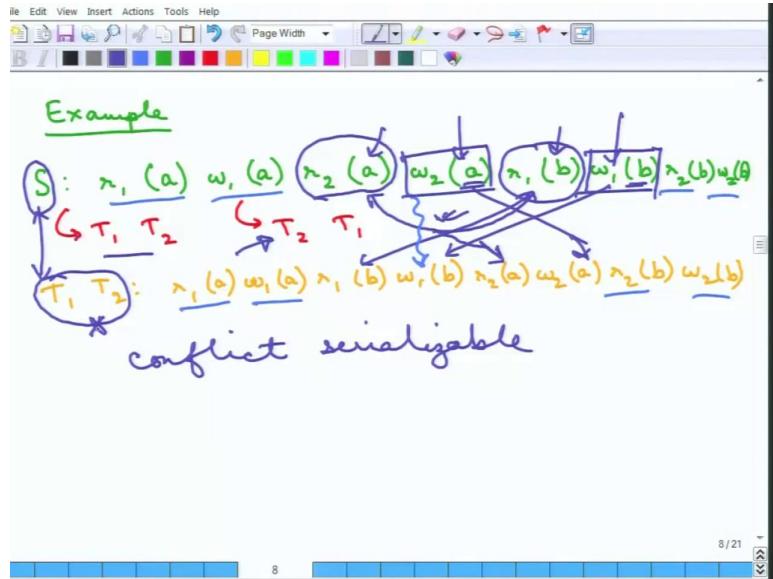
So, that is the definition of conflict equivalent. Now, this is the first definition, now  $S$  is said to be conflict serializable, this is conflict equivalent, conflict serializable. This is the definition again, if  $S$  is conflict equivalent to some serial schedule, so once more  $S$  is conflict serializable, it is given to some serial schedule.

So, it does not need to be conflict equivalent to all the serial schedule, but they must exist at least one serial schedule that  $S$  is conflict equivalent to and then, it is called conflict serializable; that is the definition of conflict serializability. Now, why are we studying it, because if  $S$ , if a schedule  $S$  is a conflict serializable that means that, it is equivalent some serial schedule.

That means, it will preserve the database consistency, because we know that if the transactions are, if the transactions take place one after another serially, then they preserve the database consistency. Because, each transaction by itself preserve the database consistency, so if transaction  $T_1$  happens, then it preserve the database consistency and then,  $T_2$  happens, then it is also preserve the transaction, then it also preserve the database consistency.

So, if there is an another schedule, which even though is interleaves  $T_1$  and  $T_2$  is conflict equivalent to it, then it also preserves the database consistency.

(Refer Slide Time: 06:17)



So, now let us see an example, so suppose  $S$  is the following, so we will use the same notation and we will even drop the semicolons. So, the first question is, is this conflict serializable, so that is what we will be trying to answer. So, that means, to say the whether this is conflict serializable or not, it must be equivalent to either. So, there are only two transaction  $T_1$  and  $T_2$ , so it must conflict equivalent to either  $T_1 \rightarrow T_2$ , or to  $T_2 \rightarrow T_1$ .

Now, let us see, whether it is conflict equivalent to  $T_1 \rightarrow T_2$ . So, first of all  $T_1 \rightarrow T_2$  is the schedule. So, what is the  $T_1 \rightarrow T_2$ , we will write down all the instructions of  $T_1$  first in the order that they appear, that cannot be violated, so  $r_1(b)$ , then  $w_1(b)$  and then, all the instructions of  $T_2$ . Now, we will see that, if we can arrive at this from  $S$  through series of non-conflicting swaps.

So, now, what is the first thing? So, first thing is that, let us see how to do it. So, this is fine, there is no problem with this too, similarly these two have no problem, because they have been starting and they have been ending. So, now what has been swapped is essentially and consider that  $r_2(A)$  and  $r_1(b)$  has been swapped, because this  $r_1(b)$  has come here and  $r_2(a)$  has come here.

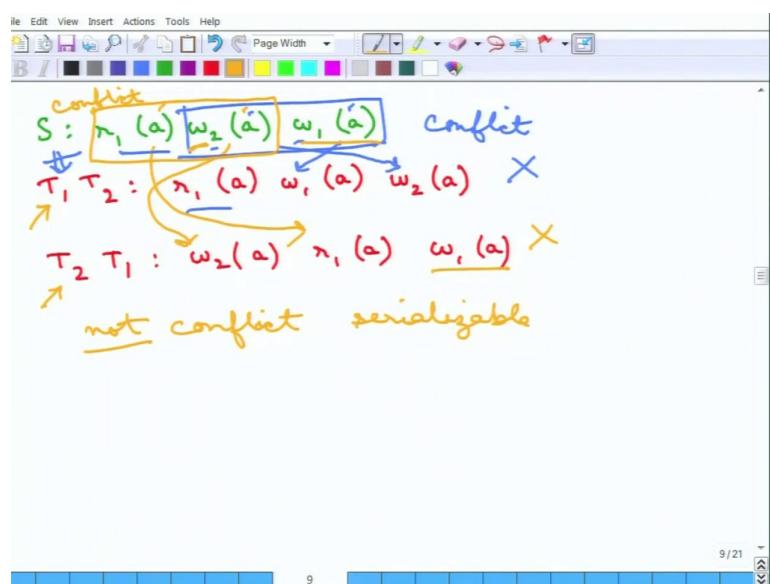
So, are these two instructions non-conflicting, yes, because they do not operate on the same data item, very simply, so these are non-conflicting and this swap is fine. And how about

these 2, then these two have also been swapped  $w_2(a)$  has been swapped with  $w_1(b)$ , because  $w_1(b)$  is coming here and  $w_2(a)$ , is coming here. Are these two non-conflicting again it is yes, because these two separate data item.

So, that means that S and  $T_1$  and  $T_2$  are equivalent; that means S is conflict serializable, so the answer to this is S is conflict serializable. So, there is no problem with this as far as database consistency is concerned. Now, note that we only needed to test with  $T_1 T_2$  and not  $T_2 T_1$ , because, it has to be conflict equivalent to one of them that is it.

So, since it is already passed in the first test it is fine, it need not be checked with  $T_2 T_1$ . If this were not equivalent to  $T_1 T_2$ , then it needed to be checked with  $T_2 T_1$ , only when  $T_1 T_2$  did not pass that is the thing.

(Refer Slide Time: 09:32)



Now, let us take another example, which is the following, S is simply  $r_1(a)$ ,  $w_2(a)$  and  $w_1(a)$ . This is a very short schedule and let us see, whether this is equivalent. So, what is  $T_1$   $T_2$  is  $r_1(a)$ ,  $w_1(a)$ ,  $w_2(a)$ . Now, let us test whether S is equivalent  $T_1 T_2$ , this is fine, but these two has been swapped. So, these two these as been swapped. Is this a non-conflicting swap, can we do that, let us test it, they operate on the same data item and one of them is a write.

So, that means, these two conflict, these two instructions conflict, so this swap is not a good swap so; that means, S is not equivalent to  $T_1 T_2$ . But, we still cannot conclude anything about S; we need to test it with  $T_2 T_1$  as well. So, what is  $T_2 T_1$ ,  $T_2 T_1$  is simply  $w_2(a)$ , then  $r_1(a)$  and then  $w_1(a)$ , let us see, whether this is equivalent to S.

So, once more  $w_1(a)$  is a last operation, so nothing to be worried about, but  $r_1(a)$  and  $w_2(a)$  has been swap. So, let us see if that swap can be happening, now  $r_1(a)$  and  $w_2(a)$  are on the same data item and one of them is a write. So, this also conflicts. So, this is not a non-conflicting swap. So, this is also not equivalent. So, S is neither equivalent  $T_1 T_2$  nor it is equivalent  $T_2 T_1$ ; that means, it is not conflict serializable.

So; that means, S will not preserve the database consistency in some manner. So, in the sense of conflict serializability, so S is not conflict serializable that is the concept of conflict serializability.

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

**Lecture - 35**  
**Schedules: View Serializability**

We will continue with the notion of serializability, so after conflict serializability we will start on the View Serializability.

(Refer Slide Time: 00:19)

**View equivalence**  
 reads get the same "view"

$T_1, T_2$        $s, s'$

1. $x$	$r_1(x)$	$r_2(x)$	initial read
2. $x$	$w_1(x)$	$w_2(x)$	final write
3. $x$	$T_1 \leftarrow T_2$	$s$	
	$T_1 \leftarrow T_2$	$s'$	

**View serializable**  
 view equivalent to some serial schedule

So, this is view serializability, so just like conflict serializability require the notion of conflict equivalence. So, we will require the notion of view equivalence, so two schedules whether they are view equivalent to each other or not that is the notion of view equivalence. So, essentially very roughly the notion of view equivalence is that if they reads and then get the same view that is they read that same thing that is produced by the writes, so reads get the same view.

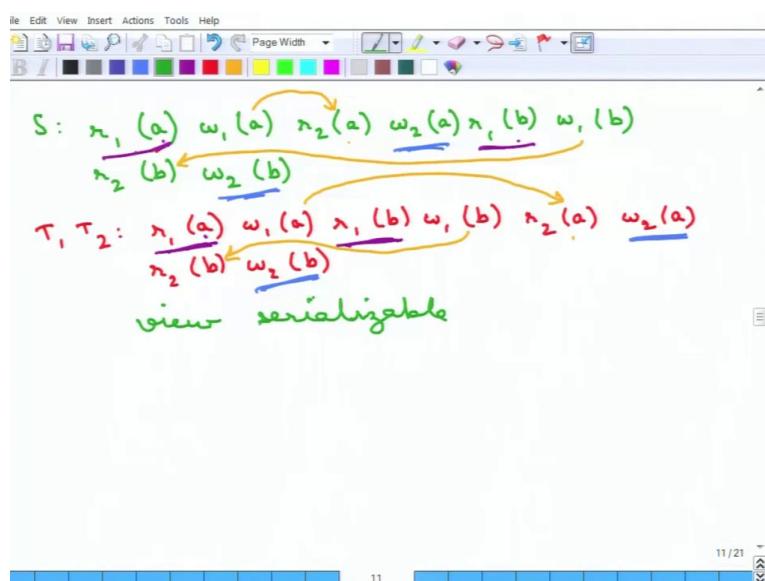
So, we will define the little bit more formally, where this view means essentially is the following. So, two transactions  $T_1$  and  $T_2$  are view equivalent if the following thing happens. So, for each data item  $x$  if there is a data item  $x$  transaction  $T_1$  reads  $x$  and transaction  $T_2$  reads the  $x$ , they read the same value of  $x$ . So, they read the same initial value, so this is the defining the initial value. So, the same initial value of  $x$  is being read by both the transactions that is number 1.

Number 2, if there is the data item  $x$  and the final writes the written values that  $x$  is written to by transaction 1 and the final write by transaction 2, the final writes are also the same. So, the initial reads are same and the final writes are same and the third one is essentially what we have been studying for the conflict serializability etcetera is that for the data item  $x$ , if  $T_1$  reads the value that is produced by  $T_2$ , then that should not be changed in the two schedules.

So, if there are two schedules that involve this  $T_1$  and  $T_2$  then there may not be changed. So, that is the notion of view serializability just to go once more, so there are two schedules  $S$  and  $S$  dashed that involve these two transactions is just for our sake of understanding, in generally it can include many number of transactions. But, if both of them starts from the same transaction that reads the value of  $x$  then of course, they get the same initial value that is read, then that is fine.

Otherwise, even if the two transactions read the two different things, then they must have the same initial read and the same final write, these are the two schedules should have the same final write and then the transaction ((Refer Time: 03:00)) for every  $x$  if  $T_1$  reads the values that is produced by  $T_2$  in transaction  $S$  the same must happen in transaction  $S$  dashed as well. This is view equivalent and the schedule is said to be view equivalent if the following three conditions happened and it is called view serializable, if it is view equivalent to some serial schedule. So, that is the thing if it is view equivalent to some serial schedule that is the same like the conflict equivalent definitions. So, to some serial schedule that is the whole point.

(Refer Slide Time: 03:45)



So, coming to the example, so suppose this is the example  $r_1(a)$  then  $w_1(a)$  then  $r_2(a)$ ,  $w_2(a)$ ,  $r_1(b)$ ,  $w_1(b)$  then  $r_2(b)$  and  $w_2(b)$ . So, the question we need to test whether this is view serializable or not. So, let us test it with  $T_1$   $T_2$  just like what we have been doing earlier. So, what is  $T_1$   $T_2$  we will write now all the operations of  $T_1$  first which is  $r_1(a)$ ,  $w_1(a)$  then  $r_1(b)$ ,  $w_1(b)$  and then that of  $T_2$ , so then this is  $r_2(a)$ ,  $w_2(a)$ ,  $r_2(b)$  and  $w_2(b)$ . So, now, we need to see if they follow those three conditions.

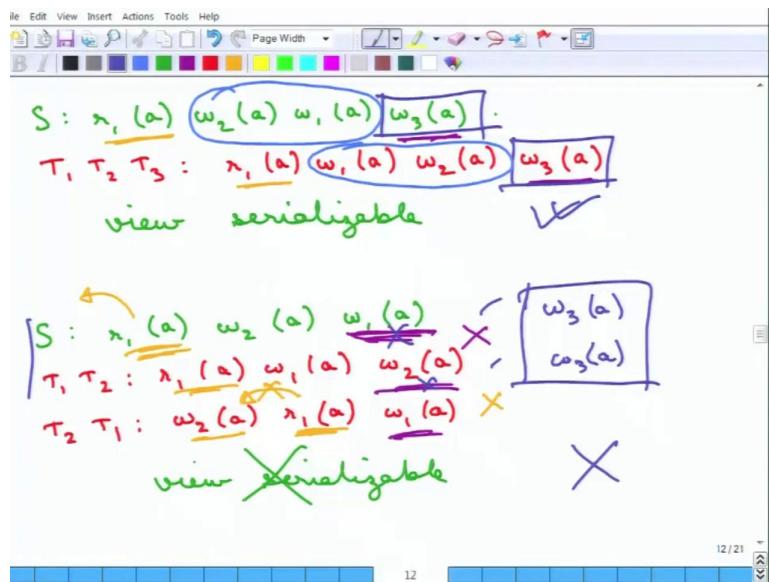
So, first of all they read the same value of a, so for a this is  $r_1$  is reading it and  $r_1$  is reading it here, so that is the same. So, if you going that to the definition, if this happens that transaction 1 reads it in one sense and transaction 2 reads then this is not following the view equivalence, neither is these following the view equivalence. So, the same transaction, if the transaction 1 reads the value then it must be the transaction 1 reads it here, if the transaction 1 writes the final value then it must be that the same transaction writes, so that must happen.

So,  $r_1(a)$  and  $r_1(a)$  that follows that is not a problem, now we have to test it for each data item. So that means, this we have only tested it for data item a, we need to also test it for data item b, now data item b is first read by  $r_1$ . So, and then it is also first read by  $r_1$ , so that is also fine. So, the initial read conditions are correct, now let us test for the final write conditions. So, the final write conditions of a is by  $w_2(a)$  in this S and in this serial schedule  $T_1$   $T_2$  it is again by  $w_2(a)$ . So, that is correct same for b it is  $w_2(b)$  and  $w_2(b)$ , so that is also correct.

Now, the third condition is essentially saying that every read that is produce by another write must be consistence, so must be the same. So, what are the read's that have produced etcetera, so let us see. So, transaction 2 reads the value a that is produced by transaction 1's write, so that condition must be present here. So, if  $r_2$  is read that must be produced by transaction 1. So, if  $r_2$  reads then the same a that must be produced, so that is also valid here.

And similarly  $r_2(b)$  is following, what is return by  $w_1(b)$  here and same as s, so  $r_2(b)$  will follow what is written by  $w_1(b)$ . So, then this is view equivalent to this transaction  $s_1$ ,  $s_2$  so; that means, this is view serializable, because this is view equivalent to this transactions  $T_1$   $T_2$ . So, that is the first example.

(Refer Slide Time: 07:07)



Now, let us take another example which is the following, this is  $r_1(a)$  then  $w_2(a)$  then there is  $w_1(a)$  then  $w_3(a)$ . So, if you remember this was not conflict serializable, but this previous example that we did was also conflict serializable. Now, let us test it with  $T_1 T_2 T_3$ . So, what is  $T_1 T_2 T_3$ ? This is  $r_1(a)$  then  $w_1(a)$  then  $w_2(a)$  then  $w_3(a)$ . So, the initial read condition is by transaction 1 here and transaction 1 here, so that is fine.

The final writes are by  $w_3(a)$  here and  $w_3(a)$  here, so that that is also correct and then there are no more reads by either the schedule S or by the serial schedule  $T_1 T_2 T_3$ . So, the third condition is also honoured trivially, so this is actually view serializable, this is very, very important to notice that. So, this is view serializable although if you remember the example of conflict serializability this was not conflict serializable, because we swapped these two, so this conflict we swap.

But, here it does not matter, so why does it not matter, so here is something that why view serializability is useful. So, why it is sometimes more useful than conflict serializable is that look at the middle two conditions. So,  $w_2(a)$  and then  $w_1(a)$  and versus  $w_1(a)$  and then  $w_2(a)$  although none of the writes actually matter. Why does it not matter? Because, both of them is actually superseded by a third write by transaction 3  $w_3(a)$ .

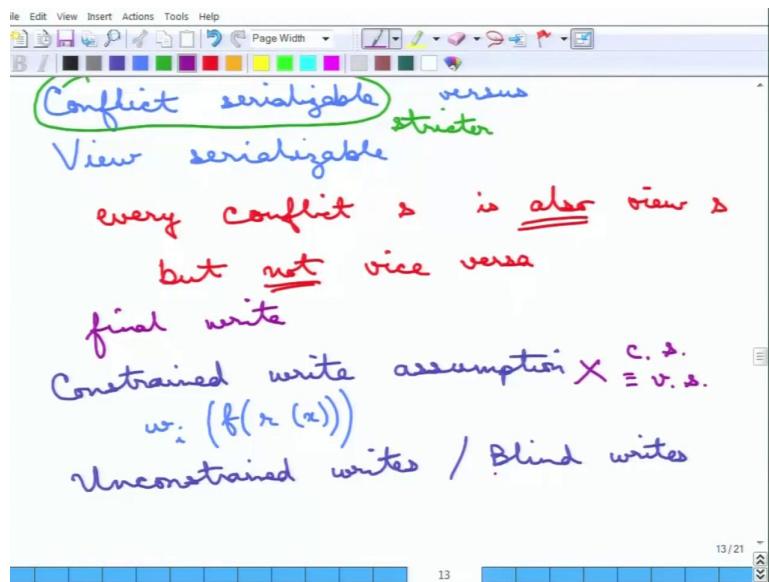
So, the condition for view equivalence essentially says that the final write is what one should be bothered about and the final writes are both produced by transaction 3. So, they are all correct, so there is nothing problem here, now just we will come to this point in a little while again, but before that I just want to show you another example just to highlight what is happening on here. So, suppose this is another schedule which is simply this, now let us test whether this is view serializable or not, this is  $r_1(a), w_1(a), w_2(a)$ .

Now, first of all the read conditions are all correct the writes; however, are not correct, this is  $w_1(a)$  and this is  $w_2(a)$ . So, this is not correct, so this S is not view equivalent to  $T_1 T_2$ .

Now, let us test it with  $T_2 T_1, T_2 T_1$  is your  $w_2(a)$  then  $r_1(a), w_1(a)$ , now here although the final writes are the same as it happens that the initial read is also the same. But, what it happens is that this is not view serializable, because this read has been produced by some other write; however, this read is being produced by this  $w_2(a)$ . So, this is not allowed, so over all this is not view serializable.

Now, so that critical difference between these two examples is that following is this  $w_3(a)$  here because of this  $w_3(a)$  between these two things, there is a final write here, there was a final write which essentially allowed to super seed both these writes. So, this final write condition was by some other transaction which remained the same for both the schedules. So, that is why this was view serializable while this was not, so that is the very important part of what to understand what view serializability is.

(Refer Slide Time: 11:03)



And now to understand it in more detail ((Refer Time: 11:04)) we will see conflict serializability. So, what we are going to study is next is conflict serializability, let us say conflict serializable schedule verses view serializable schedule. So, as we solve the two examples, the first example was that it was conflict serializable and view serializable both, the second was not conflict serializable, but was view serializable and the third one was neither conflict serializable nor view serializable.

So, this was not by accident as it happens that we can we have this following rule is that every conflict serializable schedule is also view serializable I am just abbreviating serializable by s. Now, this is true, so if something is conflict serializable it must be view serializable and you can actually prove it formally, but if you think for a little bit and look at the intuition of it, you will see that this is correct. However, of course, we saw an example that the vice versa is not true.

So; that means, that not every view serializable schedule is conflict serializable and we already saw an example. So, what it actually means is that conflict serializability is a stricter condition than view serializability, so conflict serializable is actually more strict, so this is stricter than view serializability. Now, we can see what is the usefulness of view serializability is that there are certain schedules which are not conflict serializable.

Because, conflict serializability is stricter condition, but they are view serializable, so if one enforces the conflict serializability those schedules are not going to be allowed by the

database engine. However, if one enforces only the view serializability they can be allowed by the database engine and in the end it does not matter, because the final writes are the same. So, initial reads etcetera are of course, the reads as produce by writes are also the same, but very, very importantly the final writes are the same.

So, essentially it all boils down to this final write, so the final write is the, big important condition that everything boils down to. Now, there is an interesting concept called the constrained write assumption, which is related to this constrained write assumption. So, the constrained write assumption, essentially says that every write is constrained by the value that it has read. So, write if there is a write by a transaction  $i$ , it must depend on what the value has been read.

So, every transaction  $s$ , so they it must be a function of whatever it has read so; that means, that the write before a transaction tries to write it must read it. So, that because the write is the function of the read. So, the other way round is that, so if this is not followed then we get what we call unconstrained writes. So; that means, a transaction simply writes without reading, unconstrained writes are also sometimes called blind writes.

Why it is called a blind write? Because, it is blindly writing it does not care about what it has read etcetera, it does not even need to read, so it just writes. So, this unconstrained writes are blind writes. So, if constrained write assumptions are not allowed then... So, let me just state it if these are not allowed then conflict serializability is equivalent to view serializability, so this is sometime needed to be understood.

So, the reason for this is that constrain write assumption look at the example essentially the difference between the view serializability and conflict serializability is that view serializability can allow certain write to write mismatches as we saw in the example between  $w_1(a)$  and  $w_2(a)$ . Because, there is a final write which does not depend on anyone of them and just simply writes, so that is the blind write. Now, if one removes the blind writes one can show that actually it can been shown that the conflict serializability is equivalent to view serializability.

Now, to turn it around; that means, that suppose there is a schedule which is view serializable, but not conflict serializable. So, then it must happen that, that schedule has blind writes. Because, so the only difference between view serializable and conflict serializable to

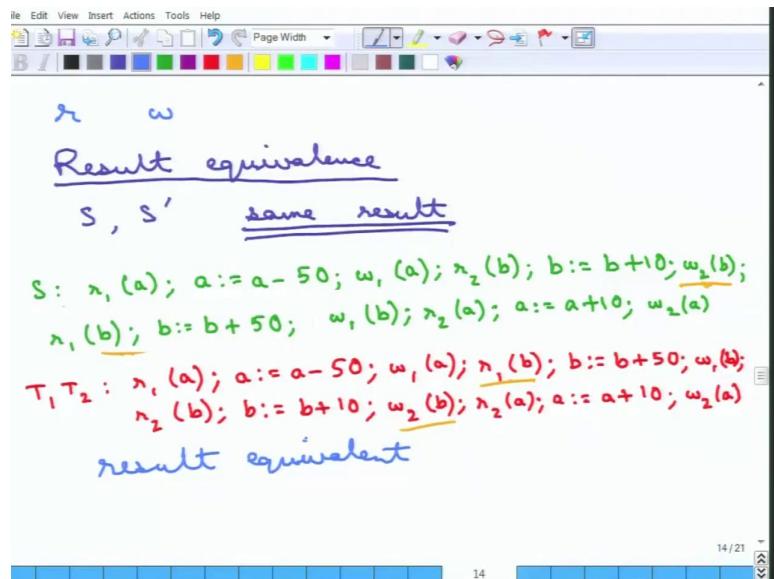
allow the schedules as view serializable and not conflict serializable is those blind writes. So, view serializable schedules, which are not conflict serializable must have blind writes or these unconstrained writes, so that is the thing to note.

So, this unconstrained writes and blind writes is very important, so if there is... So, under this constrains write assumption, if there is only constrain writes if no blind writes are allowed then conflict serializability is equal to view serializability. So, that is about the two serializable notions of conflict and view serializability.

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

**Lecture - 36**  
**Schedules: Result Equivalence and Testing for Serializability**

(Refer Slide Time: 00:15)



Now, one interesting thing or one important thing that you must have noticed is that these serializability notions are just concerned about the write and read, but not the values that is being read or write. I mean of course, there is a constant because they must be reading the same value etcetera, but they do not analyse the result part say. So, now, to extend that there is another concept which is called result equivalence, so just like there is conflict equivalence and view equivalence there is something called a result equivalence.

Now, two schedules are said to be result equivalent to each other,  $S$  and  $S'$  are result equivalent to each other. If they produce the same result, if they somehow produce the same result, then they are said to be result equivalence of course, then it means that they are correct from the databases point of view if all the transactions in those schedules are successful. Now, here is an example to highlight the point, is that suppose there is this schedule  $S$ , which says read 1 of  $a$ , then it actually does this following, now we need to worry about the result.

So, we need to know the values of  $a$  and whatever things that we are writing. So, then it writes, then it reads  $b$ , then it does  $b = b + 10$ , and then writes to  $b$  fine. And the other one

is the following it reads  $b$ , then it does  $b = b + 50$ , then it is  $w_1(b) \quad r_2(a)$ , then  $a = a + 10$ . And then, finally, so just to note this is all just one schedule, so this is one schedule  $S$  and this is another schedule  $T_1 \quad T_2$ , which is the serial schedule, which if you write down these operations in the following manner, then it produces this thing.

So, we will write down all the operations of a first, so first all the operations of transaction one and then those of transaction two. Now, look at these two transactions and let us see whether they are conflict equivalent or not. Clearly, they are not conflict equivalent, because  $r_2$  and  $w_1(b)$  etcetera must happen in the correct order, so they are not conflict equivalent just to highlight, because this  $w_2(b)$  and  $r_1(b)$  must be in this order.

So,  $w_2(b)$  and  $r_1(b)$  conflict and here it is swapped the other way round. So, this is not conflict equivalent, this is also not view equivalent, because this  $r_1(b)$  is reading the value that is produced by  $w_2(a)$ , while as this  $r_1(b)$  is reading the value something else. So, these are neither conflict equivalent nor view equivalent; however, very, very importantly these are result equivalent and that can be seen just by understanding the semantics of these two schedules.

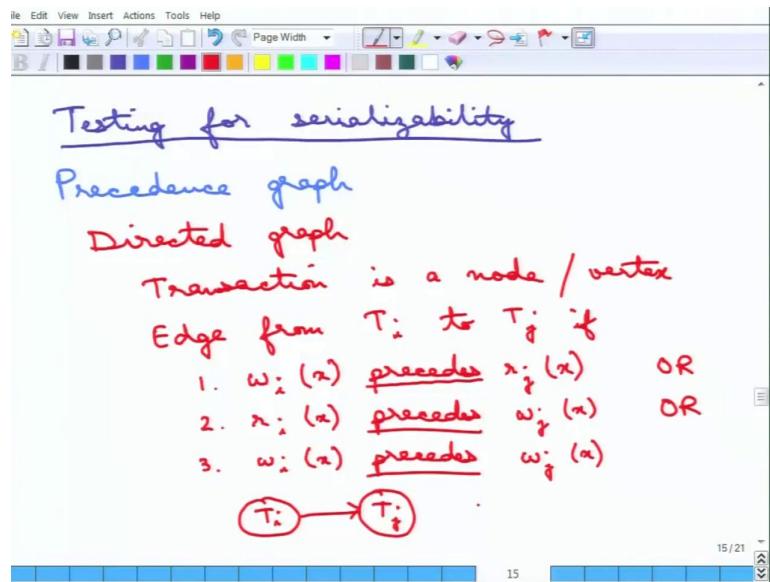
So, essentially the point is this that  $S$  should be allowed by the database, because it finally, produces the same result as the serial schedule  $T_1 \quad T_2$ , but that is very very hard for the database engine to figure it out and unless it does some semantic analysis. So, semantics is the meaning of the transactions, it is not just blindly whether it is read or write and who reads from, which other transaction reads from which other values etcetera.

But, actually analysing the value, that if you do a minus 50 first and then a plus 10 that is the same as doing a plus 10 first, and then a minus 50 or if you do  $b + 50$  first and  $b$  minus this thing, then it does not matter. So, then that records as semantic analysis and which is of course, not done by the database, because it can be very, very hard as you see. So, although you can see that theoretically there is a notion of result equivalence, database in use do not do that.

So, it is essentially just a theoretical notion of equivalence and the databases only go, only try to go up to conflict serializability and view serializability. Now having said all of these

things, let us see that how does the database find out or whether, what is the algorithm to find out whether a schedule is serializable or not. I mean, we have been doing with paper and pen etcetera.

(Refer Slide Time: 05:17)



So, essentially what we are trying to say is testing for serializability. So, whether there can be an automated algorithm for doing this or what can be done about it, testing for serializability requires the following notion. So, what is done is first of all this is something called a precedence graph is created, so there is notion of precedence graph. Now, I am assuming all of you are familiar with the graph data structure, essentially graph has certain nodes and there are edges that connect the nodes, which mean certain things. Now, we will see what the thing is that, so this is the directed graph.

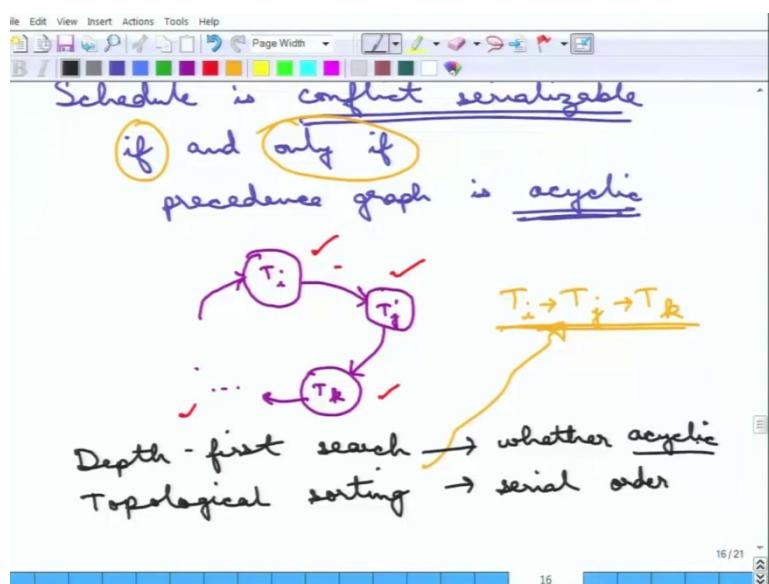
So, here in this thing, so this is a directed graph and each transaction, so every transaction in the schedule is a node, transaction is a node or vertex whatever way you want to say it is the same thing. And then, there is an edge from transaction  $T_i$  to transaction  $T_j$  note that edges are directed, because it is a directed graph. If the, this thing happens if any of that following three condition happen is that the  $w_i(x)$  precedes  $r_j(x)$ .

So, what are we trying to say here is that this is this important condition precedes. So, precedes meaning  $w_i(x)$  must be happening before  $r_j(x)$  this or  $r_i(x)$  precedes  $w_j(x)$  or, now you can guess the third condition. It is  $w_i(x)$  precedes  $w_j(x)$ , essentially it just encodes

the three conflicts the read after write, write after read and write after write. So, essentially, so if there are two, so between transaction  $T_i$  and  $T_j$  if there are two instructions that can conflict, so essentially write read, read write and write write.

And if transaction if the operation of transaction  $T_i$  happens before that of  $T_j$ , then transaction  $T_i$  has a directed edge to transaction  $T_j$  that is how you create the precedence graph. And now it should be clear why it is called a precedence graph. Now, given this precedence graph, so first of all given a schedule one can create the precedence graph.

(Refer Slide Time: 08:17)



Now, given this precedence graph a schedule is conflict serializable, so it is a particular schedule is conflict serializable if and only if, so this is if and only if this is important it is not just if, so this is both necessary and sufficient condition. If and only if the precedence graph is acyclic, so which has got no cycles, so this is conflict serializable if and only if the precedence graph is acyclic. So; that means, if there is a cycle in the precedence graph, so essentially let us trying to understand, what does the cycle means.

So, there is  $T_i$  that has an edge to  $T_j$ , then that has an edge to some other thing let us take  $T_k$  etcetera and that has an edge to something else dot, dot, dot finally, there is an edge back to  $T_i$ . Now, what does this edge  $T_i$  to  $T_j$  mean  $T_i$  to  $T_j$  essentially mean is that  $T_i$  and  $T_j$  conflict and the order of  $T_i$  the instruction of  $T_i$  must happen before that of  $T_j$ . Then, the

next edge essentially means that the instruction of  $T_j$  some instruction of  $T_j$  must happen before  $T_k$ .

And similarly it happens that the instructions of  $T_k$  must happen before  $T_i$ ; that means, that you can think of any serial order, let us a  $T_i T_j T_k$ . Now, that is going to violate the last condition the  $T_k$  is not happening before  $T_i$ . Now, let us choose the other way round suppose its  $T_k T_i T_j$ , then  $T_k$  to  $T_i$  is fine  $T_i$  to  $T_j$  is fine, but then  $T_j$  should be happening before  $T_k$ .

So, essentially if there is a cycle, then this essentially saying that  $T_i$  must be before  $T_j$  it must be before  $T_k$  which must be before something else must be before  $T_i$ , which can never happen which kept; that means, there is no serial schedule possible, which means that if there is a cycle it is not conflict serializable.

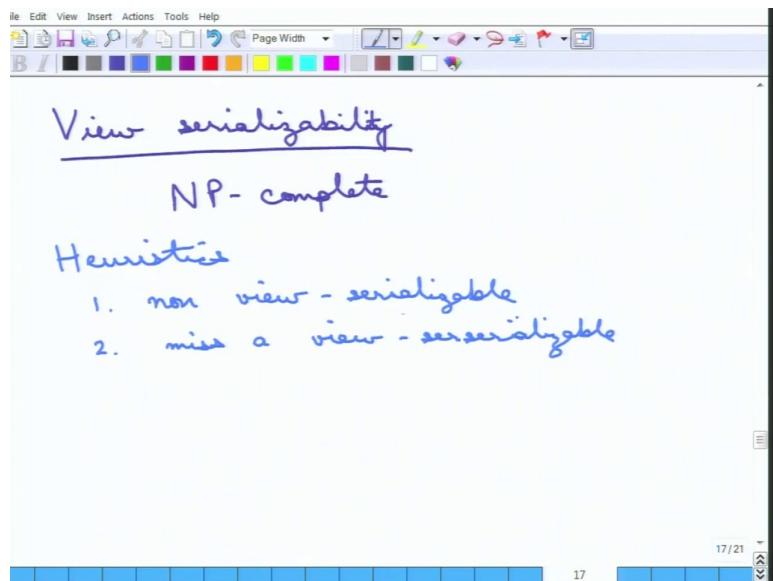
And the other way round can also shown is that if it is conflict serializable, then if a schedule is conflict serializable; that means, that it is equivalent to some order  $T_i T_j T_k$  etcetera it is conflict serializable it is equal to some serial schedule; that means, the only edges that can be present is from  $T_i$  to  $T_j$  and  $T_j$  to  $T_k$  they cannot be any back edge from  $T_k T_i$ ; otherwise it not equivalent; that means, there cannot be any cycle.

So, that is why this both if and only if, so this both necessary and sufficient condition. Now, the question then become is the given a schedule one can produce its precedence graph and if the graph contain cycles, then it is not conflict serializable otherwise it is. So, the essentially the question now boils down to how do you find if a directed graph the precedence graph contains cycles or not.

Now, for that there is an algorithm, which is called a depth first search, so if you start the depth first search from a vertex if there is a cycle it can find it out. So, assuming there is no cycle if one runs another algorithm, which is called a topological sorting the topological sorting will essentially give a serial order, so topological sorting will actually produce a serial order of the transaction. So, that means, topological order may actually produce something like  $T_i T_j T_k$  so; that means, this is equivalent to  $T_i T_j T_k$  and the depth first search can find out whether there are cycles or not, so whether the graph is acyclic or not.

So, these are the two algorithms and if you are familiar with the graph algorithms, then this there otherwise you have to just know these things.

(Refer Slide Time: 12:07)



So, this is about conflict serializability, now testing for view serializability is a much, much harder problem and actually there is a complexity class of problem called the N P complete serializability. So, testing for view serializability is actually an N P complete problem, N P complete problems are essentially hard problems. So, they are supposed to be hard it is not clear whether they hard or not nobody else found a good efficient algorithm for them yet, so N P complete problems are typically sort of as hard problem.

So, testing for view serializability is a hard problem and the database is really do not, because it can take a long, long time to the actually solve this N P complete problem. So, the database engines generally do not try to test for view serializability in the exact notions. So, what it will try to do is again it resorts to the concept of heuristics, so what is does is that this is some practical heuristics it tries to catch all non view serializable schedules.

So, it can quickly catch if a schedule is non view serializable, but then of course; that means, that it can miss a actually view serializable thing. So, it catches all non view serializable thing, but can miss a view serializable. So, what is the effect of what I am trying to saying is that given are schedule if it is non view serializable there is a practical algorithm some heuristic will actually say it is non view serializable.

But, it may also happen that it declares of view serializable schedule to be non-view serializable as well that is the mistake that it does. Because, the correct procedure is N P complete and that will take a long, long time to finish, so the practical way of doing it is that doing a mistake on only one side. So, it may means a view serializable schedule, but if a schedule is non view serializable it will surely catch it.

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

**Lecture - 37**  
**Schedules: Recoverability**

So, we will next move on to another important property of schedules, which is the recoverability. Now, what we have been studying in the serializability or in the conflict serializability and the view serializability is that, whether they are equivalent to each other. Assuming that all the transactions in the schedule are correct; that means, all the operations are successfully done, processed.

Now, what happens in most transaction, it may happen that the order, the transaction may abort etcetera. So, whether the transaction can recover or not; that is the study of recoverability. So, essentially in the serializability study, we did not take into account the commits and aborts and their order, more importantly the order of when a transaction commits and when a transaction aborts etcetera. That is what we are going to next study, so this is on recoverability of schedules.

(Refer Slide Time: 01:00)

Recoverability

Recoverable schedule

1.  $T_i$  reads  $x$  previously written by  $T_j$

$\Rightarrow$  2.  $T_j$  commits before  $T_i$  commits

$S: \pi_1(a) \xrightarrow{x} \pi_2(a) \xrightarrow{c} \pi_1(b) \xrightarrow{a_1} \times$

$S: \pi_1(a) \xrightarrow{x} \pi_2(a) \xrightarrow{c} \pi_1(b) \xrightarrow{a_1} \pi_2(a_2) \checkmark$

So, recoverability of schedules, so here the essential idea is that the order of commits and aborts in the transactions are important. Now, the first thing is, we will define which is something called a recoverable schedule. So, schedule is said to be recoverable, if certain

conditions happen. So, recoverable schedule, essentially the definition is the following. Suppose, in the schedule the transaction  $T_i$  reads a data item  $x$ ; that has been previously written by some other transaction  $T_j$  as part of the same schedule of course.

Now, it must happen that if this holds, then if one holds, then the second condition must be holding is that,  $T_j$  must commit before  $T_i$  commits. So, if this happens then the second condition is that  $T_j$  commits before  $T_i$  commits. Now, why is this, intuitively we can see, what is the reason for this is that,  $T_i$  has read something that has been produced by  $T_j$ .

Now, unless  $T_j$  really commits meaning it vouches that what it has written is correct,  $T_i$ , the reading of  $T_i$  may not be correct. So, in other words, suppose it happens that  $T_j$  now aborts. Now,  $T_j$  aborts meaning, whatever written by  $T_j$  needs to be rollback and needs to be undone, which means that whatever  $T_i$  read, which is the value written by  $T_j$  is also incorrect, because it should not have read this, because that has been undone.

So,  $T_i$  if it had already committed, then it is in a problem, because now it has committed with a wrong value read. So, that is why, this is not allowed and that is why a schedule is called recoverable, if this, if  $T_j$  commits before  $T_i$  commits, if the first condition happens. Now, let us directly jump to an example and let us see what we are trying to say. So, suppose our schedule is the following is  $r_1(a)$ , then there is a  $w_1(a)$ , then there is a  $r_2(a)$ .

So, the transaction 2 then reads and then, suppose transaction 1 read some other data item. Now, essentially what is happening is that, you see that transaction 2 is reading the  $a$  which is written by transaction 1. Now, in that case, the rule says that,  $T_j$  must commit before  $T_i$  commits. Now, suppose what happens is that, let me introduce this thing is that suppose after this, there is a commit by  $c_2$  here.

Now, the question is whether this schedule is recoverable or not, so this is not recoverable, I am assuming that  $c_2$  commits here and then,  $c_1$  commits here. This is not recoverable, because  $c_2$  should not be committing before  $c_1$  does. The reason is suppose  $c_1$ , it is not a commit here, but it is an abort by  $c_1$ . Then, what happens is that these values; that is produced is wrong; that means this read that what it has done from  $w_1$  is also wrong;

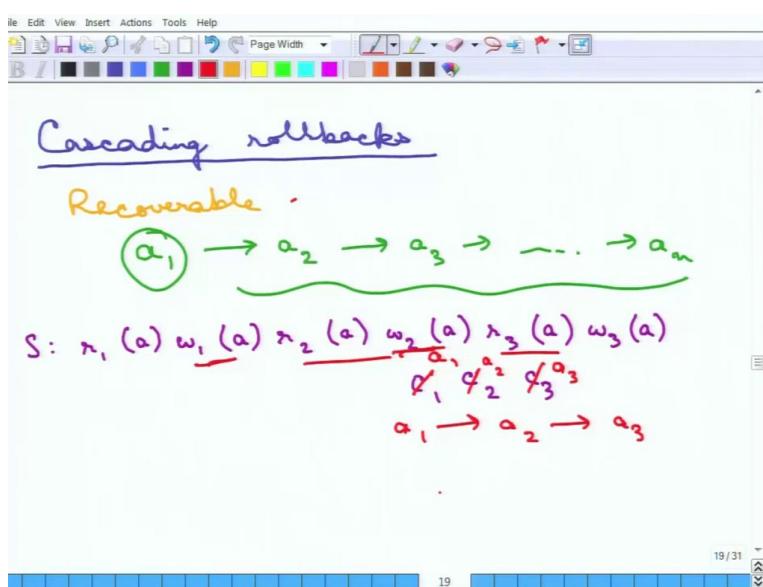
however, it has already committed.

So, by the time the system reaches here and finds that, this is abort, this has already committed. Committed meaning, if you remember the property of durability, when a transaction commit, it essentially says that, if I am committed, then everything that I am going to do is final and correct now, but that is a problem. So, this is not recoverable, so this schedule is not recoverable.

Now, to make it recoverable, what should have been done is that if this is the case, if this is the schedule, then  $r_2(a)$ , then  $r_1(b)$  etcetera, what it should be done is that  $c_1$ , so  $c_1$  transaction 1 should first commit and then transaction 2 commits. So, this order of commits is very, very important. So, that is the reason why we studied recoverability this order of commits and abort is what we are very much bothered about, because that is way a schedule may be recoverable or may not be recoverable.

So, now the reason this works is that, suppose just like the previous case, somehow this commit is not true, but this actually aborts. Now if this aborts; that means this value produced is wrong; that means, this  $r_2$  should also be wrong, which is fine, because then the transaction 2 can then simply abort. Instead of committing, because it is read wrong value it can also just abort. So, then there is no problem as for as the recoverability of the schedule is concerned. So, that is the recoverable all that seems to be fine.

(Refer Slide Time: 05:48)



But, then it runs in to one problem, which is called the problem of cascading rollbacks, by the way, this rollback is essentially what the undoing operation etcetera is. So, when an abort happens, the transaction essentially rolls back; that means that, it undo all the values and writes the old values of etcetera, so that is the roll. Now, the problem is, even if a transaction is recoverable, there may be a series of rollback as we saw in the other the previous example, if there is an abort by transaction 1, then this leads to an another abort by transaction 2.

So, in a general sense, this may lead to an another abort by transaction 3 and all these, things there may be many, many, many rollbacks, just because one transaction aborted or one transaction rollback. So, this is the series of cascading rollbacks. So, that problem may be there, even if a transaction is recoverable. So, now, just to highlight the point a little bit better and just given an example, suppose this is the schedule is  $r_1(a)$ , then there is a  $w_1(a)$ , then there is an  $r_2(a)$ , then there is a  $w_2(a)$ , then there is an  $r_3(a)$ , then there is a  $w_3(a)$ , etcetera.

Now, as we said that the, because to make it recoverable, it should happen that  $r_1$  should first commit, then  $r_2$ , then  $r_3$ , but suppose  $r_1$  somehow decides to abort, there is some problem. Now, if since  $R_1$  aborts,  $r_2$  must also abort, that means this should abort, because this is aborted and  $r_3$  must also abort, so that is the thing. So, that means,  $a_1$  leads to aborting of  $a_2$  and then leads to aborting of  $a_3$ . So, by the way  $a_1$  stands for abortion and c it for commit.

So, commit and abort are c and a, so that is the series of rollback. So, that is the problem with recoverable schedules is that, even though it is fine in the sense of correctness, but it may lead to a lot of work being undone. So, there may be a lot of time to recover, because lots of aborts are done and lots of time it is takes for the schedule for the database to recover.

(Refer Slide Time: 08:02)

So, the next concept that comes in the space is the concept of cascadeless schedule. So, to avoid this problem of cascading rollbacks, we defined what is next called a cascadeless schedule. So, essentially cascading rollbacks are eliminated, so there are no cascading rollbacks. If a schedule is cascadeless; that means, that there should not be any cascading rollbacks and a more formal definition is the following is that, if transaction  $T_i$  reads  $x$ , which is written previously by transaction  $T_j$ .

Then, it must happen that transaction  $T_j$  commits before  $T_i$  even reads, so note the definition, note the difference of the definition from the recoverable schedule, in the recoverable schedule, the first condition was the same. So, if  $T_i$  read something, which is written by  $T_j$ ,  $T_j$  must commit before  $T_i$  commits that was in the recoverable schedule.

In this new cascade less schedule,  $T_j$  must commit even before  $T_i$  has read. Now, the intuition should be very clear, why this is being made. So, intuition is the following is that, if  $T_j$  commits before the read is being done. So, if  $T_j$  is successful; that means, the read is correct, there is no problem with the reading. If; however, that  $T_j$  is wrong and it has aborted, it can be argued that the  $T_i$  has not wasted any operation,  $T_i$  has not even read from  $T_j$ .

So,  $T_i$  by itself does not need to abort, because it is not even done anything, which is

dependent on  $T_j$ . So, that is why it avoids the cascading rollback of  $T_i$ , if  $T_j$  rolls back and that is the definition of cascadeless schedule. So, here is the example, once more the same example we will follow as we did. So, it is  $r_1(a)$ ,  $w_1(a)$ , then there is an  $r_2(a)$  and  $r_1(b)$ .

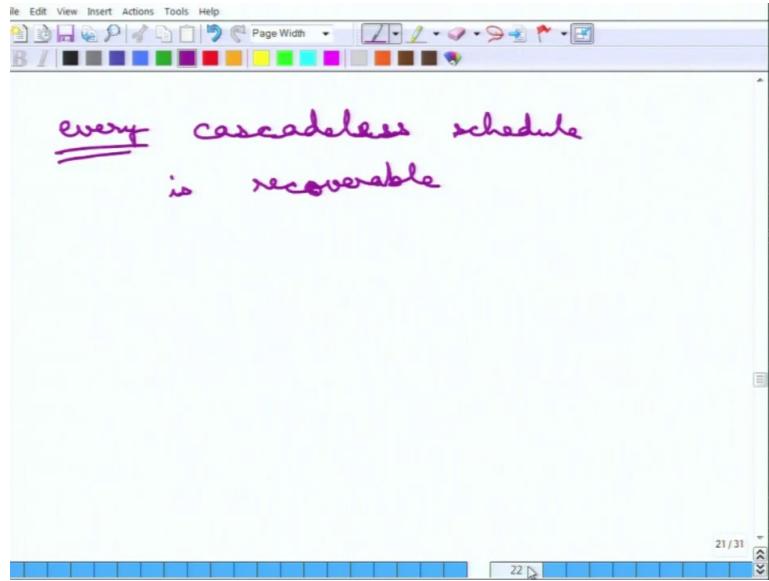
Now, the original problem was if this abort happens, then this commit should also be changed to abort, this is why this was recoverable, but not cascadeless. Now, to make it cascadeless, what needs to be done is the following is that  $r_1(a)$ ,  $w_1(a)$ , then it cannot be that  $r_2(a)$  does before  $a_1$  does. So, transaction 1 must commit, but for transaction 1 to commit, it must complete this operation.

So,  $r_1(b)$ , then transaction 1 commits and then, transaction 2 starts doing whatever it wants to do, because this read depends on this write, so this commit must happen before this read happens. So, now, you see, what is the good thing about this is that, even if this aborts, then of course, transaction 2 has not done anything at all. So, it will just simply not start, or it does not need to abort, the question of aborting or commit does not arise for  $t_2$ .

Because, it has not started doing anything at all, so this is not the first example, this is not cascadeless, but recoverable, but this cascadeless and recoverable. So, essentially the idea is that, no completed transaction needs to rollback, because  $t_2$  has not completed. So, it does not need to rollback, so no completed transaction rolls back; that is the effect of this cascadeless schedules.

So, that is the definition and the property of cascadeless schedules, now one thing is that there is a connection between cascadeless recoverable just like view serializability and conflict serializability. There is again a connection between cascadeless and recoverable. As we saw in the example that the first example was recoverable, but not cascadeless and the second example was cascadeless and if you have noticed, it is also recoverable.

(Refer Slide Time: 12:02)



Now, this is not an accident, so we can put this down as actually in the following manner is that every cascadeless schedule is necessarily recoverable. Now, this is again, this can be shown in two ways from the intuition we can see plus also it can be actually proved by easily from the definition. Now, the definition of the cascadeless says is that  $T_i$  does not read before  $T_j$  commits; that means, that  $T_i$  cannot commit of course, because  $T_i$  has a reading operation, the commit is the last operation of a transaction.

So, if  $T_i$  reads after  $T_j$  commits, then it of course, means that  $T_i$  commits after  $T_j$  commits, so that means, every cascadeless schedule is necessarily recoverable, but not the other way round and we saw example, why it is not the other way round. So, that is fine up to this cascadeless schedule, now this all seems to be correct, but there is still a little bit problem that is left, even after cascadeless schedule is that the problem of writes remain.

So, the writes, the problem of writes remains in the sense is that a later transaction. So, all the cascadeless schedule etcetera says that transaction  $T_i$  reads, what has been written by  $T_j$ . Now, which means that nothing is said about the writing of transaction  $T_j$ , suppose the same data item is written by  $T_j$ , it does not read what  $T_i$  writes, but it simply writes the same thing. And then, it tries to commit before the commit of  $T_i$  happens, now that is also of course wrong, because it should be writing afterwards not earlier.

(Refer Slide Time: 13:44)

Strict schedule

1.  $T_j$  reads / writes  $x$  written by  $T_i$   
⇒ 2.  $T_j$  commits before  $T_i$  reads/writes

S:  $r_1(a) \quad w_1(a) \quad w_2(a) \quad r_1(b) \quad c_1 \quad c_2 \quad x :$

S:  $r_1(a) \quad w_1(a) \quad r_1(b) \quad c_1 \quad w_2(a) \quad c_2 \quad \checkmark$

every strict schedule is cascadeless

So, the problem of writes is not handled not talked about at all by recoverable and cascadeless schedule so the next definition is that of strict schedule, where it tries to handle this situation with the writes. So, let us first look at definition of this, so essentially it is kind of a same thing, if  $T_i$  reads or writes, this is the change here  $T_i$  reads or writes  $x$ , which is previously written by  $T_j$ .

Then, it must happen that  $T_j$  commits before any of this read or write operation is done. So, before  $T_i$  reads or writes, so it is not just about reads also about writes. So, it enhances the definition of cascadeless to include the writes, so that is why, it is something more than cascadeless. Now, let us go to the example, suppose  $r_1(a)$ , then  $w_1(a)$ , then there is a  $w_2(a)$ , then  $r_1(b)$ , now essentially what it says is that, this is commit by  $c_1$  and then commit by  $c_2$  fine up to this part is fine.

Now, this is not strict, because this write on the same data item  $x$  is happening before  $c_1$  commits. So, this is not strict, although this is cascadeless, but this is not strict. So, this job should have happen before  $c_1$ , now to make it strict, the following thing needs to be done. So, this  $r_1(a)$ ,  $w_1(a)$ , then of course  $r_1(b)$ , then commit and then, this  $w_2(a)$  and  $c_2$ , so this makes it strict.

Now, the question is the connection between strict and cascadeless, again it can be very easily

seen that every strict schedule is cascadeless. But, of course, not vice versa, the reason is, so this is the example to show that this is cascadeless, but not strict, but otherwise it can be again very easily understood from the definition. So, strict argues when  $T_i$  reads or writes from  $T_j$  and it should the commit must happen before reads or writes.

So, if you just read, of course, the reads or write covers that reads, so that means, that every a strict schedule is of course maintains, what the cascadeless schedule tries to say. So, it is of course, cascadeless but not the other way round, because is an example the first example actually shows that. Now, all that seems to be fine, but the question is why is strict important, I mean why do we bother about this writing of a and before what happens, what is the problem with this schedule.

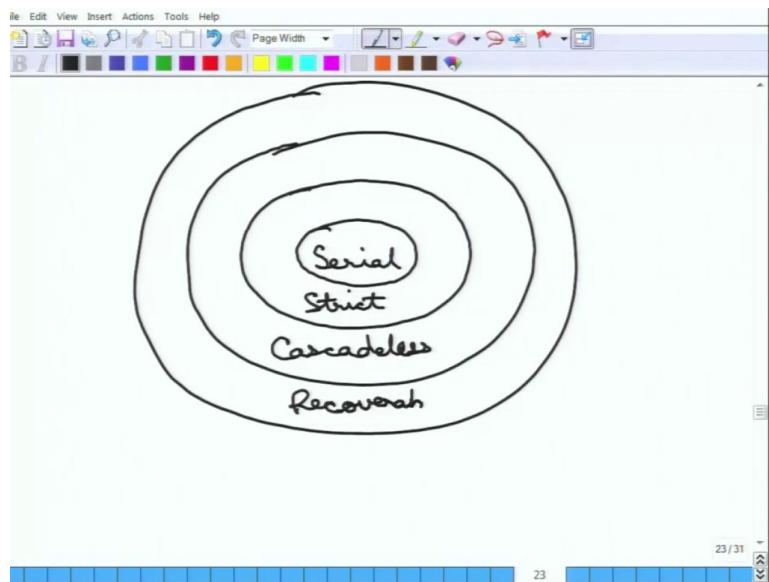
I mean, this we are saying that, this is not strict, what is the problem, the problem is the same as the earlier one is that. So, this the problem is the same as what we have been seen in the view serializability is that  $w_2(a)$  essentially of blind write Now, which means that  $w_2(a)$  maybe may have read some other value and if transaction 1 by itself aborts, it is may not be correct the transaction 2 simply blind writes and then, successfully goes to successfully commits, it should not be happening.

So, again it is a completed transaction that is allowed before it depends on some other transaction. So, essentially they write after write conflict is not taken care of, it take care of the write after write conflict, it must happen that the next write  $w_2(a)$  must wait for the  $w_1(a)$  to finish successfully, which means must wait for transaction 1 to commit. That is the big thing, so that is why, this is strict.

Now, just to complete the story, so by the way just one very interesting thing is to say that, every serial schedule, so what is the serial schedule, just to refresh your memory it is a one transaction happens then completely commits and then, the next transaction starts and then, completely commits, then the third transaction. So, every serial schedule in which this follows is of course, strict.

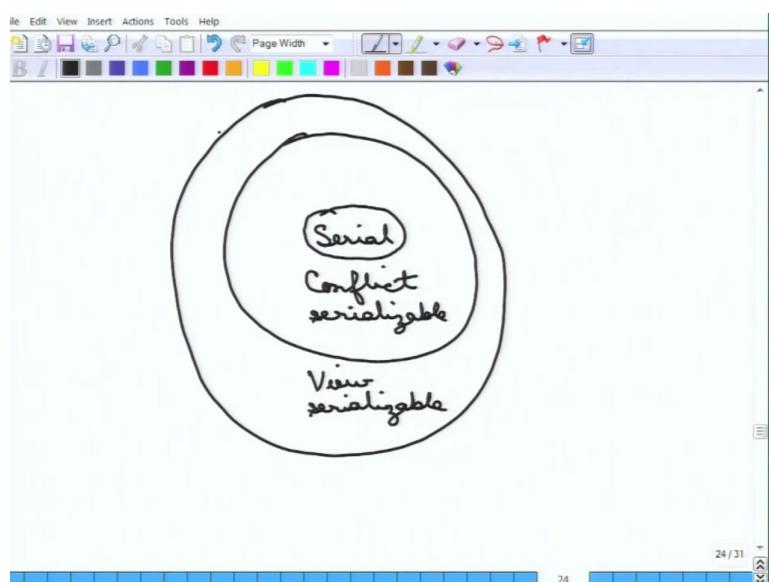
Because, before the transaction 2 does anything, whether it is read writing the same data item etcetera, the transaction 1 has committed. So, it is of course strict, which means it is also cascadeless and recoverable, so here is the nice Venn diagram to think about.

(Refer Slide Time: 18:24)



So, there are these serial transactions, this is serial, now if it is serial, it can be also thought of as, so there are then these things, which are strict. So, serial transactions are strict, but there are other schedule, which also strict, but not serial. Then, there are cascadeless schedules, which I mean every strict is of course, cascadeless, but not the other way round and then finally, there are recoverable schedules. So, that is seems to be a nice Venn diagram.

(Refer Slide Time: 19:09)

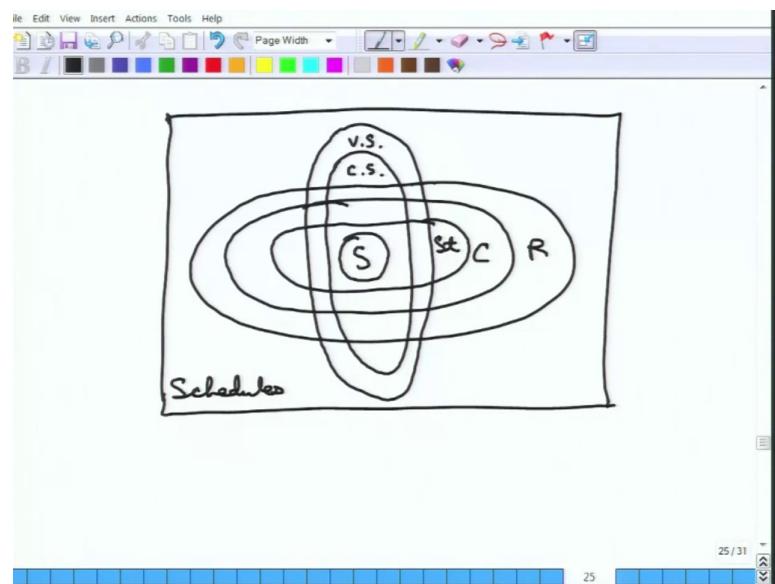


Now, if we repeat the Venn diagram for if we go back to the concept of serializability and we can write, we can produce the similar Venn diagram. So, if it is serial; that means, that it is

also conflict serializable, but of course, there are other conflict serializable schedules, which are not strict and every conflict serializable schedule is also view serializable, but not the other way round.

So, we can again think of some another Venn diagram like this, now the question is these are two Venn diagrams, let me kind of very quickly abbreviate them and write down what I am trying to say. So, suppose this serial schedule is S, now that is your S.

(Refer Slide Time: 19:51)



So, what is the connection, essentially what I am trying to say is what is a connection between recoverability and serializability and here is the Venn diagram that it shows is that suppose, so this is your let us say conflict serializable. Then, let us say this is your view serializable, then there can be schedules which are strict. So, this is strict what does it mean that there are schedules which are strict, but neither view serializable nor of course, conflict serializable. Then, there can be strict schedule, which are view serializable, but not conflict serializable.

Then, there are strict schedules, that are conflict serializable and that means, of course, view serializable, but not strict and so and so forth. And strict and then, there can be this cascadeless and then, finally there can be recoverable. So, apparently all possible combinations are there and of course, just to complete this, this is the entire space of schedules and there are some schedules, which are not at all view serializable and not recoverable.

So, but the interesting part is here is that serializability and recoverability apparently can take way in any parts of there can be schedules which so the recoverability is independent of the serializability. So, whether the schedule is recoverable or not says nothing about whether the schedule is serializable or not and vice versa. So, apparently you can find example for all of them and that is the nice thought example or whatever you can do this that's a nice homework for you to do about.

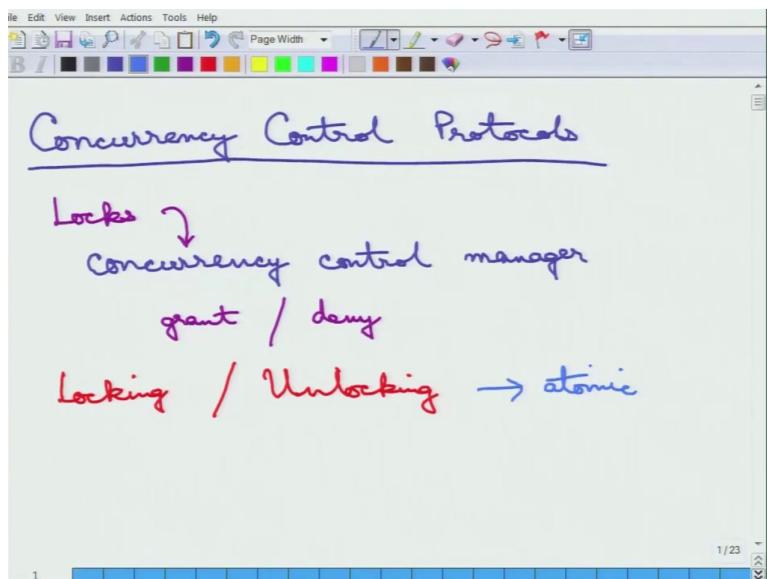
So, thing about can we get examples of schedules, which follow in one of these things and not the other. So, all the portions of the Venn diagram can you get example of all of them. So, that completes the module on schedules and we will next start on concurrency control protocols in the next module.

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

**Lecture - 38**  
**Concurrency Control: Locks**

We will continue with the transactions, we have already seen what a transaction is, what a schedule is and some recovery control systems, etcetera. So, we will next move on to concurrency control protocols. So, concurrency control protocols are essentially that is what they decide, how to manage the concurrency between two or more transactions in a schedule. So, the logs, etcetera and when to give access of a particular data item to a transaction that is what we will be studying next.

(Refer Slide Time: 00:42)



So, this is about concurrency control protocols. So, one of the ways of ensuring concurrency is through the use of locks. We all know that locks are, what OS also use, operating systems also used to ensure concurrency and the databases are no exception, they also use lock. So, locks are what that access the control. So, this control the access to a data item and the requests of lock are essentially made to, there is a module in the database, which is called the concurrency control manager.

So, this is the module in the database that handles this concurrency control protocols and handles the locks, etcetera. So, all requests for locks are made to this concurrency control

manager. The concurrency control manager decides whether to grant the lock or deny the lock, if the grant, if the lock is granted then the transaction essentially acquires that lock and goes ahead with that operation on the data item. If it denies, it can actually wait, the transaction generally just waits till it is granted later by the concurrency control manager and then it goes away.

One very important thing is that the operation of locking on data item by a transaction as well as operation of unlocking the data item by a transaction, these two operations must themselves be atomic. So, it cannot happen that two transactions are trying to lock a particular data item. So, only one of them will get the lock or it may happen that none of them gets the lock, that is the separate issue. But, so these are by itself atomic operations, the locking and unlocking, just the operations by themselves the atomic.

So, either a transaction completely gets a lock or it does not get it. So, two transactions cannot get the same lock, even if they mean request almost in the same time, they will some ways of ensuring that only at most one of the transactions get the lock.

(Refer Slide Time: 02:53)

The slide has a title 'Data item' and contains the following text:

1. Exclusive lock ( $\times$ ) : written / read
2. Shared lock (S) : read ~~written~~

Below this is a hand-drawn 'Lock compatibility matrix' table:

	S	$\times$
S	yes	no
$\times$	no	no

So, then a data item may be locked, so a data item is what it is locked or unlocked, a data item may be locked in essentially two modes, the first one is called an exclusive lock. So, this is denoted, so exclusive lock, this is denoted by an X. So, essentially this means that data item can be both written as well as read. So, if a transaction obtains an exclusive lock on a data item, then the transaction can both write and read to the data item. The other type of lock

is the shared lock which is denoted by S, which means the transaction if it obtains a shared lock on the data item, it can only read.

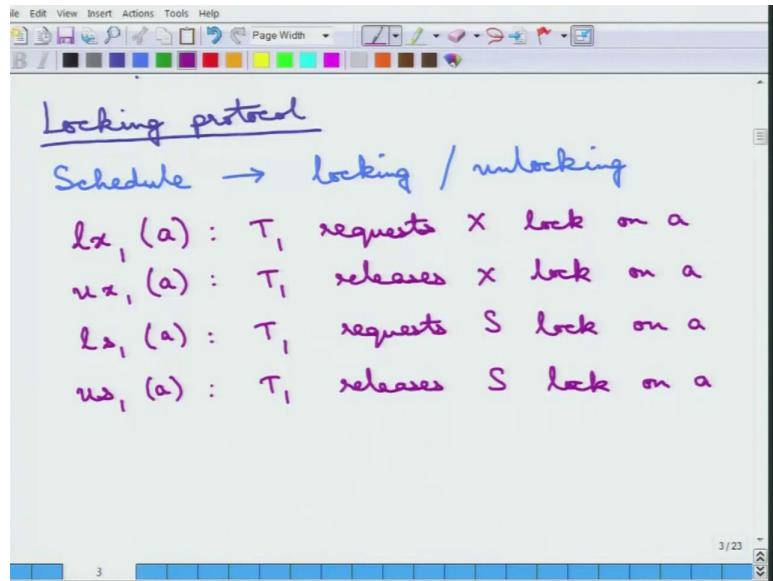
So, write data item cannot be written, so it can be only read by the transaction. So, these are the two important things the exclusive lock and the shared lock and then there is a lock compatibility matrix. So, there is a lock compatibility matrix that essentially defines the compatibility between the different kinds of locks. So, which means that suppose there are two transactions that are also, all locking etcetera we will be assuming on the same data item; otherwise, there is no conflicts and there is no issue.

But, suppose there are two transactions, the row denotes one transaction and the columns denotes another transaction. So, the point is if there is some transaction which has got shared lock and the other transaction wants the shared lock. Can we get it? Yes, it can get it. Suppose, one transaction has got a shared lock and the other wants an exclusive lock, can it get it, no it cannot. So, similarly if one has got the exclusive lock and the other wants the shared lock, it cannot and of course, if one has got the exclusive lock, the other cannot get the exclusive lock.

Now, this is should be very intuitive as to what is happening, this is essentially to guard against the conflicts with write. So, if there are two transactions that the both of them want to write to the same thing, which is both of them are wanting the X lock, that cannot be granted. Even if one of them wants read and the other is writing, the other transaction will not be able to get the S the shared lock, because other has got the exclusive lock.

Now, only case where both of the transactions can get is both of them wants to read the data item, in which case there is no conflict. So, both of them can just obtain a shared lock which is why the S to S matrix, the entry for S to S is yes and they can just go ahead with doing that. So, the read, read there is no conflict, this lock compatibility matrix does just it codes the conflict that we have already studied. So, that is the thing and as I said earlier, that if a lock cannot be granted in this case, it will be just wait for that. So, using all these locks, we will next move on to the locking protocols.

(Refer Slide Time: 05:43)



So, the locking protocol is essentially how the transactions must be granted locks, so that the concurrency can be maintained and there is no issue with the correctness. So, the correctness of the schedules will also be maintained. So, we have seen that the schedule we have already seen that the schedule specifies all the read, write and maybe the abort and commit operations as well. In a locking protocol, a schedule must also mention explicitly, all the locking and unlocking operations. So, it must say the transaction one wants a lock, etcetera and so on and so forth.

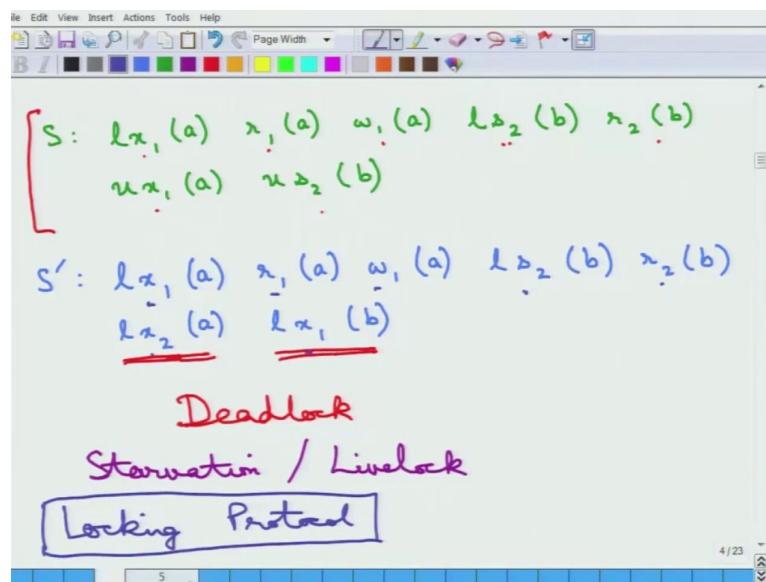
Now, essentially we cannot guarantee whether the locking or unlocking has been done, because that is done by the concurrency control manager. So, what it essentially tries to do is to request a lock and then unlock that is it. So, now, that is denoted by in the following manner, so when a transaction... So, this  $lx_1(a)$  transaction, that this notation means that transaction  $T_1$  wants, if transaction  $T_1$  requests an x lock on data item a, so that is the meaning of this, so that is why it is called a lock x.

So, l stands for locking, x means exclusive lock by transaction one on data item a. So, that is the notation  $lx_1(a)$ , so if transaction  $T_1$  request x lock on a. Now, when you say  $ux_1(a)$ , essentially it means that transaction  $T_1$  releases the x lock on a. So, these are the things, it either requests or release. Then, similarly there is this  $us_1(a)$ , which means transaction 1 requests a shared lock, S lock on the data item a and then, there is an  $us_1(a)$  which is

transaction 1 releases the S lock on a.

So, these are the four types of operations that will be added to the schedules in addition with the read, write and abort, committed, etcetera to specify the locking and unlocking of this operations. So, for example schedule can be simply specified in the following manner.

(Refer Slide Time: 08:04)



Let us just go over briefly what does this mean. So, essentially it means that first the transaction one requests the exclusive lock on a, once it gets it otherwise it must wait once in gets it, it reads a and then write to a, then transaction 2 request the shared lock on b which will be granted, if there are assuming there are no other transaction since, because these are in the different data item and this is a exclusive lock and this is just shared lock on b which is got nothing to do with a, that is granted then it reads, because it has got a shared lock it can only read it cannot write.

So, it only reads which is fine and then this transaction 1 releases the exclusive lock and transaction 2 releases the shared lock and that is the end of the schedule. So, that is how a schedule will be specified in this. Now, let us see an example of what needs to be, so this is all fine as a schedule. Now, let us consider another schedule and see what may happen with such a schedule, now note very importantly what we are doing here is that the following.

So, transaction 1 once an exclusive lock on a assuming there are no other transactions and assuming there is nothing else going on, this lock will be granted, because there is no conflict

nothing. So, transaction one gets that lock only then it reads and then writes, transaction 2 request the shared lock on b again that will be granted, because there is no problem then it reads. Now, what happens is that transaction 2 wants an exclusive lock on a and transaction 1 wants an exclusive lock on b.

Now, the important thing comes is that transaction 1 is already holding an exclusive lock on a. So, unless transaction 1 releases it or unlocks it transaction 2 cannot get another exclusive lock on a, because that is what it is not allowed by the lock compatibility matrix. So, it will not be getting, so it will keep on waiting, so transaction 2 at this point will keep on waiting for the exclusive lock. On the other hand transaction 1 wants an exclusive lock on b, now transaction 2 holds the shared lock on b it has not released it.

So, transaction 1 wants an exclusive lock again this is not going to be allowed and this will keep on waiting. So, now, essentially what is happening is here is very, very important. Transaction 2 keeps waiting for an exclusive lock on a which it is not going to get unless transaction 1 releases, transaction 1 on the other hand is waiting for an exclusive lock on b, which is again not going to be granted unless transaction 2 releases.

Now, you see this is the classic deadlock situation, transaction 1 wants a lock which it is not going to get and unless it is I mean and it is not releasing other locks and transaction 2 wants another lock which is again it is not going to get. So, it is not going to release other lock, so both the transactions are just waiting for locks and for the other transaction 2 and none of them is going to release it, so it is the classic deadlock situation.

So, the concurrency control protocols whatever this protocol in the schedule etcetera, the locking unlocking may result in the deadlock I mean it is only says when to request for the lock it does not mean that the lock is got immediately, it only request for a lock and depending on the concurrency control manager it may or may not get it and it is especially something is that it can never violate the lock compatibility matrix. So, if there is another transaction that holds an exclusive lock, it cannot get another exclusive lock or shared lock on it.

And if another transaction holds the shared lock, it cannot get an exclusive lock that is the lock compatibility matrix. So, it may result in dead locks, now otherwise what will happen is that there may be a violation of the correctness. So, the concurrency control manager will never sacrifice on the correctness it will not grant this lock. So, what may happen is that there

maybe the situation of dead lock that is the one thing, the other thing is starvation or live lock that is another phenomenon that may also happen.

So, what is essentially it is the phenomenon of starvation or live lock is that suppose there are many transactions and transaction 1 holds the lock on some item a. Now, both transaction 2 and transaction 3 wants it as it happens that once transaction 1 releases it, transaction 3 grabs the lock, so transaction 2 does not get it. Now, before transaction 3 releases it, transaction 4 again wants the same lock on the same data item.

So, once transaction 3 releases transaction 4 grabs it and transaction 2 again does not get it and then again before transaction 4 releases, transaction 5 wants it, so transaction again does not get it. So, essentially there is no dead lock in the system, in the sense that the transactions are proceeding some of the transactions are at least proceeding is not that all the system is halted, but transaction 2 is starved, it is starved.

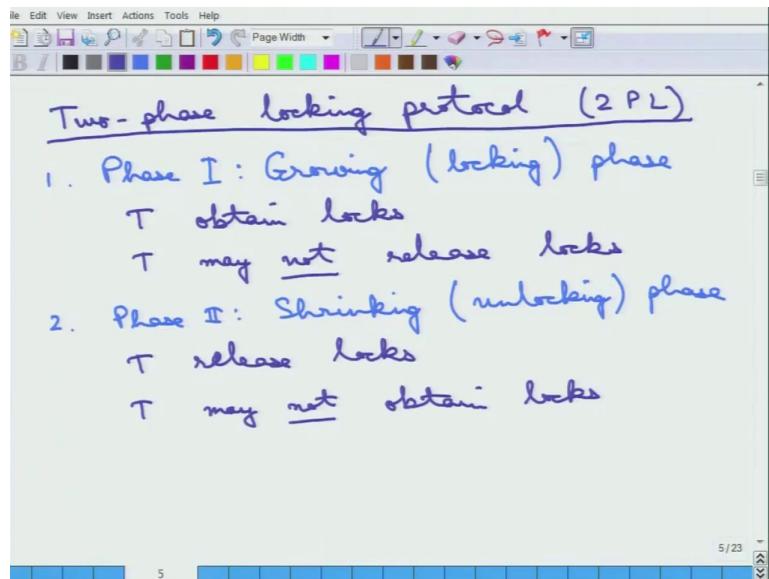
Why it is called starvation? Because, it is not getting the lock that it wants some other transactions are getting it. So, that is the phenomenon of starvation and live lock and that can also happen if this schedules are just requesting for locks and releasing it that can also happen. So, that is the point of schedule, so then what it is being done is that now what we are going to next study is this, what is called the locking protocol.

So, essentially the locking protocol will try to specify how a transaction should request for locks and how it should release the locks, such that the correctness is not violated. So, that is the locking protocol that we are going to study next.

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

**Lecture - 39**  
**Schedules: Concurrency Control: Two-phase Locking Protocol**

(Refer Slide Time: 00:12)



We are going to study locking protocols and the first locking protocol, that we will study is called the Two Phase Locking Protocol or the 2 PL. The 2 PL is a more common term probably and this is the two phase locking protocol. So, there is why is it called two phase, we will see. There is, there are essentially two phase, in the phase 1, the phase 1 is also called the growing phase, growing or the locking phase. In this growing phase or phase 1, a transaction may only obtain locks.

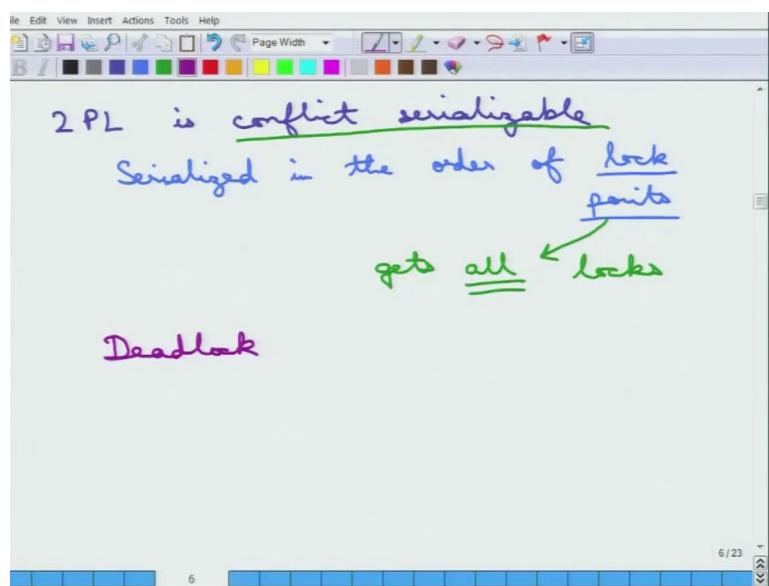
It may not release, it may not release locks, so, it can only get locks, it cannot release locks, so that is the growing phase and the complementary is the phase 2, which is the shrinking phase or the unlocking phase, whereas probably you can already guess, what is going to happen here is that, a transaction may only release locks, it may not obtain locks. So, these are the important things, so what is essentially happening is that there are two distinct phases.

In the first phase, all the transactions can simply request for locks, it can only request and it cannot release any of the locks. In a second phase, the transactions can only release the locks, but it cannot get any more locks. So, very simply the way to think of this is that, in the first

phase all the transactions is to request for all the locks without releasing any one of them, because once it starts releasing the second phase, it cannot obtain anything.

So, essentially all the transactions first request for all the locks that it will need to complete the transactions. So, the locks on all the items that it will need, that is the growing phase or the locking phase and in the second phase, the unlocking phase it will just keep on releasing these things, it will not ask for any more locks. So, these are the two phase locking protocol.

(Refer Slide Time: 02:40)



Now, it can be shown, couple of things can be done is that this two phase locking protocol, 2 PL can be shown to be conflict serializable, this is very, very interesting. So, the two phase locking protocol is conflict serializable. Now, there can be proof of it, but intuitively it can be said that, that the two phase locking protocol, the schedules this is serialized, so this can be serialized. This is equivalent to a serial schedule, where the order of the transaction in the serial thing is serialized in the order of something called a lock points.

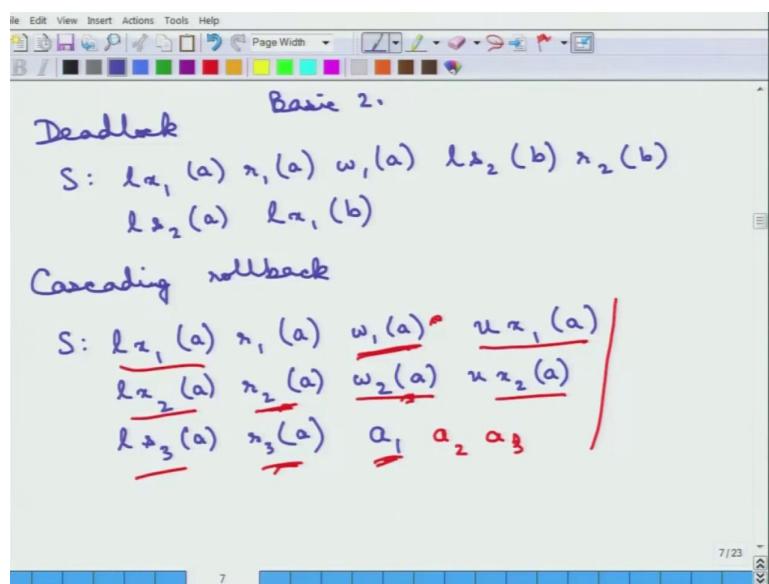
So, what is the lock point? The order of lock points, so a lock point is the time when a transaction gets all the locks. So, that is the thing, so this is called the lock point of a transaction. So, what I am trying to say is the lock point is the following is that... Suppose, transaction one wants the lock on item A and item B. So, once it gets item A, it still not got hold of the lock point, once it is gets also the lock for item B, it will enter, it will then attain what is called the lock point.

Now, do note that in the two phase locking protocol, transaction one after it has got the lock of A, it will not release A, it will only, because it has not got it because it will first, try to get all the locks, which means it will get the lock of B before it will start releasing all the point. And after it has got the locks of all the items that it wants that is when it is called the lock point; that means, the transaction then holds the locks for everything that it needs.

So, that is the lock point of a transaction, though it is scheduled, then it can be serialized in the order of the lock points. So, whichever transaction gets into the lock point first, is the first transaction in the equivalent serial schedule of a two phase schedule. So, here a two phase schedule is conflict serializable, that can be proved, so this is the important part. However, conflict serializable protocols may still suffer from the problem of deadlock, because a transaction one wants the locks and as we saw the example actually; that is actually in a transact two phase locking protocol.

Because, none of the transaction has released any locks, so it was just for trying to get it and it can of course, enter into the deadlock. Because, transaction one gets the lock of A and it wants B, while transaction two has got the lock of B and it wants A. So, this is the basic two phase locking protocol and it may simply suffer from the problem of deadlock.

(Refer Slide Time: 05:14)



So, here is probably an example to, just to highlight the point deadlock, so and the schedule that exemplifies that it may suffer from the deadlock. So, this is a very simple example of how it can deadlock, so this point the system is deadlock. The two phase locking protocol can

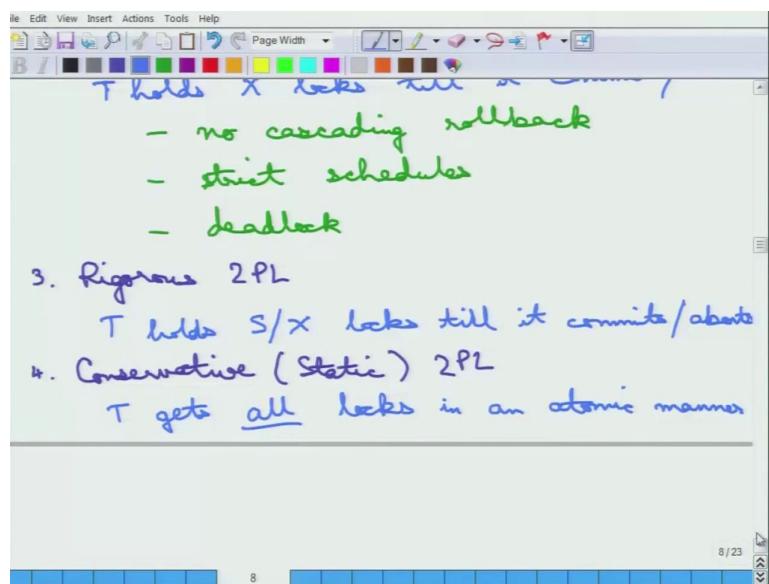
also suffer from the problem of cascading rollbacks. If you remember the cascading rollbacks is that when one transaction aborts, other transaction has to abort as a result of that and then, some other transaction may have to abort and so on and so forth, so that may also happen and here is the example for this.

So, let me first put on the example and at this point, what happens is that, transaction one aborts. So, if transaction one aborts, so everything has been going on fine, now transaction one has aborted, so; that means, this is all, this write operation is essentially invalidated. So, this should not go through; that means, this read is invalidated; that means, this write is also invalidated, so; that means, this read is also invalidated. So, essentially if  $a_1$  happens, then transaction two must abort and transaction three must abort.

So, this is the cascading rollback, but note that this is in the two phase protocol, because for every transaction it gets the locks and then it releases it, it gets the locks and then it releases it, it gets the locks and then it releases it, it is getting all the locks that it wants. So, there is a, it is a simple way to see that this is still in two phase locking protocol, but it can suffer from this problem of cascading rollbacks.

So, it can suffer from both the... So, the basic two phase protocol, this is the basic two phase protocol that we have been studying. This can suffer from the problems of both deadlock and cascading rollbacks.

(Refer Slide Time: 06:48)



There are some variants of the two phase locking protocol. So, the first thing of course, we studied is the basic one. So, this is the basic, this is called the basic two phase locking protocol. This is the basic protocol that we have just studied, then there is something called a strict two phase locking protocol. So, a strict two phase locking protocol, the following thing happens that the transaction must hold all its exclusive locks till it commits.

So, it holds X locks till it commits or transaction must hold its X locks, till it commits or abort. So, then let us see what this happens is that, so transaction holds the exclusive lock... What it means is that, once it has got an exclusive lock for writing, essentially exclusive lock used for writing, it will hold on to it unless it commits or aborts. So, what is the effect; that means that suppose the transaction commits and then, it releases the locks; that means, that the lock that it has got did its job correctly of course, and the write operation is also successful.

So, anything that reads after this will be correct, the point is since it holds the lock, nobody, no other transaction can start reading that particular data item, unless it releases the locks and it has released the lock only after it has committed; that means, only after it has made sure that the write that it has done is correct. Otherwise of course, it just aborts and then, releases it, but then no other transaction will have any effect on this, because it cannot read.

So, there is in no way any other transaction can read that particular data item, before its commits, before it releases the lock and by the time the transaction is either committed or aborted, so it has essentially taken the decision. So, this means, that there is no cascading rollback in this, so there is no cascading rollback in this strict two phase locking protocol. Because, it cannot be it cannot, so the cascading rollback happens, because another transaction reads the particular data item that has been written, but not committed or aborted and that cannot happen, because it holds all the exclusive locks.

And it produces strict schedules, because nobody can even write to the other data item, because it still holds the x locks before it commits or aborts. So, this case is called it produce a strict schedule and that is why, you can see that why it is called a strict two phase locking protocol, because it produces the strict schedule. However, it may still deadlock, so the problem of deadlock still remains and again it is easy to prove that it may deadlock and I invite you to think of an example to show that in a strict two phase locking protocol it may still deadlock fine.

So, that is the strict two phase locking protocol, then there is something called a rigorous two

phase locking protocol. So, here what happens is a transaction holds all locks till it commits, so it, so holds S X both locks till it commits or aborts. So, rigorous two phase locking protocol it of course, strict, because it holds on to X locks and it is, it also holds on to the shared lock, it does not even release the shared locks till it aborts or commits.

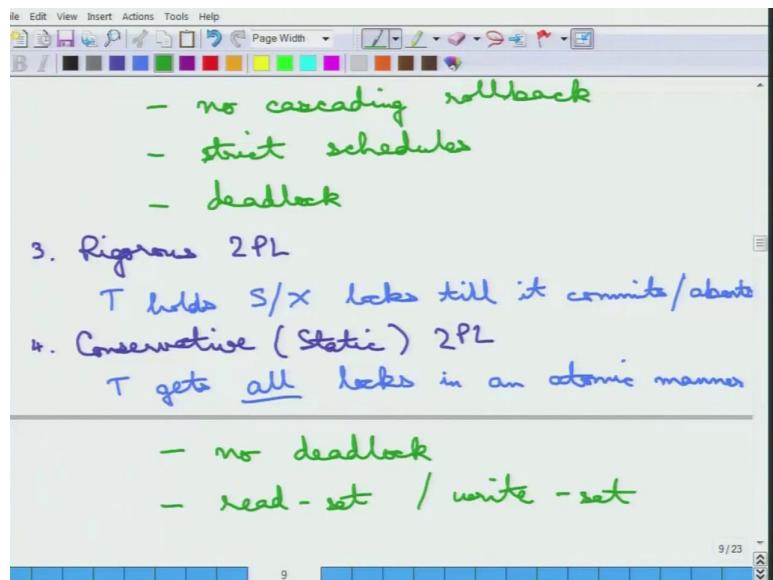
So, this is again of course, can be serialized in the order of this. So, this can be serialized simply in the order of their commits, but this is the same property it holds on to this no cascading roll to this on and all of these things. So, no cascading rollback and strict schedules it will do and it dead locks also. So, that fourth one in this is called the conservative or sometimes called the static, two phase locking protocol that this is an important thing transaction gets.

So, by the way rigorous two phase locking protocol can also deadlock, because the simply, because one transaction wants another lock the transaction one wants a lock on B while transaction two wants a lock on A and transaction one holds the lock on A and transaction two holds the lock on B. So, nothing it, so only says that it cannot it does not release till it commit or abort, but does not say how to get it, so the dead lock may also happen.

In a conservative two phase protocol transaction gets all the locks in an atomic manner and now this requires a little bit more thought of what is this happening, so what does an atomic manner means. So, a transaction essentially at the beginning of it, it declare as the intent of getting all the locks. So, it will say I want an exclusive lock on A and a shared lock on B and an exclusive lock on C and etcetera.

And it requests all of those locks together, so both transaction one does it and transaction two does it and what the concurrency control protocol manager will do is that it either honours all the lock request together and it does not occur does not honour anything. So, what I am trying to say is that if a transaction one wants the locks A B and C, either it will get all the locks A, B and C or it will get none of the locks. So, its atomic that is why it is called all the locks in a atomic manner. So, it cannot happen that it will get the lock on A, but not a B and C and it will lock on A and B, but not on C, so and so forth.

(Refer Slide Time: 12:31)



So, very simply you can see that if this happens, then this cannot deadlock, what is the deadlock situation, the deadlock situation is, that there are two transaction holding locks that the other one wants. So, now, this cannot happen, because if transaction one wants the lock on B and holds on A it cannot happen it cannot want a lock on B while holding A, because it either it has got both A and B or it has got none of them and the same thing for transaction 2, so there is no deadlock.

So, this is deadlocks free and as an implementation detail or as a more detail of these things each transaction declares, what it called it is read set and write set and it gets the shared locks if it wants it can get shared locks and read set and the exclusive locks on write set. But, essential idea is that it either gets all the locks or it gets none of the locks, so that is why it cannot deadlock. So, that is the whole point of this conservative or static two phase locking protocol.

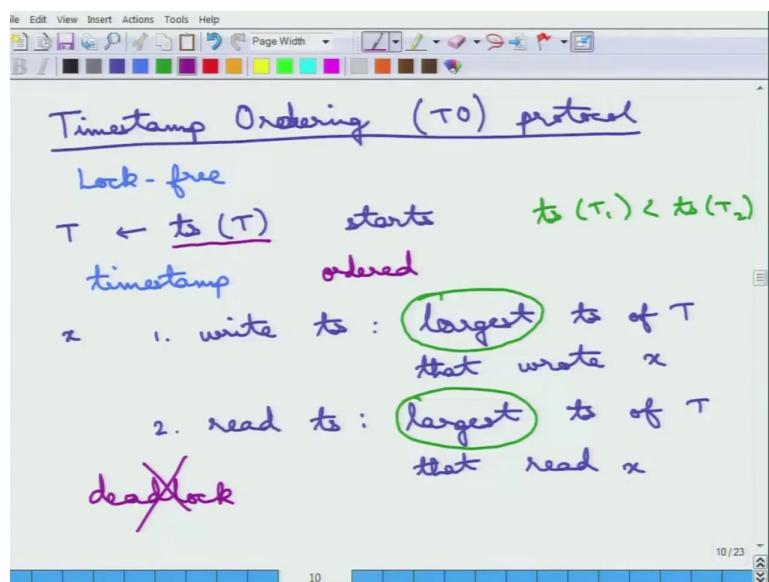
So, these are the four variants of the two phase locking protocol essentially just to summaries the two phase locking protocol, every transaction must first get all the locks and then, only it can start releasing the locks.

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

**Lecture – 40**  
**Concurrency Control: Timestamp Ordering Protocol**

Next we will move on to another Concurrency Control Protocol, which is the Timestamp based and very importantly this does not use locks, it is the lock free protocol.

(Refer Slide Time: 00:19)



So, this protocol is called Timestamp Ordering or the TO protocol. So, timestamp ordering protocol, this is very importantly this is lock free, so it does not use any lock on the data item. So, first of all what is a timestamp. So, at each transaction is assigned a particular timestamp when it starts, so when the transaction starts it is assigned a timestamp, so this is called the  $ts(T)$  is essentially the timestamp of T.

So, a timestamp is a logical counter, it is a logical time, it can be the actual wall clock time or any logical counter; such that, this is ordered, this is... So, the timestamps are ordered and it is any logical clock. So, this is an ordered logical clock, so very simply every time tick may be given and you need to give timestamp one, then two and so on and so forth. And it can be anything, as long as it is ordered and ordered meaning, so between two timestamps it can be always determined whether the timestamp of the first timestamp is lesser than the second timestamp or equal, that can be always done, so that is why it is ordered.

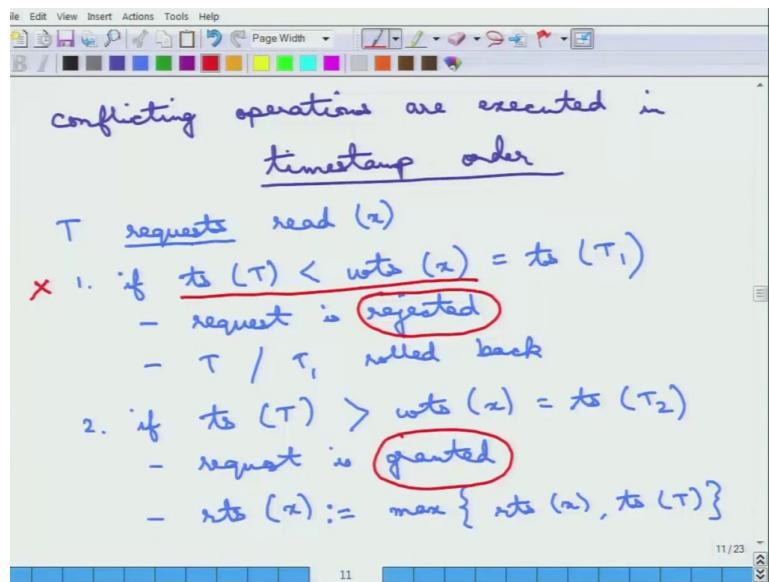
So, that is the timestamp for the transaction, so this when it starts. So, when the transaction starts; that is given a timestamp; that is the timestamp of the transaction. Now, for every data item  $x$ , so there are two timestamps that are maintained. The first one is called the write timestamp, which is the largest timestamp of any transaction that wrote  $x$ . So, that wrote  $x$  successfully, but essentially, so what does it mean, that  $x$  is the data item and suppose transaction 1 wrote it and transaction 2 wrote it both, so it is the largest timestamp.

So, in the transaction if transaction 1 starts before transaction 2, then the timestamp of transaction 1 will be less than the timestamp of transaction 2; that is because it is ordered, etcetera. And if both of them suppose as written to an  $x$ , then this will get the largest timestamp, so this is important of this is the largest timestamp of any transaction that are successfully written to  $x$ . And similar to the write timestamp there is of course, a read timestamp, which is the same definition, it is the largest timestamp of a transaction  $T$ , that successfully read  $x$ . So, once more this is the largest timestamp of something that has wrote to  $x$  and this thing.

So, the timestamp ordering protocol uses only this timestamp. So, it only uses the timestamp and we will see that, but essentially it can be shown that type, so the protocols that use timestamps can never deadlock, because it... So, there is always a strict ordering between the transactions that has started, so transaction 1 if it starts before transaction 2, it must be having a lower timestamp than the ordering 2.

And, so it can never deadlock, because there is an implicit priority order of the transaction based on the timestamp. So, timestamp ordering protocol or for that may be any protocol that you can think of that uses timestamps will not deadlock. So, this is the very important property of that.

(Refer Slide Time: 04:03)



So, coming to the actual protocol, what it does is that it the basic idea of that is that, the conflicting operations, the basic philosophy of the timestamp ordering things is that, the conflicting operations are executed in the timestamp order. So, if there are two operations  $i_1$  and  $i_2$  and then,  $i_1$  has a lower timestamp then it is executed in that particular order, are executed in timestamp order. So, this is the important part of it, this is the of course the intuition or the philosophy of what the timestamp ordering protocol is and we will go to the details of it next.

So, suppose this is this thing, so a transaction requests a read. Note that this is a request, because it may or may not successfully end doing it, correct. So, it is just requests a read on x, now couple of things may happen is that the following things may happen is that, if transaction, the timestamp of transaction T is less than the write timestamp of x. Then, what is essentially the meaning is that, the write timestamp of x essentially says that, there is a transaction. What is the write timestamp of x, it is a timestamp of a transaction and that is written to x and that transaction is greater.

So, the timestamp of that transaction is greater than the timestamp of this transaction T; that means, that write has essentially should not have happened before this read. So, essentially this read is late and there is somebody written, some other transaction which was supposed to write later has already written and only now the read request for this transaction is coming. So, this is not correct, because this is now going to read, if it now reads it is going to read,

what has been written by the timestamp, the wts of x.

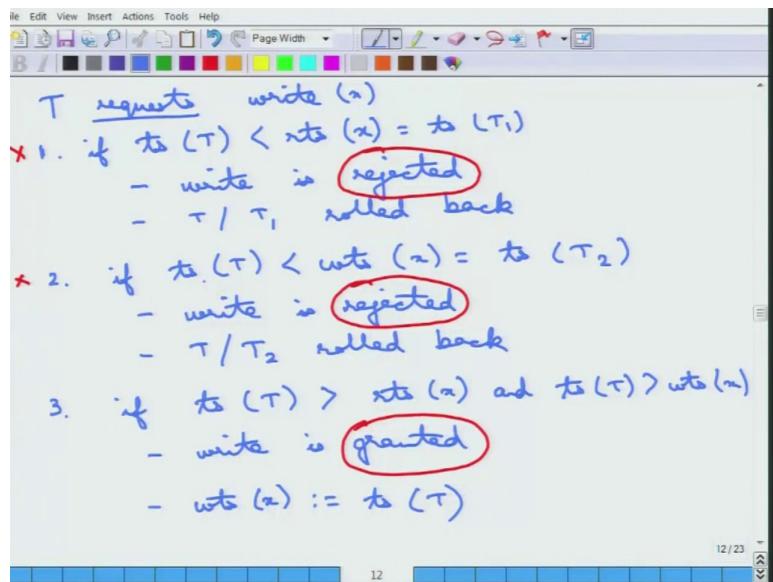
So, just to elaborate this further, so suppose the write timestamp is essentially the timestamp of some other transaction  $T_1$  that has written. So, essentially transaction of the timestamp of this T is less than the timestamp of the transaction 1 that has written, so it should have read the value before  $T_1$  got a chance to write it. So, it has not done it, so this is wrong, so this operation this is not going to be allowed.

So, what it does is that, this request is rejected and, so this request is rejected and this T or  $T_1$ , whichever one of them must be rolled back, because they have not done it in the order that they were supposed to do. So, T the read of T should have happen before the write of  $T_1$ , but they have not done, either this happen, so this is not correct. So, essentially this is the whole point if this happens, then this is not correct, so then it cannot allow, so this request is rejected, this is very, very important, so this request does not go through.

On the other hand, if the timestamp of transaction T is greater than the write timestamp of x, which is let us say that some other  $T_2$ , then there is no problem. Because,  $T_2$  was supposed to write to this x before t s has read, because the timestamp of  $T_2$  is lesser than the timestamp of T and that is what is being done, so this is fine. So, this is simply request is granted, request is granted meaning the T is allowed to read x, which also means that the read timestamp of x may be need to be updated.

So, the read timestamp of x is then updated to the... So, it is the largest read timestamp, so it is either the, whatever it is already there or the timestamp of T. So, essentially the timestamp of T is greater than the read timestamp, then it is updated otherwise it remains the old thing, so that is the thing. So, this is, very importantly this is granted. So, that is the read request; that is what happens when the read is, when a transaction T request the read.

(Refer Slide Time: 08:38)



The next thing is when a transaction T requests a write, so this is again just a request, request a write of x. The first thing is that, if the transaction if the timestamp of T is less than the read timestamp of x. Suppose this is equal to some timestamp of  $T_1$ , this means that again the same problem as in the earlier case. So, the read of timestamp  $T_1$ ,  $T_1$  has already read it, now  $T_1$  should not have read it before the T has the chance to write, because T is earlier, the timestamp of T is lesser than the timestamp of  $T_1$ .

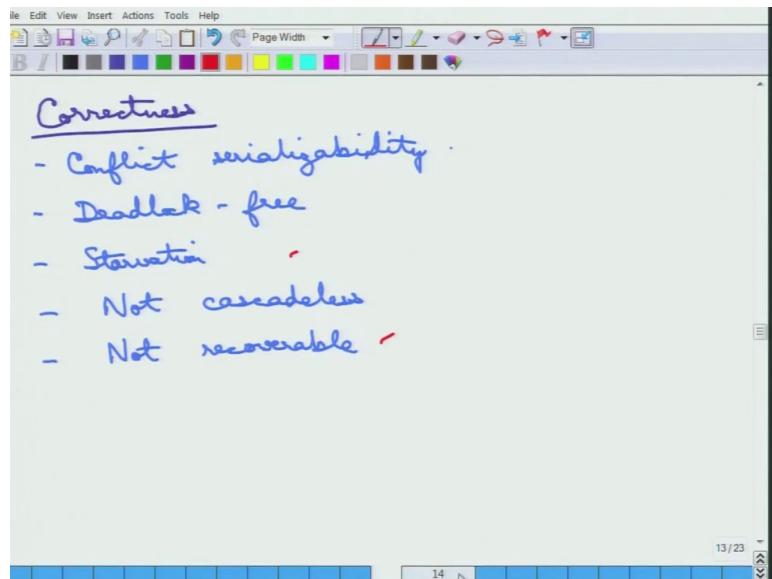
So, T should have written before the read of x by  $T_1$  has done, so again this is wrong, this should not be allow, so this write request is rejected, this write is rejected and the same thing is that T or  $T_1$ , one of them is rolled back that is the same idea, because there is a read write conflict and it has not progressed in the correct orders, so this is rejected, so this does not go through. The second one is that if the timestamp T is less than the write timestamp of x, so let us say this is the timestamp for  $T_2$ .

Again the problem is T should have written before  $T_2$ , because the timestamp of T is lesser than the timestamp of  $T_2$ . So, the write to x for T should have happen before the write of x by  $T_2$ , so the same thing, so this write is again rejected. If these are write, write conflict, the previous one was read write conflict, so this is again rejected and then, the same thing is that T or  $T_2$  is rolled back. So, this is rejected and this operation, this has been go through.

And finally, transaction T is greater than the read timestamp of x and transaction I mean the timestamp is greater than the write timestamp of x. So, which means that there is no problem, all the read that should have gone earlier has gone earlier and all the write that has gone earlier that has gone earlier, so this is simply allowed, so write is granted. And similar to the previous case, the write timestamp may now need to be updated, so the write timestamp of x is needs to be updated, but now you see that the write timestamp is just needs to be updated to t, because there cannot be any other timestamp which is greater than T.

So, simply this can be taken as the write timestamp of T, so that is the two things. So, these are the two important parts, if that when T request read and write, then what happens essentially. So, very simply if a transaction comes after and then, it is in the correct order, then it is granted otherwise it is rejected.

(Refer Slide Time: 12:09)



Now, all this is fine, we now need to argue a little more formally about the correctness. Why is this going to be correct? So, the correctness of the timestamp ordering protocol can be argued. So, this first of all, this guarantees conflict serializability, so this guarantees conflict serializability; that is not probably very hard to see, because it does not allow any conflicting operations to go through that. Hence, the whole idea of the protocol, why it rejects this thing is, because it was violating those conflicts, so that is the thing.

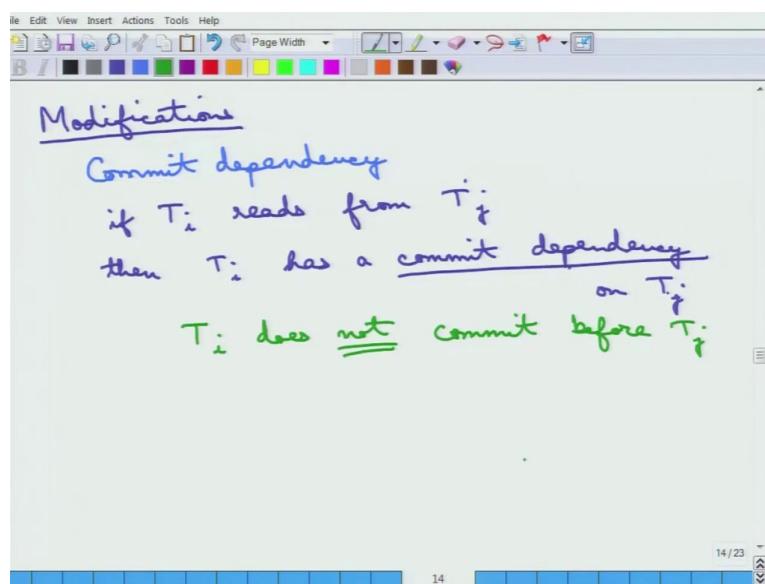
So, essentially the idea is that the conflicting operations are executed in correct timestamp order, if it is not, then if it appears, so if it is requested in a non, if it appears in a non

timestamp ordering what it should be, then it is rejected, that is called first thing. The other thing is, as I say there cannot be any deadlock, because the transaction that started earlier always gets the priority, so all these things the transaction with a smaller timestamp is always going to get the priority.

So, deadlock free, so this is called deadlock free, so but there may be starvation and what it happen is the same situation is that. The timestamp there may be other transactions with lower timestamp they keep on coming and the particular transaction is just that keeps on waiting indefinitely. It is always rejected, rejected, rejected and so on and so forth. It may cause starvation and it may not be... So, it is not cascadeless, which means that, so essentially it means that cascading rollbacks may happen, because the particular transaction aborts, the other transaction that is dependent on it may also abort, so it not cascadeless.

In fact it is very very interestingly it is not even recoverable, because it has allowed the particular a timestamp, it has allowed a particular transaction to go ahead read or write, because the other transaction has not yet come and that transaction may be just have committed or aborted, so it may have just committed and not aborted, so then it is non recoverable. Because, later we find that it should not done then, but by the time it has already committed. So, this is very important, but this is not recoverable. So, although it is deadlock free and it guarantees conflict serializability, it still has these problems of starvation and non recoverability; of course, if it is not recover it is not cascading as well.

(Refer Slide Time: 14:38)

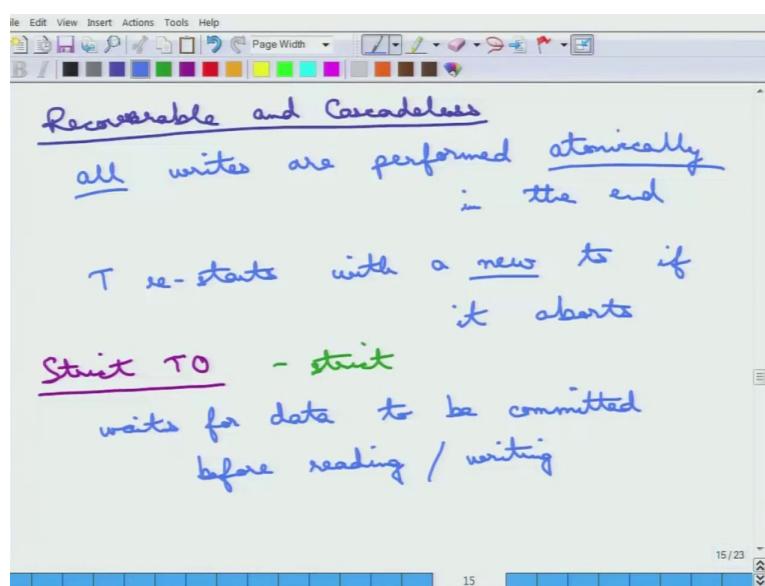


So, there are certain modifications just again the variance of the two phase protocol, there are certain modifications that can be done to the basic timestamp ordering protocol first of all you can use something called commit dependency there is the concept of commit dependency. So, this is to essentially ensure it is that the transactions the schedules are recoverable what is the commit dependency is following if  $T_i$  reads from  $T_j$  then  $T_i$  has a commit dependency on  $T_j$  which is essentially saying that.

So, the dependency on  $T_j$ ; that means, that if  $T_i$  reads from  $T_j$  what does this mean is that  $T_j$  has produced a value which  $T_i$  reads then  $T_i$  has the commit dependency on  $T_j$  which means that  $T_i$  cannot commit unless  $T_j$  commit. So, its commit depends on whether  $T_j$  has committed or not. So, that is called commit dependency. So, this essentially ensure that  $T_i$  does not commit before  $T_j$  does it does not commit because it is dependent on  $T_j$ .

So, unless  $T_j$  commits it cannot commit, it does not commit before  $T_j$ , that is to ensure that it is recoverable transaction is recoverable otherwise what may happen is that  $T_i$  reads and then commits and then  $T_j$  aborts by the time  $T_i$  has already committed and its a wrong commit because  $T_i$  has read a value that it should not have done. So, that is what this thing, correct.

(Refer Slide Time: 16:30)



Now, couple of more things. So, now to make these things recoverable and cascadeless, so this is to making this recoverable and cascadeless. So, there can be certain modification done to this thing to make this recoverable and cascadeless. What can be done is that all writes are performed in the end, all writes are performed in the end atomically are performed atomically in the end, again, what does this atomically mean is that either all the writes go through or none of write are go through, and that is done in the end.

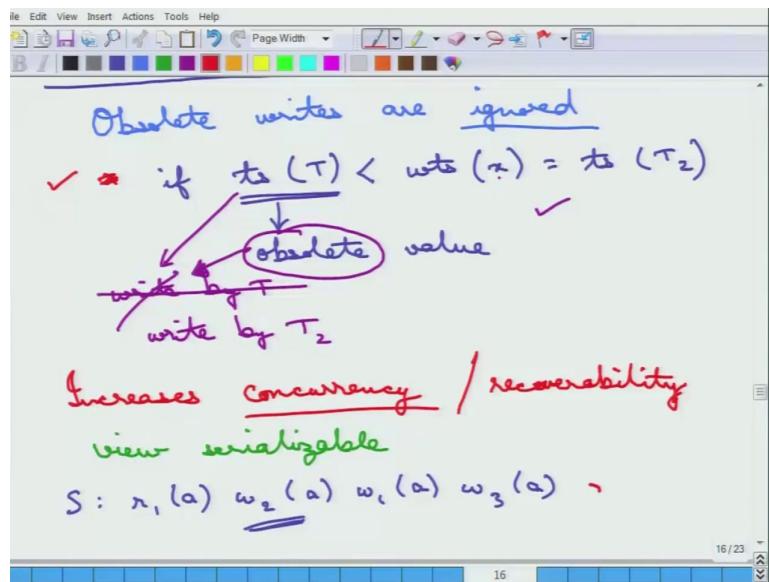
So, that's essentially just saying that none of the writes will be done unless all the writes will be done and that will be done in the end. So, first to read all of these things and then the writes is done. Now, if this cannot be done if a transaction aborts then what happens if a transaction aborts that it just restarts with a new timestamp. So, that means it will get a higher timestamp and then the commit dependencies etc the commit the timestamp because it will now start with higher timestamp all the transactions that started earlier will get the priority over this and because it is aborted and then it will just try to finish it in that way.

So, that is one way of doing the other way of doing it is that let me just write it down so transaction restarts with a new timestamp if it aborts. So, it is new timestamp if it aborts these are the two important things the other way of making this is to use the locks. But then you can see that is going to against the philosophy of why timestamp ordering protocol is needed in the first place because it supposed to be lock free.

So, to use locks to make it recoverable and cascade-less essentially having a locking kind of protocol so that is not a very elegant solution good solution anyway and then there is one version of it called strict timestamp ordering protocol which use this essentially tries to make the protocol strict just like there is the strict version of the 2 PL. So, this essentially waits for data to be committed before it is read or write. So, essentially it does not read or write any uncommitted data.

So, it only reads committed thing. So, that is essentially just mimicking the definition of strict schedules. So, it will only read or write a particular data for which the transaction which produced read or write that thing has committed; that means, that is all correct and then only it reads and writes. So, that is all serial.

(Refer Slide Time: 19:31)



So, that is. So, this is about the timestamp ordering protocol there is one extra thing that can be done which is called the Thomas's write rule. So, Thomas's write rule is to make it little bit more concurrent. So, it is to allow some more schedules which will not be allowed by the basic timestamp protocol. So, remember what we done for conflict serializability, view serializability we did not take care of the blind writes.

So, blind writes, so here the same thing so obsolete writes so its not essentially blind write but obsolete writes, obsolete writes are ignored. So, if there is an obsolete write it is simply ignored. So, what does an ignoring of obsolete write means is the following thing remember what happens in the write case is that if the transaction that wants to write has a timestamp lesser than the read timestamp of this. So suppose this transaction is  $T_2$ , then in the basic timestamp ordering protocol this was rejected because this is essentially the writes have not come in order, in this case in the Thomas's write rule this is allowed.

Because, the idea here is that  $T$  is trying to write an obsolete value why it is an obsolete value the reason it trying to write an obsolete value. Why this is called an obsolete value because, you see what is happening,  $T$  suppose  $T$  would have actually written the value of  $x$  now the another transaction two which would come after  $T$  because its timestamp is greater and would have written  $x$ . So that write by  $T$  will be overwritten by the write by  $T_2$ .

So, this was obsolete anyway so this is why it is obsolete and it is not a problem if an obsolete write was not there, so you just simply ignore, so you simply ignore that there is an

obsolete write and that was not written. So you simply... ignore the loss of an obsolete write because it was obsolete anyway, so then this is allowed and then that protocol just goes through, so what it essentially means is that, instead of the three conditions of the write rule. Two of them does not pass and the one pass the middle condition may also to pass because it is an obsolete write.

and the obsolete write is ignored. So, this increases the concurrency. So, this increases the concurrency of this because it allows some more writes to go through, so it allows some more transactions to go through to successfully complete and commit etc it increase the concurrency and it also increases the recoverability because now what happens is that this transaction will not simply abort it can continue. So, it may be increasing the recoverability, so as a note on this.

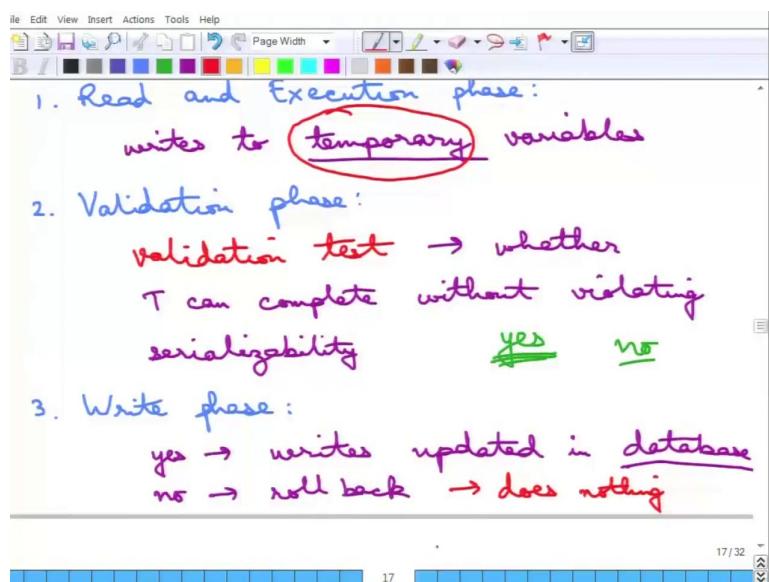
So, this allows certain view serializable things, so this can allow certain view serializable which will not be allowed otherwise by the basic things because you can see this is essentially the obsolete rule is obsolete write is essentially the difference between the conflict and view serializable things, so here is the small example that will highlight this. So, suppose this is  $r_1(a)$ ,  $w_2(a)$ ,  $w_1(a)$  and  $w_3(a)$  now you see that this  $w_2$  is the obsolete write and this will be allowed under this Thomas's write rule, but not otherwise and this protocol it goes through. So, that is the end of the timestamp ordering protocols.

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

**Lecture - 41**  
**Concurrency Control: Validation Based Protocol**

We will continue with the Concurrency Control Protocols, next we will be studying this Validation Based Protocol.

(Refer Slide Time: 00:16)



So, this is called the validation or certification based protocol. So, what happens in this is that, the transaction or all the transactions in a schedule has three distinct phases, the first phase is called the read and execution phase. So, what happens in this phase is that, all the transaction in the schedule, read the variables from the database and execute on them, but write on the temporary values in those databases. So, it only writes to temporary variables, not to the actual database values.

So, only the temporary values of this are modified; that is part of this transaction, this part of the schedule, so nothing is being changed in the actual database, so this is just a temporary variable, this is important. In the second phase, this, the second phase is called the validation phase, which is why this protocol is called a validation based protocol. What happens in this validation phase is that, now, it is actually being checked whether this writes that where going through in the first phase in the execution phase and read phase can be done without violating

the serializability.

So, this validation test is done, there is something called a validation test. So, the validation test essentially checks whether variables can be written, whether it can be written without violating the serializability, whether transaction can complete without violating the serializability. So, whether all those writes that are think are correct in the sense of, they are correct in the sense of serializability. So, without violating the serializability, so that is what the test is being done in this second phase, which is the validation phase.

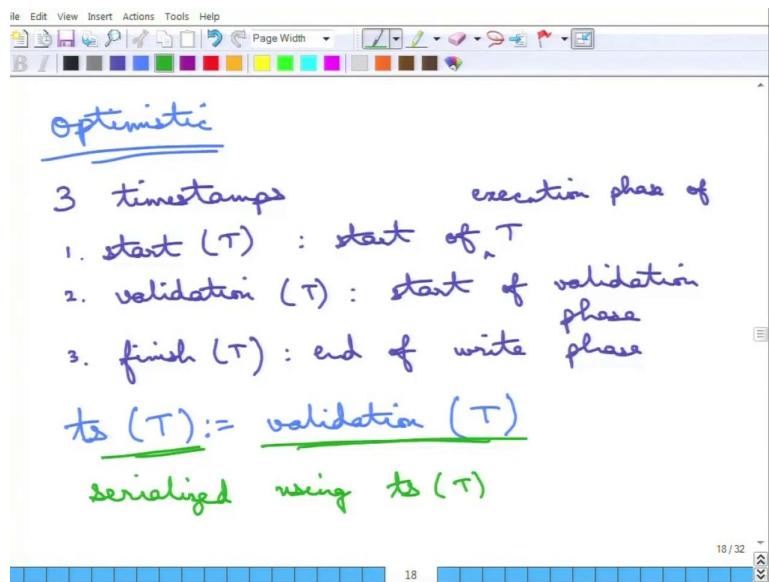
And in a third phase, the third phase is simply called the write phase. Now, there are two things that can happen in the second phase, so the transaction checks whether it can complete without violating this serializability. So, if it can complete without violating the serializability, then in this third phase this writes are actually updated to the database. So, if it is yes, then writes are actually updated to the database, writes updated in database.

So, that is essentially; that means that all the writes that were in the temporary variables are now actually propagated to the database. On the other hand, this violating serializability may also answer no, which means that if the transaction proceeds, then it will violate the serializability. So; that means, it should not proceed, so essentially this is akin to roll back, so the transaction must roll back.

Now, what does this roll back means is that, so the transaction must not write anything to the database. Now, actually notice that the transaction has not written anything to the database, anyway it has written only to the temporary variables. So, the roll back actually meaning, it does anything. It just simply, can does not updates the writes in the database, so it is just nothing. So, there is no problem, that way in the rollback phase, but it does the way to understand is that.

So, the critical thing about this is that, it first executes, assuming that everything will be correct and then, it validates whether that assumption that everything will be correct was valid or not. If it is valid, so the validation phase says yes, then; that means, that everything was actually correct, then it just simply goes and writes to the database; that is the writes are updated in the database. Otherwise if everything is not valid, it cannot write anything to the database. But, it has actually not written anything to the database, anyway it has written only to the temporary variables and nothing needs to be done.

(Refer Slide Time: 04:28)



Now, this is also sometimes called an optimistic concurrency control protocol, this is sometimes also known as an optimistic concurrency control protocol. Why it is called optimistic? Because, the transaction actually does all the operations with the hope, that the validation phase will pass successfully, with the hope that nothing is wrong with the hope that everything is well, so that is why it is called an optimistic.

So, for the transaction to actually do these things, there are generally three time stamps are maintained. So, three time stamps are maintained to do the actual validation step, so the three timestamps for each transaction is the start timestamp for the transaction, this is the start timestamp and second is the validation timestamp, this is the start of the validation phase.

Now, this is the start of the transaction, this is the start of the validation phase of the transaction, so that is the validation timestamp, start of validation phase and finally, there is a finish timestamp, which essentially says that the end of the write phase. So, everything has gone to the database, actually being done correctly to the database. So, this is start of  $T$ , which is you can also say this is start of the execution phase of  $T$ ; that is the same as saying that is the first phase, so that is the same thing as saying part of the execution.

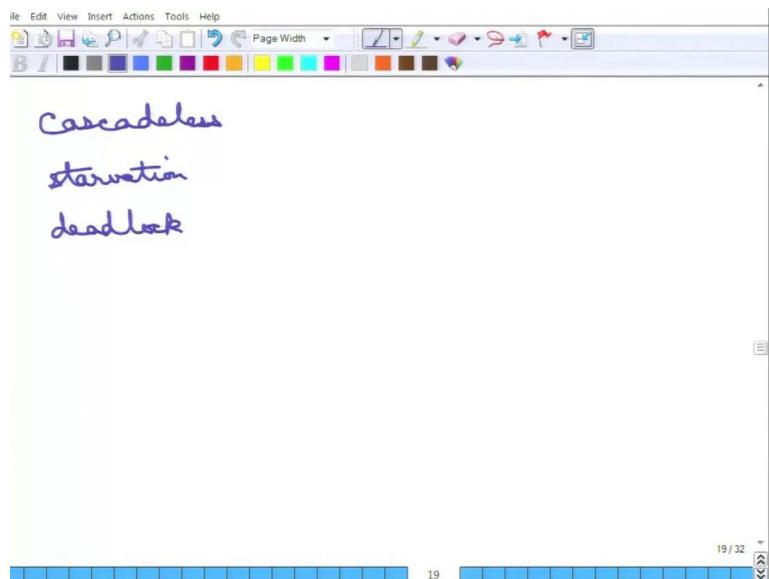
So, these are the three timestamps that the transaction uses in this optimistic concurrency protocol or the validation based protocol. Now, timestamp of the transactions, so there is a timestamp that is given to the transaction, which is set to the validation timestamp of this. So, the timestamp, when somebody talks about the timestamp of the transaction; that is

essentially the timestamp of the validation phase of the transaction, so which is essentially equal to the validation timestamp.

Now, the transactions everything is correct, then this is serialized, if everything is correct, then the serialization is done using this validation, using this timestamp, so using this timestamp  $T$ . So, that serialization is done using this timestamp  $T$ , which is the same as saying that the serialization is done using the validation timestamp. So, what does it mean essentially is that, suppose there are two transactions in the schedule  $T_1$  and  $T_2$  and  $T_1$  enters the validation phase earlier than  $T_2$  and suppose that there is no, nothing wrong in those, so the validation steps are performed correctly.

Then, the serialization order of this, of the transaction in the schedule is  $T_1$ , then  $T_2$ , because  $T_1$ 's validation timestamp is earlier than the validation timestamp of  $T_2$ . So, this increases the concurrency, etcetera.

(Refer Slide Time: 07:22)



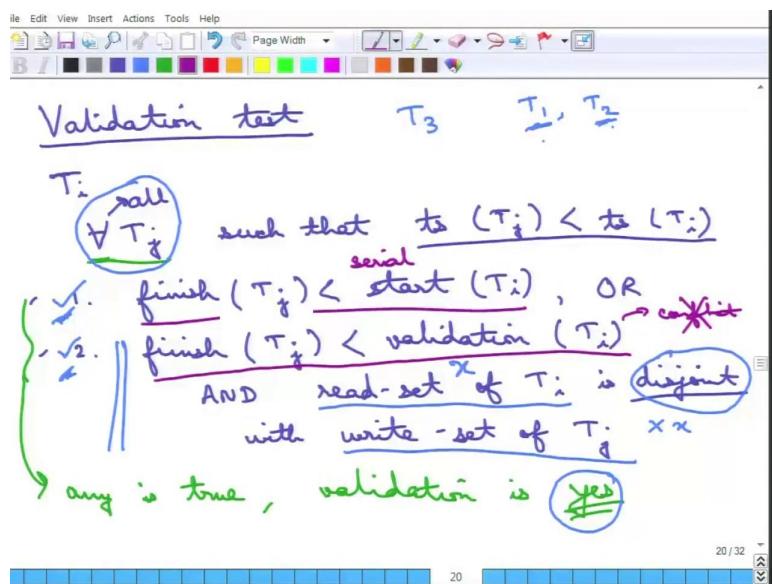
This can be also shown to be cascadeless, this is shown to be cascadeless. The reason why this is cascadeless is that, if a transaction fails, essentially it will not pass the validation phase. So, it is actually not going to write anything, so nothing needs to be roll back in that case, so that is the thing about these transactions. However, it can suffer from starvation, now this, one must understand once this timestamps are established, the timestamps of the transaction are established using this validation thing, then the timestamp based ordering

protocols can be used.

Something similar to the timestamps based ordering protocol can be used to serialize it, so the starvation may happen. In the sense, that if a transaction cannot finish, it restarts with a new timestamp and then, it must again check, etcetera and then, it may again not finished, because some other transaction has come through again and so on and so forth.

So, it keeps on starting, it keeps on restarting and keeps on rolling back or keeps on aborting, that is why it can be starved. Although, theoretically there is no deadlock, because these are timestamps based protocols and there is always a strict order of transactions. Between two transactions, there is always a strict order based on the timestamp of the transaction, so there is no deadlock. But; however, as we can see, this can have starvation, now what exactly at the validation test that is being done.

(Refer Slide Time: 08:41)



So, let us now move on to the details of the validation test. So, what how this validation phase actually does, we have been talking about the validation phase etcetera, what exactly are this validation test. So, for a transaction  $T_i$  there are two conditions, that are checked for all transactions  $T_j$ , so we are worried about transaction  $T_i$  and for all transaction  $T_j$ ; such that, so the time stamps of  $T_j$  is less than the time stamp of  $T_i$ ; that means, all transactions that started earlier the two things are checked, the first thing is that  $\text{finish } T_j \text{ finish} < \text{start of } T_i$ ; that is the first check.

The second check is that the finish  $T_j$  is earlier than the validation of validation phase of  $T_i$  and this is very importantly and the read set. So, earlier I say in a moment what a read set is, but let me just write this, now the read set of  $T_i$  is disjoint with disjoint with write set of  $T_j$ . If either of this condition is true, then the validation passes, so if anyone is true any of this condition is true, then validation phase says yes the validation is yes otherwise it is no.

So, what does this mean let us go over these two conditions one by one, so we are worried about transaction  $T_i$ , now note for this is for all transactions  $T_j$ . So, this is for all transaction  $T_j$  it is not for a single transaction  $T_j$ , so what for all transactions  $T_j$ , that has the timestamp earlier than the timestamp of the transaction  $T_i$ , that we are worried about, it must happen that  $T_j$  finish before  $T_i$  started.

Now, what happens if  $T_j$  finishes before  $T_i$  started of course, if  $T_j$  finishes remember that the finish  $T_j$  that time stamp is when  $T_j$  has written everything to the database, so if the  $T_j$  is completed. Now, if  $T_i$  starts after  $T_j$ , then of course, everything is validated nothing is wrong, because  $T_j$  was supposed to complete earlier than  $T_i$  and  $T_j$  finish before even  $T_i$  started. So, that is fine that is this one is probably easier to understand, why the validation is correct, then.

The second condition or the condition is that  $T_j$  finished before the validation started, now couple of things is happening. So,  $T_j$  finish before validation started, but before validation of  $T_i$  started, what is the  $T_i$  phase, that it is into it is the execution of the read phase of  $T_i$  so; that means,  $T_i$  is actually reading from the database reading some variables from the database and writing to the temporary copies of it, but it is actually reading.

Now, what may happen is that think of an item  $x$ , so  $T_i$  reads  $x$ , now it may happen that  $T_j$  was supposed to write  $x$  and  $T_j$  wrote  $x$  after  $T_i$  did, so there is a problem correct. So, it cannot say that it has validated, because it is not clear when  $T_j$  wrote. But, the and condition says that the read set of  $T_i$  is disjoint with the write set of  $T_j$ ; that means, if  $T_i$  reads  $x$  from the database  $T_j$  does not write to  $x$ .

So,  $T_j$  it is disjoint; that means, this they do not share anything so; that means, if  $x$  is read by  $T_i$ , then  $x$  is not written by  $T_j$ ; that means, if  $T_i$  read some variable  $y$ ; that is guaranteed that  $T_j$  has not written to  $y$ ; that is why it is called disjoint so; that means, that the reading of  $y$  by  $T_i$  is correct, because there is no transactions earlier than earlier to it which has read which has written to it that way also this is correct.

So, that is why, even if this condition is met, then the validation is also correct and, so that is why it can written validation as yes. So, this the two conditions, that must be these are the two conditions, so either of one of them must be happening for the validation to be passing. Just to remind you once more that this has to be for all  $T_j$ , now so; that means, that, so suppose you are worried about the transaction  $T_3$  and before transaction  $T_3$  there is transaction  $T_1$  and  $T_2$ .

Now, for each of these transactions either condition one or condition two must hold, so for  $T_1$  and  $T_2$ , so it may happen the  $T_1$  for  $T_1$  transact condition one holds and for  $T_2$  condition two holds, so that is fine. Either of these conditions must hold for  $T_1$  and  $T_2$  only, then the validation test is supposed to pass correctly. Now, what is the outcome of all of these things if these condition happens this is essentially just a serial I mean this is  $T_j$  finishes and then,  $T_i$  starts, so that is just essentially saying this a serial.

And the second condition says that the reads of  $T_i$  are not affected by the writes of  $T_j$ , so there is no conflict so; that means, there is no conflict, so that is why these two are correct. So, that is the whole thing about validation test this is how the validation test is done and that is how the validation based protocol concurrency based protocol goes through.

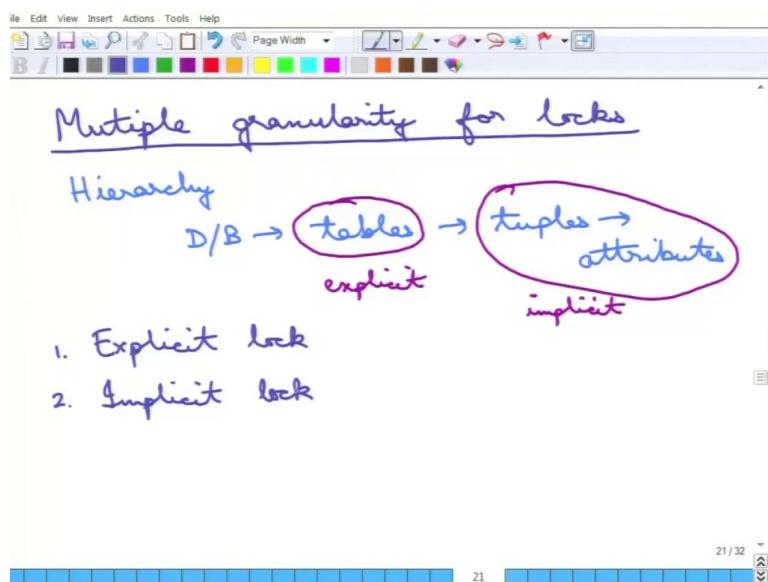
**Fundamentals of Database Systems**  
**Prof. Arnab Bhattacharya**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kanpur**

**Lecture - 42**  
**Concurrency Control: Multiple Granularity for Locks**

We will next continue with some other kinds of issues in the Concurrency Control Protocols. So, we will return to locks, but more importantly, we will return to locks in what is called a Multiple Granularity. So, if you have noticed, then we have been understanding locks, we have been studying locks for various name data items. But, we have not bothered about, what the data items were actually.

Now, what happens in that, in some cases is that, suppose there is a table; that the entire table needs to be locked and then, within the table, there are records or tuples that needs to be locked again and within in a tuple, there may be an attribute that needs to be allowed. So, the lock is essentially applied at multiple granularity, so the lock may be applied to only an attribute or to a table or to a record or to a entire database and so on and so forth and then, there are some issues that we will discuss themselves.

(Refer Slide Time: 01:05)



So, this is the issue of multiple granularity for locks, so multiple granularity. So, the granularity essentially means at which level the lock is applied. Now, what one may do is that, so the hierarchy, so if this takes into account the hierarchy of the database. So, the

hierarchy may be there is a database, then there are different tables, then there are tuples or records within that, then there are attributes within that and so on and so forth. This is the hierarchy and it may be, locks may be applied at different levels.

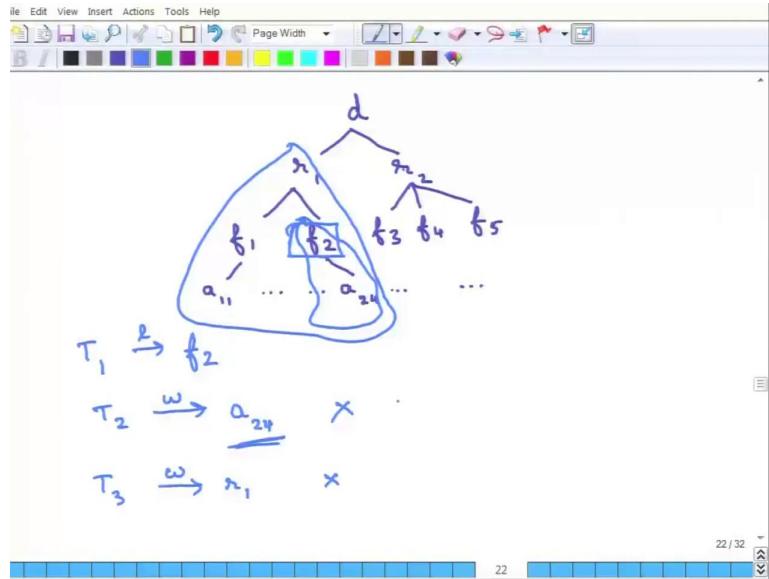
Now, the locking, when it is done at different levels, then the following thing happens. So, suppose a particular table is locked. Now, what does locking a particular table mean? It is essentially, even a particular table is locked that is an explicit lock, but implicitly what happens is that all the tuples and all the attributes in all the tuples are also assumed to be locked implicitly. Because, otherwise you can see what is the problem is that.

So, suppose a table is locked; that means, the table needs to be read, etcetera and if the tuple is not locked, if the tuple is not supposed to be locked, then what may happen is that some other transaction may come and writes to the tuple. So, then this problem of summarization, problem of maintaining, statistics, etcetera may get harm. So, that is why these are also supposed to be locked, but these are not locked in explicitly, this is locked in an implicit mode.

So, then essentially there are they; that means, there are two kinds of locks which is an explicit lock and then there is an implicit lock and locking something, explicitly locks everything under it in a implicit mode. So, the granularity of locking essentially is that, this can be in a fine granularity; that means, this is lower in the hierarchy, so something like attributes are being locked, etcetera and; that means, one can assume higher concurrency, because if only an attribute is locked, other parts of the tuple can be accessed, other tables can be accessed, etcetera.

But, it is a high locking overhead, because every attribute is being locked, etcetera. On the other hand if it is a coarse granularity, then something coarse such as tables, etcetera is locked. So, it reduces the concurrency, but the locking overhead is low.

(Refer Slide Time: 03:30)



Anyway, so here is an example of a hierarchy, so this is the database  $d$  and then, under it, there are assumed, there are two tables  $r_1$  and  $r_2$ , under  $r_1$ , this is just an illustrating example. So, suppose there are these tuples  $f_1$  and  $f_2$  and under  $r_2$ , there are this  $f_3$ ,  $f_4$  and  $f_5$  and under  $f_1$ , there are these attributes  $a_{11}$ , etcetera and so on and so forth. So, this is the thing that to be done.

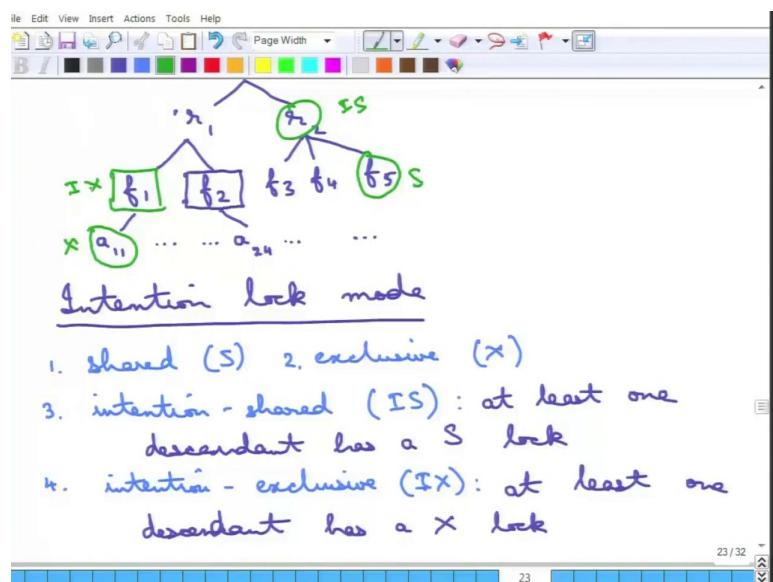
Now, suppose there is a transaction  $T_1$ ,  $T_1$  has locked tuple  $f_2$ , so  $T_1$  has locked  $f_2$ , now  $T_2$ , so this has locked already, now  $T_2$  wants to lock  $a_{24}$ . What is going to happen is that, this should not be allowed, because even though  $f_2$  is locked explicitly, this attribute  $a_{24}$  is locked implicitly. So, this request cannot be granted, because this is an implicit lock that is been taken up  $a_{24}$ . So, that is because, it has locked  $f_2$ , so this goes to because  $f_2$  is locked, everything under it is supposed to be locked as well. But, suppose  $T_3$  wants to lock  $r_1$ , should it be allowed,  $T_3$  wants to lock  $r_1$ ; that is also not allowed, because if  $T_3$  locks  $r_1$ , then it means that  $T_3$  will be locking this entire thing.

So, suppose  $T_3$  wants to lock  $r_1$ , it is not going to be allowed, because if  $T_3$  locks  $r_1$ ; then it means, this  $T_3$  is going to lock this entire thing, which means that the lock at  $f_2$  again clashes with that at  $T_1$ . So, it is not going to be allowed, so essentially what is being meant is that, if  $f_2$  is locked, all its descendants are locked, as well as all its ascendants up to the

database cannot be taken a lock on any of its descendants up to the database. And none of its descendants in its sub tree cannot be locked either, so that is the thing about implicit locks.

So,  $r_1$  could not be locked, even though, there is no such implicit lock on  $r_1$ , but  $r_1$  still cannot be locked, because locking  $r_1$  would actually mean an implicit lock on  $f_2$ . So, that would clash with the explicit lock of  $T_1$ , so that is why, it is not going to be allowed, so that is the thing. So, this you can see that, this is clearly going to be a little inefficient, etcetera.

(Refer Slide Time: 05:47)



So, what is being done is that, there is something called an intention lock mode; that is being used. So, intention locking mode, intention lock mode that is used, so that is essentially what we have been discussing. So, if  $f_2$  is locked, then all its descendants are in the intention lock mode. So, what is descendant is in the intention mode, so it is not an implicit lock, but it is an intention mode lock, so that is the intention mode.

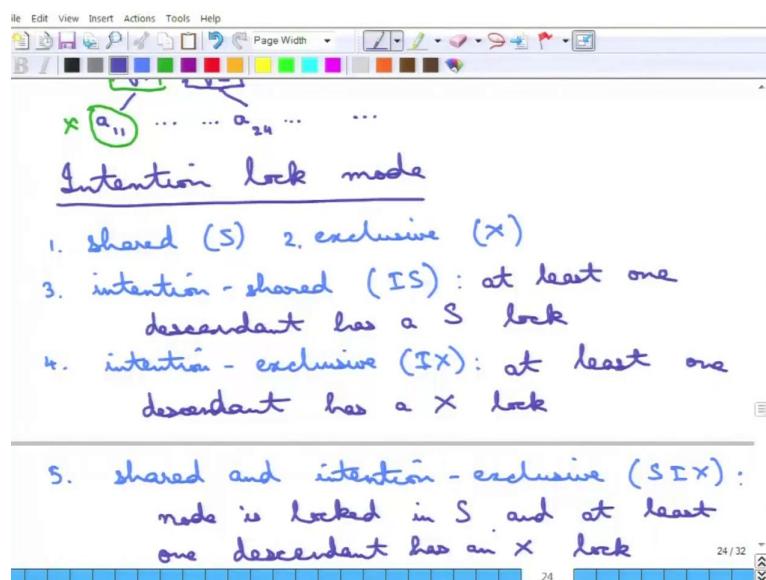
So, we have been studied, we had already studied this shared lock S and the exclusive lock X; that we had studied for the read write. If you remember, there was the lock matrix as well S, X, etcetera. The intention lock mode introduces three additional locks, so this is 1 and 2, but it introduces three more locks, the first one is called an intention shared lock, so this is the IS lock.

So, what is an intention shared lock? This means that, at least one descendant has a S lock. So, at least one descendant has a S lock, so then it is called an intention shared lock. So, what

does this mean is that, suppose this is locked in S mode, then  $r_2$  is supposed to be in an IS mode, because at least one of its descendants is in an S lock. So, this is why the intention lock mode is being used.

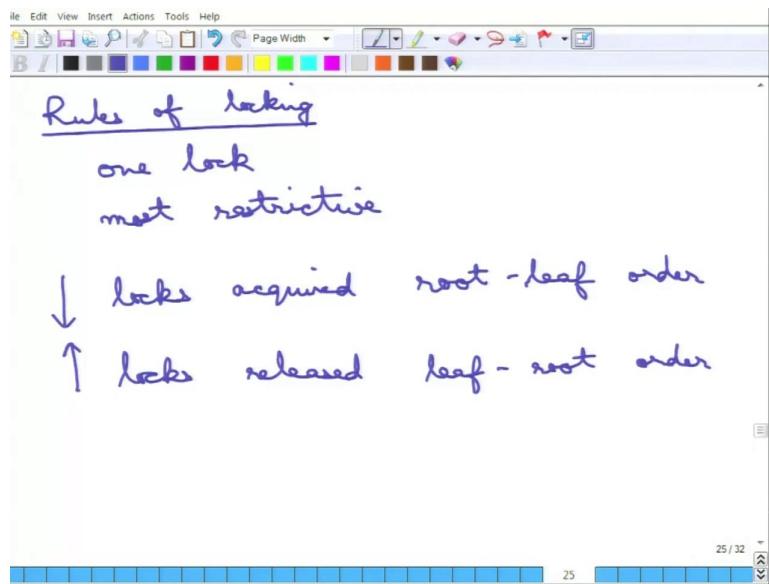
Then, similar to IS there is something called an IX, which is intention exclusive lock mode IX. So, this is the definition is similar, so at least one descendant has an X lock. So, that is why, so this is called an intention exclusive, because this is really not an exclusive mode, but it is an intent of an exclusive mode, because some descendant has an X lock. So, again this is easy to see, what can be the example, so suppose, this is X mode, then this is in IX mode.

(Refer Slide Time: 08:18)



Very, very importantly, there is a 5th kind of lock, which is called a shared and intention exclusive lock, this is both shared and intention exclusive lock, this is called the SIX lock. This is a famous rather an unfamous lock, this is a SIX lock. So, shared and intention exclusive lock, this thing is that, the node itself is in S mode and at least one descendant has X mode. So, this is what a shared this one is in X mode, so this is a shared intensive lock. So, the shared intensive lock is a little bit complicated, we will come to that.

(Refer Slide Time: 09:19)



But, before that, let us go through the rules of locking. Using this now, we have five locks in our hand and so we must have the rules of locking, we must know the rules of locking. So, the first rule, actually some simple rule is that a transaction may obtain only one lock at a time, on an entity at a time. So, that means, if a transaction wants a lock on one data item, it can only get one lock, it cannot get two locks, so it cannot for an example get an S lock and an X lock, it should get only one lock. Now, the question is, which lock will it get? It will get the most restrictive lock.

So, it should get only one lock and that is the most restrictive lock that it should get. So, if it wants both S and X, it should get the most restrictive, which means, it should get X. And every lock must be given notice of all low level locks, so that means that whenever something, whenever transaction locks a data item, it sends the notice up to all the descendants.

So, the descendants must be known, in that something has happened in the descendants, otherwise it cannot get this intention lock mode, so that is there. So, the locks are acquired in a root to leaf order. So, from top to bottom; that means and the locks are released in a leaf to root order. Little bit of thinking will tell you why that is necessary is that, before the locks are released, some other transaction may come and take another locks.

For all those cases of conflicts do not happen, the locks are acquired in a root to leaf; that means, the highest lock that is needed is first got; that means, that everything under this is

supposed to be locked and then, the next lock in the level is acquired and so on and so forth. On the other end, when it is released, it is released in the opposite order, because otherwise what may happen is that, suppose a higher leaf is released, then anybody checking the higher leaf would think, higher node will think that everything else below is not locked; that is wrong. So, it is released in the leaf to root order. So, these are the rules of locking, but very, very importantly the one thing that is left is that, the actual locking matrix.

(Refer Slide Time: 11:45)

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	n
IX	Y	Y	n	n	n
S	Y	n	Y	n	n
SIX	Y	n	n	n	n
X	n	n			

Now, we have five kinds of locks and the lock compatibility matrix must be noted for all of them. So, this is the lock compatibility matrix between all these five kinds of locks. So, I am going to draw the lock compatibility matrix in a moment, so the locks are the following, so this is a IS lock then there is an IX lock, then there is an S lock, then there is SIX lock, then there is an X lock, this is one thing.

And on the rows, there is the same thing IS, then this is IX, then this is S, then there is SIX lock and then, there is a X. The reason I need to writing this is the least restrictive to most restrictive. Let us complete the lock compatibility matrix and then, we will understand what this is why I am saying this. So, this is S; that means, if one transaction gets an IS lock on one data item, another transaction can then get an IS lock on that same data item.

This is also y; that means, IS and IX also do not conflict IS and S do not conflict, IS and SIX do not conflict. However, IS and X do conflict; that means that, if a transaction is holding an intention shared lock on one data item, another transaction cannot get an X lock on it, so that

is very important. So, no that this is no, next we continue with the IX to IS is yes, this we have already seen and IX to IX is also yes, if both transactions can hold an IX lock; that means, some of the one of the descendants is in X for both the things and we will worry about it later, which descendent etcetera.

But, otherwise this is no problem; IX to S is however no. So, one cannot get a shared lock on a node, if one of it is descendent is in a exclusive mode, because what does it mean, why is this a problem, a shared lock on everything means that, it is probably reading that entire table etcetera. Then, IX meaning, there is something inside is locked for writing.

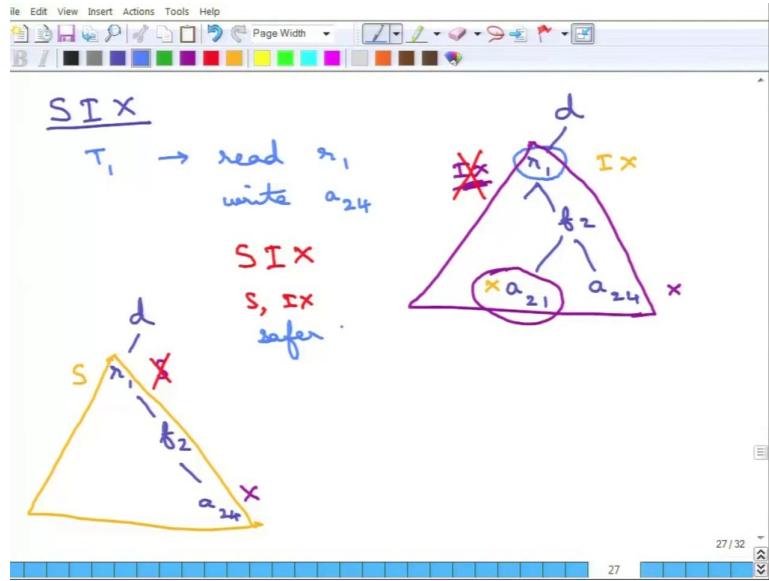
Now, that means, that if the writing is being done, while it is reading then the summary etcetera will be wrong, that is why, this is not allowed. Now, IX to SIX is also not allowed and IX to X of course, not allowed. Next comes here the S lock, which is the shared lock, now shared to IS is y; there is no problem, shared to IX we have already seen, this is no. Then, shared to shared will be yes, both of them can get a shared lock, no problem.

But, this SIX lock cannot be here within the X lock and X is no of course, then comes the SIX lock, the SIX lock is this is yes, then with IX, SIX lock cannot be shared. So, then SIX lock cannot be shared with S as well, SIX lock cannot be shared with SIX lock itself and nothing can be shared with X. And finally, X lock cannot be shared with anything. Now, if you notice this matrix is symmetric, as it should be, because it does not matter which transaction is on the row and which transaction is on the column, but here are the things.

So, couple of important things to notice that, X lock cannot be shared with anything, if something is being exclusively locked for writing; that means, that is supposed to be written and; that means, it cannot be shared with anything else. And if something is being exclusive and something has been intentionally locked.

So, in the intention mode of shared lock; that means, something below it is only read and even this is not being read, then it can share with most of the locks except of course, the X lock, because something inside this being written. So, that cannot be done, this is the lock compatibility matrix that one must remember to get an understanding. Now, it is probably easier to understand all those locks except the SIX locks. So, what it exactly is the SIX lock, the SIX lock is something little bit more interesting etcetera.

(Refer Slide Time: 15:55)



So, let us go over the SIX lock in a little bit more detail. So, what does SIX lock, why is SIX lock needed, how it is useful. Now, suppose there is a transaction  $T_1$ ; that wants to read  $r_1$ , this is reading  $r_1$ , but modify, but it wants to write  $a_{24}$ . So, if you remember this was one is being done etcetera and then, there was an  $f_2$  under which there was an  $a_{24}$ .

So, transaction  $T_1$  wants to read  $r_1$  and write to  $a_{24}$ , now if transaction  $T_1$  wants to read and it writes to  $a_{24}$ , then what happens is that, how on which mode will it lock  $r_1$ , so that is the question. So, the thing is that, if  $r_1$  is locked in the let us say  $r_1$  is being locked in the IX mode, so that is the first thing  $r_1$  is locked in the IX mode, which means that, because there is something X going on here it can be IX mode.

Now, which means, that any other transaction that wants to lock  $r$  into get this IX mode. Now, let us check up these things, IS to IX is y, so that means, any other transactions, which wants to lock  $r_1$  in a IX mode may also be granted, so that means, some other transaction may also lock this  $r_1$  into IX mode. So, this is unsafe, why because if the some other transaction is locking this, then what may happen is that some other transaction may be actually modifying  $a_{21}$ , which is unsafe, because  $T_1$  is supposed to be reading the entire  $r_1$ ; that is why, that is  $T_1$  is reading  $r_1$ , so that is these thing.

So, that means, this is unsafe, this is where the problem will happen. So, it should not allow it into IX mode. So, that is the first part of it and what about if it only locks it in an S mode, so

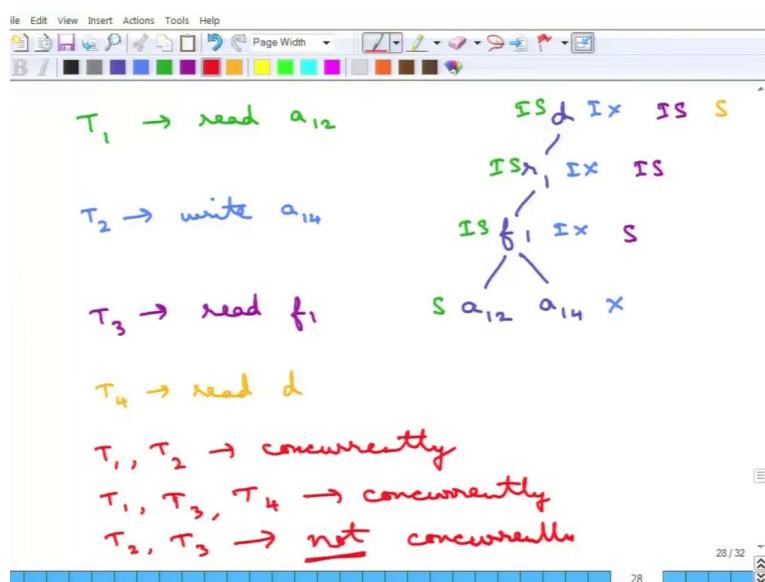
suppose on the other hand, the same example. So, let me redraw the example here once more, this is  $r_1$ , then this is  $f_2$ , this is  $a_{24}$ . Now, what if this is locked only in the S mode, because it is only reading, then some other transactions may also come and lock it in a S mode.

So, which means that, what happens if this is locked in the S mode, then some other transaction may come and also lock it in the S mode because S to S is allowed. So, then; that means, that although there is an X here, these transaction the other transaction when it is reading this entire thing will have a problem, because this is being modified. So, S lock cannot be allowed as well.

So, the abstract is that neither IX mode should be allowed nor S mode should be allowed, now as a compromise, what is being done is that, a new mode is being invented new mode is being designed, which is called the SIX mode. That is it wants to say that this is both X, this is both IX and S. So, SIX mode as you can see the name this is S and IX both; that means, that I want that this transaction is saying that, this is being read completely.

So, that is why, which is S, but something inside it also be written that is why it is IX. So, that is why, this is a SIX mode. So, this SIX mode this SIX lock is essentially a compromise from IX and S and this is safer SIX mode is much safer than either as S mode or an IX mode.

(Refer Slide Time: 19:33)



So, this is the thing, let us now do some example. So, suppose  $T_1$  wants to read  $a_{12}$ . So, it simply wants to read  $a_{12}$ . So, then what is the thing it applies, it applies this in S mode, then

this IS mode, this is IS mode and this is IS mode simply that is fine done that is the first example. The second example suppose  $T_2$  wants to write  $a_{14}$ , so what it should be doing is that, it is going to lock this in X mode, then this is IX mode, this is IX mode and this is IX mode.

Then, suppose the third example is that there is a transaction  $T_3$ , which wants to read  $f_1$ , so that is one it is doing it wants to read  $f_1$ . So, what it does is that, it locks  $f_1$  in S mode and then,  $r_1$  in IS mode and d in IS mode. So, that is again very simple and the fourth example is that  $T_4$  wants to read the entire database, so read d. So, the only thing that it does is that, it just locks d in the S mode and that is what it is being done.

So, these are all the four locks, then doing go going ahead, let us do a little bit analysis of which examples, I mean of all these transactions, which can be going through concurrently. So, now, the thing is  $T_1$ ,  $T_2$  can if they execute concurrently  $T_1$  and  $T_2$ . So,  $T_1$  is trying to read  $a_{12}$  and  $T_2$  is trying to write  $a_{14}$ . So, can they go ahead concurrently, yes they can, because there is no problem, because this is an S and X in a different things and this is IS and IX. If we go back to our compatibility matrix IX and IS is compatible.

So, there is no problem with this. So,  $T_1$  and  $T_2$  can be done concurrently. So, there is no problem with that. Then, let us think of some other kind of operations. So, let us say let us now argue about  $T_1$  may be  $T_3$  and  $T_4$ , what happen what can be done about them. So,  $T_1$  is this is all in S IS mode,  $T_3$  is this is in S mode and these are in IS mode and this is S. So, can they go ahead, so we have to check IS versus S which is S. So, there is no problem. So, this can also go through concurrently.

So, there is no issue with them, because these are either in IS and S modes and they do not conflict, they are compatible with each other. So, this can go ahead concurrently. Now, let us say  $T_2$  and  $T_3$ , we will just going to do some examples  $T_2$  and  $T_3$ . So,  $T_2$  this is in S IS, IX mode and  $T_3$  is S IS IX. So, now, can they go ahead concurrently, we need to check and S with IX is no; that means, that this cannot go through. So, there is a conflict here S with IX cannot be done.

So, this cannot be read one transaction cannot be reading  $f_1$  completely, because something

inside it is being executed. So, this cannot go ahead concurrently this cannot go ahead concurrently this is not compatible  $T_2$  and  $T_3$  are not compatible. And let us just do one more example, which is  $T_2$  and  $T_4$ , can they go ahead concurrently, the thing is that  $T_2$  is IX. So, this is the same problem S with IX.

So, again they cannot go ahead concurrently that is it can be understood from the... So, if one remembers the lock compatibility matrix, then this can be done. Even, otherwise one does not remember to really memorize the compatibility matrix, the compatibility matrix is quite intuitive and if one draws this diagram, it is probably easier to understand why S, is not compatible with IX.

Because, as you see in this example, if somebody is reading  $f_1$ ; that means, it is trying to read everything, including  $a_{14}$ ; that is why it is getting a lock S on this, while some other transaction is writing on this. So, which is of course, got a read write conflict and that cannot be allowed. So, that is why these things can be done.

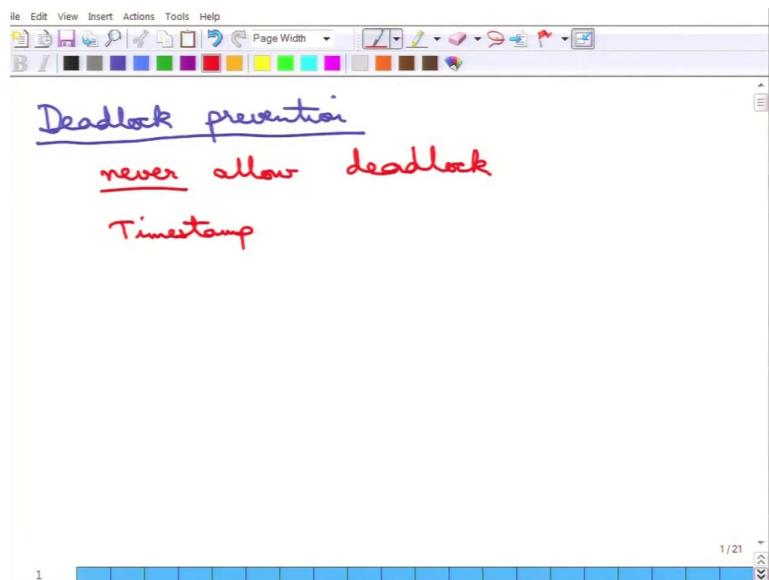
**Fundamentals of Database Systems**  
**Prof. Arnab Bhattacharya**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kanpur**

**Lecture - 43**  
**Concurrency Control: Deadlock Prevention and Deadlock Detection**

We will continue with the Concurrency Control protocols, as we have already seen many protocols, locks and timestamps based ordering protocol and so on, also the multiple granularity locking. But, if you have noticed in all of them, there is one recurring problem which is the deadlock. So, most protocols can fall into this trap of deadlock, so the basic time, the basic 2-PL protocol can fall in the trap of deadlock.

But, the timestamp ordering protocols cannot of course, but the locking protocols can always go into this deadlocks state. So, next what we will be studying is to see how to handle this deadlock.

(Refer Slide Time: 00:51)



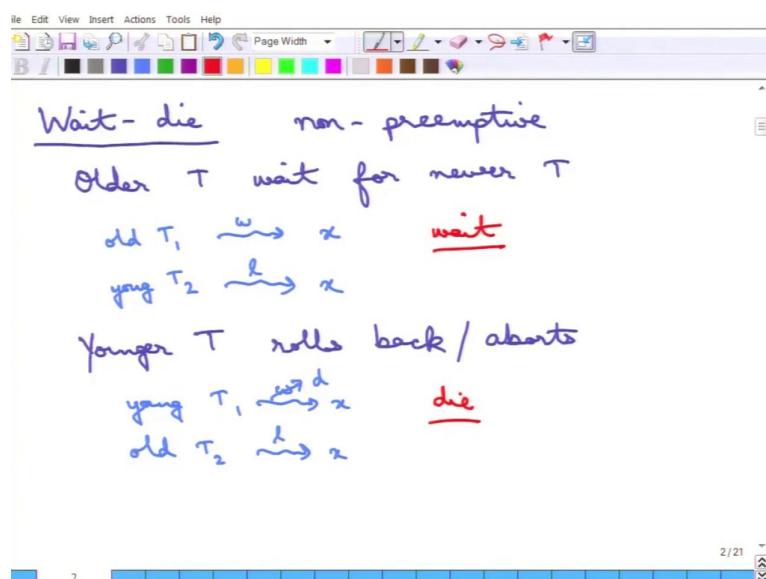
So, the first thing is the deadlock prevention that we will study. So, what I mean by deadlock prevention is that, this type of scheme, the deadlock prevention schemes never allow a system to enter into a deadlock. So, they never allow deadlock, so if these schemes are run, the system will never deadlock. So, we have already seen an example of this, the timestamp based ordering protocols are of course, deadlock prevention schemes.

So, the timestamp based, the protocols that use the timestamp, the basic timestamp ordering

protocol and the validation based protocol, they are we have seen that these are, this deadlock prevention scheme, because they never allow deadlock, then there are two as... So, what happens in that is that, so in this deadlock prevention schemes is essentially... In the timestamps based ordering protocol what happens is that, when a transaction or when an operation of a transaction appears out of order, then one of them must roll back.

If you remember in that read timestamp while we are checking the read timestamp and the write timestamp, there are two ways. We said either the transaction T that is trying to write it should roll back or the transaction  $T_1$  or  $T_2$  for which it is not correct should roll back. So, now,... So, which one should roll back that is a question. So, whether it is T or which is it is  $T_1$  or  $T_2$  that is the thing, so for that there are these two schemes.

(Refer Slide Time: 02:21)



So, the first one is called a wait die scheme. In a wait die scheme, this is also, this is a non preemptive scheme and we will see what does a non preemptive mean, but just remember for the time being that this is a non preemptive scheme. What happens is that, here the older transactions wait for newer transactions. So, suppose there is an old transaction, the  $T_1$  that wants a particular lock on an item x and  $T_2$  which is a younger transaction. So,  $T_1$  is old and  $T_2$  is young,  $T_1$  is currently holding that lock.

So, then the older transaction simply waits for the young one to finish, so the old waits that is fine. So, if the older comes after the younger and the  $T_2$  is already holding the lock and the

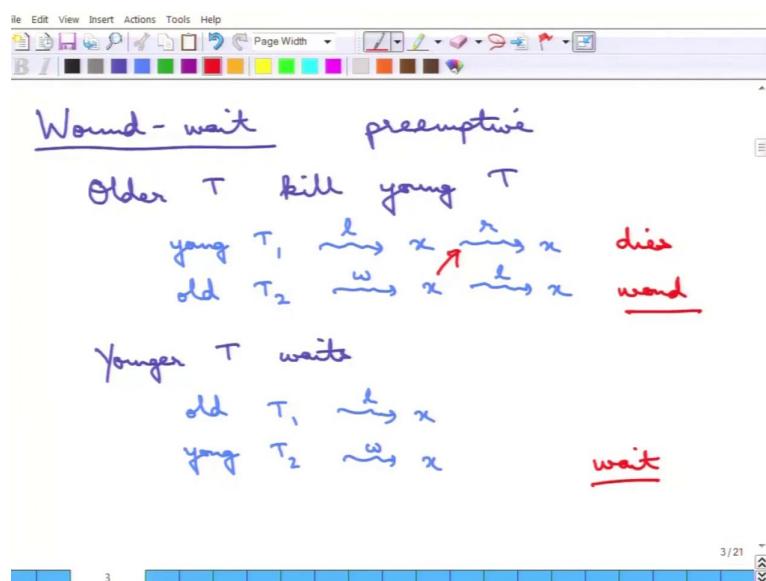
older one wants to it, it just waits. The younger ones; however, do not wait, if a younger transaction wants this thing wants a lock, they simply roll back. So; that means, the younger ones are do not wait for the older ones.

So, if the younger one comes and sees that an older one is already holding the lock, it simply dies, so that is why it is the thing. So, this is what is happening. So, the younger one, young one is holding, is waiting for a lock it does not wait for a lock, it essentially just dies. Because, an older transaction is currently holding the lock, so this is why it is dies. It just rolls back or aborts which is the same as abort so; that means, it is dies and this is why this is called a wait die scheme, so this is the thing.

And the younger ones as you can see can die many, many times, because what happens is that when a transaction rolls back or dies or aborts, whatever is the terminology, it restarts with a new timestamp. So, when a young one restarts, it restarts with of course, a younger timestamps, so the chance that it will die again is more. So, it can just keep on dying again and again and this is called non preemptive, because nobody removes any other transaction.

So, the old transaction simply waits for the young one to finish and the young ones simply dies. So, nobody preempts another transaction, so nobody hurriedly removes another transaction, so that is why this is called a wait die transaction.

(Refer Slide Time: 04:59)



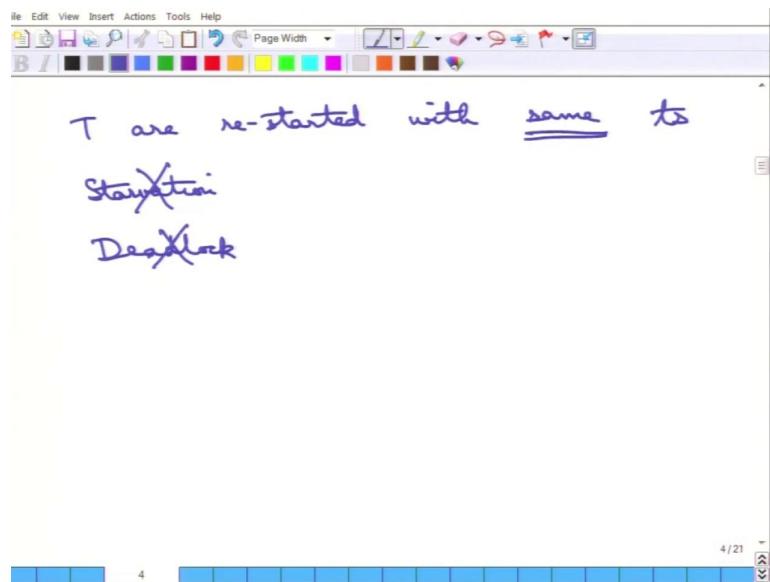
So, the other thing in this is the wound wait scheme, this is preemptive and again we will see

what does preemptive mean later. But, let us see what happens is that the older transactions kill newer ones, so this is interesting, so kill young transactions. Now, what does this mean? So; that means, that suppose there is a young transaction  $T_1$  which is holding the data item x and an old transaction  $T_2$  wants the data item x. So, at that point since the old transaction has come, the young is holding to it, the young is made to release the lock and the young is essentially it is, it dies.

So, the young one dies and the old one essentially now gets the lock and continues with this. So, the young one is made to die, so that is why it is the older transactions kill young ones that is the thing. And what happens to the younger ones? The younger ones simply waits, so an old transaction is holding the lock and the young transaction comes, it simply waits. So, the younger one is simply waiting, so in the terminology the old one is set to wound the young one. So, it stabs the young one, it kills the young one whatever is the thing that is why it calls and the younger one dies. So, this is it wounds it, so that is why it is called a wound wait protocol.

Now, this is pre emptive. Why is this pre emptive? Because, when an older one comes and it sees, it wants the lock which is held by young one, it pre empts the young one, it forcibly aborts the young one, it forcibly removes the lock from the, it forcibly makes the younger one, unlock it. So, it removes the named data item whatever the data item from the young one, it makes it, release it and it preemptively grabs the data item, even though the *old* ones holding it, so that is why this is a pre emptive data item. Now, transactions in this case what may happen is that, transactions may be started with the same timestamp.

(Refer Slide Time: 07:22)



So, transactions are restarted, it may be restarted with a younger timestamp, with a new timestamp or restarted with the same timestamp, so this is the other thing. Now, one can show that there is no starvation in this schemes, there is no starvation. Why is that? If the transactions are re-started with the same timestamp, why is that, is that at some pointing time, if the starvation... When the starvation happens is that, remember that when a transaction wants a data item and it cannot get it, the next time it wants it is, by the time some other transactions have got it and so on and so forth.

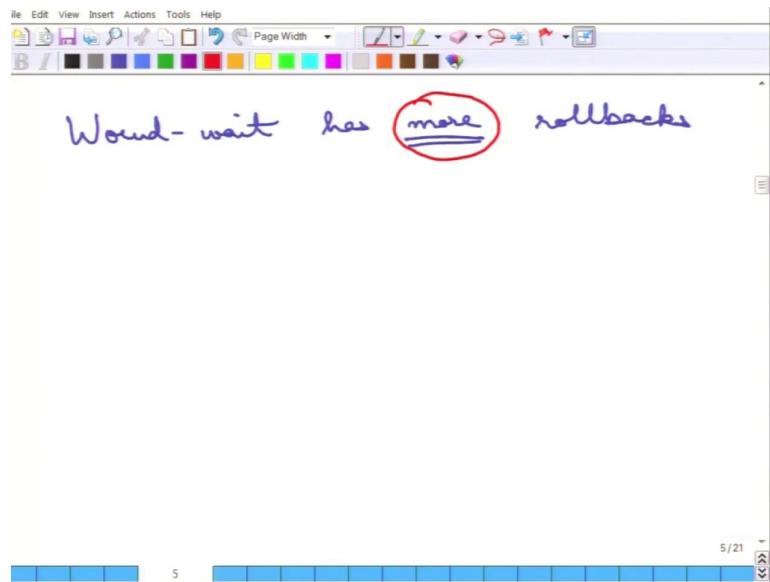
Now, if a transaction is restarted and it is restarted with the same data item, at some point in time this transaction must be the oldest one that is around. Because, every new one, every other one has got it and so on and so forth, every other older transaction has finished and this is what is being starved. So, then it will be getting it, it will wound all the young ones and it will get it, so this there will be no starvation.

On the other hand, the same thing what happens for the wait die is that, at some pointing time this transaction must be the oldest one. So, it must simply get it, because it will wait for the young ones to finish and another transaction which is also wants it will not be getting it, because it is younger than this, so this is the older one. So, and the end it will get it, so there is no starvation and as we have already understood that there will be no deadlocks. So, this is free from starvation and this is free from deadlocks.

Now, among these two things which is the better one as it happens that in the wait die

transaction the young ones keeps on dying. Because, every time it is restarted there is some old one that is being done. So, young one reaches to this place, but the old one is already holding the lock and it dies it restarts and it reaches the same place and it dies. So, it keeps on happening, so it dies many, many times.

(Refer Slide Time: 09:29)



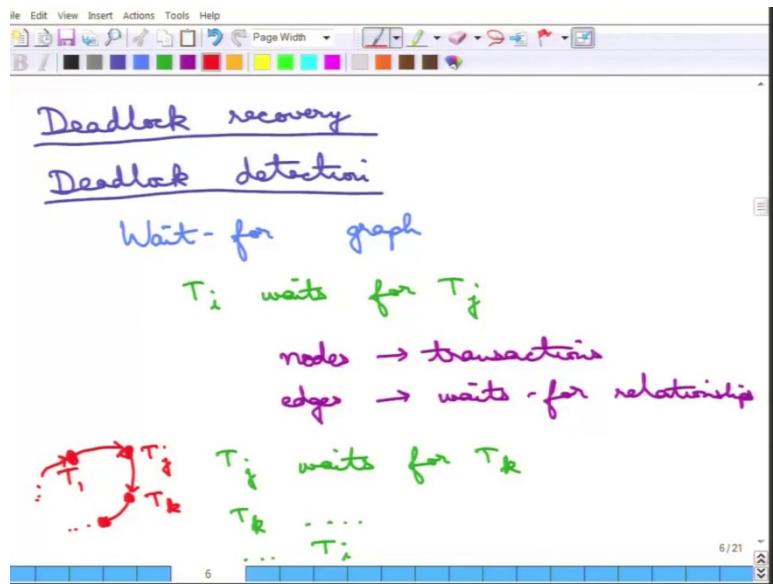
So, in that case essentially the wound wait has more roll backs. So, wound wait protocol, the wound wait scheme rather has more rollbacks than the wait die scheme. So, among the two things the wait die is preferred. So, between the two schemes the wait die and wound wait the question is which one is more preferable. Now, what happens in the wait die is that young transactions may die many times, now what does it happen is that when a young transaction reaches to a point where it wants a lock it looks for an old it sees that there is an old transaction that is holding to it, so it dies.

And it is restarted by the time it restarts reaches the same point it may be happening that the old transaction is again holding it, so it again dies. But, the chances of that the old transaction finishing is larger than in the wait die in the wound wait what happens is that the wound wait, the older transactions kill the younger ones. So, if a old transaction some very, very old transaction that was supposed to happen much earlier comes it will kill all the young ones.

So, even though the younger transactions die many more times in a wait die and older transaction may kill more transactions in a wound wait. So, in general wound wait has more roll backs. So, what is preferred is the wait die scheme? So, this is about the deadlock

prevention we can also talk something about the deadlock recovery.

(Refer Slide Time: 11:00)



So, deadlock prevention schemes will never go into deadlock, so there is no question of recovery. But, this locking schemes they do not they are not deadlock prevention schemes, so they may go into deadlock. Now, the question is once a system goes into deadlock it must be detected and it must be recovered. So, the system must not keep on waiting of course, it must be the deadlock must be broken. So, how is the way to be done?

So, the first thing to recover from a deadlock is to identify the deadlock. So, that is called the deadlock detection. So, how is a deadlock detected? This is very similar to the to some concept that we did earlier anyway. So, the concept that it uses is something called a wait for graph, so this is a wait for graph and we in the conflict serializability we saw graph, this is kind of the same thing and what happens is that in this wait for graph suppose  $T_i$  waits for  $T_j$  that what does this mean, the  $T_i$  waits for  $T_j$  is that  $T_j$  is holding a lock on an item that  $T_i$  wants.

So, then in the graph... So, the graph the nodes are transactions and the edges are according to this wait for things, so the edges are the waits for relationships. So, in the above case when  $T_i$  waits for  $T_j$  there will be a node  $T_i$  then it will have a directed edge to node  $T_j$ . Now, what happens in a deadlock is that  $T_i$  waits for  $T_j$  then  $T_j$  waits for  $T_k$ , then  $T_k$  waits for something else and finally, that something else waits for  $T_i$  that is why it is a deadlock.

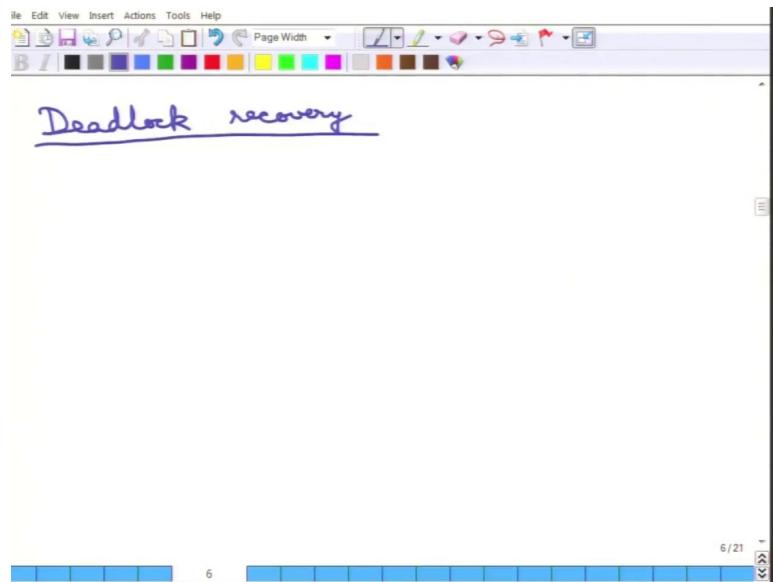
So, now, you can clearly see what is going to happen in this graph is that now  $T_j$  waits for  $T_k$  then  $T_k$  waits for something else, then something else, something else that waits for  $T_i$ . So, that is the... What does that mean is that, for this graph to say whether there is a deadlock or not, there must be a cycle in the graph. So, only when there is a cycle in the graph it can be detected that there is a deadlock. So, the how does one detect if there is a cycle in this directed graph that is the same as we did in earlier in the conflict serializability graph, etcetera.

So, one can run a depth first algorithm to find out if there is a graph, if there is a cycle then there is a deadlock. Because, in that cycle all the transactions are waiting in a circular manner, if there is no cycle on the other hand then there is no deadlock. So, it is a, this thing.

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

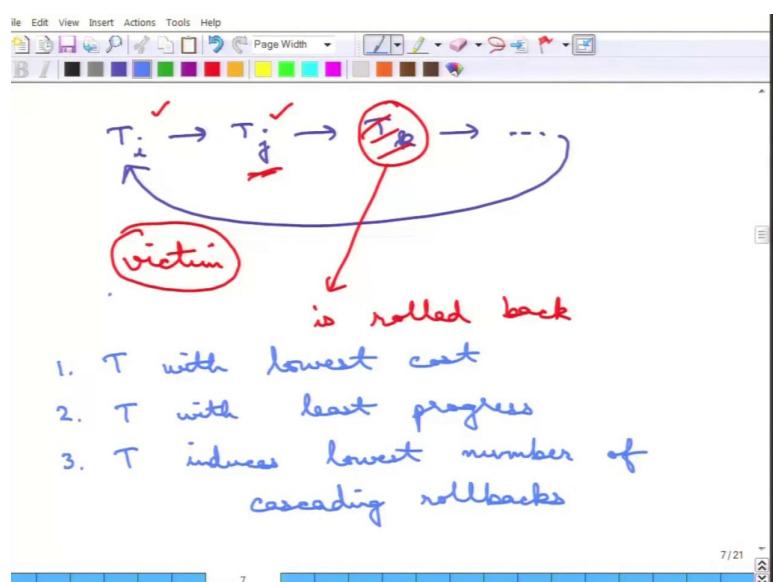
**Lecture - 44**  
**Concurrency Control: Deadlock Recovery and Update Operations**

(Refer Slide Time: 00:09)



The question now is, suppose the cycle has been detected and it now needs to be recovered.

(Refer Slide Time: 00:17)



So, how does one recover; that means, so here is the idea. So, cycle meaning  $T_i$  goes to  $T_j$ ,

$T_i$  is waiting for  $T_j$ , this  $T_i$ ,  $T_j$  waits for  $T_k$  and this dot, dot, dot and finally, that is waiting for  $T_i$ . Now, to break the deadlock one of them must be aborted, one of these transactions must be aborted, so one of them must be made a victim. So, suppose arbitrarily  $T_k$  is made the victim; that means,  $T_k$  is now rolled back, so all the resources that  $T_k$  holds is released, which means  $T_j$  can now pass; that means,  $T_j$  will finally, finish and; that means,  $T_i$  can then pass and so on and so forth, so this is the whole effect of this.

So, one of them must be chosen as the victim of this. Now the question is, which transaction is the victim? Now, there are many schemes to do that, but there are essentially some heuristics to this. So, the first one is the transaction that has the lowest cost is chosen as the victim. Now, what does that mean? What does it mean to say the lowest cost? Remember, that these transactions are essentially operations, the read, write of those kind of operations and each such read, write operations will have some cost, this is by the query processing engine, they will estimate some cost.

So, the transaction that has the lowest cost if that is roll back; that means, that the least amount of work is supposed to be redone. So, it essentially is that, it can finish off the most quickly, so if that is redone that is not a problem, so that is the one with the lowest cost; that is one heuristic. The other heuristic is that the transaction with the least progress. Now, what does that mean is that even though, we were in the first heuristic, even though the transaction with the lowest cost is being chosen, that may not be a good one, because even suppose the, it has the lowest cost, but it has almost completed.

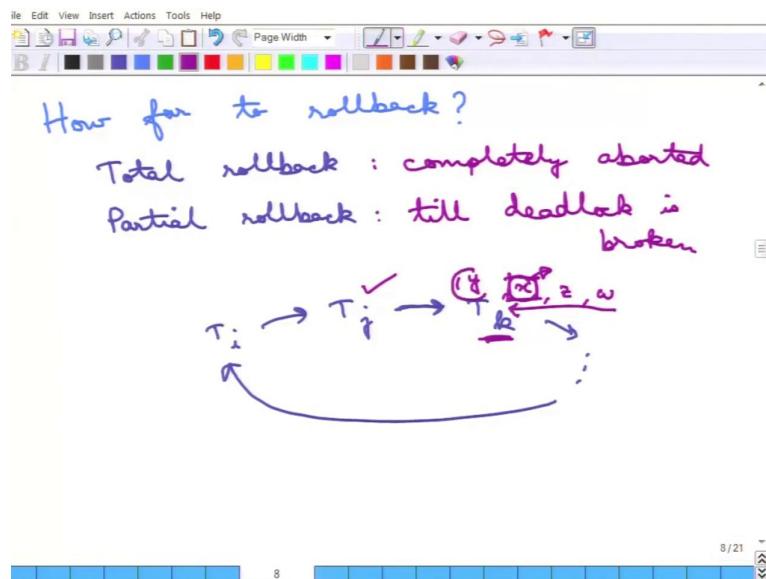
Now, if that is rolled back, then all that work needs to be done. In the second heuristic what is done is that, the transaction with the lowest progress is being chosen, lowest progress meaning, it has gone into the order of operations into the transaction the least amount. So, it has essentially done the least amount of work, so it has essentially completed the least amount of work. So, if that is being rolled back or that is being made the victim, then the least amount of work is being lost.

So, that is the rational of the second heuristic and the third heuristic is the one that induces the lowest number of cascading rollbacks. Now, this is should be very, very intuitive and this is very, very important to understand, that what happens in the case of a database is that if a transaction rolls back, it may have a series of cascading rollbacks. So, even though the

transaction may have the greatest progress or greatest cost, etcetera or if, so and if it has being and it has gone too much far etcetera and it has or the other way around it has not gone too much far etcetera, but if it rolls back, then it will cause a lot of cascading rollbacks.

So, essentially all those cascading rollbacks meaning all the transactions that are now roll back as the effect of this roll back will need to be redone, so that is a big cost. So, essentially the transaction, which has minimal effect on the other transaction in terms of cascading rollbacks is chosen. So, these are the three heuristics of how to choose the victim or how to choose the transaction to roll back.

(Refer Slide Time: 04:01)



The next question comes is how far to roll back, so the earlier question was, which transaction to roll back and then, the question is how far to roll back. Now, of course, intuitively one may say that, if a transaction is rolling back it must roll back to the complete beginning of it, the complete start of this, so that is called a total roll back. So, total roll back means the transaction, essentially all the operations done by the transaction are undone and the transaction essentially just restarts, it is completely aborted.

So, this is essentially, it saying that completely aborted, the transaction is completely aborted. What can also be done is something called a partial roll back can be done. Now, what is the idea of a partial roll back is that, may be not all the operations in the transaction are wrong, may be not all the operations in the transaction have caused the problem. So, it is rolled back, so only that for which there is a problem. The problem is in the sense of... Remember, that

we are not talking in the problem, in the sense of recoverability or etcetera, we are talking in the sense of deadlock.

So, roll back till deadlock is broken, now that is the interesting thing to do now. Now, let us just go back to the example that we did earlier, to understand what is being happened. So, suppose  $T_i$  waits on  $T_j$  that waits on  $T_k$  and that waits on so on and so forth, then this goes back.

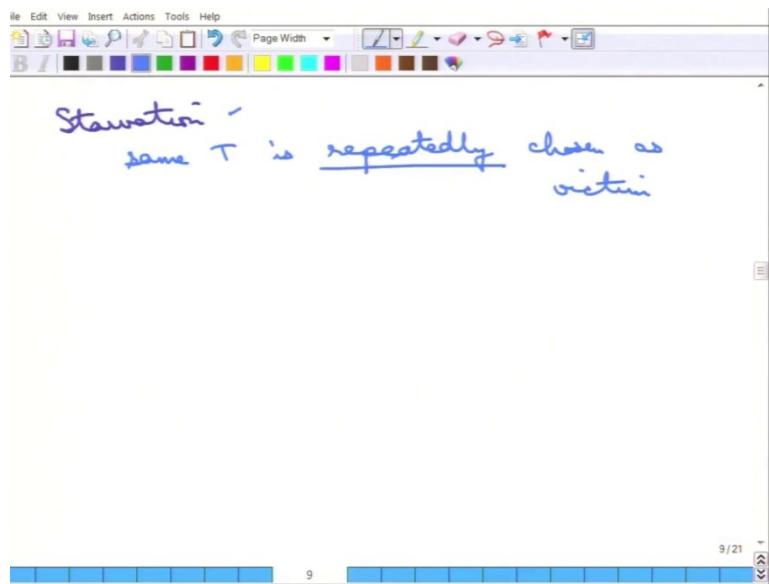
Now, suppose this  $T_i$  waits for  $T_j$  is being done, because of a data item x. Now, before that suppose  $T_k$  has held the data item y and suppose after that it held's the data item z, then it holds the data item w and so on and so forth.

Now, the problem is with this x, so  $T_k$  is a victim, now if x is released, then  $T_j$  can succeed and then, once  $T_j$  succeeds  $T_i$  succeeds and the deadlock is broken. So, the point is it will be a rollback only till the point of the problematic data item due to, which  $T_j$  is rolled due to which  $T_j$  is waiting. So, why this operations that involve y, the data item y did not be rolled back? Because, y is not causing any problem, so it rolls back only to the point, where there is a problem and then it is released.

But, this requires a lot of schematic analysis and the thing is that, it has to be known that, which is the data item it can hold, which is this waiting for which is of course known, but it must be also known that this partial roll backs can be problematic. Because, even though it can be said that all the operations of y may not be rolled back, but there may be some operations of y, that affect the later operations of z and w and so on and so forth, so all of these must be taken care off.

In general, by keeping a count of which operation comes after which operation, which the transaction they anyway keep at it, then it can be rolled back till the point, where the deadlock is broken, so that is the thing about how far to roll back.

(Refer Slide Time: 07:03)



Then, the question is the other thing is that starvation in this deadlock recovery schemes, the starvation may happen, because the same transaction is repeatedly chosen as the victim. So, that is why this starvation may happen. So, repeatedly chosen as victim, this may happen, because this transaction is doing the least progress or transaction has a lowest cost, etcetera, this is repeatedly chosen as the victim, so start starvation, so it may go into this thing.

So, one interesting way or one way of breaking this problem of starvation is that when a transaction is chosen as a victim, a count is kept. So, a victim count is kept for each transaction and to break the starvation problem, there may be a threshold number of times, when a transaction is a victim, so suppose it is three. So, once a transaction is chosen victim for three times, it will not be chosen as a victim for the next time.

So, it can be starved for three times, but not any further, so that is to just break the thing about starvation. So, that is this all the things about this locks and the deadlock recovery, etcetera.

(Refer Slide Time: 08:16)

Insertion and Deletion

- insert ( $x$ ) } write ( $x$ )  
- delete ( $x$ )

Errors:

- $n(x), w(x)$  before insert ( $x$ )
- $n(x), w(x)$  after insert ( $x$ )
- $n(x), w(x)$  before delete ( $x$ )
- $n(x), w(x)$  after delete ( $x$ )
- insert ( $x$ ) after delete ( $x$ )

write ( $x$ )

Now, there is one thing that is left for the concurrency control protocols is we have been talking in the concurrency control protocols all along about read and write. Now, there are two more important operations in databases that are quite frequent actually in transactions, that we have not talked about at all, these are insertion and deletion. So, what happens when a new tuple is inserted into a table or when a tuple is deleted from a table or some attributes are updated, etcetera? Remember, that updates can be handled always as a deletion and then, an insertion.

So, what we will talk about here is the insertion and deletion, so insertion and deletion, so what happens in the case of insertion and deletion. So, for a data item  $x$  there are two operations, that we need to worry about, which is the insert( $x$ ) and a delete( $x$ ). So, insert( $x$ ) meaning this  $x$  is newly inserted into whatever place it holds to and delete( $x$ ) is from ((Refer Time: 09:18)). So, let us summarize the logical errors is that, so the errors that can happen due to insertion and deletion the logical errors that can happen is that, there can be a read of  $x$  or write of  $x$  before insert has taken place.

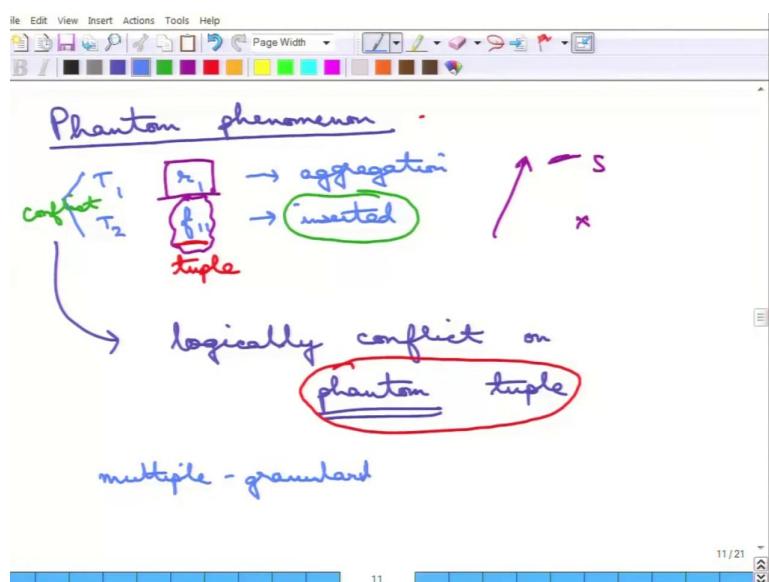
So, one transaction wants to read to an  $x$  or write to an  $x$ , but which has the other transaction, which is supposed to insert it has not yet done, so that is of course, an error. Then, the read  $x$  and write  $x$  may be happening after the delete of  $x$ , so one transaction to suppose to read or write this was late and the another transaction came and by the time this read and writes came for this transaction another transaction came and deleted it, so that is also an error.

Then, there are other kinds of errors, which are happening, because these are not idempotent operation. So, once the data item is deleted it cannot be deleted again, so the deletion operation is not idempotent; that is error actually the same for insertion. So, once a data item is inserted another transaction cannot issue another insertion operation, because it is already in the place already in the record or database etcetera, so these are the problems with insertion and deletions.

Now, note that everything can be very simply handled by assuming that read insert(x) and delete(x) by assuming that insert(x) and delete(x) at similar to write x. So, all the conflict that happen with write is supposed to be happening with insert and delete; that means, any insertion is a conflict with write any insertion is a conflict with another insertion any insertion is a conflict with the delete. So, one thing that I have left out is that delete(x) is a problem before insert(x), so once something is supposed to delete it, but it did it before, so that is a problem as well.

So, if insertion and deletion are treated like write x and all the conflicts, that write x has is propagated to insertion and deletion and then, the lock compatibility matrix and then, the conflicts and everything is handled in the same manner as the write x, then everything will be correct. So, that is the case with insertion and deletion. So, that is the way to handle insertion and deletion.

(Refer Slide Time: 11:41)



To finally, finish the concurrency control protocol we will talk about one interesting

phenomenon, which is called a phantom phenomenon, so a phantom phenomenon is the following is that. Suppose there is a transaction  $T_1$ ; that is reading a relation  $r_1$  for, let us say competing an aggregation or something like that ((Refer Time: 12:02)), so this is the this thing.

And then, in the meanwhile there is a transaction  $T_2$ , which comes and updates some field units some in, so this is inserted. So, this some field is inserted or some tuple is inserted and so on, so forth into the thing. So, then  $T_1$  and  $T_2$  of course, as we can see this is a conflict this is of course, a conflict, because the aggregation will be wrong if it is allowed to insert. But, note that strangely enough this is hard to detect, because where is the insertion happening the insertion is not happening at the relation  $r_1$  the insertion is happening inside a tuple or inside a field or this is a new tuple; that is being inserted.

So, this is a new tuple, so just to highlight this is a tuple, so this is a new tuple; that is being inserted, that tuple of course, must not locked by  $T_1$  was there was no tuple earlier. So, when  $r_1$  is locked even in the is mode and all of these things everything that is locked below cannot lock this  $f_{11}$ , because  $f_{11}$  does not exists at all this was this tuple did not exists, so this is being inserted.

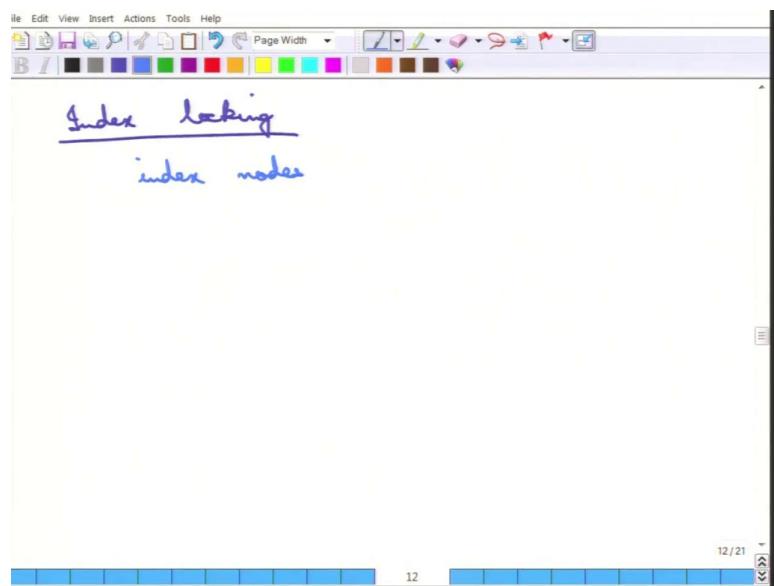
So, how to handle this is why it is called a phantom phenomenon, because when  $r_1$  is being locked they do not lock this, but, they logically conflict on something called a phantom tuple, so this is logically logically conflict on a phantom tuple. Why is this phantom? Because  $r_1$  does not know the existence of it, so for  $r_1$  this  $f_{11}$  is a phantom tuple  $f_{11}$  does not even exist for transaction one, so for  $r_1$  with for transaction.

So, this is a phantom tuple; that is why this is one of the hardest problems in databases to catch. So, the insertion one of thing is hard actually this phantom phenomenon, but, so this is known as a phantom tuple this  $f_{11}$  and this whole phenomenon this is known as the phantom phenomenon. So, that is the thing of this, so exclusive lock know how to solve this essentially is that one must get an exclusive lock on this relation r. So, this is that exclusive lock; that is being brought, so whenever this is being read etcetera this exclusive lock.

So, the multi granularity locking actually can solve it, whenever this thing is trying to insert it checks all the way up its parents and see that  $r_1$  is locked in S mode, because this is being

done, so this cannot be inserted. So, insertion essentially an x lock, but this is already locked in a shared mode, so it cannot get an exclusive lock. So, the multiple granularity locking actually solves it, so the solution is a multiple granularity locking that we saw earlier, but this is just to highlight that this is one of the very hard problems, so interesting problems to catch, so this is the phantom phenomenon.

(Refer Slide Time: 15:10)



And just one thing to end the concurrency control protocol, we have been talking about the databases relation etcetera and all of these things, but the databases also use the index structures, so the index locking must also be taken care of. So, the varieties of things in the index must be locked. So, the index nodes may need to be locked and the index nodes locking the index nodes essentially meaning locking the data items that it the index node corresponds to etcetera.

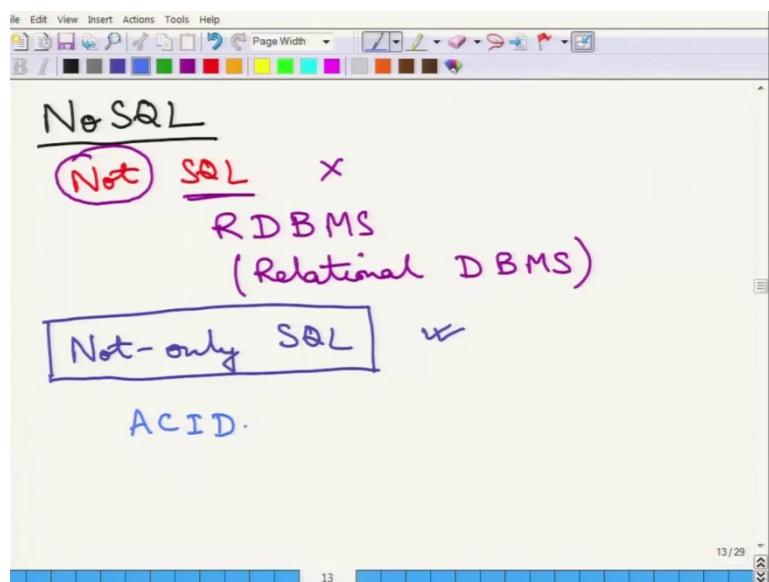
So, this may also helps to avoid the phantom phenomenon, because the insertion etcetera is always happening through the index lock and the index through the index node and the index is already locked, because it will touch that particular tuple. So, that finishes the module on concurrency control we will next go over no SQL and big data.

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

**Lecture - 45**  
**NoSQL: Introduction and Properties**

Welcome, we will start on a new module today which is on NoSQL. So, NoSQL and big data at the last two modules that we will cover as part of this course. Note that these are new additions to the undergraduate database courses and these are actually, because these are new developments in the database paradigm.

(Refer Slide Time: 00:32)



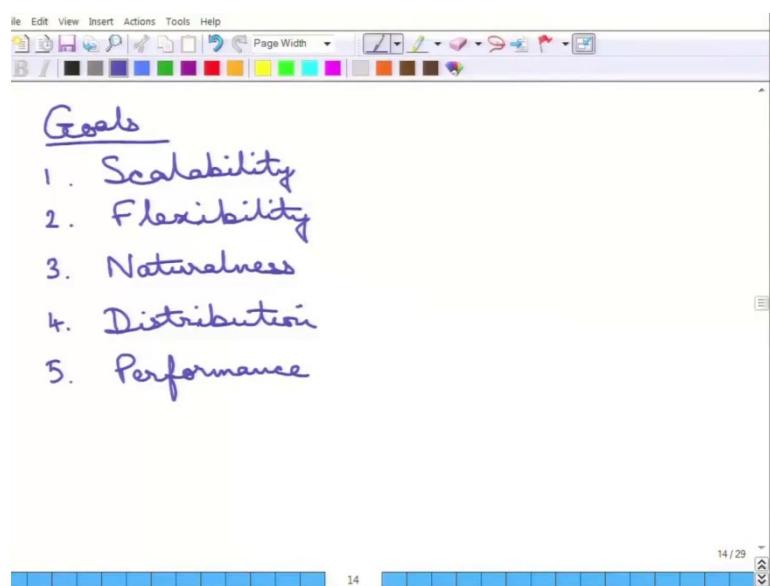
So, we will start off with NoSQL, this is NoSQL. So, as you can probably guess the name NoSQL is that, why is it called NoSQL and what does NoSQL mean actually. So, SQL stands for the Relational Database Management System, so the RDBMS. So, SQL is a quintessential programming tool to handle relational databases and the NoSQL started off by saying that this is we will not use SQL. So, it says it is not SQL, it starts off saying not SQL, because SQL essentially stands for the Relational Database Management System.

So, this is the RDBMS that is the term. So, it is essentially relational, this is the relational model that we have been studying so far in the beginning of the modules, we have been studying the relational algebra, the relational model, etcetera. So, it started off as a not SQL; however, it is no more a not SQL any further, because the... So, the originally the NoSQL

model started the NoSQL databases started by saying that we will not use relational database model, we will not use SQL, we will not use the properties of this ACID properties, the Atomicity, Consistency, the Isolation and Durability this we will not use, we will use something different.

Then, the database researchers and the database community all over the world realize that the RDBMS is just too powerful to ignore. So, then it switched on to something which is now essentially called not only SQL. So, note that NoSQL stands now for not only SQL, it is not SQL, so this is the correct term for this thing. So, it does not restrict itself to using only SQL or rather SQL here stands for the relational database management system, it does not use only SQL or the RDBMS it uses something more on top of that. So, essentially the idea is to get out of acid properties, so not to follow acid properties on that, but to follow something that is not just acid property.

(Refer Slide Time: 02:53)



So, the goals of this NoSQL system whatever it, the goals of NoSQL systems is can be generically summarized as the follows. So, the first one is scalability, so scalability, now one important thing about scalability is that, the database says the relational databases are very scalable, they can go up to large amounts of data, etcetera, etcetera. But, the still there is a problem with scalability when you think of a very large project.

For example, the entire Google web pages or the entire facebook graph or things like that when it is really, really large, the relational database models may or may not scale. So, all the

searching, etcetera they may or may not scale, all the transactions, the schedules, etcetera may or may not run well. So, the scalability essentially the volume of data that it tries to handle. So, that is one of the goals is to make the system as much scalable as possible, so you keep on adding data without much trouble.

The next is flexibility, so flexibility... So, what does the flexibility mean is that the relational database models part into a particular schema and that schema must be very rigidly followed. So, if there is a student table which requires an id, which is a non null value and a roll number and name, etcetera it must have all of those things, so that is not flexible. So, this NoSQL paradigm, the NoSQL databases wanted to make it flexible. So, it can have different kinds of students for example or different kinds of attributes for a student.

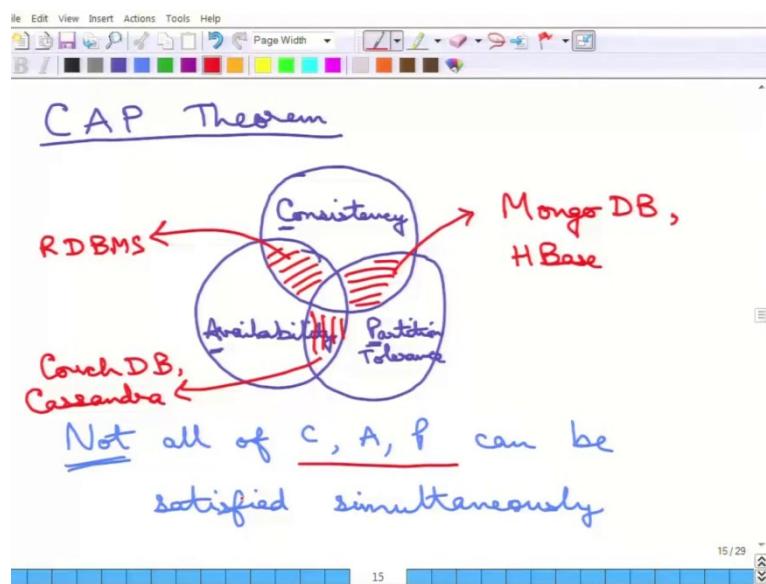
And now if you think of these social networks of the face book, etcetera, not every people will have lots of friends, not every people will upload the photos, not every people will have blogs, etcetera, but it is flexible. So, every people, every person is an identity, is an entity, but it can be flexible. So, that is the other goal or flexibility, then it is naturalness, so again naturalness what it means is that sometimes just to fit it into the relational model, things have to be broken up into this the relational schema in a slightly unnatural way, I mean this is not too much of a concern for RDBMS, in generally they are quite natural.

But, sometimes if you saw that bank account thing, the depositor, etcetera then they may not be natural. So, they can have this account and loan number and all of these things together, so that is the naturalness, so this... So, the goals of NoSQL system is to make it much more natural, so that is the thing. Then, the fourth point is extremely important, this is called distribution. So, distribution essentially means is that no matter how larger my SQL system or oracle or whatever, the sum the relational database systems can handle, they are essentially rooted to one machine.

So, they the essentially the entire data lies in one machine, one big disk, whatever you can get it, but there are mostly not handling the distributed systems. So, distributed systems are not generally handled very well, very elegantly by this SQL systems. So, the distribution essentially is to say that the data can be distributed across different machines, it is a normal distributed system paradigm and that has to come naturally within the database system, it is not has to be imposed on top of it, it should be part of the database design itself, so that is the distribution.

And the last thing of course, is that it has to still give it a good performance. So, the system has to still perform very well in a sense that if I want to update something in my face book account or something, it should be quick, it should be almost immediate or my other friends should be able to see it and so on and so forth. And somebody writes a comment I should be getting it and so on and so forth. So, these are the goals of the NoSQL systems.

(Refer Slide Time: 06:56)



And; however, very, very interestingly there is something in this space which is called a cap theorem. So, a cap theorem essentially says that there are three things, so first is consistency, there are three properties of a system that is very desirable, consistency. The second is availability and the third one is partition tolerance, so partition. So, how much it can tolerate partition, so that is the distributed net essentially. So, this is why this is called a cap theorem. Because, this is C, A and P, so this is why this is the cap theorem.

So, essentially we want the ideal system should have all these three properties, but what the cap theorem essentially says is that not all of... So, not all of C, A, P can be satisfied simultaneously. So, not all the three properties can be satisfied simultaneously, so that is the cap theorem. Now, very, very interestingly we have all learnt that what is the theorem, the theorem must be rigorously proved etcetera. However, the cap theorem although it is called a cap theorem, there is no proof for it, it is just a conjecture that there cannot be any system that has got all of these things.

So, note very importantly that although this is called a theorem, there is no proof it is just a

conjecture that is widely believed to be true and accepted to be true, but there is no formal proof that there cannot be a system, where all these three properties can be satisfied. Anyway, so there are systems on the other hand that can satisfy two of it at one time. So, for example, if you take this space here which is that something which satisfies consistency and availability, but not partition tolerance, this is our traditional relational database management systems.

So, before that what does consistency mean is that... So, consistency we have been studying consistency for these transactions etcetera is that things must be consistent. So, this is one of the acid properties that for example, this account balance or sum of account balance should be, the total should be the same before and after the transaction and so on and so forth this is consistency.

Availability is that the database is always assumed to be available. Because, even if it crashes it comes up, it recovers there is this whole recovery protocols and then as if that nothing has happened it to behave as if has nothing has happened and then new transactions can behave that it is again available that is from this thing. So, RDBMS's shared this properties of consistency or rather exhibit the properties of consistency and availability, but not partition tolerance. It is not very good for distribution that is the thing that is the biggest criticism of RDBMS. So, that is why it is consistent and available, but not partition tolerance.

On the other end, we can have systems here, which is let us say consistent and partition tolerance. So, it can maintain consistency and it can be quite nice to distributor, this kind of systems. Some examples are mongo DB, you may have heard the names of this, this is mongo DB and H base. So, these are certain things which are consistent and partition tolerance, but they are not guaranteed to be available all the time. So, some data items in mongo DB or H base may not be available. So, what essentially happens is that?

So, H Base let us take an example of H Base, there are different tables that go to different machines very simply put. Now, what happens is that some data item has been inserted into a table which is in machine a. Now, if machine b wants to query it is not immediately available it may or may not return it, because it is in some other machine, so it will take some time to update and so on so forth. So, it may not be available in that sense and machine a may be down.

So, machine a maybe may have this is because this is a distributed system machine a may

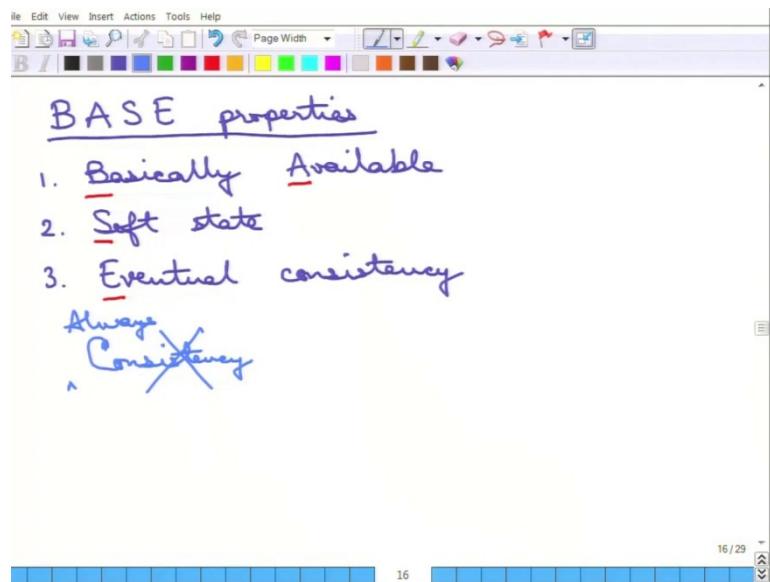
have some faults etcetera. So, it is not available till it comes up and so on so forth so, that is a mongo DB and H base. On the other hand there is this other kinds of systems which is this availability and partition tolerance and these kind of systems, the examples are your couch DB and the probably a bigger example is Cassandra.

So, these systems are available they will always make the data available, because they will essentially what they will do couch DB and Cassandra they will replicate the data. So, they will ensure that if machine a gets a copy, then machine b also gets the copy and so on so forth. Of course, they are partition tolerant they nice scale very nicely with distributed system etcetera, but they may not be consistent. Why are they not consistent? Because, suppose the copy in machine has been updated.

Now, the copy at machine b does not still know about the update in the machine a. So, it is not consistent, because the states now vary between machine a and machine b and somebody which who queries the data may get it from machine b in a inconsistent state. Because, the consistency is maintained or the updates is maintained in machine a, but machine b has not yet got it, so that is the problem with all of these things.

So, every of this systems has suffered from one lack of one of the properties, so that is the famous cap theorem. So, essentially this is what the cap theorem says that not all of this just to repeat that not all of CAP can be satisfied simultaneously. So, then let us move on to the next part of NoSQL is that. So, NoSQL started off by saying that they do not care about acid properties.

(Refer Slide Time: 12:49)



So, what they do care about is something called the base properties. Now, you may see that the name of the properties base is essentially to counter acid and we will see what the properties are, there are three base properties, the first one is basically available, so it is saying that the this is basically available. Now, what does it basically available is that the system does guarantee availability, but not right then I mean may be it is available after some time etcetera, etcetera, but it is not right then, but it is basically available, it is mostly available that is the first property, so that is basically available.

The second property is called soft state. Now, what is soft state? The soft state is the following, so the state of the system is supposed said to be in a soft state I mean it is said to be in a soft state what it means is that the state must change without any input. So, one important thing about RDBMS is that suppose there is an RDBMS that is around and without any new query coming, without any new insertion, deletion etcetera coming the state of the database does not change.

However, for this NoSQL system the state may change, the state may change because it needs to make it available, it needs to make it consistent so on so forth. So, it just it is synchronizes with other machines, other distributed setups etcetera, etcetera. So, that is why the state may be changed, so what it means is essentially is that it is just in a soft state. So, the state may change without apparently any input coming, without apparently any querying done that is why it is called in a soft state.

And the third and the last property is eventual consistency. So, eventual consistency these are the... So, eventual consistency is that data will be eventually consistent without any interim perturbation, so what does that mean is data will be... So, at some point in time data may not be consistent, because as I said that machine a updates and machine b does not get it. But, eventually after some time machine a will translate or will transfer this update to machine b will let the machine b know inform it about the updates, so then machine b will also be updated.

So, it will be eventually consistent without I mean, so without any perturbation in the sense that you do not need to give any more inputs to machine b to say that get go and get yourself updated etcetera machine a will finally, update it. So, it is eventually consistent, but not so one important thing to remember is that this NoSQL systems are not always consistent, they are eventually consistent. So, at some point in time if you take a snapshot, it may not be consistent.

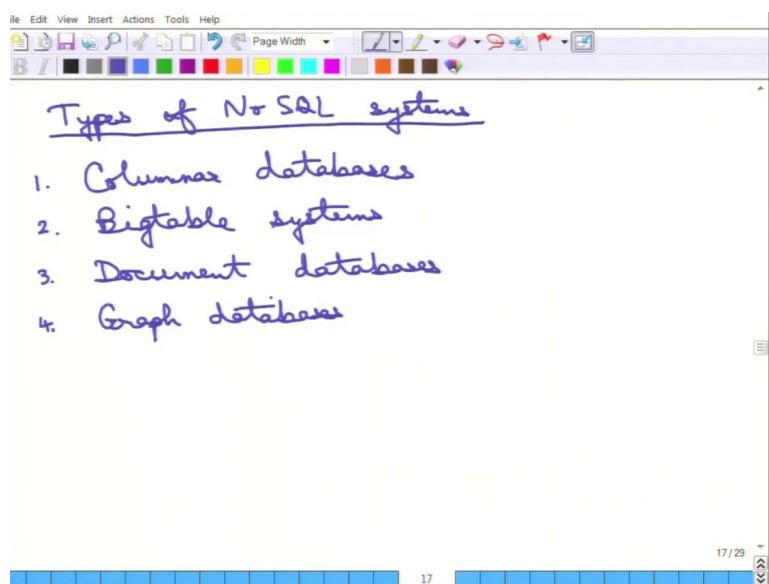
So, why is this called base properties? So, these are the things it is basically available soft state and E that is why it is called base. Now, this you see is just to counter the acid properties that is the thing. So, very, very importantly what it does is that the consistency as we understand in a database system is that it is consistency always, so it is not always consistent. So, always consistency is sacrificed, so it is only eventually consistent. So, this is just to counter the acid properties.

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

**Lecture -46**  
**NoSQL: Columnar Families**

What are the types? There are different types of NoSQL system.

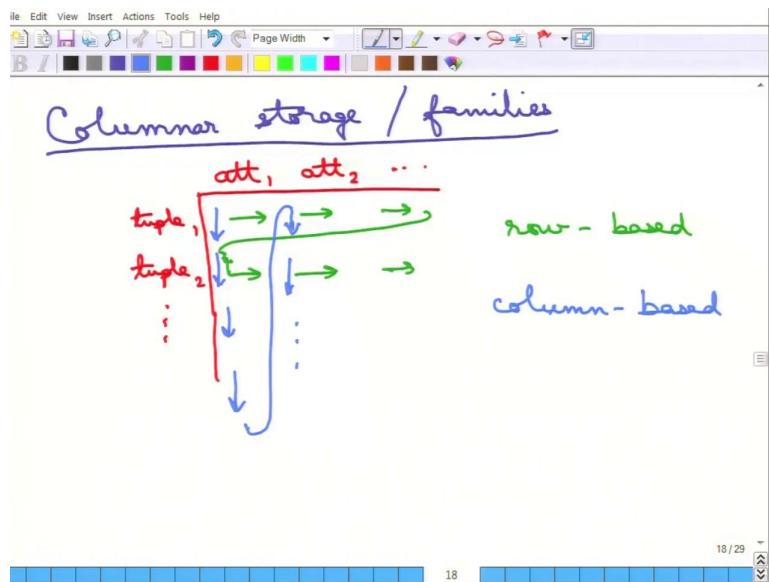
(Refer Slide Time: 00:15)



So, there are four main types of NoSQL systems. So, we will just briefly go over them, note that the whole issue of NoSQL systems may be another half course or whatever I mean, it can be very large, but... There are four main types of NoSQL systems. The first one is the columnar database family, so columnar databases, the second is the big table systems, the third one is the document databases. You may or may not have heard the names of this.

So, these are nowadays becoming very, very popular and so, may have heard the names of these things anyway and the fourth one is graph databases. So, let us go over each one of them one by one.

(Refer Slide Time: 01:17)



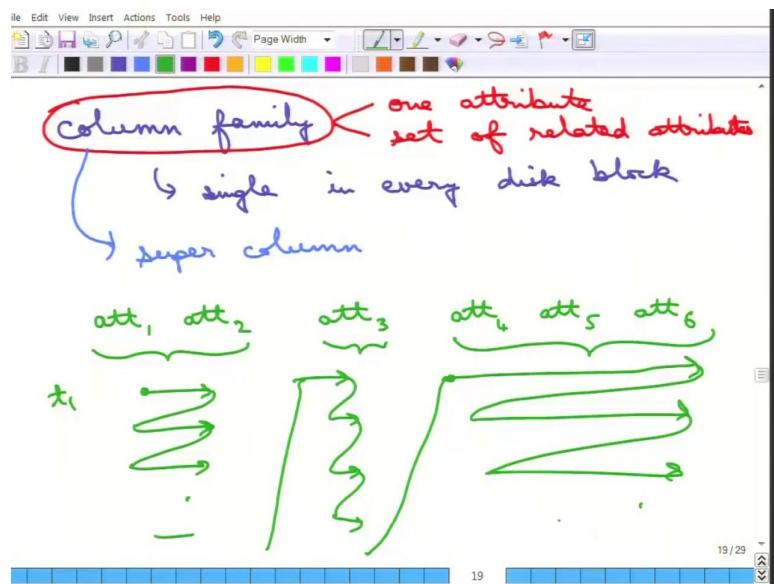
So, the first one is columnar storage, so columnar storage or columnar family let us just say columnar storage or columnar families or columnar databases, etcetera. So, one important thing is that, how is data stored in DBMS is they are stored in a row manner. So, what does it say to store in a row manner is that, think of a typical relational table. So, there is tuple 1, then tuple 1 has attribute 1, then attribute 2, attribute 3 and so on and so forth and after that it is tuple 2.

So, how is it actually stored in the disk? Disk is essentially, I mean the one dimensional right, because logically it is stored in tracks, sectors, one sector after another and so on and so forth. So, what it done is that, each tuple is taken, all its attributes are stored and then, the next tuple's attributes are stored and then, the third tuple and so on and so forth, so that is called the row wise storage. So, to highlight, what I am trying to say is, this is a normal database table, this is all the tuples, so this is tuple 1, tuple 2 and so on and so forth and while, this is attribute 1, attribute 2 and so on and so forth.

So, in a row based storage, what is being done is that, this is stored, then this is stored then this is stored, then this is stored, then this is stored, then this is stored, so that is, so this is called a row based storage. On the other hand, what is being done for the columnar families is that, this is stored, then this is stored, then this is stored and so on and so forth. So, all everything up to this is done, then attributes 2 is stored. So, what is this is being done is that, this is the column based storage.

So, in column based storage, what is essentially done is that, the first attribute of all the tuples are stored. So, the attribute 1 is taken, it is value for tuple 1 that is stored, then for tuple 2 it is stored, then tuple 3 and so on and so forth and up to all the tuples. Then, the attribute 2's values are stored, tuple 1, tuple 2, so on and so forth. So, it is stored in an attribute wise manner; that is the column storage; that is why it is called a columnar family or column based storage or columnar family, columnar databases, etcetera.

(Refer Slide Time: 03:42)



So, essentially this is the idea of storage and there are couple of things is that, so a single disk block or this is a single, what is called a column family, there is a concept of a column family. So, a column family is that instead of storing it one column at a time, you can group certain columns together and that can be stored. So, a column family this is stored in a single I mean, so every disk block contains only a single column family, so single column family in every disk block.

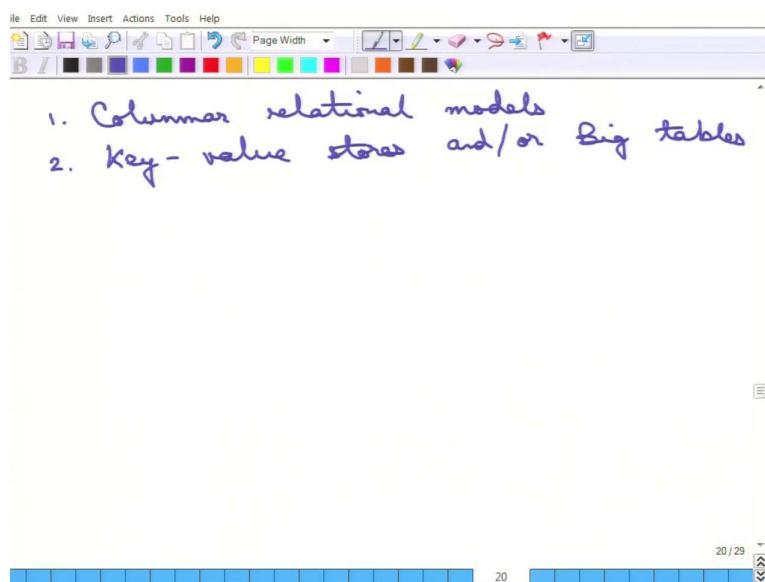
So, every disk block stores only a single column family. So, if it is a new column family it is a next block, it is stored and so on and so forth; that is the, this is the concept of the column family. So, either a column or a set of related columns together, so this can be either one attribute or set of related attributes, so that is called a column family. So, then this set of columns is sometimes also called a super column, so the column family storage this is set of things is also called a super column, so that is the super column.

So, essentially suppose the example is that attribute 1 and attribute 2 is pertaining to one

super family, then attribute 3 by itself is one super family, then attribute 4, attribute 5, attribute 6 is another super family. Then, what happens is that this is stored, so this is tuple 1 this is stored, then this is stored, then this is stored and so on and so forth. And after that is finished a new disk block starts, how many disk blocks it takes and then, this is stored, then this is stored, then this is stored and so on and so forth.

Then that is finished, then this is stored, because this is all part of the same column families, so then all the attributes are stored one after another, then this is stored and then, this is stored and so on and so forth. So, that is the idea of a column family or a super column.

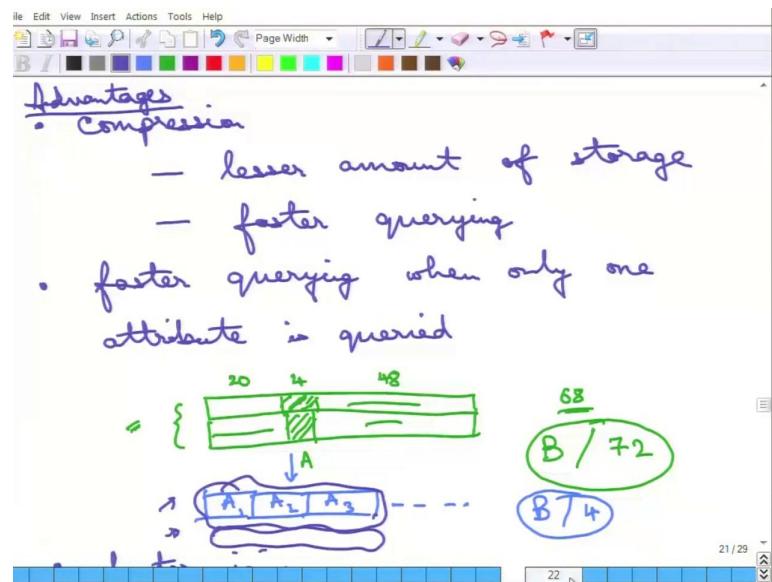
(Refer Slide Time: 05:54)



Now, out of these columnar families there are two main types of things, so first is the, so this is the columnar relational models. So, there are essentially two main types of things, the first is the columnar relational model and the second is the key value stores, also sometimes these are also or the big tables, what are called the big tables. So, very, very importantly, whenever we say column storage, so even the column storage is considered to be part of NoSQL, note that column storages can be relational models.

So, that is why it is called not only SQL, because these are part of these relational models or also NoSQL. So, columnar storages just by itself does not mean, just the fact; that is NoSQL does not mean it is not RDBMS, columnar storages can be RDBMS, so these can be relational models as well. So, this is essentially, what we have been just studying about, so this is RDBMS and it is stored one after another.

(Refer Slide Time: 07:05)



Now, couple of things about columnar relational models is the following is that, see when attributes are stored, when these are stored in an attribute wise manner, it allows compression of attributes. Now, what does it say means to say it allows compression of attributes? Suppose, there are many tuples which share the same attribute, suppose there are 100 such tuples, which share the same attribute.

So, what can be done is that, only one couple of that attribute value can be stored and it can be said that this is shared by tuples, whose ids are from whatever 900 or some 1 to 100 or 213 to 312 and so on and so forth, so that will allow compression. Now, compression saves two things, first of all it takes lesser amount of storage, because it is here of course, compressing it that is the thing. Also very, very importantly, this can allow faster querying. Why will it allow faster querying? There are two reasons, why it will allow faster querying.

So, first of all when... So, suppose you may need to find all tuples whose value is greater than something, etcetera. So, now, you go one tuple and instead of going over all the tuples you go to that value and you skip over that 100 tuples all together that is one way of doing it, that is the one good thing about compression. Even, if this is not compressed, so these are the gains of this thing, even if it is not compressed, faster querying can be allowed, faster querying is also true when only, I mean when only one attribute is queried.

So, what does this mean, so this needs to be understood in a little bit manner. So, suppose there is a selection query, which says select everything from the relation, where A greater

than 4. What is the basic manner of doing it is that, it goes over the tuples, then it goes to the attribute of A, checks it and if it is correct it says it is correct, it is output otherwise it is not, then it goes to the next tuple, checks the attribute of A and so on and so forth.

Now, how much, so it is essentially going over the disk is essentially going over all the other attributes, which are not A and skipping it is not checking it, but it is actually going over. Now, in the disk, what does it mean to go over it is actually taking more blocks to study more blocks to query, so this is what I am trying to say. So, suppose this one row this is attribute A that I am interested in this takes let us say 4 bytes and suppose there is another 20 bytes here and another some 48 bytes here.

So, to go from attribute A to here you are essentially wasting a going over 68 bytes this  $48 + 20$  only, then you are reaching the next A. So, in a disk block, how many such tuples can be stored or how many such tuples, where only A search can be stored this is essentially

$\frac{B}{20 + 4 + 48}$

the disk block size  $\frac{B}{20 + 4 + 48}$ , so that many number of tuples can be stored. So, that many number of attributes A can be searched by accessing one disk block.

However, if this is stored in a columnar manner, then what is being done, is essentially attribute A of tuple 1 is stored, then attribute 2 of tuple 1 is stored and so on, so forth, this is all four blocks and so on, so forth these are all stored together. So, then how many such

$\frac{B}{4}$

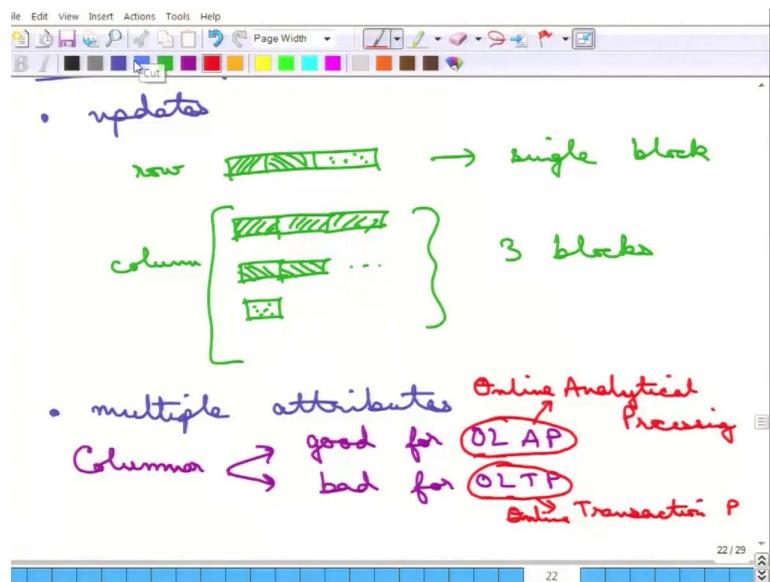
attribute AS can be stored in 1 block this is  $\frac{B}{4}$ . So, you see that is a larger number much larger number, so it is a larger number and, so it is quite beneficial if it is stored by attributes, because just by one disk block access more tuples and more attributes or more, more attributes and more tuples can be searched.

So, the single attribute in more tuples can be searched, so that is the, so more tuples can be decisions about more tuples can be taken by searching them. So, that is the one good thing about this thing and just to continue this allows also faster joins. So, why what are joins, what joins are generally on two columns, one column from table A and another column from table B.

Now, essentially it just go over that table A and other attribute in the table B the attribute A in table the first table and attribute B in the second table and that again by accessing lesser

number of disk blocks more such joins the join condition between more tuples can be checked. And only if that attribute A values and attribute B values are matching, then the actual tuple needs to be bothered about the actual tuple needs to be output. So, the even the joins are faster, having said all of these are essentially the advantages of columnar storages.

(Refer Slide Time: 12:11)



Now, what are disadvantages there are certain disadvantages as well. So, the disadvantages are the following is that updates, so updates are not good updates are not good at all, because, now what does updating mean updating generally means that particular tuple is updated of course. I mean the particular attribute either, even if it is one attribute is updated or multiple attributes are updated. So, multiple attributes suppose there are multiple attributes are updated for a single tuple.

Now, how many blocks does it require in a row based storage to get the complete tuple it is just one disk, because tuples are stored particular tuples are always stored in a single disk block. Now, here whatever is the number of column families or super columns, that many blocks need to be accessed. So, here is the idea is that, if it is row based storage this entire tuples of suppose this is attribute A this is attribute B and so on, so forth and this is attribute C and everything is stored in a single block.

So, if there are three updates that needs to be done, the tuple is brought from the memory and this is one single block tuple is brought to the memory from the disk one single block that is it. However, if it is row based storage this is in one block this again is in another block,

because in this other block there are attributes only of the same type this is not other attribute. So, that is there not there this is again attributes of this type only.

And again, so the third attribute is again in another block, so this will require three blocks, so that is the difference between, so this is in row and this is in column. So, updates are much slower if the update involves more than one attribute it is much slower for columnar families columnar databases. Also if the query or the join, if the query generally, if the query touches multiple attributes, then it is also not faster probably it may not be faster it is not clear it will depend on lots of other parameters it may not be faster.

Because, it again goes to it accesses the tuple when if it checks if that value passes it needs to go to the other disk block to access the attribute of the same tuple and then, it needs to see that and so on, so forth, so it may or may not be faster. So, when multiple attributes are done it may not be any faster. So, that is anyway the algorithms need to take care of this handling different things and keeping the idea of the tuple id etcetera and so on, so forth can be done.

So, then essentially there are two terms in the database that use in the commercial databases, that is being done is that. So, columnar families or columnar things are good for OLAP and bad for OLTP, so the OLAP and OLTP these are two new terms, so OLAP this stands for online analytical processing or analytic processing and this stands for online transaction processing. So, what does this mean analytical and P stands for this is online analytical processing and the other is of course, online transaction processing.

So, the same thing, so what is OLAP essentially versus OLTP, OLTP is essentially transaction. So, just single updates are being done or some small part of, very very small part of the database is updated. So, a single tuple or a couple of tuples like that account A account B it is only account A tuple is updated and account B tuple is updated. These transactions touch only a very small part of the things, but they touch probably multiple attributes of these things.

So, for that columnar families are not good, because the relational databases can do it much faster if they are doing it. However, for this analytical, so the online analytical processing, what happens is that suppose you are trying to find out the summary of certain things the average over all balance all accounts, then the relational databases the traditional row based relational databases will take a longer time as opposed to columnar databases, because they access that column much faster and can do this summaries and aggregation operations much

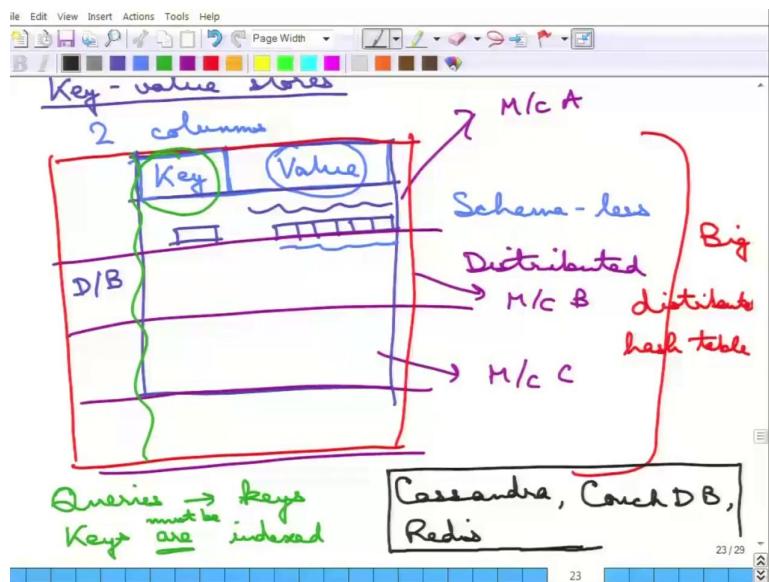
faster.

So, these are the two standard terms OLAP and OLTP and one big example of this relational database is you must remember the name, which is Monet DB. Monet DB is the example of a columnar relational databases Monet DB.

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

**Lecture - 47**  
**NoSQL: Different NoSQL Systems**

(Refer Slide Time: 00:09)



Let us now move on to key value stores. So, the key value stores, the basic idea is that, there are only two columns. Even, if you want to think of it, there are only two things, the first is a key and then, there is a value; that is it. So, that is the entire database schema you may say or whatever, there is a key; that is a value. It is almost like hashing.

So, what does it mean is that, the key can be anything, but it is generally only the text. So, the key can be generally anything, but it is an id either assigned by the database system itself or one can assign its own key, but it is more like a hashing. And the value can be anything; in fact, it is so, I mean it can be even said that the value can be internally broken up into multiple attributes, multiple values and so on and so forth, multiple dimensions and anything it can be done.

But, the key is a single thing, which is generally assigned by the database itself. So, how does one do? Searching it, still it just like one wants to find the particular Tuple. So, one has to specify its key, once a key is done, it finds it and then, the value is all returned as a single

entity, single may be text or may be something else to the system. Now, the system has to parse the value, the system has to understand that, for that particular database, the value consisted of multiple attributes or it is a text or it is an image or it is something else, but that is up to the particular application.

The database by itself does not say anything about, what the value is, it just say it is value; that is it, it just want to block to it and there is a key and then, there is a value; that is all and it is simply essentially an object for the key value systems. So, then the whole database is essentially just one big table with just the keys and values, this is just one big table and that is your whole database, that is the entire database, this is one big table.

So, you can again consider it as a table thing; that is all. So, the keys are stored and then corresponding to each key. Why is it called a columnar sometimes? It is the keys are stored and then, corresponding to each key, there is a pointer, there is a values are stored to where the object that which the value, the object of that value is stored. It is essentially just like hashing, so the hash keys are stored together and from each hash key bucket; there is a pointer to the actual value or the object; that is it.

So, this in some sense then becomes there is a term called schema less, because it does not matter, what the schema is, the database per say does not care about, what the schema in this value, it just say, there is a key and there is a value. So, for all such databases, the schema is just key and value. So, it does not care, what the schema inside it, whether it is one big object and image or it is actually stored in a relational database manner, etcetera, it does not care.

So, this is highly scalable and this is nicely distributed. Why it can be very easily distributed is that, you cut this database into multiple portions and give these things to machine A and this to machine B, this to machine C and so on and so forth. And if you have more data, you just cut it out more and you give it to machine D, etcetera. So, it can be very easily distributed, because all it needs to do is to just cut it out like this and give the keys essentially; that is all.

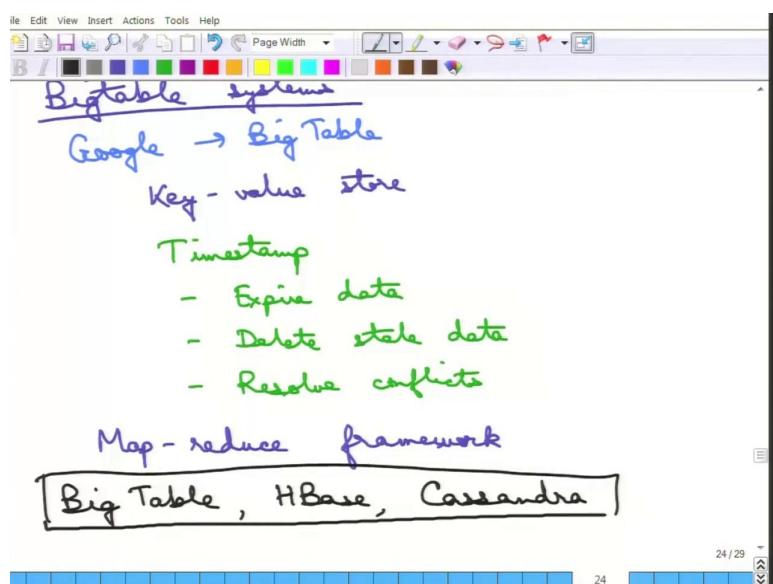
So, this is essentially, very much essentially, let me state it once more, this entire thing is one big distributed hash table; that is all. So, because this is like one big hash table, but this is distributed into different machines. So, that is all that is there, so it is just a big distributed hash table. Now, all the queries are on the keys only, so queries are only on keys, one cannot query the value itself, just like in the hashing, one cannot say give me the key for which the

value is this, it cannot be done.

So, all the queries are on the keys only. So, that is one disadvantage of this thing, because the values cannot be looked inside. So, the keys are necessarily indexed. So, the keys, there must be some kind of keys or keys are generally indexed, because otherwise one cannot keep on searching all the keys. So, keys are indexed, these are must be indexed actually, keys I should say are must be indexed, because otherwise there is no gain in using such a thing.

So, this is all big table and it can use this memory as a cache. So, certain machines, so wherever the queries land up in, the memory of those machines can be used as a cache. So, certain keys which has been queried, it can be again queried just from the cache; that is the thing. So, what are examples of this kind of system? This is Cassandra of course, Cassandra is one big example. Then, there is a Couch DB; that is one example, then there is Redis and then, there are many such examples of this key value stores. So, that is a, these are the examples of this key value stores.

(Refer Slide Time: 05:40)



So, let us move on to the Big table systems. So, you may be knowing that in some time back about a decade or so back, Google had this big table, Google had a schema as a relational, I mean Google had a database not relational, Google had a database, which is called the big table. So, this started the big table system started from that big table; that is why it is called big table, Google called it actually big table. It is essentially a key value store only.

So, this is just a key value store and it is nothing more than that and but, data can be replicated, it provides better availability and so on and so forth. So, the big table system uses a Timestamp. So, a Timestamp is used to store whatever is happening. So, whenever a data is inserted into the table or whenever it is modified, etcetera different Timestamps are given.

So, Timestamps essentially help to finally achieve consistency in the sense that, if an update happened at particular Timestamp, it can be later on, if the query comes at some other Timestamp, it can be later on synchronized and so on and so forth. So, there are different types of Timestamps and the Timestamps can be used to do something else. So, it can expire the data, so one can say that this data will be valid till a particular time point.

So, you can then use that to delete stale data, an old data can be deleted and of course, to resolve conflicts. So, that is the most important thing about Timestamp and we have already studied a whole lot of how the Timestamps can resolve conflicts and this is the read, write conflict set, so that can be resolved. So, that this is all used and then, there is another important term that is here. So, there is a Map-reduce framework, that goes hand in hand with big tables to compute certain things to do certain operations on this tables, the Map-reduce framework can be used.

So, the Map-reduce framework essentially, if you simply understand the Map-reduce is the following manner is that, suppose you want to compute a function on different things, you can map portions of that input to different machines, each machine does its job and then, you can bring in those, this I mean summaries and can then reduce it to one numbers.

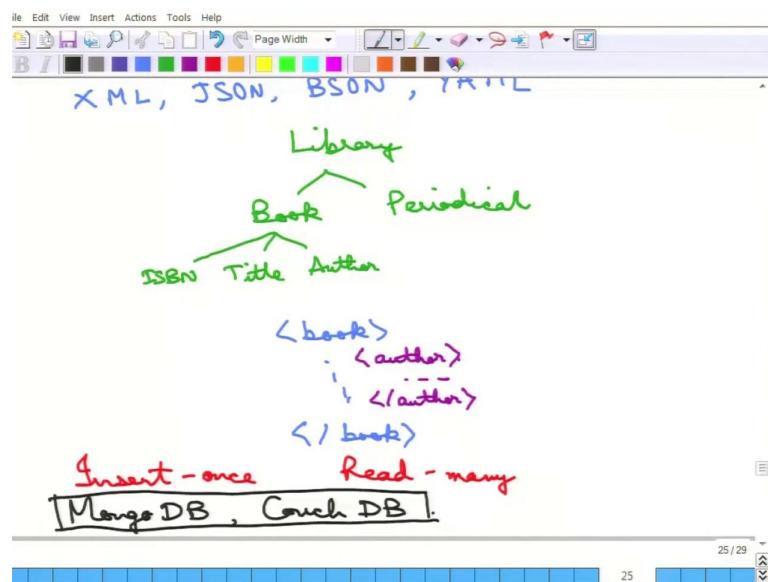
So, an example is that, suppose you want to add 100 numbers and you have four machines in your hand. So, what you will do is, you will map the first let us say, this is a very simple scheme, we will map the first 25 numbers to the first position, the next 25 numbers to the second machine and so on and so forth. Now, each machine will compute the sum of 25 numbers. Why it is done? This is being done in parallel and this is a much lesser work and can be done much faster, etcetera as well.

So, then it reduces a map; that is a map, the first part is a map, then it reduces to these four numbers that you bring it in and submit up. And now of course, this gives you the sum of all the 100 numbers. So, you have to ensure the correctness of I mean of course, sum is a very easy example to see, why it is correct, but there are other things, where the map and the reduce may not be so easy, but that is the basic idea of the Map-reduce framework and that is

being used for this big table systems.

What are the examples of big table systems? Of course, big table itself that I have already said, this is big table, then H base, H base is one very important thing. And again Cassandra, because Cassandra, you can use this Map-reduce framework and you see these are all key value stores at the end anyway. So, these are all examples of this big table system.

(Refer Slide Time: 09:28)



So, then let us move on to document databases. So, document databases, what is the basic idea of the document is that, it uses documents as the main storage of data. So, there are different document formats, one is XML, one is JSON, you may have heard XML and JSON. You may not have heard BSON, this is a binary JSON and then, there is something called YAML.

So, essentially XML, JSON, etcetera, these are just storages in the document form and these are certain formats of data. So, the data can be stored in certain formats, for example, the XML format, data can be stored in an XML format. An XML is essentially a tree kind of thing and the data can be said that, it can be stored in a nice tree kind of format and that is the XML.

The flexibility of XML is that, one can define the own schema and that can be part of the XML document. So, one can define that suppose I will have a library, the library is one big chunk. So, you can define in XML like this, so suppose there is a big library, a library will

have different things may be a book, may be a periodical, etcetera. So, you can define this entire structure, the tree structure. Book will have a title, will have authors and it can have a, I mean ISBN number, so on and so forth.

And it can have, so you can, then this is your schema, this part becomes part of the XML and you can define your entire library collection using this thing. So, what you say is you first define a library scope, then within that there are book scopes, then within that, there are title values and so on and so forth; that is the XML thing that can be done. So, the document that the XML, the whole thing itself is the database, so the particular book, so within library, there will be particular book, so the XML tags are given like book. And then, within book you will mention certain things and then, there is a closing thing for books, this is just like the HTML tag.

So, this itself becomes a key, because this is all inside one big XML document, so that is the thing. So, document this can be put to a particular location; that is the URI, etcetera and can be these things. But, very, very importantly just like the other systems, what is inside a document, so to find out whether there is an author field etcetera that needs to be parsed.

The database per say by itself does not provide any mechanism to parse it is up to the application to say, I want to find out the authors. So, it will look for that particular author thing etcetera. Note that, this is one of the very, very important advantages of the relational database format, once the schema is fixed, once the schema is given, then the database does all the parsing etcetera.

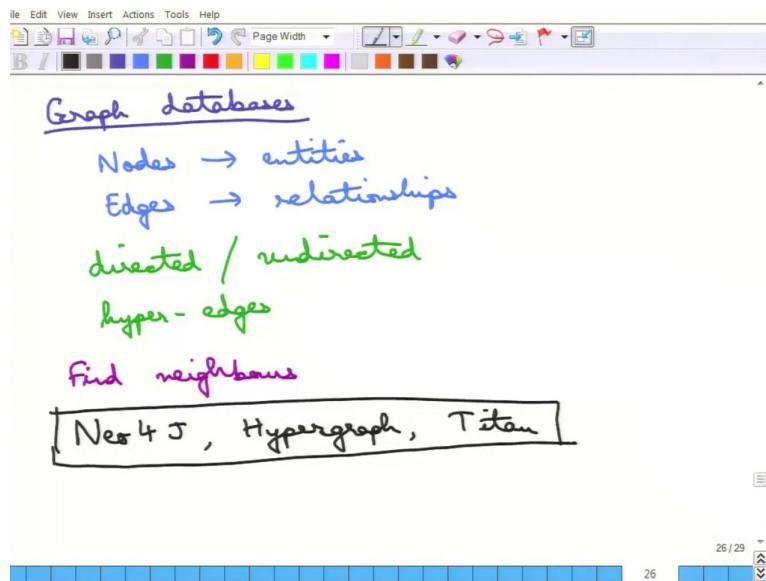
So, once you say I have the schema, I have the following attributes that roll number, name and year and department, etcetera, then you just say I want to find a query on department and the database does it. You do not have to then get that entire student tuple and find out the where the department is, the database does this for you. But, not for these kind of document databases or in general for this key value stores, it is not big even you, you have to parse the database has to be...

So, again it can use the Map-reduce framework and this can be very useful. So, the document databases are actually very, very useful in the following sense, where you insert so the data is inserted once, so this is only insert once and read many. So, for example, in typical library etcetera, the library will acquire a book particular once and many, many people will search that book or do something with that book read it. So, that is very these thing and these can

scale up much better manner.

So, that is the good thing about document databases and the examples of the document databases are, this is your Mongo DB, it is a Document Database and so is Couch DB couch DB is another Document Database. So, these are the two big examples of this thing.

(Refer Slide Time: 13:58)



So, finally, we have the graph databases. So, the graph database the entire all the Tuples etcetera is represented as a graph. So, what are the nodes, the nodes are the, so the nodes represent the entities in the graph. So, this is the facebook graph that we have been talking about nodes. So, what are the nodes in these thing, the every person every account holder is a may be a node and then, the edges encode the relationship between this. So, the friend relationship is essentially an edge.

So, this entire thing is stored like a graph database. So, it is not easy to store this in a relational manner, but it is much more natural and easier to store it in this graph format for a social network. For example, it is a very natural node edge kind of a relationship, it can be directed of course, it can be directed or undirected depending on what the application is or what the kind of database it is saying, it can have hyper edges as well.

So, what are hyper edges is that, it is not a single edge between two nodes, it can a particular edge may be over multiple nodes, for example, a group in a facebook can have multiple nodes. So, that one group may be represented as a hyper edge of all the members in that, all

the nodes in that. So, why are graph databases useful? Graph databases are useful for certain types of queries; these queries are let us say you want to find neighbors of everyone.

So, I want to find friends of friend and these kinds of things that, if it is in a relational format or if it is in some key value format, etcetera it is much harder to find it. You have to get all the values of myself and parse them and then, get the values of those etcetera. But, in a graph thing, it is much more natural to find friends of friends or find neighbors etcetera, this is much more natural and it is actually being shown by facebook and other people, other companies that this is much faster as well, much more efficient, much more scalable.

And what are the some examples of this is, there is this NEO 4J, then there is Hyper graph, then there is Titan so on and so forth, there are many other examples that one can find out, so these are the examples of these thing. So, what are the things about NoSQL is that, so NoSQL, although it started as an anti SQL movement, actually it said that no we do not want SQL any further, it is not so more, it is just not only SQL, it cannot live without the relational database.

The relational database as I said earlier is just too powerful and so it is essentially a compromising saying that in some cases, the relational database does not scale or the consistency requirements, the transaction requirements are just too strict for certain applications and it may not be required. So, acidity is sacrificed and you rather go for this distributed setup, which is much more scalable, so that is the thing.

But, NoSQL as is not good for every scenario banking scenarios must use this consistency and transaction, protection, durability, etcetera in NoSQL system just fail. And even nowadays, it is people are realizing more and more that consistency matters a lot, so eventual consistency may not be desirable in all kinds of applications. So, there are now people are although NoSQL is very much hyped and it is taking over, it is nowhere near relational databases.

And actually database researchers are again going back to this traditional DBMS things, I am trying to fix certain problems in NoSQL using this traditional RDBMS methods. So, most legacy systems still use this RDBMS, etcetera and NoSQL is currently well it is as I said it is currently too hyped etcetera and NoSQL. But, however, NoSQL there one important thing about NoSQL is that, with the advent of big data and cloud computing etcetera most of those try to use No SQL system. So, they try to use the properties of NoSQL or do rely on NoSQL

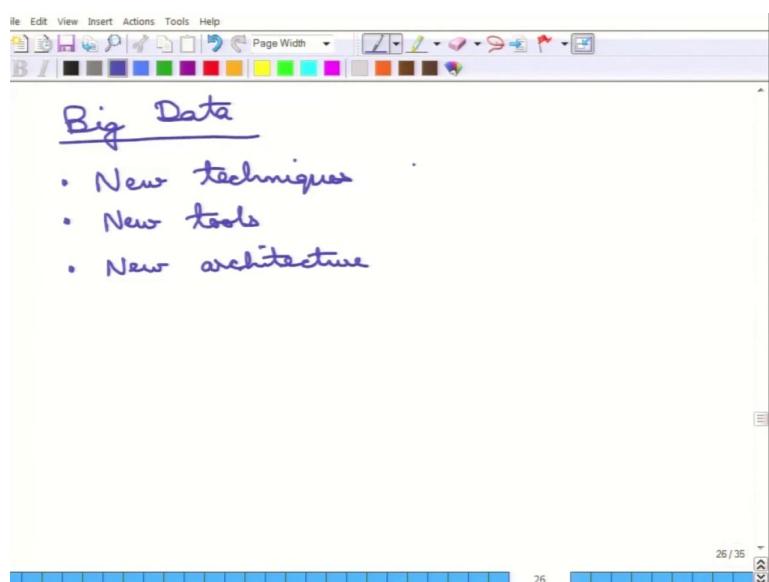
as the backend databases as the support systems for all of that. So, that ends the module on NoSQL and later, we will touch on a little bit on big data.

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

**Lecture - 48**  
**Big Data**

We will start the last module which is on Big Data. So, big data is the term that I am sure all of you must have heard it.

(Refer Slide Time: 00:21)



So, what is big data? So, we will cover very briefly about big data. The first question that everybody has in mind is, what is big data, now big data very simply, data which is big. Now the question of course, is what does it mean, how big is big. So, for example, if you go to sociology, sociologists will do experiments, field experiments, will ask people questions and will take their answers and so on and so forth. For sociology, may be even 10 persons data is big, because it is very hard to get a complete set of 10 persons, we will answer all their questions honestly and reliably and so on and so forth.

So, for them 10 is big data, on the other hand you go to physicist, who takes astronomy. So, people who gets these images from astronomical telescopes and so on and so forth, for them terabytes of data is what they get in a day, single day they get terabytes of data. So, terabytes of data is, even terabytes of data is not big for them, so peta bytes and so on so forth is probably big for them.

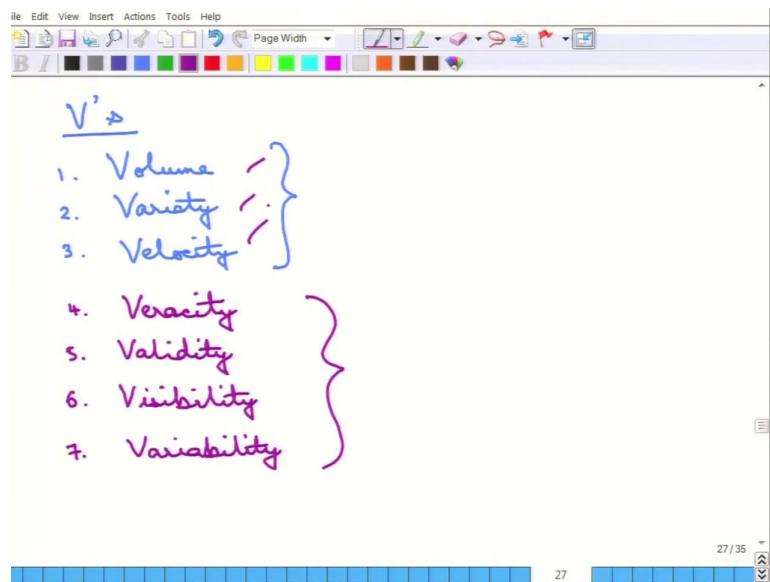
So, there is no threshold of, there is no very nice threshold of saying, oh after this, this is called big data and below this, this is not big data, there is nothing like. It critically depends on the application that you are talking about. So, once more there is nothing concrete about what big data is, it critically depends on the application. Generally, what is being done is that, what is considered big data is that, when the data or the algorithms that will run on the data, this cannot be handled by a single machine that is typically considered to be big data.

So, when that data is too big, that cannot be stored in a single machine or that cannot be handled by traditional algorithm, because it probably requires too much time, impractical amounts of time or too much resources that is being considered as big data. But, do note that this is all are very fuzzy definition, there is no strict crisp definition of what big data is, it depends very much critically on the application that it is talking about.

So, that is big data that, but and let us see a little bit of what the characterization of big data is essentially, now this large volumes of data as I am saying, the traditional algorithms may not be able to handle it. So, essentially you may require new techniques to just to store the data, just to query, handle the data. So, new tools, because the traditional tools such as the traditional RDBMS systems may not be able to handle and many new architectures of computing systems.

So, because single machines may not store it, so you will require probably cloud computers or super computers or distributed setups and so on and so forth. So, that may be necessitated and so the big data may require all of these things. Now, for big data what are the, there are certain properties of big data that is being done.

(Refer Slide Time: 03:19)



So, the three most important properties of big data are the... So, the big data is typical this properties which are called the v's, v properties. So, the three important v's are the first three important v's is volume. So, big data generally has a very large volume, so when it is very large, even how to load even all the data into one machine or into whatever, I mean there are system to load it, index it, query it etcetera that is a problem, this volume. Then it is variety, because big data may not always be pertaining to data that is of a single types, so it is not one schema or one cascade of things, then we will various kinds of things mixed in a big data setup, so that is the thing.

And it can be the data can be structured like RDBMS or semi structured like key value stores or completely unstructured like text, written memory I mean there is no structure almost no structure, so that is the thing and the velocity. Now, what does it mean to say velocity? Velocity is essentially the big data may be arriving at real time, so it may be streaming data, so it may be as the algorithm process it, more and more data keep coming. So, and it can be so big that the entire data cannot be stored and then it processed upon, it needs to be processed a faster, it need to be processed in real time in a streaming manner, so as data is coming in, there is some processing going on to it and that is being processed.

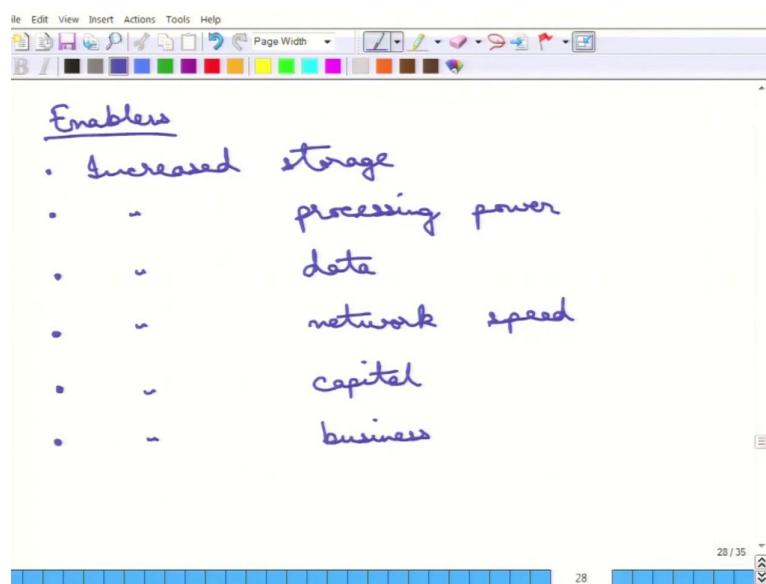
So, these are the three initial v's of big data, then as happens with any hyps transfer big data is very much hyped and there have been some other terms that have been associated with this. So, veracity, veracity means whether the data that is coming or the authenticity of the data,

the truthfulness of the data whether it is correct or not, the validity same thing whether the data is still valid or the data that one is processing big data has expired.

So, by the time I write a post into the face book and I delete it that post is no longer valid, so that is the validity. Then visibility, how does one visualize the entire data and whether it is visible to all parts of the system can everybody. For example, seeing my facebook post and so on and so forth, it can even if all the machines want it and so on and so forth, the visibility is another issue and the seventh one is the variability.

So, how much variety can this big data handle in the single setup? So, how much can it be anything or can it does it need to pertain to certain kinds of structure at least, certain kinds of rules at least. So, these are the issues with big data, so these are the three I mean the volume, variety and velocity are the three initial v's and then there are many other v's that has been talked about.

(Refer Slide Time: 05:59)

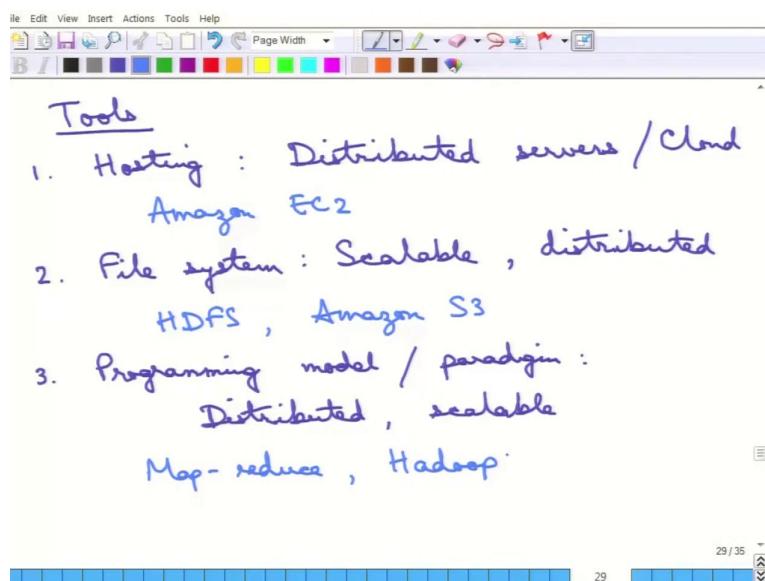


Now, to for big data systems to happen we require certain enablers, enablers meaning certain things have to fall in place, so the big data systems can be talked about. So, first thing is that it requires increased storage space, so suppose terabytes of data are coming in a day. So, you will require beta bytes storage that so 1000 terabytes is one peta byte. So, you will require that kind of storage, so there must be those hard drives or those systems must be made, so that is the thing, so this must be available.

So, increased storage volume and increased type of storage, then increased processing power. So, we will require faster and faster machines, more and more better and better CPU's, etcetera just it is very understandable and of course, increased data. So, big data is all about this data, so more and more data is being produced and that is why it is called big data, then many times the data is available only over network. So, increased network, speed or network capabilities, whatever you want to say, because data may be streaming in from different networks and you may send it to multiple database, because it is a distributed system, so it must be sent to different places, etcetera.

And of course, capital or I mean increased capital essentially money, we require money to store all of these things. So, all this is required to do big data things and well why will you do big data, there must be increased business. So, the last two things are essentially trying to say that if you wants to do a big data something on big data some company wants to do big data. So, it must get the profit out of this to... So, that it will be it is worthwhile to engage in this big data principle, so that is the thing about big data and these are the enablers of big data and then there are certain tools for big data that one has to talk about and that is it.

(Refer Slide Time: 07:52)



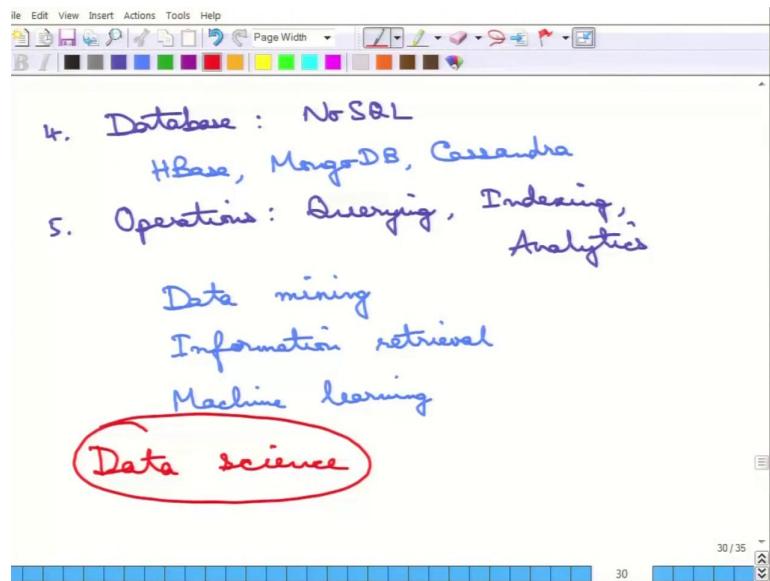
So, there are certain tools that are already out already available, so tools for big data. So, the first thing is that for hosting big data there are this clouds. So, the cloud the distributed servers or the cloud that are being... So, these are already available, so this is the famous cloud. So, all of you must have heard this term cloud computing and again that is not a very

crisp term, but essentially cloud it is a distributed server. So, one can store data and there are examples is that your Amazon EC2 that let us you store lots of data and of course, you have to pay by it is not free, but at least let us you do that.

Then the file system, there are certain tools, there are certain file systems available which will let you do this thing. So, in the file system the properties of this file system is that this must be scalable and they must be distributed, the file system itself must support this data, a traditional file systems may not be useful. So, again HDFS the file system this thing that supports this is already available and then there is a Amazon has its own file system called S3 that is there then there is a programming model.

So, the programming paradigm itself changes, because big data cannot be handled by traditional algorithms may be. So, that is why the programming model or the programming paradigm needs to change, so this has to be more scalable and distributed. So, this is again distributed and scalable, so this the new programming module itself has to have this constructs built into it, it is not it must support this naturally and the map reduce framework, as I have been talking about map reduce framework will handle this and Hadoop is another tool that let us one do all of these things.

(Refer Slide Time: 09:51)



Then it may require of course, it will require database support. So, these databases may need to go beyond RDBMS and as we have been saying this essentially this is all clubbed under this NoSQL term as we saw earlier. And these are examples are of course, your H base then

mongo DB and Cassandra and all of these things and all of these are enablers for big data. So, these are tools for big data then this will let you do certain analysis on big data that is all and then there are finally, this is operations.

So, operations are your, what are the kinds of things? So, it must let you do... So, tools for querying, tools for indexing, tools for doing analytics. So, one has to do run analytics and this O lap etcetera these are all coming from that querying indexing and analytics and then there are... So, for this there are this data mining, data mining there are different data mining tools etcetera there are different...

So, essentially the entire data mining is **proliferating** because of this big data things, because it has to have this operations doing for this thing. So, data mining then information retrieval these... So, these are all generic terms of course, and these are not particular tools that I am mentioning, because there are many, many tools in these spaces. But, all of them essentially can handle as enablers as tools for doing big data.

And machine learning of, so machine learning. So, one particular example of machine learning which is interesting is that there is a mahout tool which is built on top of Hadoop. So, it can do certain machine learning algorithms directly with Hadoop database, so with the Hadoop this thing. So, there are many open source tools, the many of them are open source and are free and some of them are not everything is free cloud for example, Amazon EC2 etcetera, etcetera are not, free quite costly, so we have to store this.

So, one has to take look at the application to see whether actually big data is suitable term for that whether it makes sense to have big data for that the application must clearly define whether what it is big data or not and just saying that my application has large amounts of data. So, I will use big data tools or big data application that is not fair sometimes that is not correctly actually sometimes traditional algorithms can do quite well.

So, one has to clearly define and see whether big data is doing all of that to handle all of these there is a new paradigm that is coming out which is there is the new term that is coming out which is called the data science and there is the, there are data science courses and there are data science things that is there. So, essentially to handle all of this big data cloud computing all of this things weather, so that is the emerging term.

So, again currently just like NoSQL big data is not clearly understood, what is big data it

depends critically on the application and it is way to hype, but these are certain enablers there are certain tools that can be done for that. So, that ends the thing on the this course that ends all the lecture modules on the course. So, I hope all of you have enjoyed this course, I hope all of you have learnt something from this course, you have done the assignments honestly and correctly and have learnt something from the assignments and the slides have been put up and the assignments solutions have been given the final certification exam, if you are taking do study for it, it is not going to be easy.

And, but I hope you all pass and we will get to know and we will have a rough idea at least of what can be there from the assignments and at the end of this course and at the end of all certification exams, you will have a quiet a good grip on what database systems are all about the UG database systems. And you will be very much comfortable and confident about handling issues for databases.



**THIS BOOK IS NOT FOR SALE  
NOR COMMERCIAL USE**



(044) 2257 5905/08

|



nptel.ac.in

|



swayam.gov.in