Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 40
XML Databases

Hello and welcome. In this final session on introduction to XML, we shall be delving primarily into XML databases. That is until now we have been talking about what is XML and what are the benefits of XML or data interchange using XML and XML queries or schemas and so on.

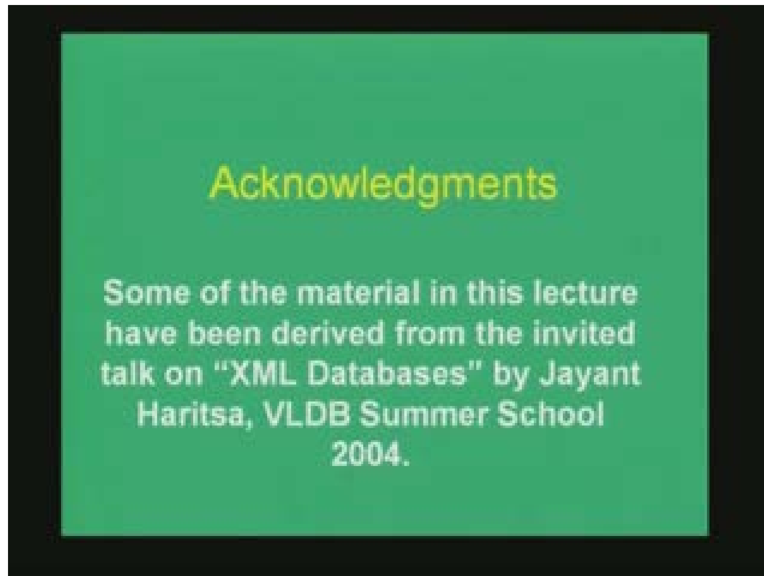
(Refer Slide Time: 01:49)

But today in this session, we shall be looking into XML databases and what are the issues that come up when XML data has to be stored or transmitted and exchanged over preexisting or some set of existing data stores. We shall also be looking into the larger problem of what is called as semi structured data management. Semi structured data as you might have imagined from the name is data which do not have specific rigid structures. And we shall show that more and more the data requirements of today's world what we call the post internet world is being defined by semi structure data rather than structured organized data. And semi structured data possess some unique problems in fact some fundamental problems in database management. And what we shall do here is that in this session is we shall only motivate those problems rather than providing any solutions as such many of which are in some sense areas of active research.

So we shall be motivating the problems with some examples or some analogies and instances and we shall conclude this session with a list of what one could expect in terms of semi structured data management.

So let us proceed further with this session and as with earlier session some of these material have been derived from an invited talk by Jayant Haritsa in the VLDB summer school held in Bangalore in June 2004 and of course including many of the analogies and jokes that have been there in the slide and so on.

(Refer Slide Time: 03:20)



So here is an acknowledgement for those set of material that have been derived from those slides. So let us look back at XML again and recap what we have learnt about XML and what are its characteristics. First of all before we look into the points in the slide itself, let us remember that XML is a platform independent, standardized and extensible markup language.

So, in a sense that its platform independent because it is written in character data and every platform should at the very least support character data and it's an extensible markup language that is it does not have a specific set of keywords as such. I mean of course it has key words where you say, where you give declarations but for describing the data itself, it doesn't have any keywords.

The user who is describing the XML document can come out with his or own set of tags or meta data that describe how the data is organized and what semantics to attach to each data set in an XML store. And XML syntax has a plain hierarchical structure which is easy to navigate and easy to enforce and easy to parse as well. So we have, as a result we have query languages like XPath which translate an XML query into a directory like path expression which is quite intuitive for users who have been using computers and so to traverse directories and look for something in a specific directory and so on.

And last but not the least, XML is understandable or parsable by both the machine and the computer. So if everything fails what you can do is just open an XML document in notepad or vi or Emax or some such text editor and look through the document and see

where there could be a problem especially, if may be starting tag doesn't have an ending tag or some problem of that sort.

And of course there are issues like what about the amount of space, extra space that XML takes up and this can be answered by the fact that one need not store XML in character form on disk. You might want to actually compress XML data and then store it on disk, so that you can and decompress it when reading it, so that the actual disk space that is taken up by XML is reduced.

So let us address the question of data interchange now using XML. XML is said to be a platform independent source of data representation and representing data that is self-describing in the sense that the meta data is embedded within the data. Now, but what is the problem, what is real and of course the advantage of this being that you don't have to worry about whether the end user is using the same platform as yours.

(Refer Slide Time: 03:38)

If you are using Linux, the end user could be using windows or Solaris or Mac or whatever and the end user could still use your services using SOAP or some other form of XML based message parsing protocol and you can still answer or you can still provide those services via an XML wrapper. But what is the interchange problem actually or in its entirety? For the interchange problem, we have to first note that most data is already stored in some existing databases. It's quite unlikely that databases that have been existing for large periods of time now will be replaced by an XML data store or will be replaced by something else that is unifying.

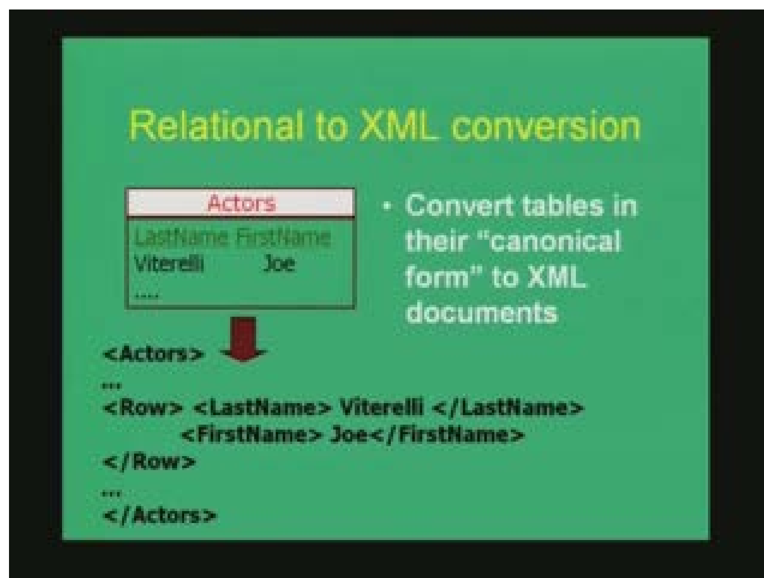
So databases are going to continue to exist and the question of data interchange will boil down to the fact that how to interface these databases using XML or using a common interchange language. And databases will not only exist, will be updated through existing interfaces. It's again quite unlikely to expect that all updating and all interface to the

databases will come through a common interchange format. So what is really required is to provide XML wrapping to existing data bases and SOAP for example in this regard that is message parsing between objects existed long before XML came into the picture. There were several different object middle wares like CORBA or DCOM and so on which supported message parsing between objects over a distributed system.

But however one needed to be CORBA complaint in order to be able to, a platform needed to be CORBA complaint in order to be able to support CORBA that is CORBA had to be returned for that platform. On the other hand when using XML, it does not bother about what kind of platform that's being used because every message is parsed using an XML wrapper.

So SOAP is some kind of a generalization over message parsing frame works in distributed object base systems which has been extended from simple, small distributed systems like CORBA to a larger web base services using SOAP. So a similar analogy also exists in data integration where data exists in different databases in preexisting forms and preexisting applications and they have to be integrated using, by wrapping the data around using XML wrappers.

(Refer Slide Time: 10:18)



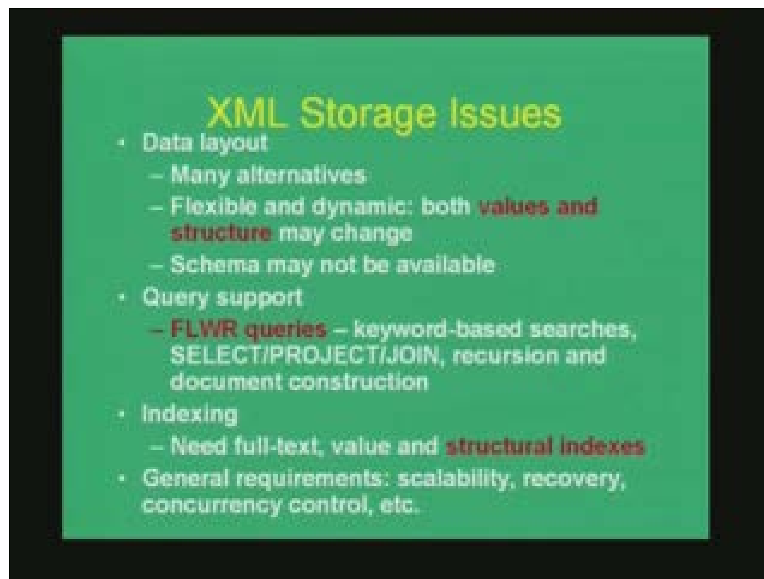
So one simple way to convert databases and now here we are assuming relational databases. And for the most part we would be correct because majority of the database implementations around the world are relational databases. So assuming that existing databases are relational databases and we need to be able to interface between these different relational databases which are across different platforms. A simple way to perform this is to convert relational tables in their canonical form to XML documents.

And this slide shows (Refer Slide Time: 10:49) such an example where there is a slide called actors containing of two different columns last name and first name. And this table

became one XML document actors and slash actors, it's a rooted XML document. And it is some kind of a flat XML document in the sense that the number of levels in this document is fixed. That is the actors file or the actors document comprises of several children called row and slash row where each row comprises of again several children, each child corresponding to a specific attribute or attribute name and this is an attribute value within the attribute name.

So it's quite simple to map a given relational table on to a canonical flat XML file and given a flat XML file, it's again straight forward to map it back into a relational table.

(Refer Slide Time: 12:07)



Now let us come to the question of storing XML documents itself that is having XML databases itself. In many cases for example in several commercial implementations, oracle 9i and DB2 and so on, the database gives an impression that it supports XML storage and XML based updates. However what it actually underneath, it's still a relational database and then there is an XML wrapper around the database.

However that is there are several other approaches to storing XML documents and in many cases it becomes necessary to treat XML not just as an interchange language but also as the language in which the data is stored. And some of the issues that occur in this regard would be issues like data layout. How would you organize the XML data there are again many alternatives to this question.

One can map XML data on to a relational database or one can map an XML dataset as a special kind of object in an object relational database or one can store XML data in native XML form or in text form and so on.

And what about updates? I mean XML databases are prone to updates and here in XML unlike in say relational database it's not just the values of attributes that can change, the

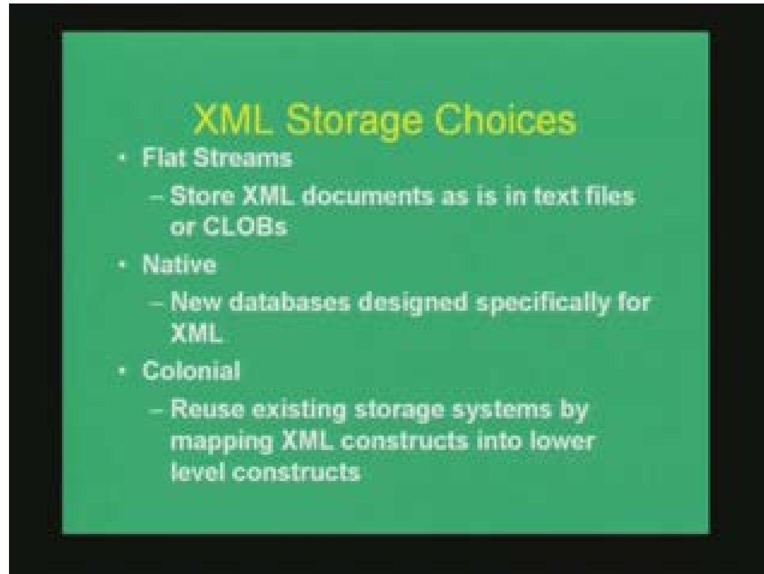
actual structure of the database can also change very frequently. In a relational database we basically assume that once we fix the schema, it is intact and it's only the data sets that are updated. That is new attributes are added or existing attributes might be deleted or modified and so on. But the evolution of the schema itself that is change in a structure of the database itself is considered to be far less frequent if at all it happens. So the database is modified, database is oriented, the relational database is oriented towards fast updates and retrieval of data and not the structure as such.

But in an XML database that may not be the case that is there could be updates of not just the values but also of the structure of in which data is stored in the database. And in many case there may be no explicit schema that's available, so well formedness itself is the schema. so we don't we don't really know which is a valid structural update and which is an invalid structural update and what kind of constraints exists between different elements and so on, unless there is a XML schema or a DTD that's available for us.

Similarly there are different kinds of query supports that need to be supported by an XML database. this standard flower queries that we saw earlier where for let where return kind of queries and also select, project and join based queries where which is also supported in XQuery in addition to recursion or document construction that's transformation from one XML document to the input XML document to the output XML document. And a far bigger problem is of indexing. How do we maintain meta data or how do we maintain indexes into an XML store? Such that we can retrieve elements as quickly and efficiently as possible whenever required. And it's not just the normal attribute and value index that we need to store but we also need to store full text indexing, what are called as inverted indexes, so to be able to search a full text elements or data sets quite efficiently.

In addition to these kinds of indexes we also need what are called as structural indexes, I mean structural indexes essentially talk about what are the relationships between, structural relationships between different elements is one a child of the other is one reachable from the other or what are the siblings of a given element and so on. And in addition there are general requirements like scalability and recovery in case of failures, concurrency control, updates and during updates and so on.

(Refer Slide Time: 17:00)



So let us look at XML storage structures itself and what are the different kinds of choices we have for XML storage. Basically we can divide the different kinds of XML storage that storage structures that are, storage paradigms or storage mechanisms that are available into three different classes, what might be termed as flat stream based storage or native XML storage and colonial storage. So what do each of this terms mean? A flat stream based XML storage essentially stores XML data as some kind of objects in an object relational database. They are stored as CLOBs or remember what is a BLOB. A BLOB is a binary large object and a CLOB is a character large object.

So you just store an object comprising mostly of characters, so character large object which contains its own mechanisms of access and retrieval as one of the attributes in a relational database. So just like storing any multimedia object or some such object, you can store an XML dataset in a relational object relational database itself. But of course the advantage of this is that, you don't need to reinvent anything in the sense that there are several object relational databases that are already available. And it's just a question of using one of the object relational databases to store XML documents but however of course the flip side of this argument is that the database itself does not support any XML centric queries. So you can query the database based on object relational constructs but not an XML constructs itself. And all those XML specific query constructs like flower queries or XPath queries and so on cannot be directly supportive.

The next kind of database storage strategy for XML is the native XML storage. Native storage essentially means that you design a new database from scratch for storing XML data, optimized for storing XML data. So here you have to worry about everything that you thought about let's say for designing a relational database, one is to think about the storage structure, block accesses that is the physical storage structure, indexing structures then some kind of query mechanisms and recovery mechanisms, concurrency control,

transaction I mean throughputs and I mean how to maintain efficient transactions and so on.

So everything that goes into designing any conventional DBMS should go into designing this native XML storage as well. And there are quite a few examples of databases supporting native XML storage. The third strategy is what might be termed as a colonial strategy. A colonial strategy essentially means that colonize an existing paradigm using XML. That is use some kind of, use an existing paradigm like say relational paradigm and then map every XML construct to a relational construct rather than note the difference between a colonial strategy and the first strategy which you just put XML storage as character large objects or one of an attributes of a relational table.

However a colonial storage essentially decomposes or deconstructs an XML document into relational constructs and reconstructs them back. So whatever XML related query that you provide like XPath expressions or flower queries and so on, they have to be rewritten in SQL and you have to have a mapping, one each to have a mapping between XQuery constructs and SQL constructs and vice versa. And of course this can be both an advantage and a limitation. The advantage primarily being that you don't have to worry about number of techniques that the native XML storage has to worry about storage structures, block storage structures, concurrency control and recovery and so on.

However there is a significant if not huge performance overhead in terms of mapping an XQuery construct into an SQL construct. Especially when there is a recursive kind of a query that's given, mapping it into an SQL construct can take quite avail and running the SQL query on the database can also be quite inefficient.

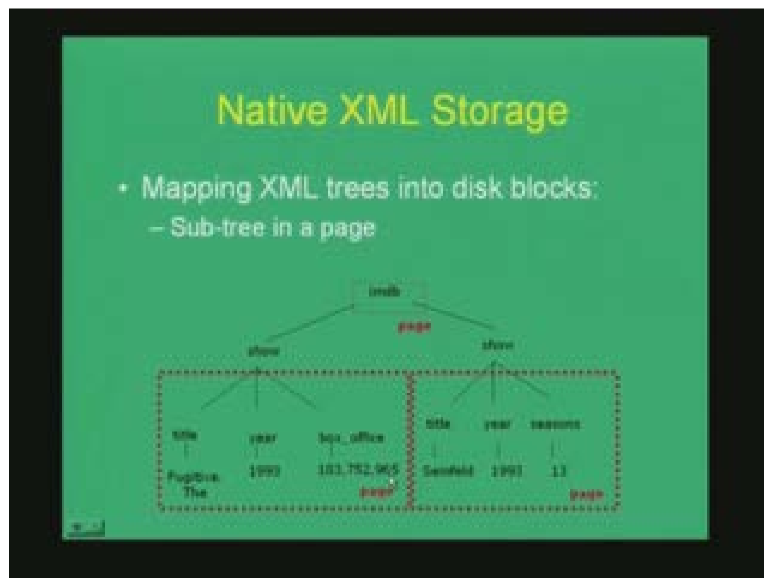
So let us look into the second strategy in a little bit more detail namely the strategy of native XML storage. That is redefining or redesigning an XML database by looking at all the different aspects that needs to go into an XML store. So what are the issues, what are the typical kinds of issues that one needs to worry about when designing a native XML storage. One need to think about data layout, how is data organized on the disk and physical data layout is essentially that is if my disk is organized in terms of pages, disk pages or disk blocks what should each block contain and how should the blocks be organized and so on.

Indexing is again a major requirement like we said before it's not just attribute and value indexes that are important, we need to also look into full text indexing or indexing of phrases within text or structural indexes that talks about how elements are related to one another and so on.

(Refer Slide Time: 22:25)

And it needs to also address query support, what kinds of queries are supported in this store and how do we optimize queries and how do we preprocess, do we need to preprocess or and if so or how do we preprocess for managing efficient query retrieval and so on. Then access control, concurrency control, updates and so on transactions, recovery and so many other issues.

(Refer Slide Time: 24:00)



Now let's look at the question of data storage in a little bit more detail. And what are the different approaches that are used for managing or storing physical disk blocks or managing the physical data storage that is how is an XML tree mapped on to physical

disk blocks. Essentially one can think of, essentially an XML data set is a tree. So this slide shows a particular tree like this where there is an imdb at the root node and there are different show elements in the second node and each show element has different sub trees like title, year, box office and so on and so forth. And finally the leaf node contains the dataset that is available in this XML tree.

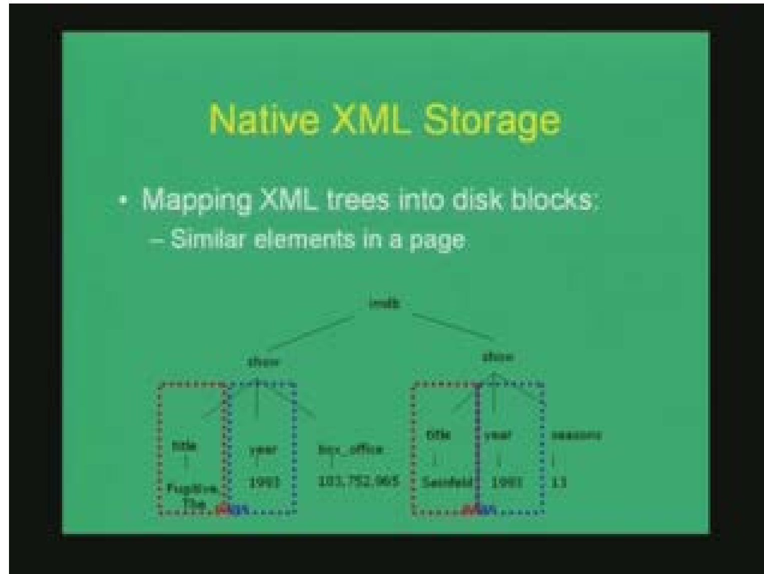
Now how do we divide this tree into physical data blocks and which kind of division makes sense? One way to think of, one way to look at dividing a tree into disk blocks is to cluster trees based on sub trees. That is this show sub tree would go into one disk block and this show sub tree would go into another disk block and the imdb would just be in an index which in turn just stores pointers to each of this disk blocks.

Now what is the advantage of such a storage? The simple advantages that the entire sub tree is in one disk block, hence if I am looking at navigational queries where the user just navigates from like opens an element and looks at the sub tree and opens another element underneath and looks at that sub tree and so on like in a explorer kind of file system navigation. For such kinds of navigational queries, this storage structure is very efficient because when the user is navigating, the navigation path is quite close to the actual way in which elements are related in the tree itself. It is quite unlikely that the user would open this sub tree and start navigating here. So in one disk access, one can read an entire sub tree.

However there is a flip side to this kind of data access itself. Suppose the user gives a XQuery kind of, a user gives a query saying show me all elements, show me all show elements where the title contains whatever the fugitive or something like that. Now if the query has to be searched on a particular element like say title and even though we know which element has to be searched for, we still have to access every data block in the disk because every data block which contains the show element, which stores the show element would contain a title element. So we still have to access every data block containing the show element in order to access such a query.

So to answer attribute kind of queries, such a kind of organization is a file organization is actually counterproductive or is not very efficient. However for answering navigational kind of queries, such a kind of access is quite useful.

(Refer Slide Time: 27:23)



The second kind of database storage structure is to cluster similar elements within a database, within a data page that is within a disk block. So, this slide shows such an example where the same tree is taken imdb and show and title, year, seasons and so on. However rather than storing an entire sub tree within a data block here, each element at a particular level are clustered together. So, all title elements are clustered together under stored in one data blocks, so in the red data block let us say. And all year elements are clustered together and stored in one single data block that is a blue data block and so on. And then there are pointers that point to each of this data blocks and so any of these elements that don't contain CDATA or PCDATA, you can cluster all of these into one data block and then maintain pointers from them to each of these data blocks.

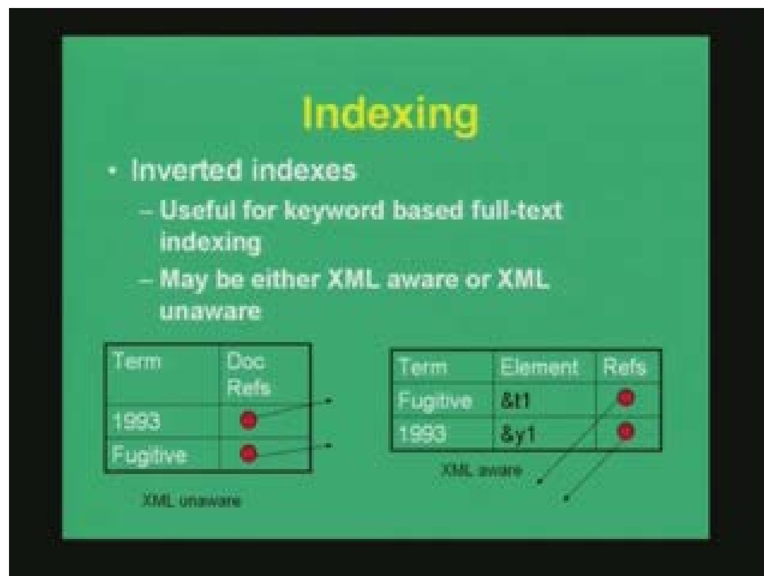
Now, again this is in some sense the dual of the earlier mechanism where such a kind of organization is very useful or very efficient for answering attribute queries. So if I want to say show me all show elements or return all show elements where the title contains the term the fugitive then all that you need to do is to first access this block which contains the show element and find a pointer for the data block containing all the title elements. And with just one data block access, for one will and of course one or more I mean depending on how many title elements are there but generally with far lesser data blocks than that are necessary in the previous case, we can access all title elements that are there in this XML store.

So answering an attribute query is far simpler, however answering a navigational query becomes difficult in this case. And there are other techniques for this thing, it depends on how something is clustered. If clustering is performed based on what may be termed as the lowest level elements where the elements contain just CDATA or PCDATA, it becomes difficult to navigate. That is from show, you need to open a title and year and from title to year it requires a different block access and so on. However it might be

possible to cluster based on paths rather than based on single elements, so rather than clustering similar elements, some other techniques cluster similar paths.

Therefore one would say that cluster all paths of the form imdb show and title in one block and all parts of the form imdb show and year in another block and so on. So there are different variants, however if one were to ask the question which is the best way of storing, which is the best storage structure for XML databases, the answer would be depends. Essentially there is no single, there is no single technique that is universally most efficient way of accessing or storing XML databases. And to a large extent it depends on what kind of queries that you expect from the users. So if it is navigational queries, it might be better off to cluster the tree based on sub trees, so store sub trees within data blocks. On the other hand if it is more of attribute searches or even say full text searches, it might make sense to cluster similar elements in a page rather than sub trees.

(Refer Slide Time: 31:54)



Let us look into the problem of indexing XML documents and what kind of indexing requirements arise for XML document. Firstly, we said that in addition to attribute value indexing, for which we can use traditional RDBMS indexing like say B plus tree or a B tree, here we need two other kinds of indexes. one is what may be termed as full text indexing were we should be able to efficiently search on free text data that are written within an XML document.

So one might just write a paragraph within an XML document and be able to search for some key word in that paragraph. And very common mechanism of indexing full text is what is called as an inverted index. An inverted index is very similar to what you would find at the end of a book like say text book were you have an index and there are certain keywords and if page numbers or section numbers in which the keywords are, in which those keywords are either defined or used or whatever.

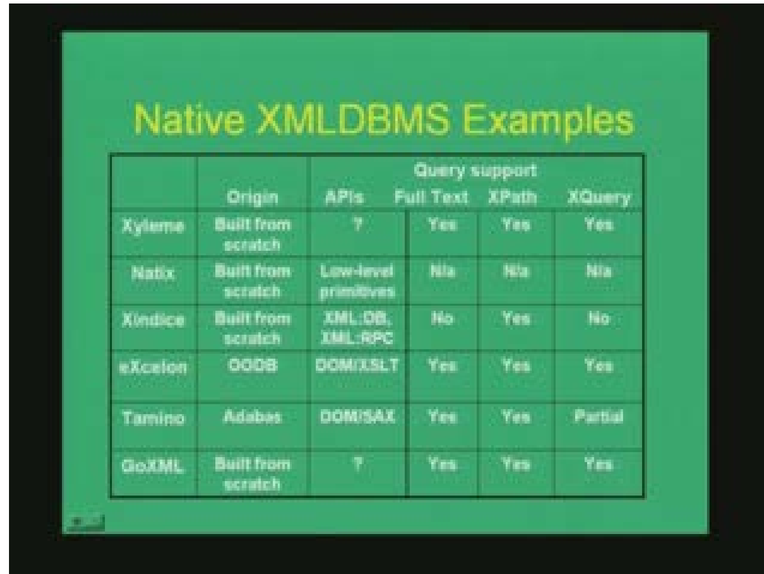
So an inverted index on an XML document would index different keywords that appear in the document and then maintain links in to the XML document saying where each of these keywords can be found. And keyword based indexing can be of two kinds, it may either be an XML aware keyword indexing or an XML unaware keyword indexing. So what is the difference between the two? XML unaware keyword indexing just looks for keyword searches. So you just give a keyword search called the fugitive or jerry sign field or whatever that appears in the XML document. And keywords are searched, this table here shows XML unaware keyword searches that is for every term that appears, it just stores the document reference which document contains this or probably the element reference or whatever. So it doesn't bother about where keyword appears and it only bothers about the keyword and the value in the keyword.

On the other hand an XML aware keyword or an XML aware index not only contains the keywords but also another element or another column which says in which element does the keyword appear from. So there is one more index that this ampersand t1 here is an element index or is a key into an element index where it identifies each element uniquely that is present in the XML store. So what this says is that the term fugitive can be accessed or the term fugitive appearing in the element whose key is t1 can be accessed using this pointer from wherever. So if you are looking at attribute based searches and we want to say return all show elements where the title sub element of show contains the term the fugitive.

Then an XML aware indexing makes much more sense than XML unaware indexing. and of course the flip side of XML aware indexing is that as the number of elements increase and keywords are repeated across different elements, there is a huge amount of combinational choices that appear between a given term and an element pair. So the term 1993 for example may appear in different kinds of elements, it might appear in births date, it might appear in release date, it might appear in show date or whatever and so on. so different kinds of elements, so each of these have to be indexed separately and which leads in increase in the size of the index structure. So this table shows different XML, native XML databases that are available and quite a few of them are already available like Xyleme or Natix or GoXML and so on.

And most of them have been built from scratch and some of them like eXelon or Tamino have been built over existing databases. So it's not in a pure sense, a native XML storage but they do they are called native mainly because they do address many of the questions that are typically addressed in native XML storage. And there is a wide diversity or wide ray, wide diversity in the kind of features that are supported.

(Refer Slide Time: 36:43)



	Origin	APIs	Query support Full Text	XPath	XQuery
Xyleme	Built from scratch	?	Yes	Yes	Yes
Natix	Built from scratch	Low-level primitives	N/a	N/a	N/a
Xindice	Built from scratch	XML:DB, XML:RPC	No	Yes	No
eXcelon	OODB	DOM/XSLT	Yes	Yes	Yes
Tamino	Adabas	DOM/SAX	Yes	Yes	Partial
GoXML	Built from scratch	?	Yes	Yes	Yes

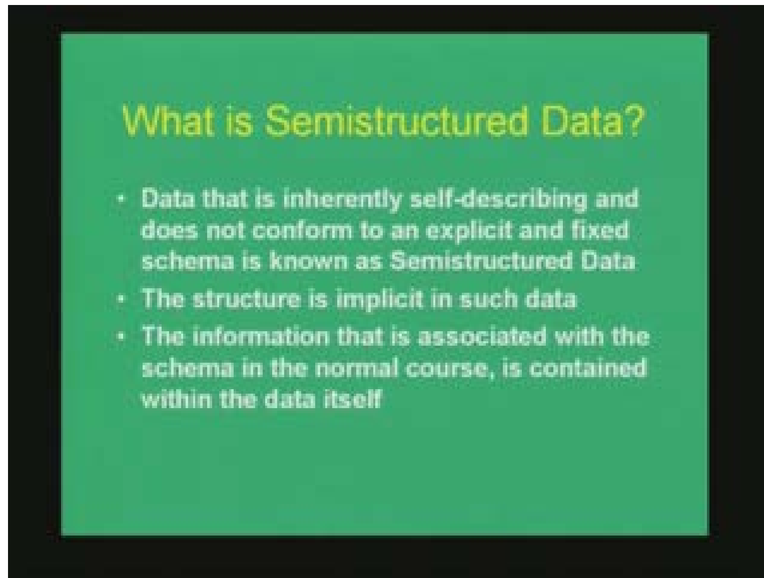
So Xyleme for example supports full text searchers and XPath searchers and XQuery searches but it doesn't have any what kind of APIs that it provides or unknown and Natix doesn't support any of these but just supports some low level primitives and so on. and there is partial support for XQuery and so on.

(Refer Slide Time: 37:10)

Let us now move into the second part of the stock where we look into managing semi structured data.

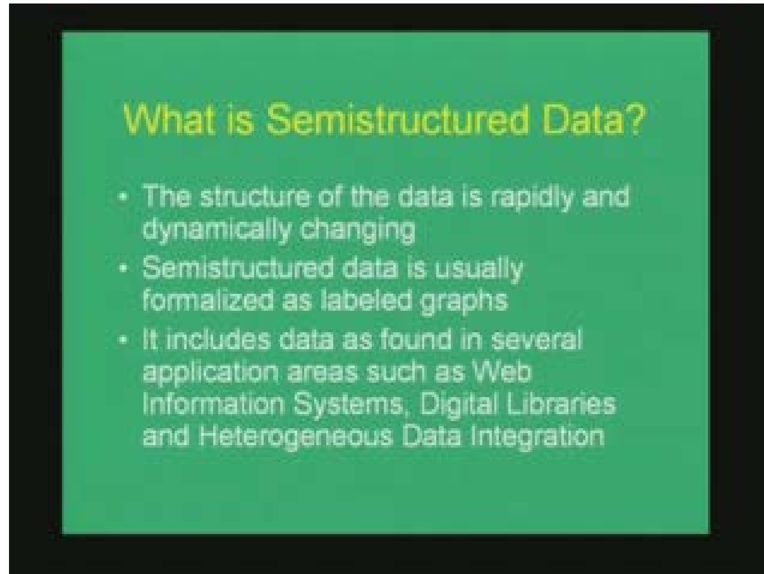
So until now we have been looking primarily into XML and XML is a very comprehensive tool to manage semi structured data. And so what is semi structure data and what is the significance and why is it important to study about semi structured data?

(Refer Slide Time: 37:53)



So, let us first define what is semi structured data. There are sever several different definitions and of course semi structured data what we understand often is data that whose structuring is not rigid and data which doesn't conform to a very rigid structuring mechanism. There are other kinds of definitions as well like data that is inherently self-describing and self-describing data, so with no rigid schema which basically implies there is no rigid schema, the data itself defines its schema and that is known as semi structured data. And again data which are generated half hand and without planning and so on, there also called semi structured data.

(Refer Slide Time: 38:29)



Now in today's world semi structured data is getting more and more prevalent and data structuredness is becoming much harder to impose and define an impose. For example what is the structure of the world wide web, I mean the world wide web is the huge data store but without any structure, in fact but one cannot even say that it is an unstructured data store because there is some semblance of structure that is present like one can think of some meta data tags that are available for each HTML text, some HREF hyper link references and so on and directories and so on and so forth. But on the whole it is not possible to define specific structure and then impose the structure on the world wide web.

So the world wide web is the best example for huge semi structured data store and most semi structured data stores are characterized by rapid or rapid changes or very frequent changes to the data set. So not only is the data not defined apriori or the structure of the data not defined apriori, it is also changing dynamically or it's also changing rapidly. And it is not able to, it's not possible to formalize semi structured data using a nice formal model like the relational model and the best data structure that's used to formalize a semi structured data are usually graph structures.

And there are several different examples for semi structured data like web information systems and digital libraries or even data integration from heterogeneous data sources can be considered to be a semi structured data problem where there are several different databases that are already defined. And there are so many such databases that it becomes impossible or impractical to be able to impose a common structure over all of them and it is easier to treat them as a large semi structured data store. So the very common example of semi structured database is of course the internet movie database which we have been seeing continuous examples of when we are talking about XML as well.

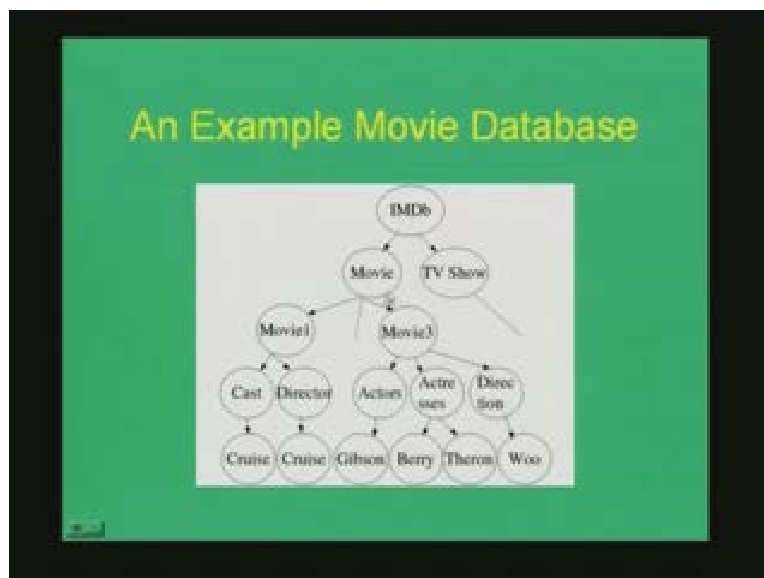
(Refer Slide Time: 40:54)

IMDb – A Motivating Example

- The Internet Movie Database is a classical example of a collection of semistructured data
- Although the information pertaining to different movies may be essentially similar, their structure may be different!
- Let us consider an example movie database

So imdb is a classic example of a collection of semi structured data and even though I mean what makes a semi structured? The fact that even though it just stores information about movies, each movie is different from the other. Each movie may belong to a different journal and it may belong to different country, some may have language fields, some may have some star cast fields, some may have some other kinds of fields which may not be available in the other records and so on. So let's consider an example from a movie database. Let's say imdb is the database and of course imdb not just stores movies, it also stores information about tv serials and documentaries and other such movie related or movie like data that have been released.

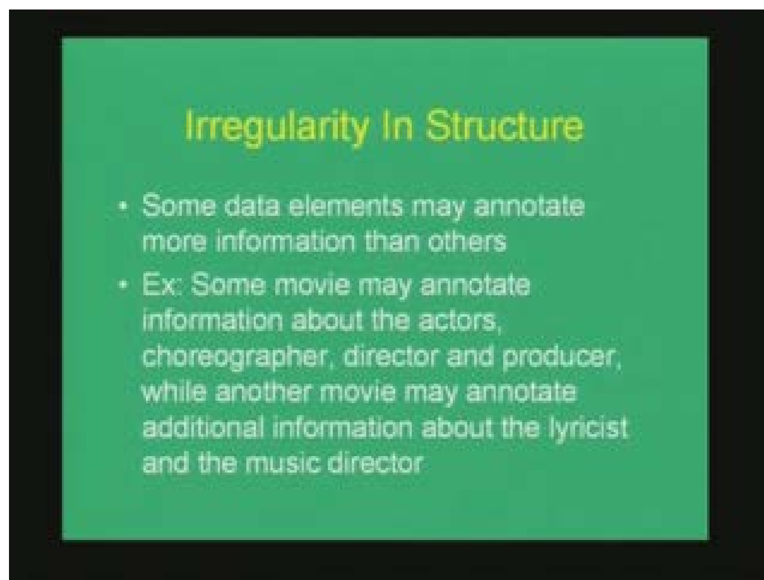
(Refer Slide Time: 41:49)



So even within a movie, let us say even within the movie category different movies could be different, movie one may have information about the cast and the director in the movie and who could be the cast in the director but movie three may have something called actors and actresses, it may make a distinction between who is the actor and who is the actresses and the direction that is rather than just the director, it can talk about a direction team or who directed it and so on.

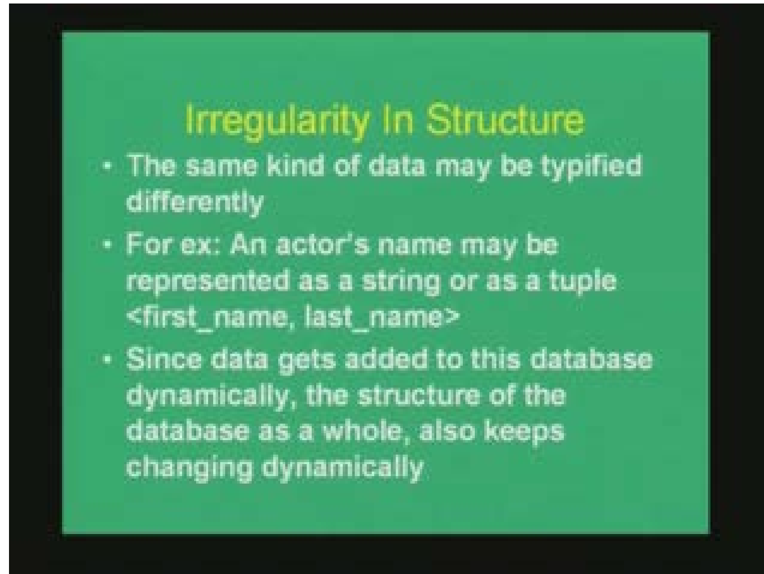
So the structure of each record that makes up a movie element in imdb may be different from one another. And some data elements may annotate more information than others and some may have missing fields and the kind of relationships that exist between each of these different elements may also change, may also vary between different records.

(Refer Slide Time: 43:02)



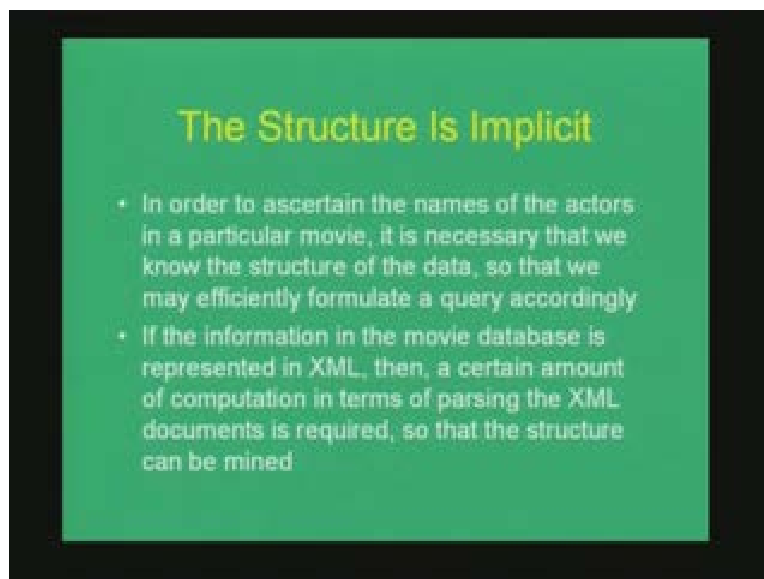
And in addition to that, in addition to changes in structure the way in which data is organized itself could change. For example one might represent an actress name as first name, last name and one might represent it as last name, first name or one might, some other record might have its something like just a name and so on.

(Refer Slide Time: 43:20)



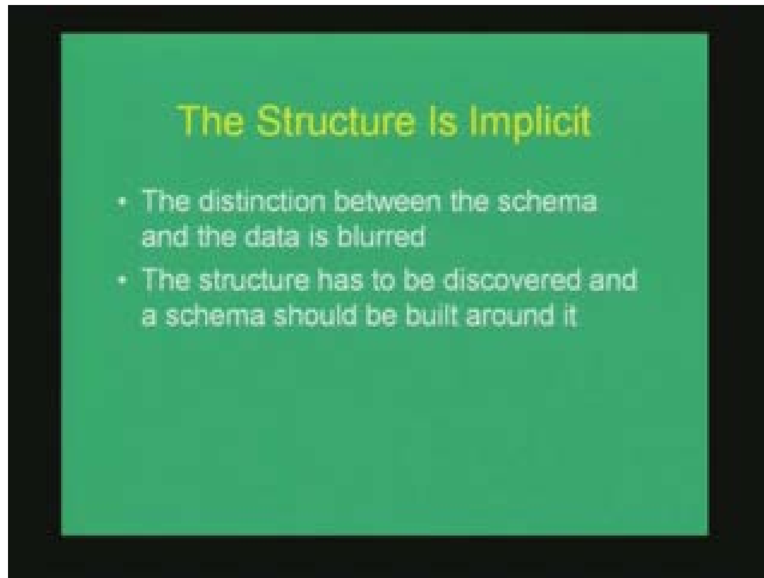
And data gets added to this database dynamically and as a result and dynamically and from different sources from different independent sources. So it becomes difficult to enforce a particular kind of schema restriction on this database. So what is the problem here or what is the main problem in managing semi structured data? The main problem is trying to ascertain what structure or what is the common structure to use for different data sets that are being added to the database and to be able to formulate queries and formulate query languages and so on.

(Refer Slide Time: 44:15)



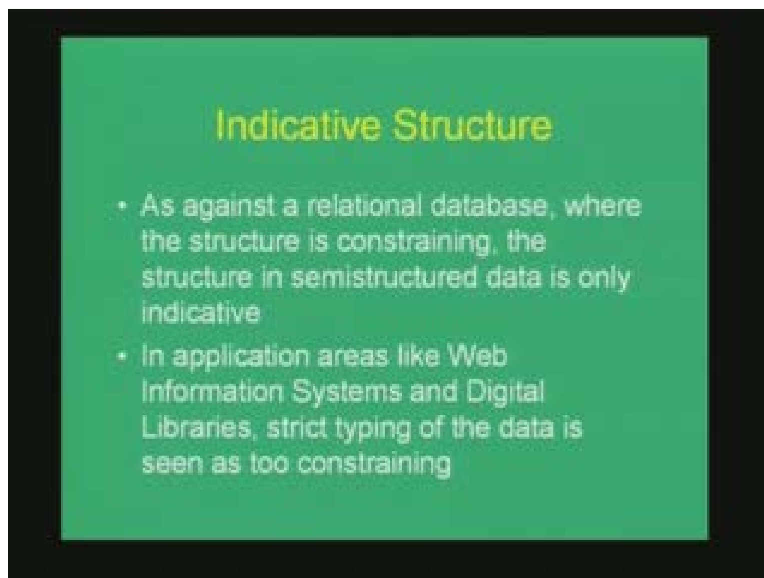
And in addition we should also note that the structure of data element is implicit. So it's not that the user providing the dataset first defines the structure and then provides data according to it but structure is embedded within the data itself.

(Refer Slide Time: 44:33)



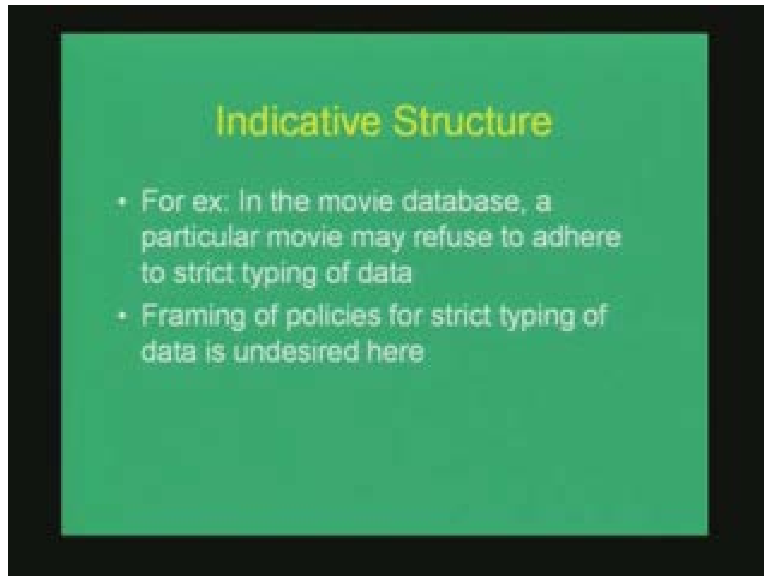
Now the structure has to be first discovered and then a common structure has to be evolved over the entire data set. And this is what is called as the problem of discovery of structure that is the structure should be discovered such that the structure is indicative in nature rather than constraining in nature.

(Refer Slide Time: 45:09)



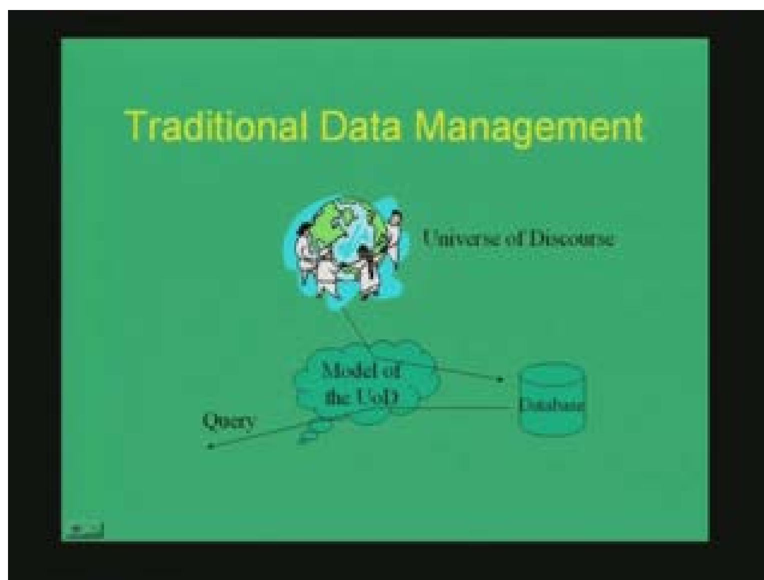
That is the common structure that evolves out of a database should not constrain the database to adhere to a specific structure but rather should be indicative of what kind of data is available in the database and how they are interrelated to one another.

(Refer Slide Time: 45:45)



So here is an example. This slide shows an example of what is the main problem in or what is the main challenge in semi structured data.

(Refer Slide Time: 45:58)



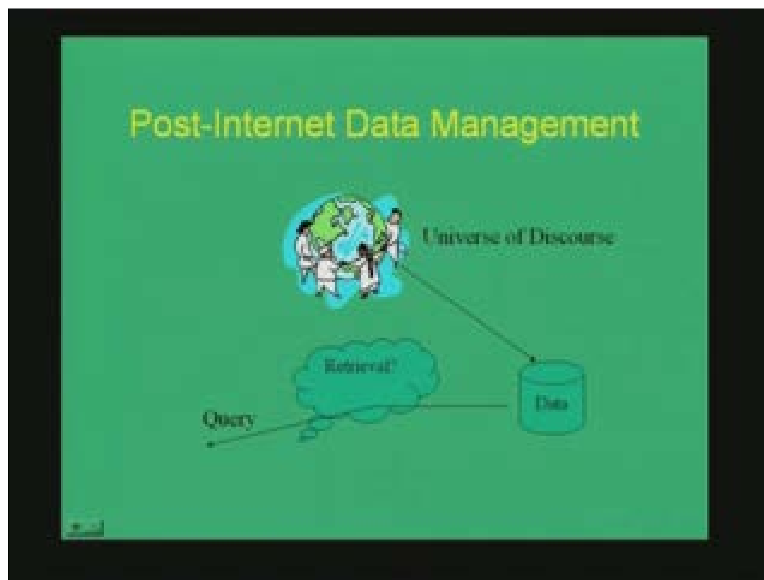
In relational or what may be termed as traditional data management what really happens is you have a UOD or a universe of discourse like company or an academic institute or

university or whatever. And then you have a model of the UOD that is there is the schema that defines how data in the UOD should be organized. It's not ease organized but how the data should be organized and data that is collected from the UOD is first taken through this model and populated into the database.

So when we say that an employee should have a pan number as the primary key and name and dependents and salary and so on, it's only those sets of data that are extracted from the UOD and then sent into the database. And especially for example, if an employee doesn't have a pan number and the pan number is the primary key then it is not possible to add that employee record into the database because the primary key has to be not null, it has a not null constraint and so on.

So constraints are enforced when the database is being populated and the query also is formulated within the model of the UOD. So the query just takes the model of the UOD and queries the database accordingly.

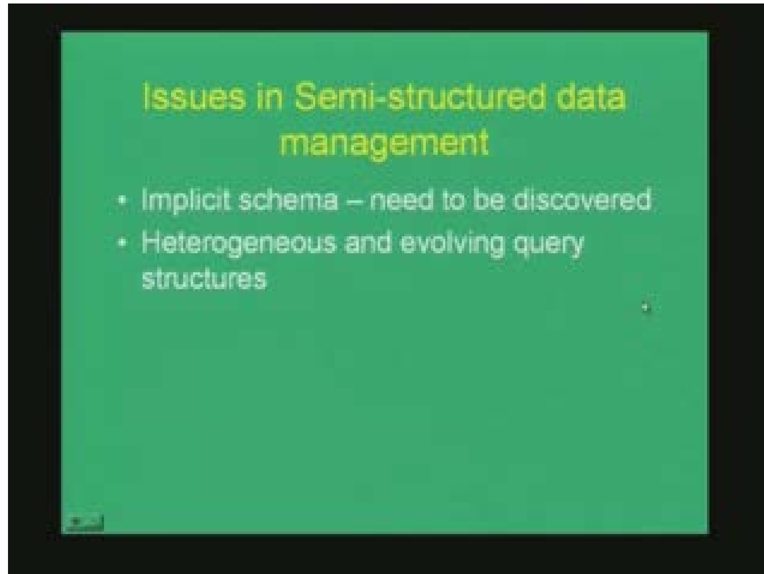
(Refer Slide Time: 47:35)



However in what might be termed as the post internet data management which is the main problem with semi structure databases, the universe of discourse whatever is the universe the world wide web or the internet movie database where users can independently add movie data into the database, the universe directly populates the database.

It doesn't go through any common mental model by which the database is populated, in fact there might may or may not be any mental model here as to how the data is organized but the data is directly populated by the UOD.

(Refer Slide Time: 48:29)

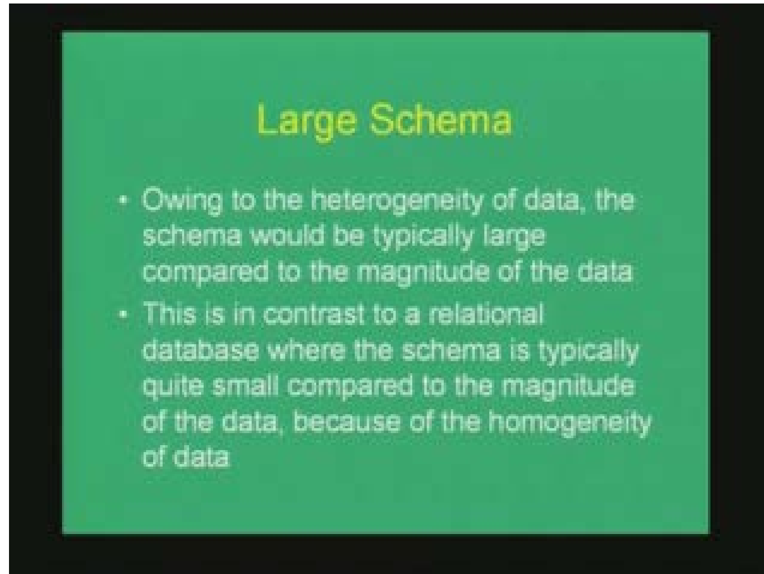


Now the query, when the query is searching the data it should not only know what data to search, it should first try to find out what is the mental model or how is the data organized, what is the schema for this by which the data can be searched. So that is basically the schema discovery problem or the implicit schema discovery problem.

And in addition to that, the schema discovery problem often encounters the problem of what is called as the large schematic structure. That is even when we discover a schema, this is again called the maximalist world notion, in contrast to the minimalist world model of a traditional database system.

In a traditional database system whatever is not allowed by, whatever is not explicitly permitted by the schema is forbidden. So everything is forbidden unless explicitly allowed by the schema. So it's a kind of exclusivist data model where things are thrown away unless they are permitted.

(Refer Slide Time: 49:05)



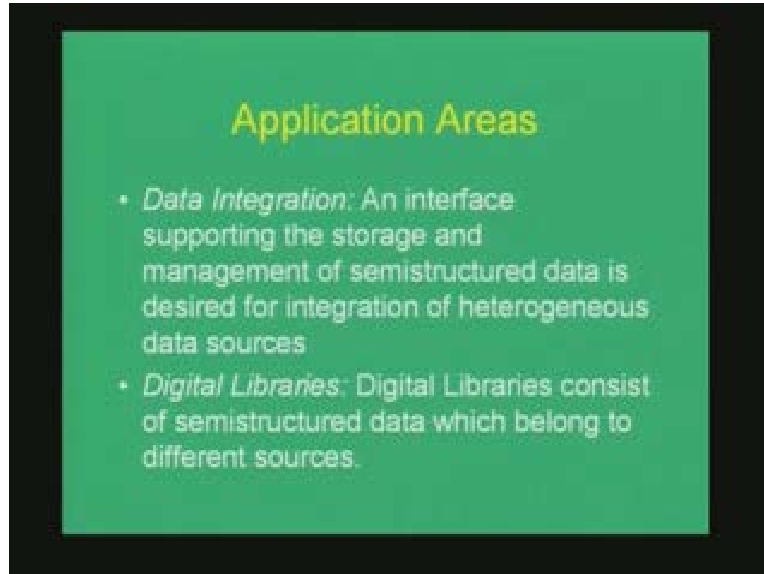
However in a schema discovery process, it's an inclusive model where everything should be permitted unless it is sure that it is forbidden, that is unless it is sure that some kind of a relationship cannot exist, all kinds of relationships between data elements are permitted. So it is not, one cannot apriori define what kinds of relationships exist between data elements unless of course we know that some kind of relationships do not exist in the database or cannot exist in this UOD.

So the associated problem from this is that the discourse schema can be quite large rather than in contrast to a relational schema where the schema is much smaller than the data set itself. so in the internet movie database for example, we might discover that lot of different, we might discover lot of different things that go into a movie based on what people add into the database. And we should allow for all such relationships, unless of course we know explicitly or unless of course we know specifically that some kind of a relationship cannot exist.

For example we might know as a rule, I don't if this is a true but we might know as a rule that in the universe of discourse that in the world of movies, it is not possible for a director to be the boss of the producer or something like that. So the producer reporting to the director or whatever. So unless of course we know that some kind of relationship does not exist, we have to accept all kinds of relationships that are, we might have to accept a dataset about a movie which does not contain any movie star.

We might have to accept a data set where a movie contain 10 different movie stars and so on. So all such relationships should be accepted unless explicitly forbidden. And as a result, the actual schema that is generated is far bigger than a typical relational schema.

(Refer Slide Time: 52:26)



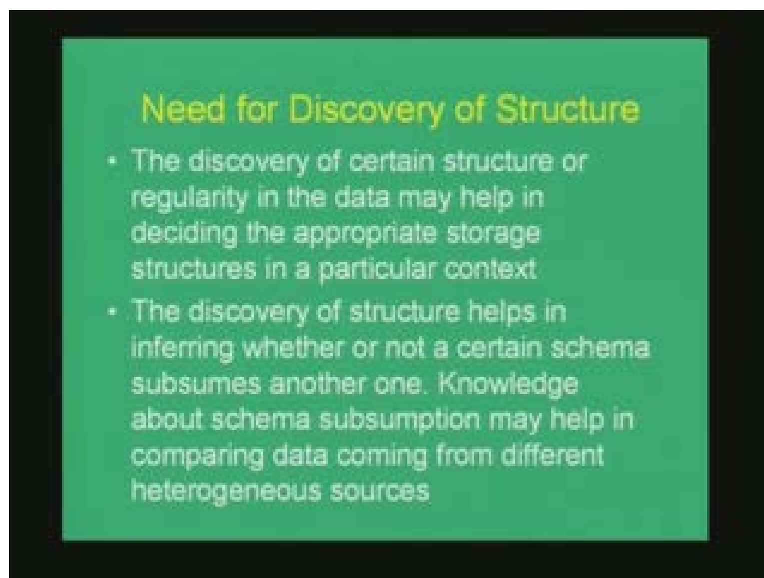
There are several different application areas where this is useful and of course where these have been tried out and these include data integration where you design an interface to integrate different desperate data sources coming from different locations, each having their own schematic structures. And the second major area of application is in digital libraries which consists of again different kinds of semi structured data coming from different sources.

Several more application areas like genome databases or scientific databases that talks about scientific documents, similar documents, citations, references and abstracts and where it was published and ratings and so on and so forth. And of course in E-Commerce applications where the discovery of structure problem becomes very important in business to business systems where each business if it is quite big, it might be difficult or impractical to impose a very specific schematic structure over the entire business house.
So one needs to be able to resort to semi structure data management, when managing B two B business systems.

(Refer Slide Time: 52:44)

So let us skip through these slides where the need for discovery of structures are motivated even more or which talks about how discovery of structures can go about and addressing the discovery of structures problem in itself would take a complete, would involve a separate session and it is clearly beyond the scope of this particular lecture.

(Refer Slide Time: 53:38)

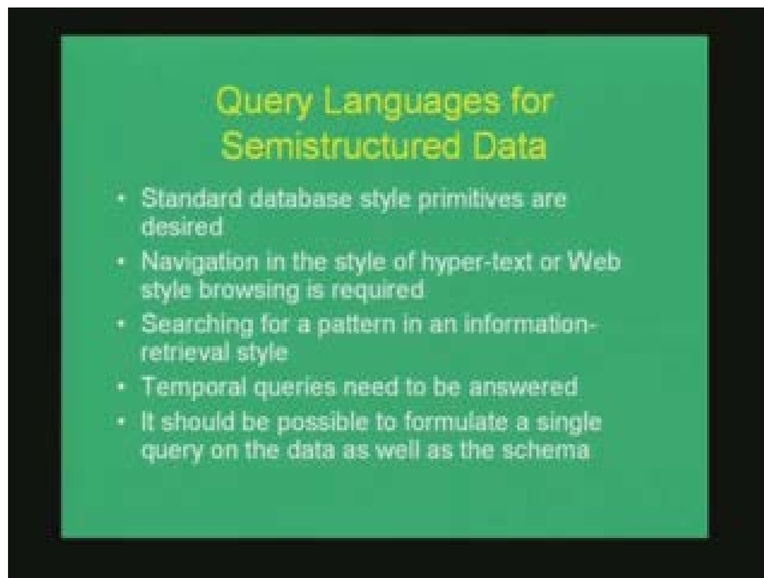


However we can give some kind of thumb rules which talk about how implicit structure can be discovered from a set of desperate data sources. And most of these revolved around looking at some kind of regularity in the data set and then generalizing based on this regularities.

And so several different kinds of data mining and machine learning and artificial intelligence kind techniques are explored for trying to fit a structure on to a data set. And there is also a notions of what is a best fit. A best fit data structure or a schematic structure should not be too general and should neither be too specific and so it has to be, the structure discovery process should be able to generalize based on whatever examples were encountered while passing through the data.

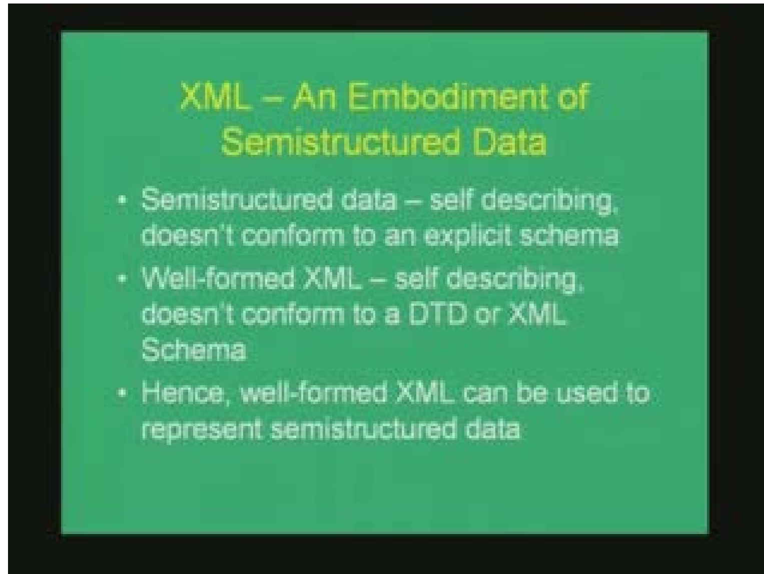
However it shouldn't be too general in the sense that it can accept anything or any structure in the data set. That is it should also identify what are the forbidden relationships among data elements in addition to what are all the possible relationships among the data elements.

(Refer Slide Time: 55:41)



And several different kinds of query languages are also supported for semi structured data in addition to XQuery which is primarily meant for XML databases and of course keyword based searches which are useful for, which are useful for full text searching. There are other kinds of primitives like navigation based queries or searching for patterns or temporal queries based on how particular data element evolves over time and so on.

(Refer Slide Time: 56:16)

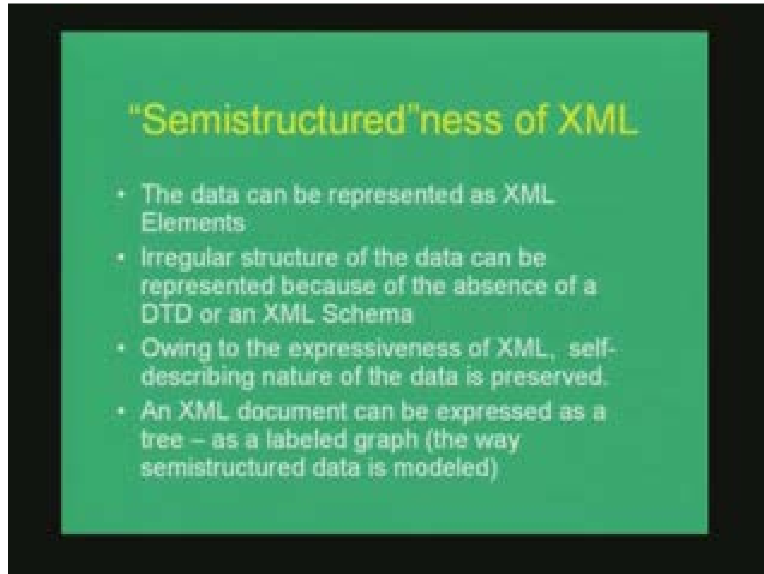


So XML is an embodiment of semi structure data in the sense that XML is the natural choice in which semi structure data can be organized.

(Refer Slide Time: 56:27)

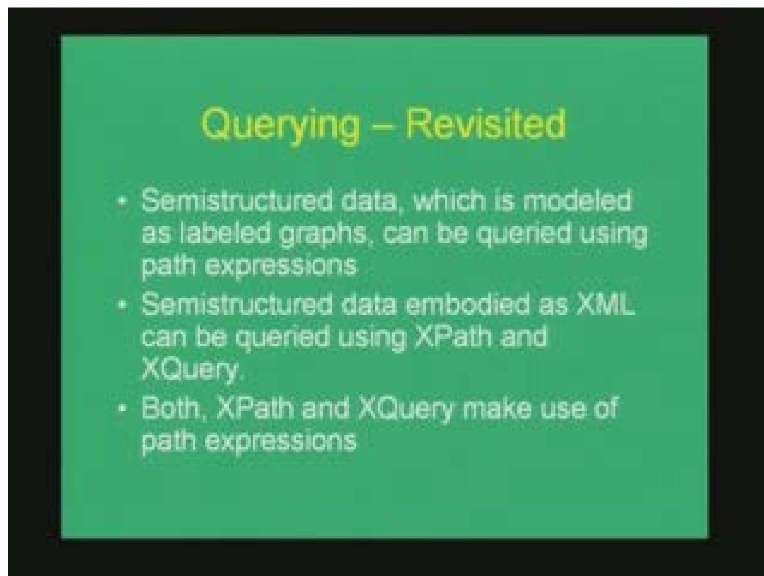
And the problem of discovery of structure over XML, over semi structured data can reduced to discovery of a XML schema given a desperate set of XML documents.

(Refer Slide Time: 56:27)

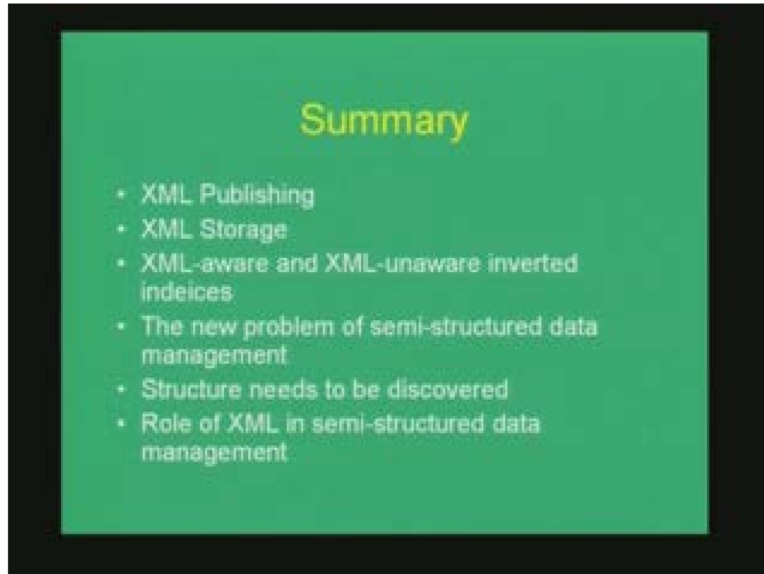


And queries can be revisited using XPath and XQuery expressions based on whatever structure that have been discovered.

(Refer Slide Time: 56:43)



(Refer Slide Time: 56:51)



So let us summarize whatever we learnt in this session and of course the idea of semi structure data itself is a vast ocean and it is beyond the scope of this lecture to explore all of them. So therefore we looked at native XML storage and XML publishing and different kinds of storage structures that have been proposed for XML and mainly we touched upon the problem of the semi structured data and the larger problem of discovery of structure which is very important for semi structured data management. So with that we shall end this session.