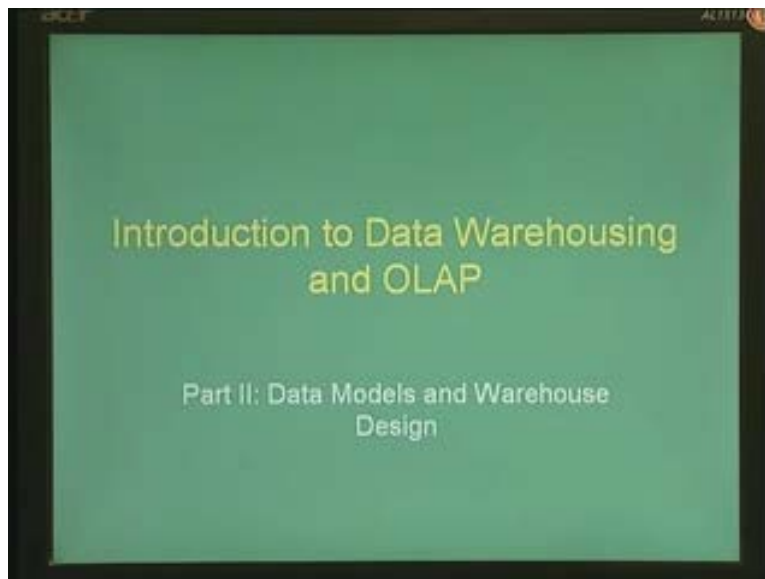


Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 31

Introduction to Data Warehousing and OLAP – Part 2

Hello and welcome back. In the previous session, we were looking into specific kind of databases kind of any sort or any kind of databases mainly meant for managing historical data or archival data, namely what are called as data warehouses. That is we are looking at how the database design changes if the application domain is changed from an operational data management that is operational data management to data management for strategic purposes that is asking queries that are of a strategic nature.

(Refer Slide Time: 01:41)

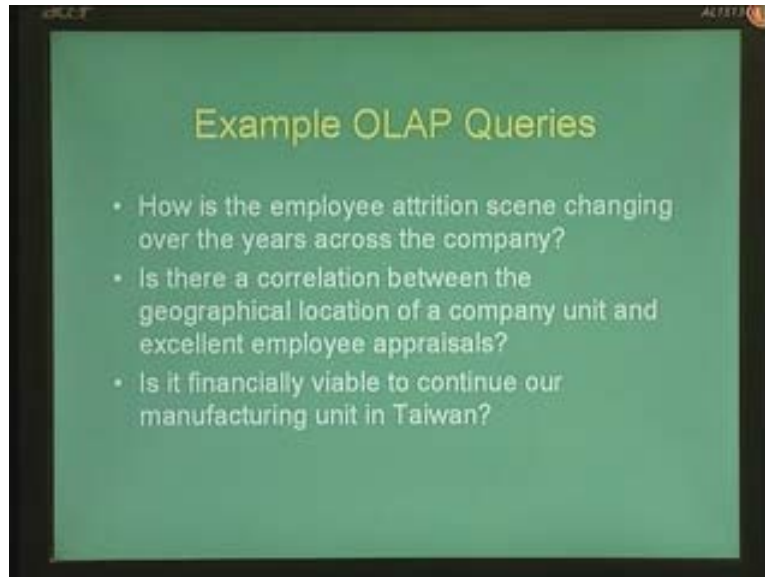


And we saw that, we saw the overall architecture for a data warehouse were you have to populate a data warehouse from different, several different data sources mainly the OLTP data sources and these data from these sources are routed or source through a data cleaning and integration engine where we saw what is really meant by data cleaning. That is there are several different sources of dirty data being creeping in different kinds of inconsistency, different kinds of missing values, spurious abbreviations and so on. And we need to clean those data elements before populating it on to the data warehouse and of course the data elements are to be integrated under a common schematic structure before the database can be populated.

And we have of course things like back flushing of clean data back in to the OLTP databases and we have feedback from the data warehouse to the cleaning engine and so on.

Let us move forward in this exploration of data warehouses to see what forms the core of the data warehouse. What kinds of data models are used to the core, what kinds of index structures or what kind of storage structures and such things are actually applied to the core of the data warehouse.

(Refer Slide Time: 03:56)



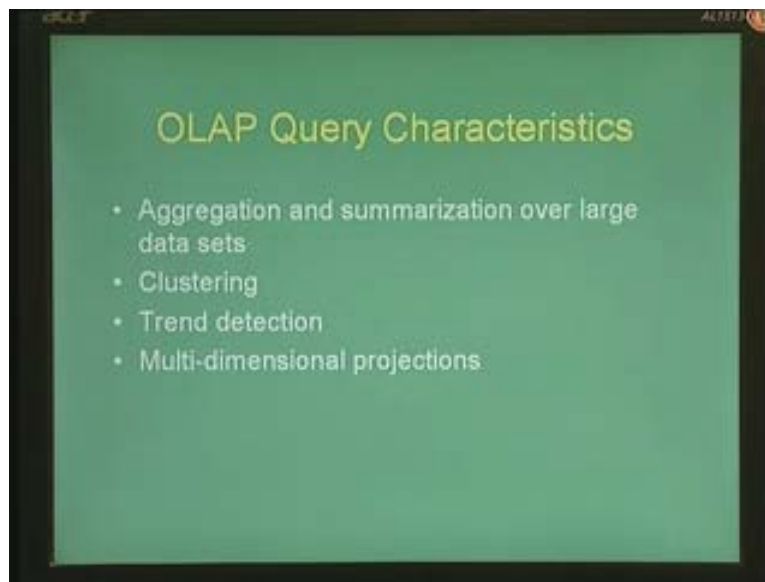
So in this session we shall be concentrating on data models and warehouse design. To begin with let us have a look at some examples OLAP queries like we saw in the previous session. Have a look at this OLAP queries something like which says how is the employee attrition scene changing over the years across the company or is it financially viable to continue our manufacturing unit in Taiwan. Now the thing is a characteristic nature of these queries is that most of them involve huge amounts of aggregation. That is we need to span or we need to read huge amount of data elements before we are able to conclusively answer a query.

Look at the first example. How is the employee attrition scene changing over the years across the company? See we need to detect the trend here. That is we need to detect the trend or employees more likely to quit the company now or employees more likely to stay or they happy or they unhappy or they finding better opportunities away outside and so on. Now in order to answer this question, we need to take employee data from across the company that is different data sources how many ever branches I have, we need to collapse such data from across the company and across the period of time. It's not sufficient if I just look 1 year data, 2 years data. We need to collect data over a period of time may be 5 years 10 years or something like that and aggregate it across employee behavior across the entire company and then we get a result something **which something** of the form which says that employee attrition seen or employee attrition rate is so and so percent.

That is given whenever an employee joins, there is so much of a probability that he is going to leave in say 1 year or 2 years or 3 years and so on and so forth. So just to answer this small query that is just to get the small set of percentages, you see we need to actually go through the huge amount of data that's present in the data warehouse. So what are some of the typical characteristics? One, we saw of course aggregation that is we have to aggregate over a large set of data and summarize results over a large data set.

We need to also often cluster data based on different parameters, is there a correlation for example in the previous slide we saw some queries talking about correlation. So suppose you want to find out correlation then we want to, we may actually want to find out clusters along a particular dimension and or along a particular set of dimensions to see whether there is actually some kind of correlation for some values in the dimension to some kind of behavior that we are searching for.

(Refer Slide Time: 06:03)

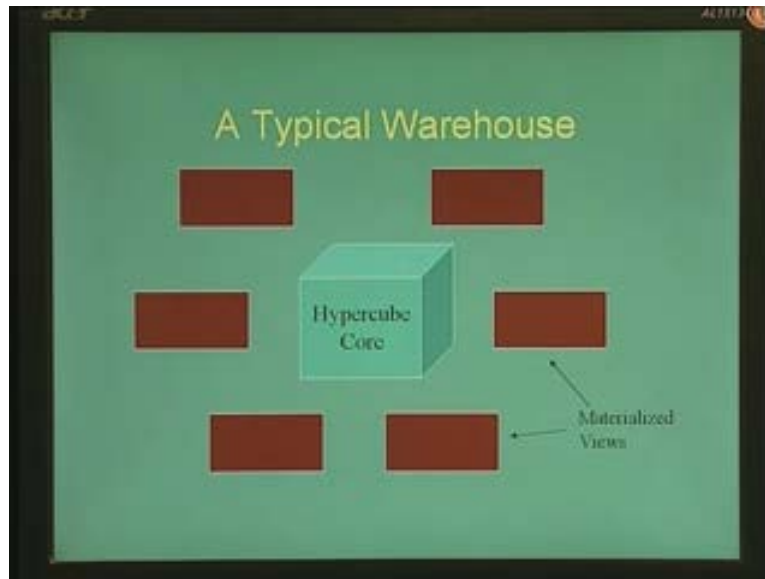


There is also trend detection that it is common in OLAP queries where you need to, you need to in some sense integrate over time. That is you need to start from the earliest source and start working your way through the through till the latest source of data and project the temporal behavior of the data elements along certain dimensions and say this is the trend that the trend is increasing towards this thing, the trend is decreasing, the trend is fluctuating and so on and so forth.

Then there are what are called as multi-dimensional projections that is a behavior could be attributed to several different dimensions and we may want to ask how each dimension contributes to certain behavior. How is the total sales dependent upon say the location of my branch, let us say I am having several branches, a retail branch across the country and I want to see how sales are correlated to branches or how sales are correlated to different events like say some festivals and some kinds of national holidays and so on or the weather is it, the sales is more in summer or winter or what and so on.

So you may have to actually project these facts, this fact called sales across several different dimensions and this has to be performed as efficiently as possible no matter what the dimension is. This slide shows the typical design of data warehouse at the core of the data warehouse say as you can see in this slide, there is a kind of something typical about this slide. That is there is a core here and a lot of other surrounding elements here. The core is what is called as the hyper cube of or also called as the data cube and several other names which maintain facts that are sorted across several different dimensions.

(Refer Slide Time: 08:29)



For example I may want to store sales data across several different dimensions like say the branch dimension, the year dimension, the event dimension, the product dimension, which product has the most sales or which branch has the most sales or which year has the most sales or which event brings in the most sales or which, whatever I mean product or the category of products and so on and so forth. So we will look shortly in to how this hypercube itself looks like or how we can design such hyper cubes and that basically stores facts that are contained in the data warehouse.

However maintaining purely just the hypercube would make the data warehouse pretty inefficient. This is because again the good old as aggregation problem that is even if there are facts that are stored efficiently in a hypercube. We still require to access let us say tens or thousands or probably millions of different facts before we are able to answer a query, before we are able to answer a query like is it financially viable to continue this branch or whatever. We need to look at several different facts before we will be able to come out with the response.

So in order to speed up such processing, there are number of what are called as materialized views. Remember the notion of views from traditional data base. What is a view? View is basically a logical substructure of the overall schematic structure. That is all of these materialized views are in some sense sub cubes of this hypercube but in

traditional databases, views are essentially virtual tables. That is you store the view as a query not as a base table that is not physically in the database. So whenever you're querying a view, you first have to execute the view that is generate the view first on the fly and then query the view. But, here that completely defeats the purpose of what we are looking at essentially speed up in performance that is query response times.

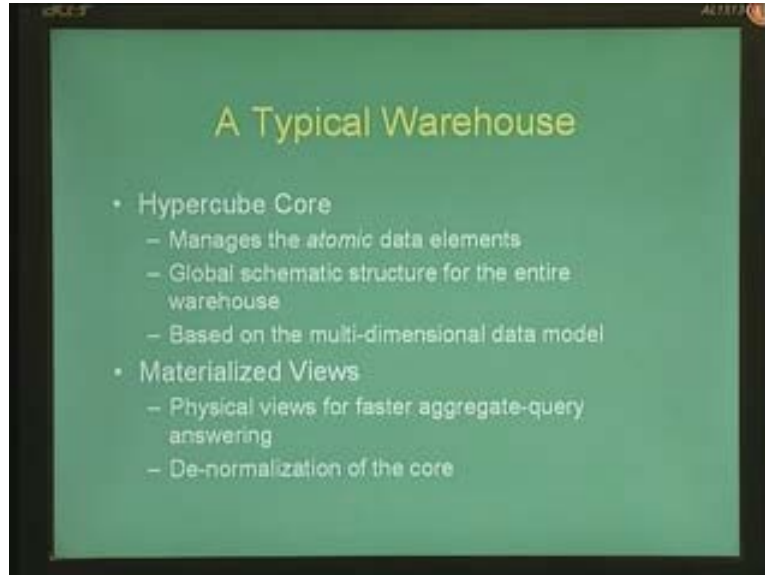
So these kinds of materialized views are actually physical views that is views which are physically computed from the core, from the hypercube core and stored on to disc. These could be something like, some kind of partial aggregate information, something like let us say I have sales data across all branches. Now I might want to store sales data for a particular product. Suppose I have a retail store, I may want to store sales data for let us say some kind of vanaspati oil or whatever across all different branches. I may want to store partial aggregate information like sales data across quarters, across different quarters in a year and so on and so forth. So **such kinds of** such kinds of data which are queried often are materialized from the hypercube core and stored as materialized **piece**.

Now I am sure you might be wondering, why this would make sense, why don't we do that in traditional databases. Because, we cannot do that in traditional databases because it hampers the concept of normalization. That is materialized view is essentially creating extra functional dependencies which over and above whatever is present in the core itself. Now what happens if there are extra functional dependencies? Maintenance is a big problem, that is whenever an update happens because there is redundant information or derivable information or maintaining all those derivatives of information becomes a huge problem.

For example if I have a particular amount of sales data for a particular product in a particular branch and let us say tomorrow I update this data element with a new number and this automatically affects all aggregate information that I am storing here. What is the aggregate information for all products in this branch or for this product in all branches and so on and so forth. So maintenance of materialized views is a problem. That is whenever the databases or data warehouse is updated the materialized view should also be updated.

However this is not too much of an issue in data warehouses mainly because I am sure you would have got, what is the reason because the number of updates is far less frequent than in a typical OLTP set up. In an OLTP set up that is in an operational database there are frequent updates and queries. The number of updates is comparable to the number of queries but here the number of queries is far more than the number of updates. The number of updates is very infrequent, possibly of the order of once a month or once in 6 months sometimes and so on where you can actually afford to bring down the warehouse system, the OLAP system, update the OLAP engine with the new data it has come in and then compute all the materialized views.

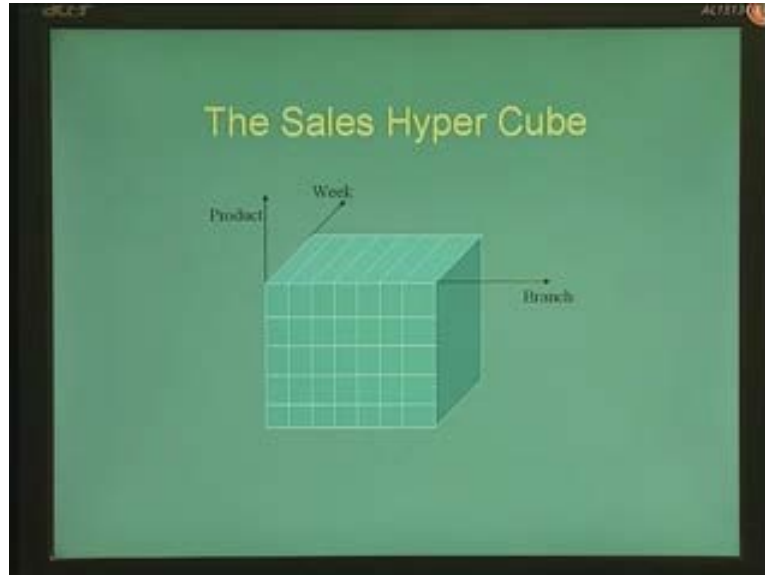
(Refer Slide Time: 14:32)



So this cube, this slide essentially summarizes what we have just talked about that is there is a hypercube core that makes of the core of the data warehouse. And the core manages what are called as atomic data elements that is data elements which cannot be sub divided into other kinds of data elements. And it's the global schematic structure for the entire warehouse. Remember this global schematic structure is what we have derived from the different OLTP sources. And this is based on the multidimensional data model and we will see mechanisms by which we can actually implement it in practice.

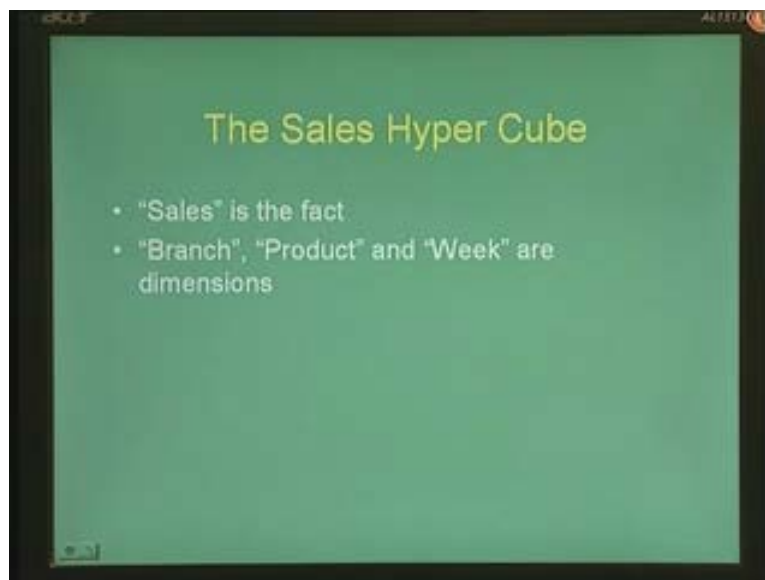
And materialized views are physical views for faster aggregate query answering and it is, it essentially amongst to de normalization of the core. And this de normalization is performed to increase performance, is done to increase performance rather than updation and updation is an issue **which** but because updation is relatively infrequent, it's not too much of a problem. So let us look at the hypercube core again in a little more detail. This slide here shows a particular hypercube having three different dimensions. Of course the reason why we are calling it as a hypercube is that it can have any n different dimensions need not just be 3, it could be 5, 10, 25 whatever. So it could be any different dimensions.

(Refer Slide Time: 15:27)



Now as you can see here each dimension talks about specific aspect of the data that we are storing. Let us say this is the sales hyper cube, the amount of sales that has happened. Now this dimension stores the amount of sales across each product and this is across each week and this is for each branch. Therefore each cell in this hyper cube refers to sales of a particular product in a particular branch in a particular week. So in this hyper cube, the constituent of the cells that is the sales data is what is called as the fact that is stored by the hyper cube.

(Refer Slide Time: 16:21)

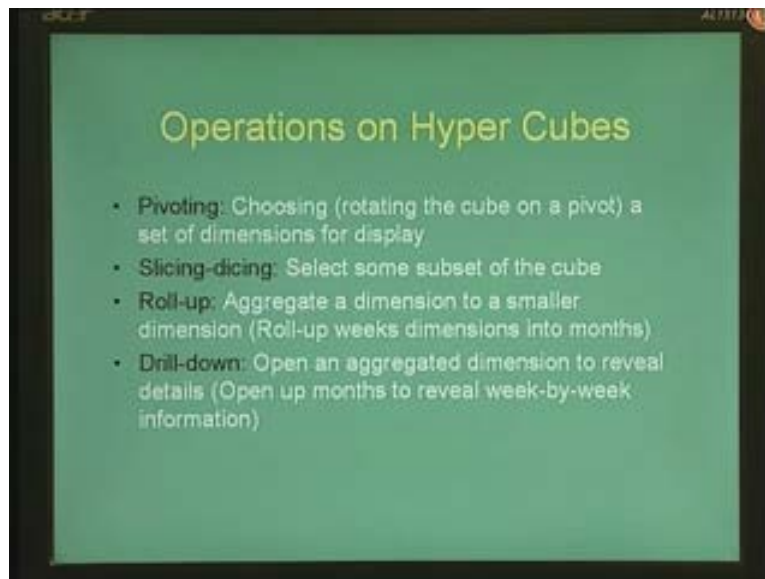


So the hyper cube manages the fact called sales and branch, product and week, along which the sales are projected, are what are called as dimensions. So the sales fact is projected across three different dimensions. In addition to normal relational database operators or rather in lieu of usual relational databases operators, there are several other operators that are defined on the hyper cube itself. Here are some of the operators that are commonly used for data warehouses.

Pivoting: pivoting essentially means, what do you understand by the terms pivoting? Essentially it is rotating the cube given a pivot that is suppose you want to see the sales data projected across some 3 different dimensions or some 4 different dimensions or whatever. We choose this set of dimensions so that these can be displayed on the visualization engine. Data warehouses or OLAP engines usually come with a, usually have a huge visualization counterpart which helps the user to visualize the inherent knowledge contained in the database or in the data warehouse in different ways. That is one way is that of pivoting the warehouse across different dimension.

Then there are slicing and dicing operators that is selection of some sub set of the cube. You can think of it as slicing the cube across different dimensions or dicing that is forming a sub cube out of the larger hyper cube and so on.

(Refer Slide Time: 17:16)

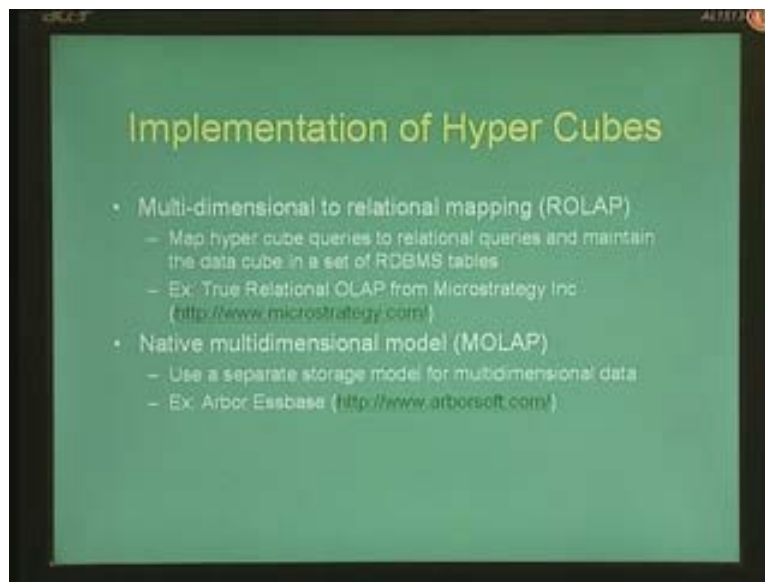


Usually data cubes also supports this role up operator were you can aggregate information on one or more dimension. For example we may want to roll up the weeks data into months data that is in the previous example (Refer Slide Time: 18:28) we have data per week. Now instead of data per week, we might want to collapse certain sets of dimensions like this in order to form month wise data. So that means we have rolled up in the week dimension across one aggregate level so that the week has now become months and so on.

Similarly the opposite operation of roll up is a drill down operation where given an aggregated dimension, you open it up in order to reveal more details. That is open up months to reveal week by week information or open up weeks to reveal day by day information and so on and so forth provided that exists in the database, in the data warehouse.

How are these hyper cubes implemented in practice? In practice there are two major approaches towards implementation of hyper cubes, what are called as ROLAP and MOLAP approaches. ROLAP essentially means the relational OLAP that is where the multi-dimensional hyper cube is transformed by certain set of transformation rules to the relational model and vice versa. That is the relational model is transformed back in to hyper cubes for answering questions. So every hyper cube queries are mapped on to relational database queries and vice versa.

(Refer Slide Time: 19:23)

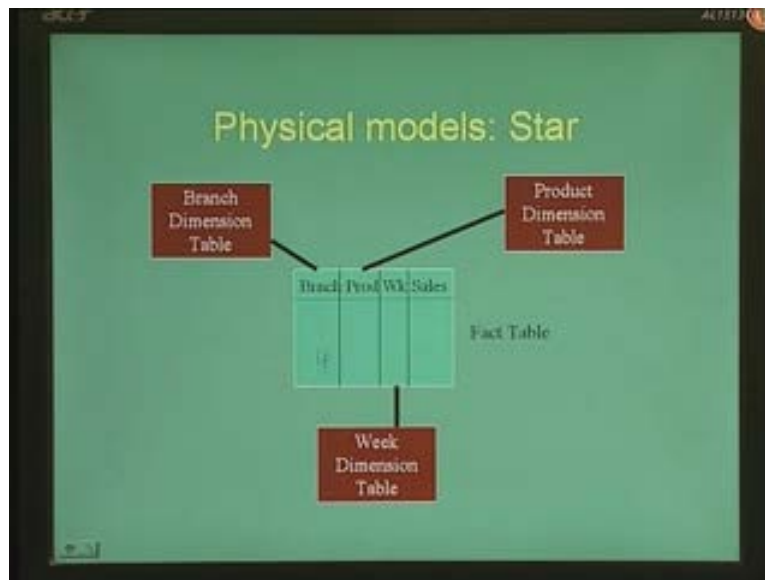


And the other kind of implementation strategy is called MOLAP or naive or native multidimensional model where the storage structures and index structures are redesigned for storage of multi-dimensional data itself and which has got nothing to do with the relational model in itself. So when you provide a data warehouse query, a MOLAP query it is directly translated on to disc accesses and further such low level query primitives and answer directly.

So there are, this slide also shows some examples like say true relational OLAP, it's from micro strategy which is an example of a ROLAP and Arbor Essbase which is an example of a MOLAP system. Let us look at ROLAP systems in a little bit more detail that is how are ROLAP is actually implemented, how is a hyper cube implemented, how is a collection of hyper cubes implemented and so on.

The most fundamental form of implementing a hyper cube using a set of tables is what is called as the star schema structure or the star schema architecture. The star schema architecture is shown in this slide here. As you can see the name star schema comes from the way that the schema is organized. That is the schema has the central fact table and the fact table obviously stores facts. That is their sales data for which is the fact which is stored for all possible combinations of branches, branch products and weeks.

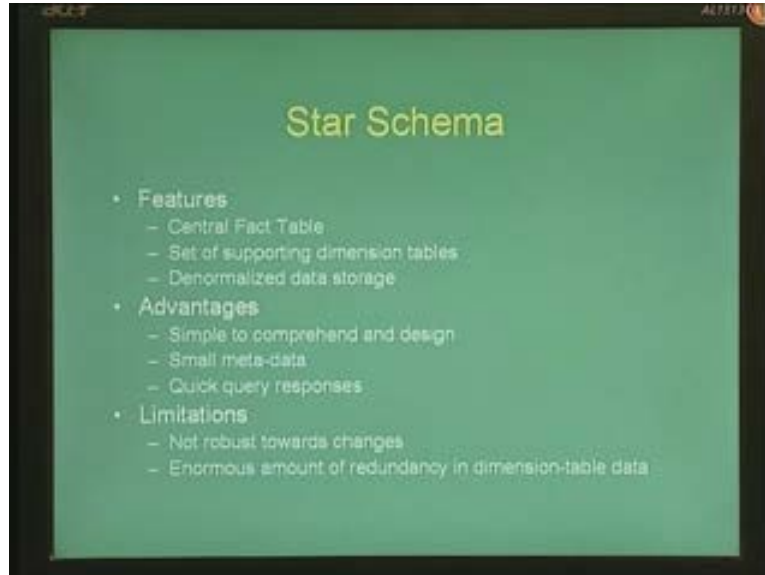
(Refer Slide Time: 22:16)



That is branch one, product one, week one sales so and so. Branch one, product two, week one sales and so on. And each of these columns here that is branch, product and week contain ids which are foreign keys in to the respective dimension tables. That is there is a branch dimension table which contains information about branches. So if branch number one is stated here, this would be a foreign key here which would give details about what is branch number one that is the name of the branch, the address, the manager, whatever and so forth. And similarly product, each product is a foreign key here which gives details about each of this product and so on. So this is the most simplest form of representing multi-dimensional hyper cube in a relational data model.

Now let us briefly summarize what a star schema is about. There is a central fact table and there are set of supporting dimension tables that is in the previous slide, the green table is the fact table and the other three brown tables are the dimension tables. And the advantages of a star schema, it's pretty simple to comprehend and also to design and there's very small amount of meta data that's required, small numbers of foreign key relationships that have to be maintained and its very quick for query responses as long as the queries are on the facts that are stored in the database.

(Refer Slide Time: 23:38)

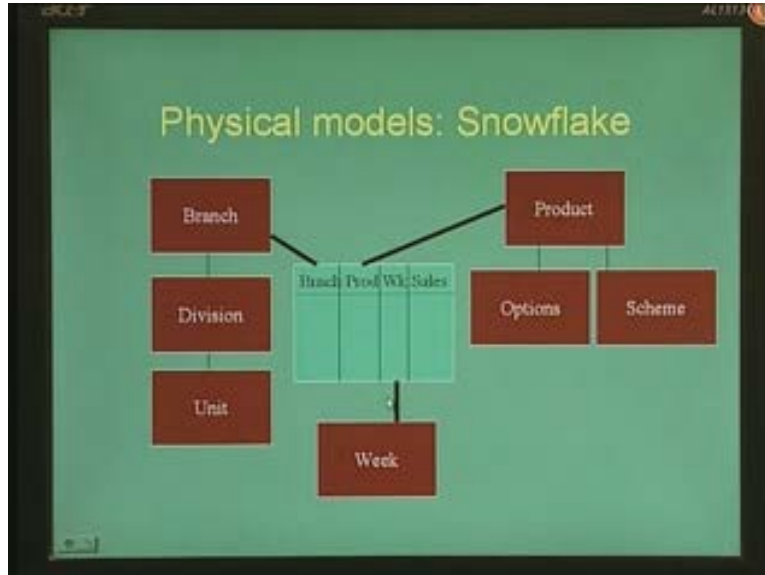


However there are certain limitations as well. That is its not robust towards changes especially in the dimension tables. Suppose if I want to add something to the dimension table, it's quite difficult and there is an enormous amount of redundancy in dimension table data because dimension tables need not be normalized and we may just be repeating information in the dimension table.

The next kind of relational model that is used is what is called as the snowflake schema. The slide here shows an example of the snowflake schema. Well, it doesn't look like a snowflake in the slide but you can imagine why it is called as snowflake schema. That is it is a star schema that is augmented with something more. That is the branch dimension here itself consists of 3 different tables. The product dimension here consists of 3 different tables and the week dimension remains as it is.

What are these three different tables? These are basically normalized data structures or normalized versions of the branch dimension table. So if the branch contains several information the manger information, the location information, the turnover information and so on and so forth, you may want to essentially break it up into different table so that accessing data about branches becomes more efficient, similarly for other products.

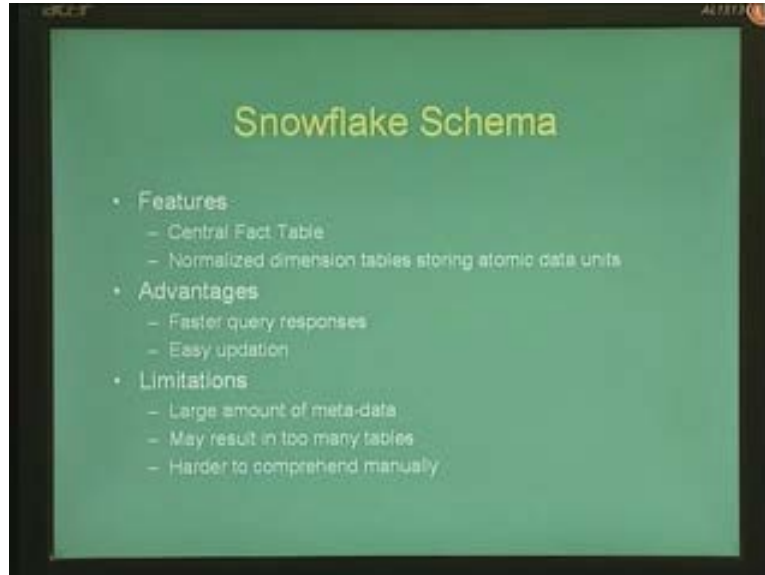
(Refer Slide Time: 24:14)



And the result is that each such dimension here has sub dimensions which kind of gives the gives an overall structure like a snowflake where there is a central fact table and branching out dimensions and so on. So the features of a snowflake schema detail here that is like always there is a central fact table as in the star schema. And however rather than just dimension tables, in storing de normalized or un normalized data it's not of de normalized it's more of un normalized data.

In each of these dimension tables, its actually dime normalized data that is stored in the dimension tables. And what are the advantages of this thing? The query responses are faster, especially on the dimension tables. We don't have to work search through the dimension tables for searching on a specific queries because it's normalized and you have several foreign key relationships and index structures and so on. And it's more easier to update especially on the dimension table but there are certain limitations as well. That is there is large amount of meta data in fact the because we are talking about aggregated data that is archival data, we might very well end up with a large amount of tables especially when we try to normalize the dimension tables.

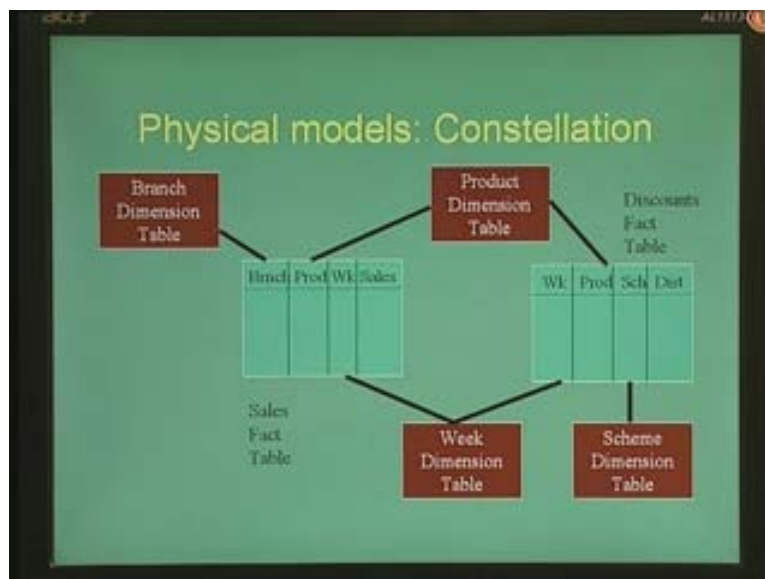
(Refer Slide Time: 25:27)



So a given branch information may create possibly hundreds of different normalize tables, if we have to normalize it to say let us say fourth normal form and **it becomes** as a result it becomes much harder to comprehend when we are looking at it manually, when we are looking at the overall schema manually.

The third kind of ROLAP model or the relational schema model for implementing multi-dimensional structures is what is called as the constellation and this is perhaps the most used model, most practical model because it contains multiple fact table not just one fact table.

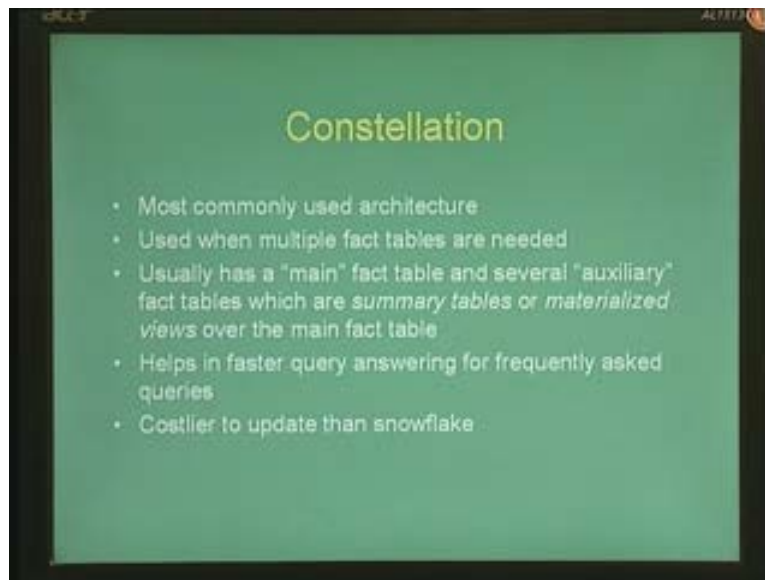
(Refer Slide Time: 26:40)



And in a data warehouse, we usually require to have multiple fact tables that is let us say each fact table corresponds to one data mart. So one data mart stores information about sales, the another stores information about salaries, the other stores information or facts about procurements and so on and so forth or cause and so on. So when we have several such fact tables, we may want to actually, we may want to actually have several stars schemata intermingling among one another. And in addition, suppose whenever we have some kind of materialized view for example let us say this is a sales data and this is some kind of a materialized view that talks about aggregated sales data across different or it just talks about the discounted sales that is sales which have been made whenever discounts were offered and so on.

So it is some kind of a materialized view that has been materialized from the main fact table. So as a result what happens here is this fact table here shares some of the dimension tables from the main fact table, from the sales fact table. So as a result you have a dimension table actually having foreign keys in 2 or more fact tables and 2 or more fact tables having foreign keys in to this dimension tables.

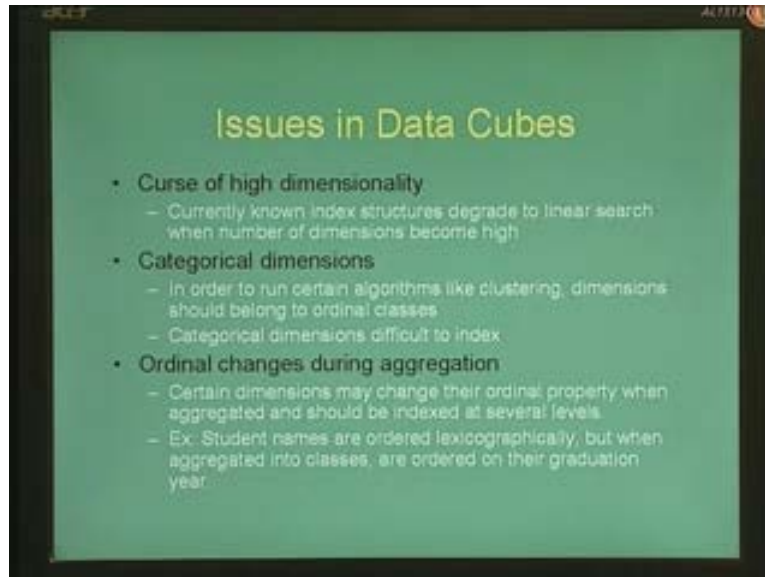
(Refer Slide Time: 28:47)



So the result looks like a constellation of different stars that is the different stars and they are staying together so that it forms a kind of a constellation. So the constellation is the most commonly used architecture and used when multiple fact tables are required. And usually **not** and that is even when multiple fact tables are not necessary or constellation schema is used in order to maintain materialized views in addition to the central fact table. That is it usually has a main fact table and several auxiliary fact tables which are materialized views over the main fact table. It helps in faster query answering for frequently asked queries, however it is costlier to update than a snowflake schema which is kind of obvious in this.

What are some of the issues in data cubes? There are certain especially managing data cubes in a ROLAP architecture that is using a relational data model there are certain issues which in some sense refuse to go away.

(Refer Slide Time: 30:20)



That is they can be sometimes so severe that it may render, it may render the data warehouse useless. That is it may not able to provide interactive response times which is the most important factor for a data warehouse. One of, probably the main such challenge is what is called as the curse of high dimensionality. There is several known index structures by which we can index on different tables or on different dimensions as efficiently as possible but most of such known index structures degrade in performance, degrade to almost linear search in performance when the number of dimensions become high.

So with a large number of dimensions, searching efficiently for a particular data element becomes really difficult. Then they are what are called as categorical dimensions. It may not be possible to order the elements of dimensions across a given scale. For example week it is easy to order. Let us say I have a week information, so I can say week 1, week 2, week 3, week 4, week 5 and so on and there is an implicit ordering that is we know that week 3 should come after week 2 and before week 4 and so on. It's not possible for week 3 to appear after week 5 and so on.

So there is an implicit ordering information. Now this ordering information is sometimes exploited for faster retrieval and computation of clusters and so on. But then there are certain kinds of dimensions which are categorical in nature, not ordinal. For example products or say branches, products let us say vanaspati oil, rice and ghee and so on and so forth.

Now which comes first and which comes second? Should oil come before ghee and should ghee come before rice or the other way around. There is no such ordering that is given for the elements in this dimension. So, categorical dimensions becomes difficult to index. We look at some index structures later on where we will see that why it becomes difficult to index categorical dimensions. We cannot identify a region and say this data element falls between this and this region because there is no ordinal information about the dimensions. And even when there is ordinal information about dimensions, sometimes ordinal classes vary whenever we aggregate. For example this slide shows a given example here (Refer Slide Time:32:38), let us say at the lowest level, we are having names about different sets of students that have come through a given university and passed out and so on.

So we have ordered the set of names, student names according to lexicographical order. However when the set of students are aggregated into classes, let us say batch 1, batch 2, batch 3, batch 4 and so on. Then the ordering becomes, the ordering is now based on their graduation year, the batch 1 graduated in 94, batch 2 graduated in 95 and so on and so forth. Now it's no longer lexicographical ordering. So the same index structure that is used for the lower level dimension cannot be used when the data cube is rolled up. That is when we have aggregated the set of students to a set of batches and so on.

(Refer Slide Time: 33:56)



When we are designing a data warehouse there is another, there is a very important dimension called the time dimension which probably needs to be addressed separately and that's why there is a separate slide on the time dimension because there is certain properties about time which other dimensions may not really adhere to, may not really hold.

The time dimension as you see is usually mandatory in most warehouse applications otherwise that the whole idea of archival data or historical data becomes meaningless if

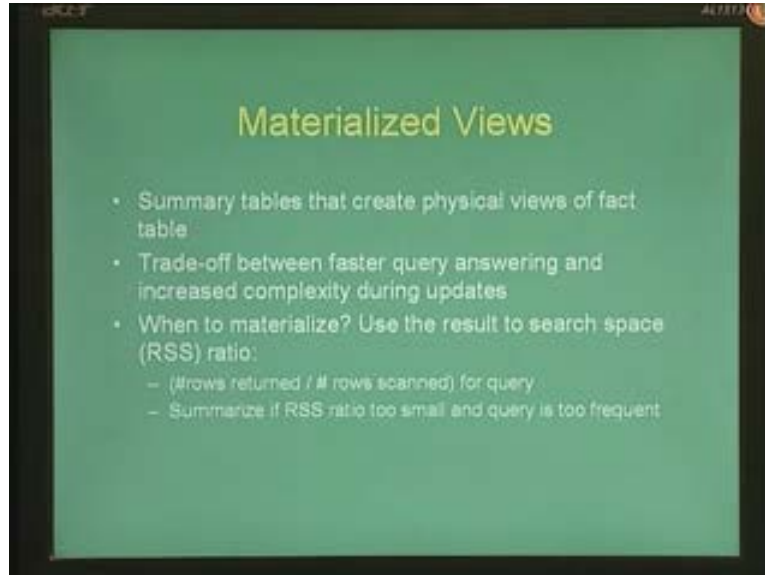
there is no time dimension associated with it. And if you are talking about trend detection and so on, we cannot do it without time dimension. And the time dimension has several different meanings based and for ROLAP techniques depending on the application context. that is suppose we rollup weeks into months or months into year, should we follow the simple calendar based rollup or should we follow the physical calendar, let us say if it is a company or should we follow the academic calendar if it is a university and so on.

So it has several different meanings when we are talking about rollup. And we need to also, in time we need to not only just order based on time, we need to also index some special events like graduation date or releases or some festivals or so on and so forth which actually affects the facts like the sales or whatever it is that we are storing. And we have to we have to timestamp these events with specific whatever time scale that we are using whenever we are having the time dimension. And lastly the order of traversal of the time dimension is important.

If we have let us say a set of all students or set of all products and we want to find out some aggregate information, it does not matter whether we traverse this dimension front to back or back to front but when we are talking about time and when we are talking about trends, is the trend changing over time, the order of traversal becomes extremely important. And let us look back at the materialized views. Coming back to the materialized views, how do we know when to create a materialized view?

As you know materialized views are summary tables that actually create physical views of the fact table. And creating a materialized view is a tradeoff between faster query answering and increased complexity during updates. That is whenever I update the data warehouse, there is a question of view maintenance, materialized view maintenance. So how do we know when to materialize? A good measure for making such a decision is what is called as the RSS ratio or the result to search space ratio. The result to search space ratio is a simple ratio which is shown in the slide here.

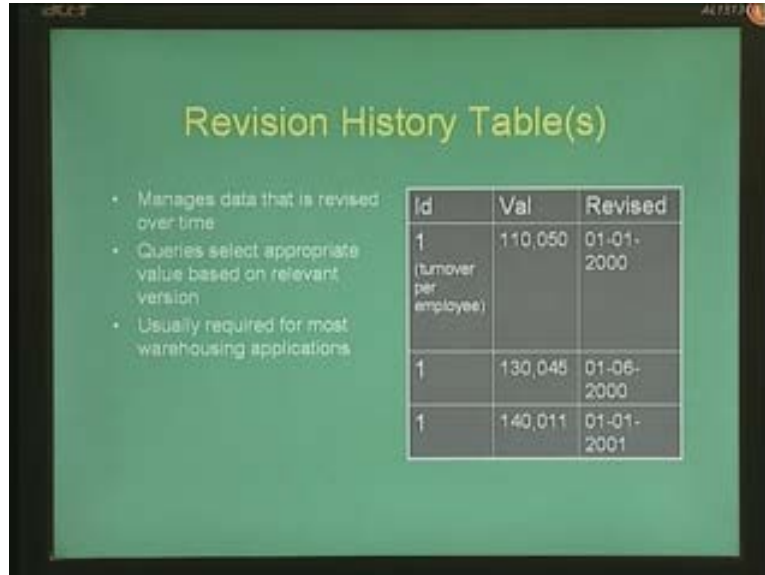
(Refer Slide Time: 36:03)



That is how many number of rows of results are returned divided by the number of rows that are scanned for a given query. For example if I say what is an average age of **what is the average age of** the employee or how was the average age of the employee change over the years. So let us say every year we have some 4000 employees, employee records and we have 10 years of data. So the number of rows that are returned are just 10 for each year the average age.

However the number of rows that are scanned are 4000 times 10, about 40000. So that gives us the RSS ratio. So, we decide to summarize if the RSS ratio is too small and the query is too frequent. That is if 10 by 40000 is considered too small given the frequency of the query, we say we have to materialize this. That is we have to physically create a separate table for this.

(Refer Slide Time: 38:11)



The slide displays a table with the following data:

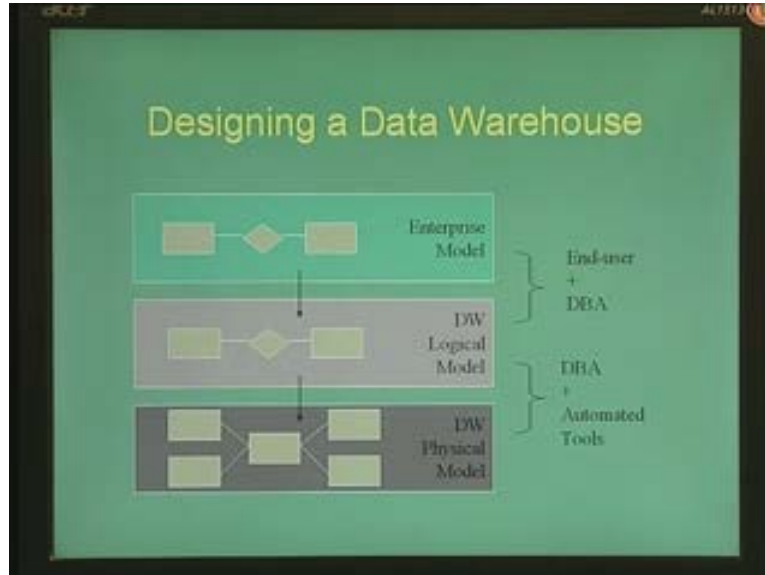
Id	Val	Revised
1 (turnover per employee)	110,050	01-01- 2000
1	130,045	01-06- 2000
1	140,011	01-01- 2001

There is also another kind of table in ROLAP structures which are again useful, which are called as revision history tables and which are again quite crucial when answering different queries. Remember in data warehouses, we are storing archival data that is data over a period of history. Now archival data essentially means that there is more of a probability that some kinds of data elements are revised over time. For example turnover per employee as shown in the slide here that is what is the turnover that the company is getting per employee, that is the total turnover divided by the number of employees. Now it keeps changing every year.

Now let us say, suppose we just have to, we don't have to, we don't want to store it as historical data that is we don't want to store it as how turnover is changing over time but rather what is the turnover per employee. So the revision history table essentially stores the different values that this turnover per employee has taken up over each different updation. That is the first updation, the turnover per employee was this and the updation was on this date. Whenever this value was updated the second time, the value is this and the updation date is this and so on. So this information is used based on what time frame is the query addressing, when it is asking a question. For example I might say what was the turnover per employee in the last decade, then I might want to take up some value that was updated in the last decade rather than this decade and so on. So that's where revision history tables become important.

So let us quickly look at what are the different steps that it takes while designing a data warehouse. See when we are designing a data warehouse, we are looking at a warehouse from a managerial perspective. That is we are essentially starting out with an enterprise model that is we have an enterprise model of how our enterprise works like whether it is a company or whether it is a large organization or university or whatever. There is certain kind of enterprise model, a university is divided into departments and labs and faculties and so on and so forth and so on.

(Refer Slide Time: 40:00)

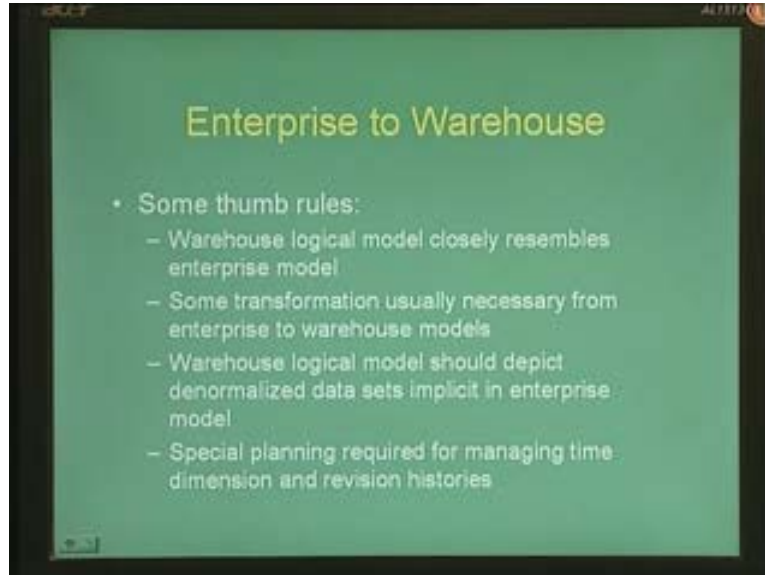


Now based on the enterprise logical model, we built the logical data warehouse model or the data warehouse logical model and from the data warehouse logical model, we come down to the data warehouse physical model which is the star schema and snowflake schema or constellation and so on and so forth.

Now this part is handled by the data warehouse administrator or the database administrator along with the end user. The database administrator has to clearly understand the end user requirements before building the logical model. And here it's usually a question of the database administrator using one or more automated tools that can translate a given logical model on to a physical model. There are some design for the most pattern art where how to get good designer, how to calibrate design is actually quite a tough problem to answer.

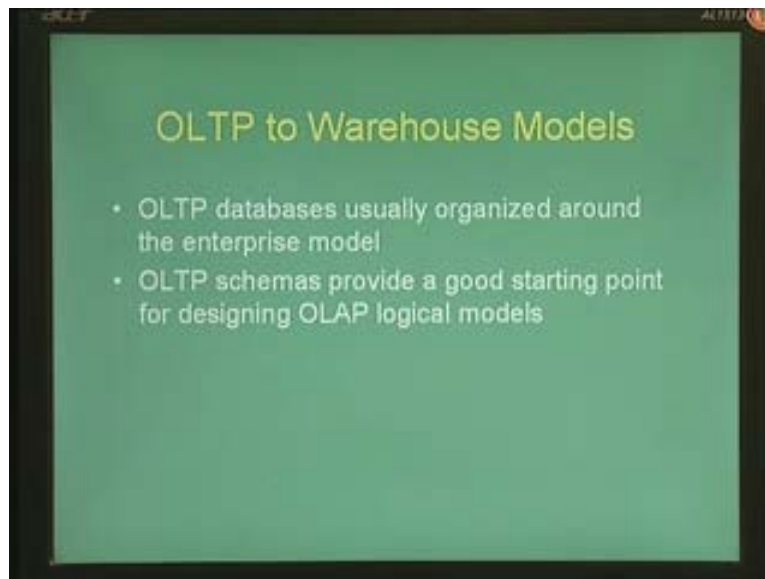
However there are some thumb rules that can help the designer in designing good data warehouse models. And this slide summarizes some of the thumb rules, things like the warehouse logical model should closely resemble the enterprise model and not let say the operational model. The enterprise model is the overall strategic model of the enterprise while the operational model is the model of let us say given particular operation. Let us say withdrawal from an ATM has a particular operational model. But this operational model rarely figures in the overall enterprise model of the bank itself.

(Refer Slide Time: 41:49)



So the warehouse logical model should resemble the enterprise model rather than the different operational models. And note that the OLTP sources are based around the operational models rather than the enterprise model. And there are some transformation rules which are usually necessary from enterprise to warehouse models.

(Refer Slide Time: 42:44)

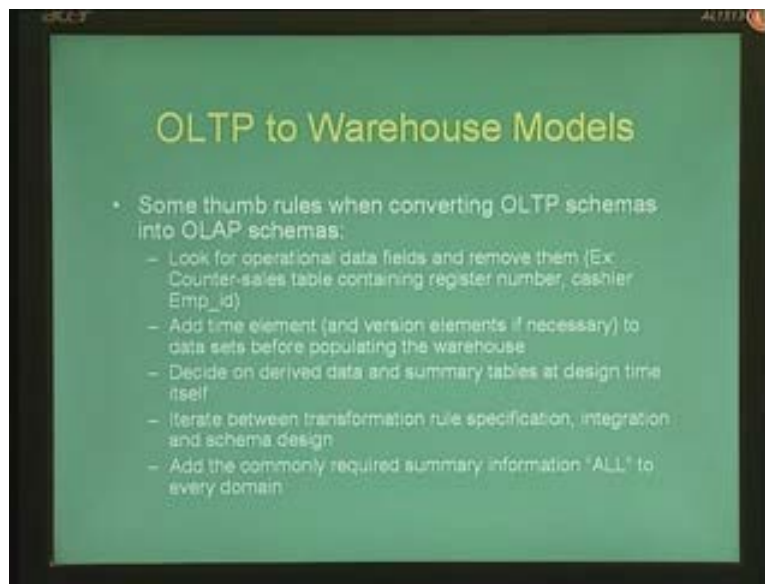


And special planning is required for managing time dimensions and revision history which we saw a special tables that we saw in the previous slides. So how do we, let us say how do we design this data integration that is from OLTP sources to data warehouses. What it is entitled for creating or integrating schemata from OLTP sources to data

warehouses. So OLTP to OLAP or OLTP to warehouse models are also based around certain sets of design principles which again have certain kinds of thumb rules again like there is no simple way of or there is no algorithmic procedure for designing schema.

However this thumb rules help in creating a good design or evaluating different designs. For example several OLTP schemata has a number of operational data fields like counter sales table that is the cash register number or the cashier identifier and so on. Such operational data fields are usually not useful for strategic decisions something that talks about what is the trend in sales and so on. We usually don't care which cashier did what and so on. So we have to remove all such operational data fields, it's important to identify which data fields are purely operational and which data fields have some kind of strategic importance.

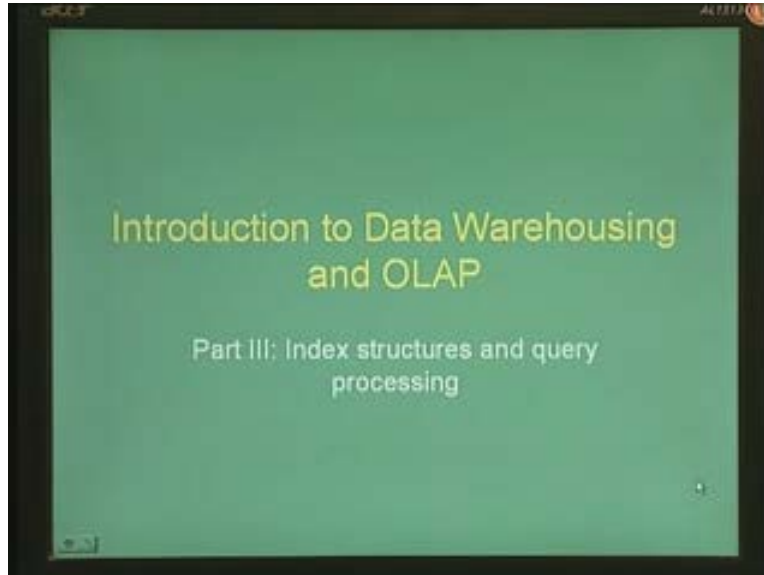
(Refer Slide Time: 43:20)



And of course we have to add a time element. Time element may be implicit in the operational model but here we have to add them explicitly for the data warehouse model and also of course version elements if necessary. And we have to decide on the derived data or the materialized views as early as possible and usually commonly used materialized view is what is called as the ALL view. That is for example if I have product wise information let us say product is a dimension, so we have product 1, product 2, product 3 as different values in the dimension and the last value will be ALL.

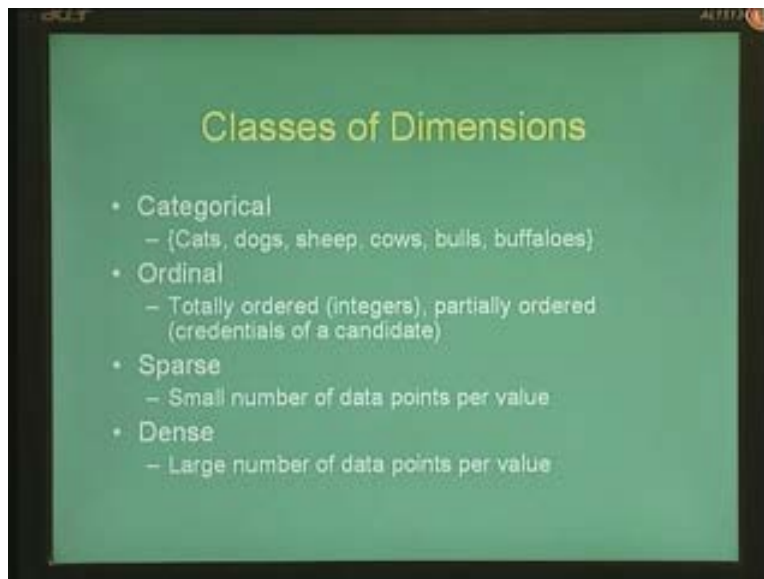
So, ALL basically says what is the sales value for all products. Similarly ALL value in the branch dimension would say what is the sales value across all branches and so on. So that was about ROLAP structures and let us briefly look at what kinds of strategies are used for MOLAP structure that is native multi dimensional databases or hyper cube maintenance and what kinds of index structures are used for handling such multi-dimensional retrievals.

(Refer Slide Time: 45:09)



So, just to summarize we have different classes of dimensions, there could be categorical dimensions, there could be ordinal dimensions. Dimensions could be sparse where there are small number of data points per value or dimensions could be dense where there are large number of data points per value.

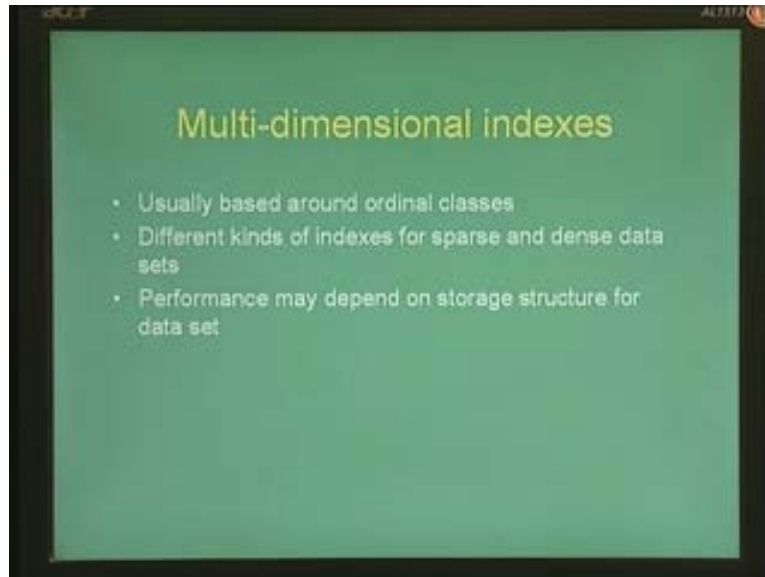
(Refer Slide Time: 45:45)



Now if we are talking about indexing note that when we are talking about MOLAP, I directly went into indexing because this is the main indexing in addition to the storage structures are what constitutes the native support for MOLAP.

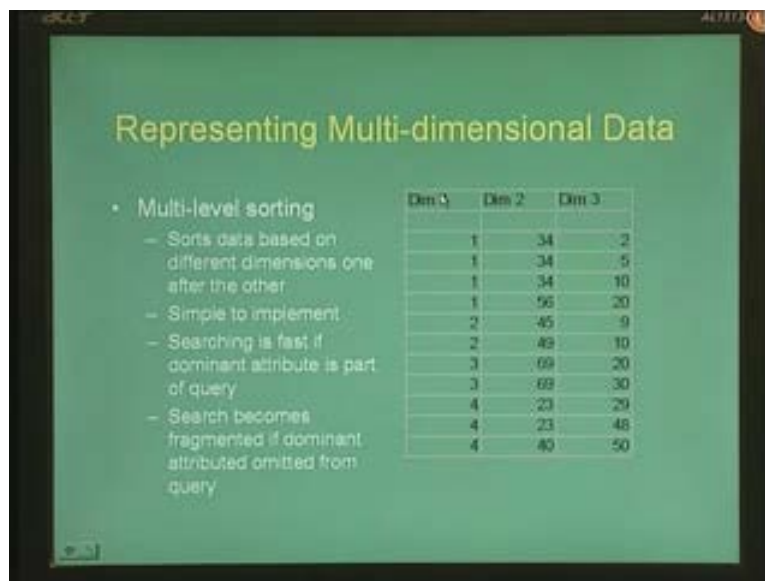
So when we are talking about indexing, indexes are usually based around ordinal classes but there are categorical indexes as well, as we will see and the performance may depend on the storage structure for any given data set.

(Refer Slide Time: 46:22)



So what kinds of storage structures are used for multi dimensional data or how do we store multi dimensional data or hyper cubes on to disc.

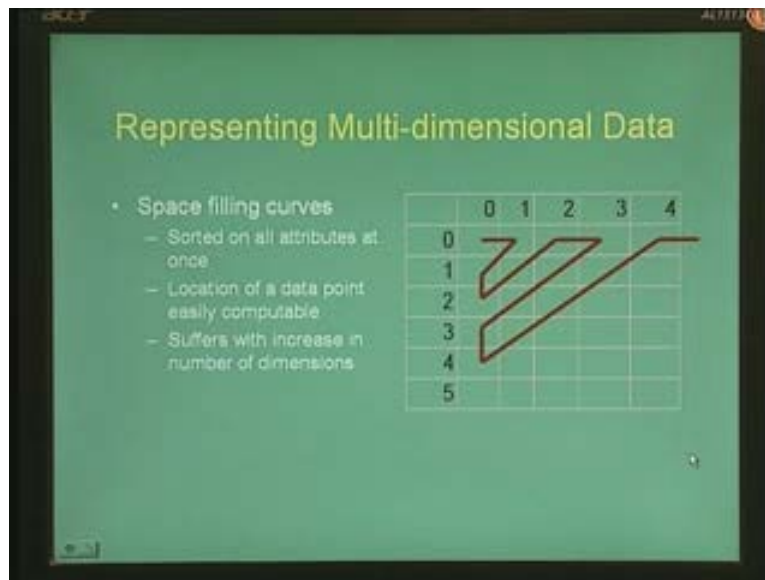
(Refer Slide Time: 46:44)



Remember discs are just stored in blocks and sectors and so on. So, one way of storing is simply multi dimensional sorting that is let us say there are 3 dimensions and there is a

fact. So you just sort the table along all 3 dimensions, first along dimension 1 then along dimension 2 then along dimension 3 and so on, so as simple as this. So, it just becomes a table one long table, the hyper cube is just one long table and this is very simple to implement and searching is very fast, if the dominant attribute that is in this case dimension one is part of the query, else it becomes fragmented and it becomes slower. Another kind of storage structure that is commonly used are what are called as space filling curves. Have a look at the slide here carefully. As if you see in the slide, this slide shows a two dimensional space discrete two dimensional space that is 0 1 2 3 4 like this, 0 1 2 3 4 like this and there is a curve that has gone through the space like this.

(Refer Slide Time: 47:16)

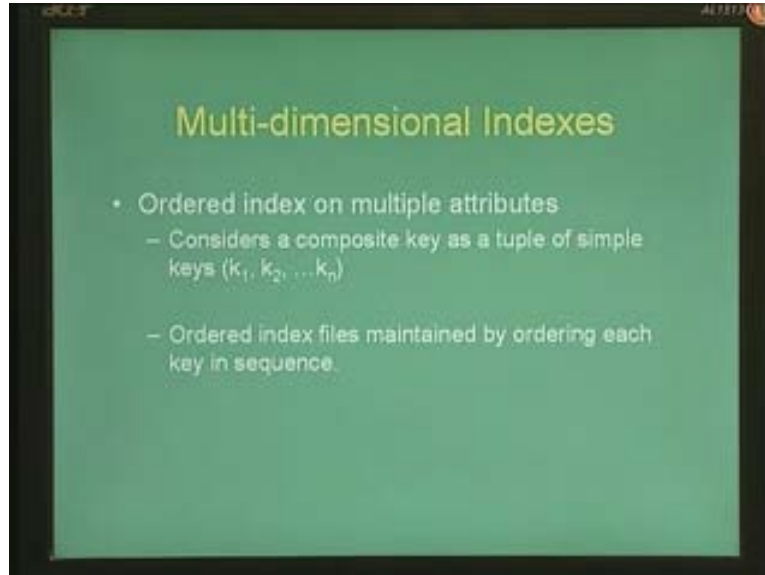


That is it has covered the entire space without revisiting any point again or without revisiting any point twice. So it is quite simple to calculate the position of a data element whenever a space filling curve is used like this. That is the first data element 0, 0 is here and 0, 1 is the second data element and 1, 0 is the third data element and 2, 0 is the fourth data element and so on. So, it is very easy to compute the location of a data point.

However, it suffers whenever the number of dimensions are high or especially when there are sparse dimension that is when there are large number of cells which have no data points associated with them. Then there are, so from these storage structures let us move on to some kinds of index structures that is how do we index elements based on this storage structures.

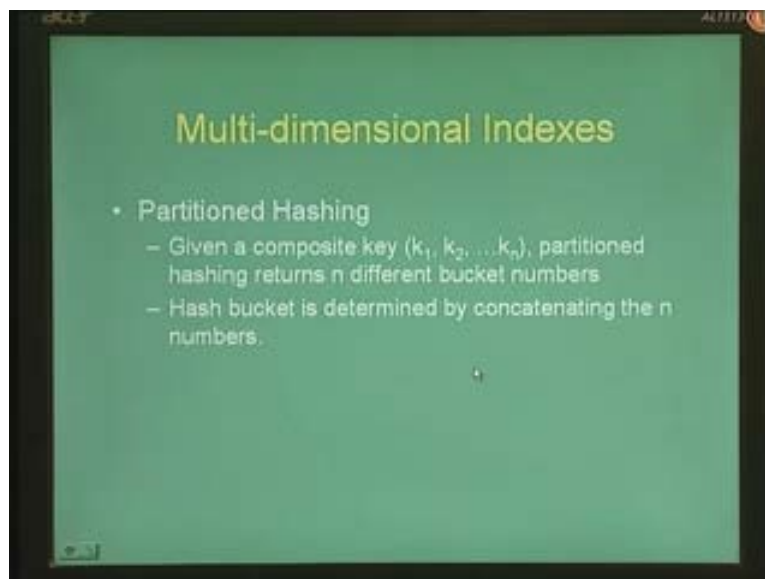
The simplest kind of index is the ordered index on multiple attributes that is instead of when you have to index based on multiple dimensions, index on each dimension separately.

(Refer Slide Time: 48:47)



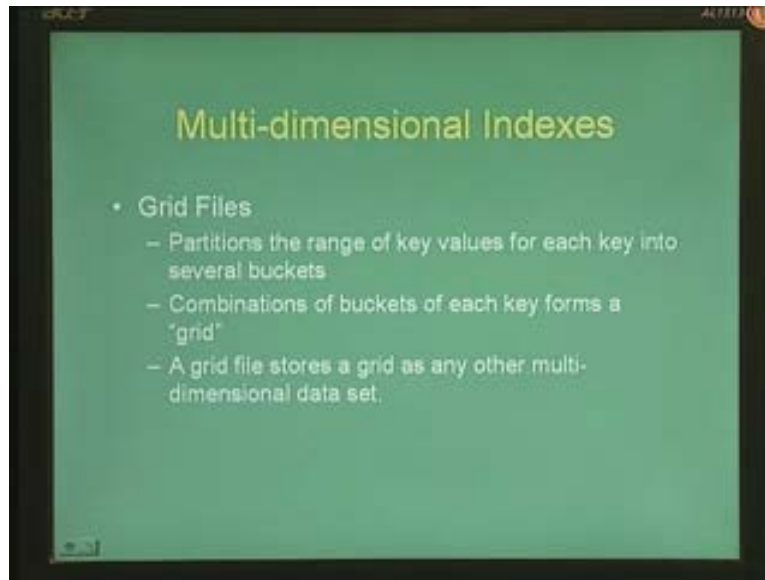
So suppose you want to index on n different dimensions k_1 to k_n , index on k_1 separately and k_2 separately and k_3 separately and so on and ordered index files are maintained by ordering each such key in sequence. But of course ordered dimensions suffer from several short comings especially when the number of dimensions is high or when the number of data points is high, you need to compute intersections across these different dimensions which becomes a lot of overhead. There is also what are called as partitioned hashing where given a composite key like this that is n different dimensions, a partitioned hashing returns n different bucket numbers and the hash bucket is simply the concatenation of the n different bucket numbers.

(Refer Slide Time: 49:42)



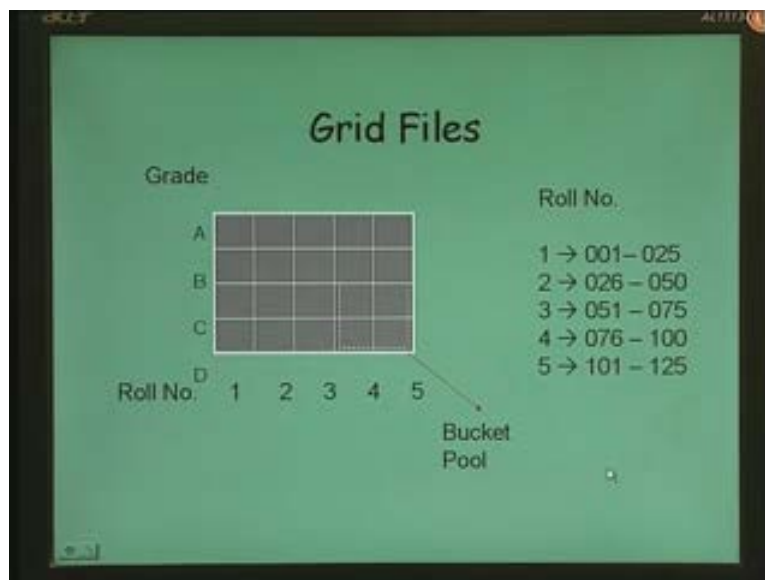
And other kinds of multi dimensional indexes include the grid files. We have seen the grid files when we are talking about index structures for normal databases.

(Refer Slide Time: 50:00)



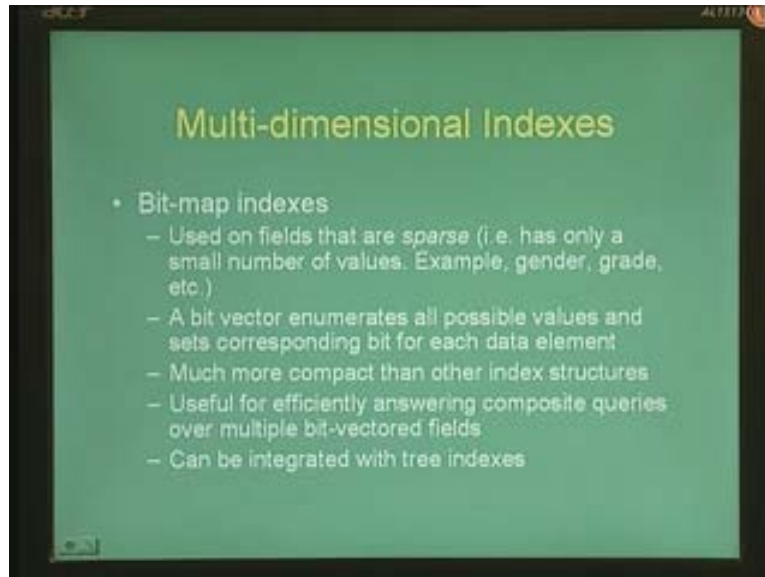
So, we shall not be covering grid files now in great detail except to state that we can think of set of buckets forming a grid and a multi dimensional index would in turn map on to a set of buckets in this grid are what is called as the bucket pool and then we essentially find out what is the or find out the corresponding data points.

(Refer Slide Time: 50:18)



Then there are what are called as the bit map indexes which are quite again, quite commonly used and these are used on fields that are sparse that is where fields have only a small number of values. For example gender has only 2 values or grades in a university setting may be has 5 values a b c d e f, a b c d e and so on.

(Refer Slide Time: 50:55)



And essentially what we do is we store a corresponding bit vector that enumerates all possible values and sets the corresponding bit for each data element. And it's much more compact than other index structures because we would be able to answer composite queries like this product and this branch and this week and so on and so forth, quite efficiently. This slide shows a particular example. Let us say there are 3 subjects database, artificial intelligence and say parallel distributed systems and 6 different grades a b c d e f.

(Refer Slide Time: 51:21)

Multi-dimensional Indexes

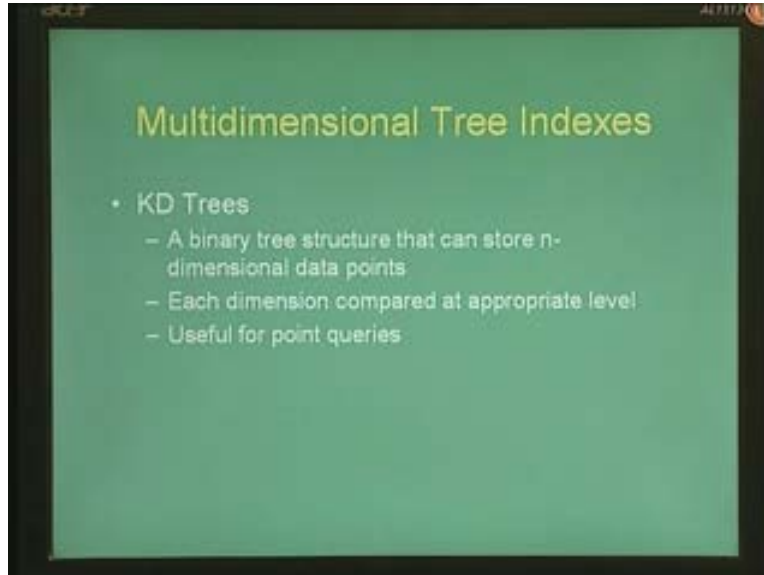
Encoding Bit-map indexes

Grade = {A, B, C, D, E, F}	Subject = {DB, AI, PDS}
A = 000001	DB = 001
B = 000010	AI = 010
C = 000100	PDS = 100
D = 001000	no value = 000
E = 010000	
F = 100000	
No value = 000000	

Student who has scored A in DB and AI
(000001 && 001 && 001)

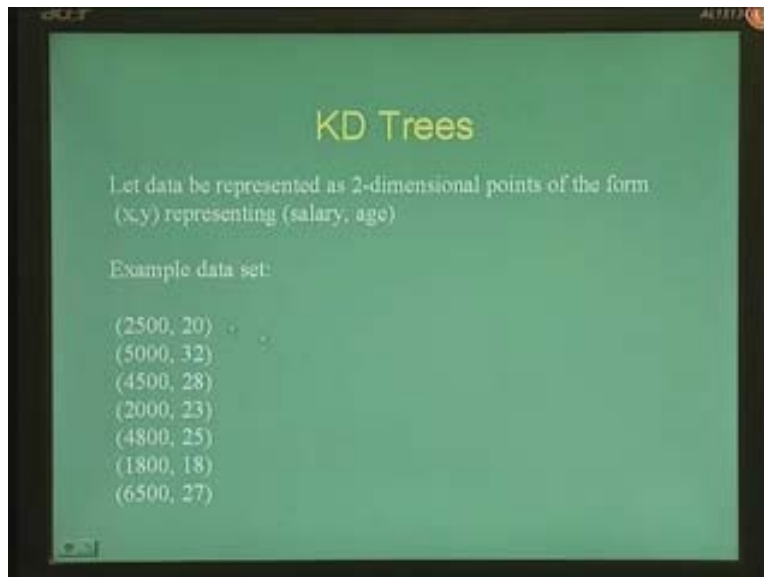
Now each grade is denoted by a specific bit that is set in a 6 bit vector that is there are 6 vectors in this and each grade has a particular bit set and no values, all zeros. Similarly there are 3 different subjects, so there is a 3 bit vector here and corresponding bit is set. Suppose we have to search for all students who have scored a in databases and AI. What is that we need to do? We just need to compute the logical and between databases and AI and grade A and that would be the logical intersection of the corresponding buckets or the corresponding sets that are stored in this values. Another kind of multi dimensional index is what is called as KD trees which is again used for efficiently answering point queries in a multi dimensional data space.

(Refer Slide Time: 52:25)



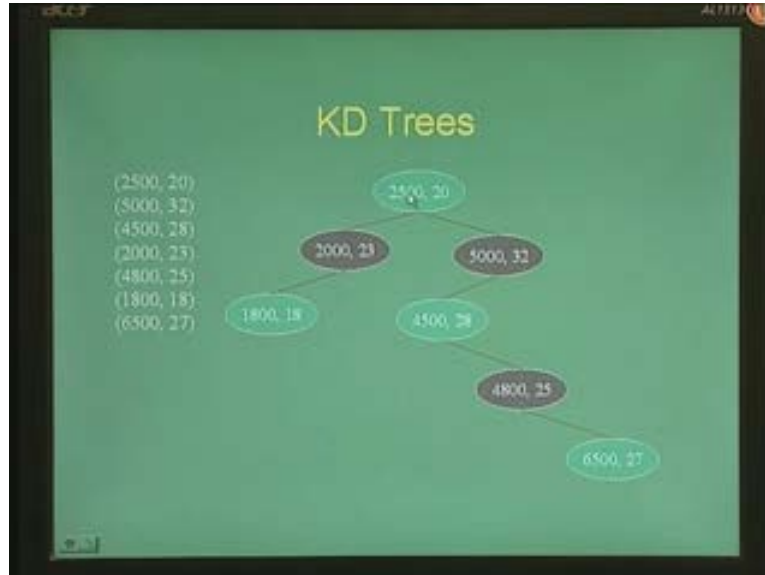
We shall not be looking into KD trees in more detail. Let me just explain KD trees by an example.

(Refer Slide Time: 52:37)



You might of heard of binary search trees were given a particular data element, all data elements having a key greater than this particular data element is in the right sub tree and all keys lesser than this given keys in the left sub tree. The same strategy here is expanded to k dimensions that's where the KD trees come from k dimensional tree.

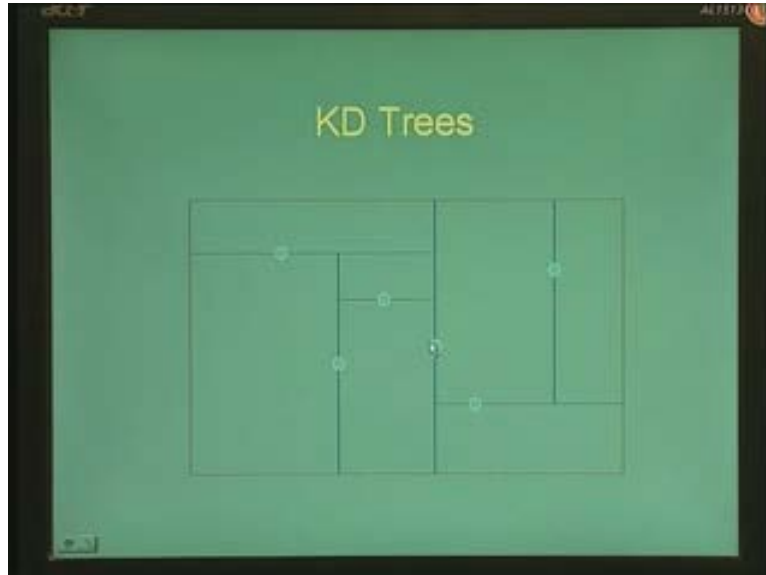
(Refer Slide Time: 53:38)



So let us say this is an example data set we have. There are 2 dimensions x and y , so representing let's say salary and age and these are the set of data points we have. The way we build a KD tree is simply like this. The first element becomes the root, for the second element we start by comparing the x dimension, the first dimension here now because 5000 is greater than 25000 it comes here. For the third one we start by comparing the x dimension here the first dimension, so 4500 is greater than 25000 comes here.

Now here we compare the second dimension that is 32 with 28 and because 28 is lesser than 32, it comes to the left of this tree and so on. So as you can see here, all the odd number of places corresponds to dimension 1 and all the even number of places correspond to comparing it dimension 2. So it's a simple extension of the usual binary search tree data structure for k different dimensions but of course there is an other way of looking at KD trees as where each data point here is found to bisect a given space into two different sub spaces. We shall not go into much more details here because KD trees in itself is a vast subject by itself.

(Refer Slide Time: 54:15)

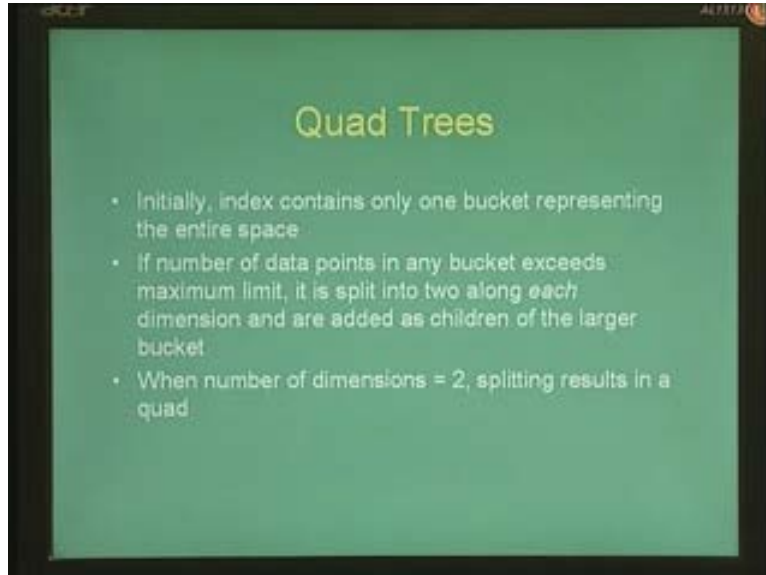


(Refer Slide Time: 54:25)

-
- KD Trees**
- Each point divides search space along one of the dimensions
 - Structure of the tree (and hence its performance) sensitive to the order of insertion of data points

And there are other kinds of data structures which I shall not be going through like quad trees and R trees which is again a very commonly used data structures where we manage regions and there are certain user specified regions and there are certain virtual regions.

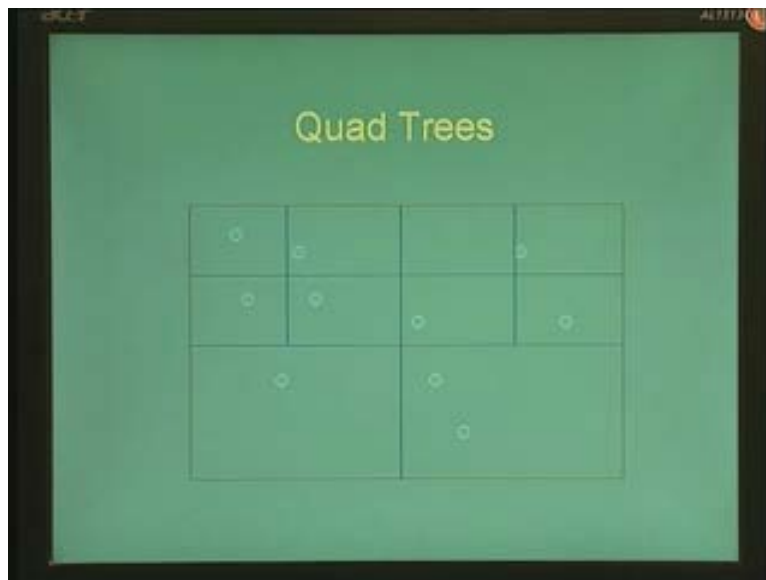
(Refer Slide Time: 54:30)



Quad Trees

- Initially, index contains only one bucket representing the entire space
- If number of data points in any bucket exceeds maximum limit, it is split into two along *each* dimension and are added as children of the larger bucket
- When number of dimensions = 2, splitting results in a quad

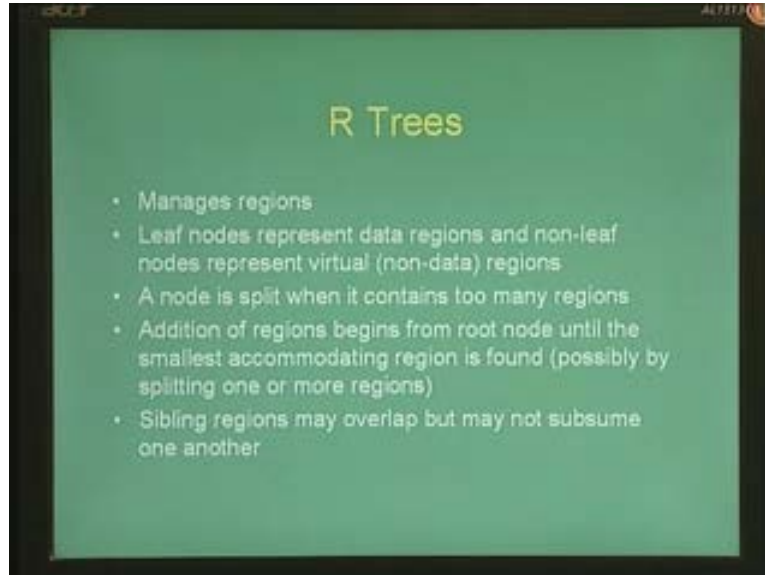
(Refer Slide Time: 54:36)



Quad Trees

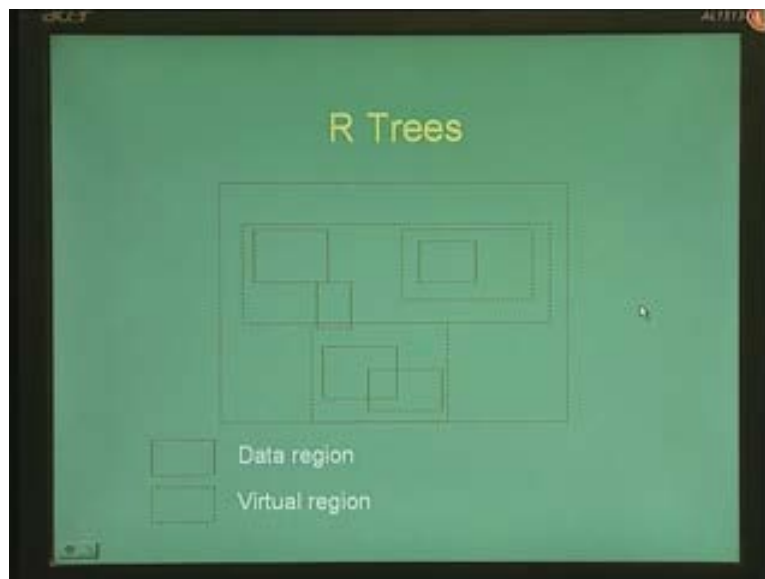
A 2D grid representing a quad tree structure. The grid is divided into four quadrants by a vertical line and a horizontal line. The top-left quadrant contains one data point (circle). The top-right quadrant contains one data point. The bottom-left quadrant contains one data point. The bottom-right quadrant contains two data points. The grid is further divided into smaller cells, illustrating the recursive splitting process.

(Refer Slide Time: 54:37)



And so each data point is also represented as a region of zero area and so on. So a region is supposed to consider, is supposed to contain other regions within it and it is supposed to have a particular capacity. Whenever a region exceeds its capacity, it is split to form 2 different regions. So the slide here schematically shows an R tree where this is the root node which is the overall region and which has two children having two virtual regions.

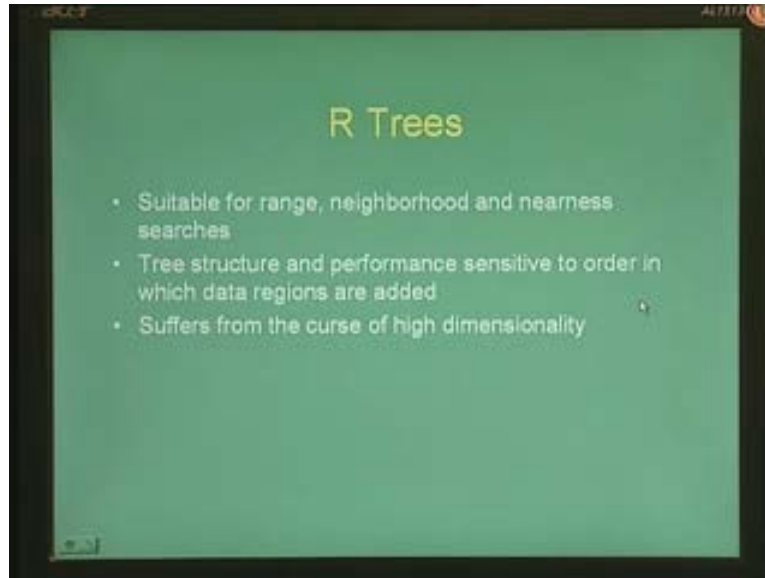
(Refer Slide Time: 55:45)



And each of these two children have two other children that is children that is child 1 and child 2 and there are to be a virtual region here, so child 1 and child 2 and so on and it is

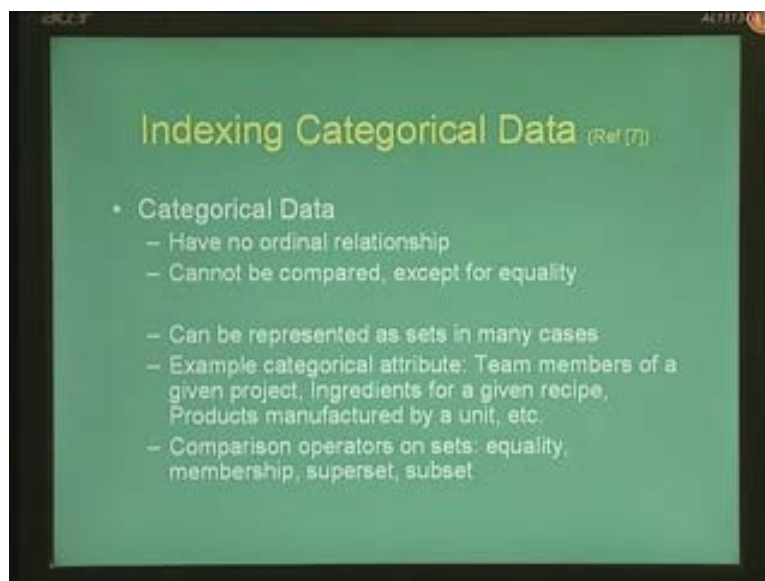
split correspondingly. And R trees are suitable for range searches, neighborhood searches, nearness searches and so on.

(Refer Slide Time: 55:47)



But however all of this suffer from the curse of high dimensionality and let's not try to prove this here but it's obvious when we analyze this data structures more closely. Lastly let us look at one data structure for indexing categorical data. Here as you can see until now we have been looking mainly at ordinal data whether it is K tree, KD trees or R trees it is important that there is an ordinal information associated with each dimension.

(Refer Slide Time: 56:08)



That is this is lesser than this and this is lesser than this and so on but here in categorical data we use what are called as signatures or bit map indexes by which we can say if the set of all ingredients whichever appear in a particular element is set to 1 here and all other elements are set to 0.

(Refer Slide Time: 56:38)

Signatures

- Represent a set as a bitmap where each bit corresponds to an object in a larger UoD

UoD = {set of all ingredients}
S, T \subseteq UoD : ingredients for two recipes
s, t : corresponding bit maps of S and T

Queries:
S \subseteq T \leftrightarrow s \wedge \sim t = 0
S \supseteq T \leftrightarrow t \wedge \sim s = 0

(Refer Slide Time: 56:43)

Signature Trees

- Leaf nodes contain (signature, datapointer) pairs
- Non-leaf nodes formed by bit-wise ORing of its children nodes
- Traverse the tree by AND'ing the query signature with the node signature

```
graph TD
    A[1111] --- B[1100]
    A --- C[1011]
    B --- D[1000]
    B --- E[0100]
    C --- F[1001]
    C --- G[0011]
```

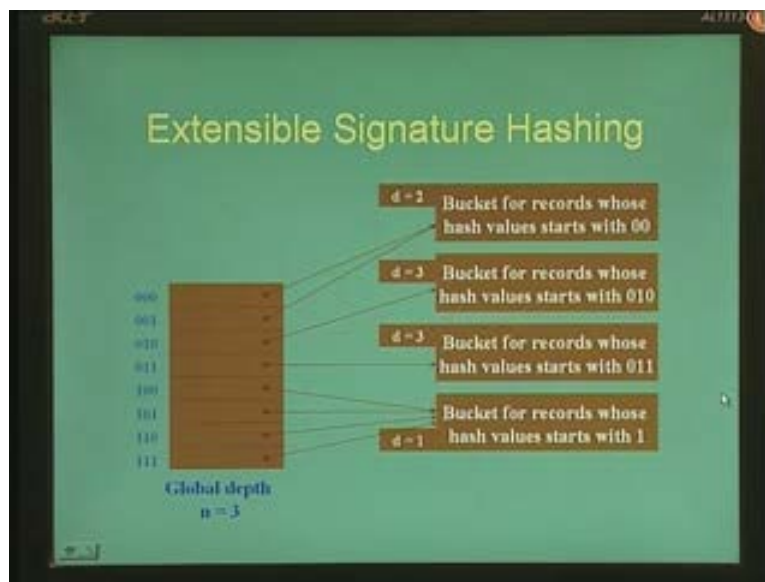
So which basically forms what is called as the signature tree for categorical data. And there are other kinds of extensible signature hashing and so on and so forth which we are not going to cover in more detail here.

(Refer Slide Time: 56:56)

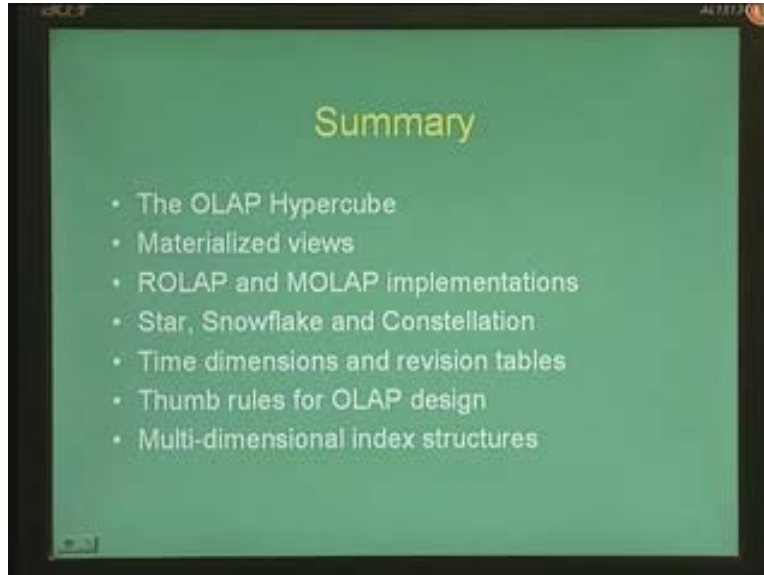
Extensible Signature Hashing

- Hash tables constructed based on the most significant d bits of signature
- Hash levels extended by extending d whenever overflow occurs

(Refer Slide Time: 56:58)



(Refer Slide Time: 57:10)



So let us summarize what we learnt in the data model section of data warehouses. We saw that a data warehouse contains the OLAP hyper cube at its core and having different materialized views and then we also saw different kinds of ROLAP and MOLAP implementations of this materialized views and different kinds of index structures, so that brings us to the end of this session.