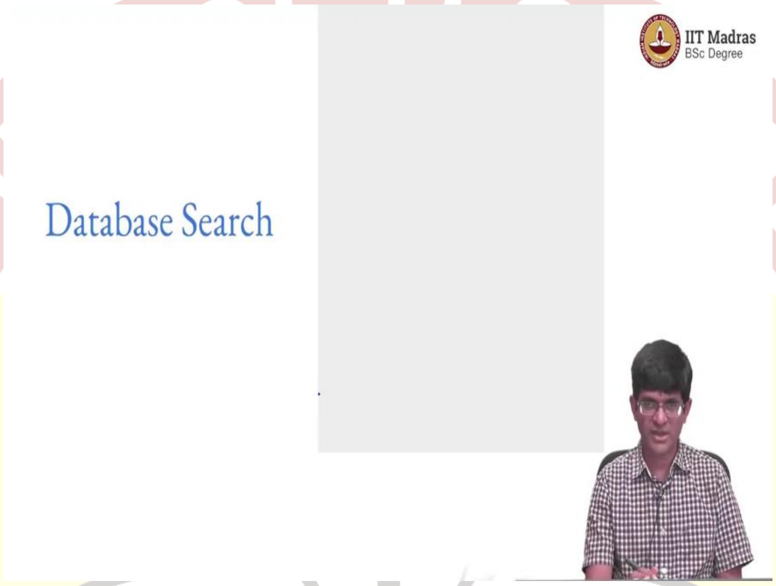
IIT Madras

ONLINE DEGREE

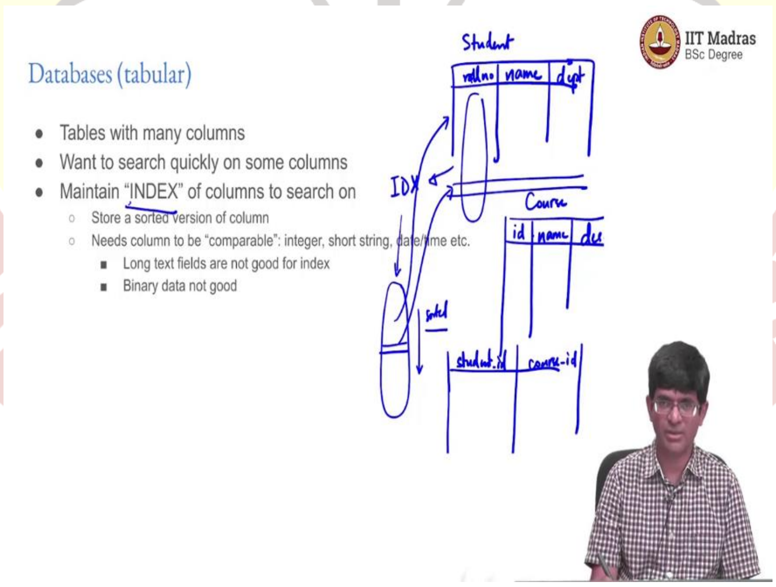
Modern Application Development – I
Professor Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology Madras
Database Search

Hello, everyone, and welcome to this course on modern application development.

(Refer Slide Time: 00:16)



Database Search



Databases (tabular)

- Tables with many columns
- Want to search quickly on some columns
- Maintain "INDEX" of columns to search on
  - Store a sorted version of column
  - Needs column to be "comparable": integer, short string, date/time etc.
    - Long text fields are not good for index
    - Binary data not good

So, what did all of these things that we considered in terms of searching for data in some storage mechanism? What has that got to do with searching in a database? The most common databases have a so called, tabular structure. There are many tables, each of which has many columns.

So, for example, I might have something called a student table where as you can imagine, it would have a roll number, it would have a name, it would probably have a department, and so on. And I might have another course table, which has an id number for the course, a name, a description, and then I could also have a sort of combined table where I have a student_id, and a course_id, which basically tells me, which student is registered for which course.

So, as long as I can create these kind of tables, it means, that I can store a lot of useful data, and we have already come across this in the previous thing. This is basically the most important part as far as model structure is concerned. We need to be able to create relationships between different columns in between different tables, and those relationships are what ultimately drives the usefulness of the application.

Now, if I want to search through all students, I cannot guarantee that the order in which I get the list of students is going to already be sorted. I have to basically make the entries as in when they come and register. So, what should I do? Should I have something, build a big array somewhere in memory, in order to store this table, because after all, tables are well suited to arrays they have the same kind of structure. There is a particular memory location or a row number, where if I go I will find all the data corresponding to one particular entry.

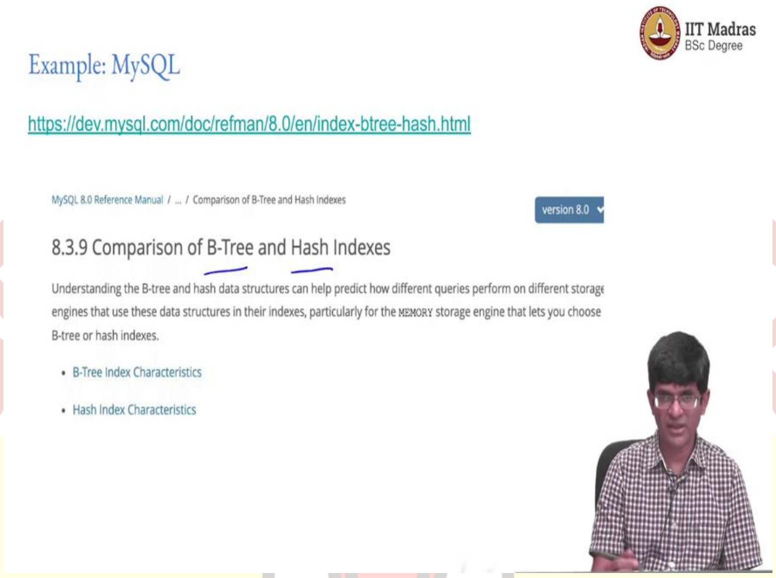
So, it looks as though maybe I can use arrays. The problem is two people can have the same name. So, I then need to find something that is unique. I assign a roll number. That is better, because, roll numbers are guaranteed to be unique. So, what should I do? Should I just sort of index, I mean, just store the data as it is or I can create something called an index.

An index is actually pretty much just a copy of the data present in one of the columns of a table, but now, in sorted order. So, you basically take all the roll numbers, and you create an index out of it. And what does that index do? It basically creates another copy, but now this is sorted and also has the appropriate links into the correct row in the original table.

So, let us say that I want, I have an index on roll number and I want to search for the entry corresponding to a given roll number, I just search in this sorted list. I know I can do that in O(log N) time. And once I have the particular entry that I have found, I just go to the actual table and pull out that row, and get all the other entries, the student’s name, the department and all other information I need.

So, building the right kind of index is very important to the performance of your application. You need to know, what kind of index to build and what is it that will change the behavior in terms of, if you do not construct the right kind of index, it can do pretty poorly in searching. If you create too many indexes, it might end up taking up too much space. So, there are a lot of engineering trade-offs that come in over here.

(Refer Slide Time: 04:11)


Example: MySQL

https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html

MySQL 8.0 Reference Manual / ... / Comparison of B-Tree and Hash Indexes

version 8.0

8.3.9 Comparison of B-Tree and Hash Indexes

Understanding the B-tree and hash data structures can help predict how different queries perform on different storage engines that use these data structures in their indexes, particularly for the MEMORY storage engine that lets you choose B-tree or hash indexes.

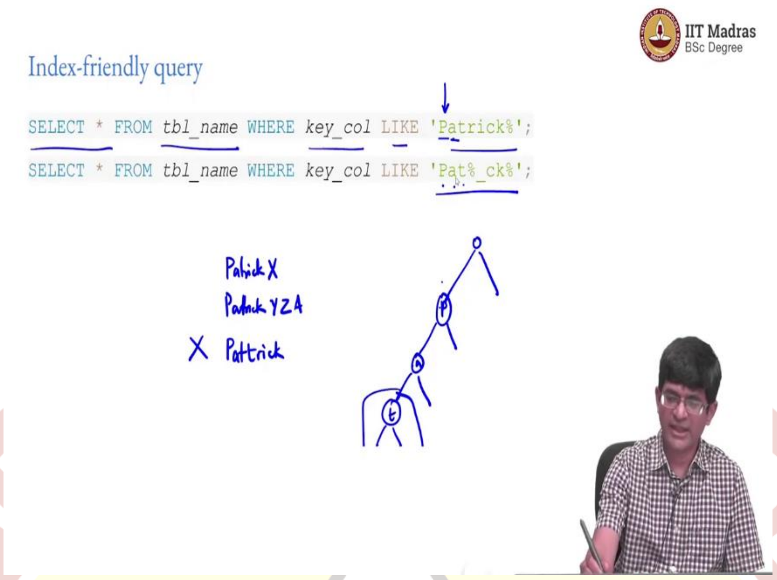
- B-Tree Index Characteristics
- Hash Index Characteristics

Now, obviously, we do not have the time to get into all of this in detail. And in fact, it is not even possible simply because you might not choose the exact same database that I recommend or that I have used in some application. You might have a different reason for finding something else. So, the point is, you should be able to look up documentation and find out more about it by looking for it.

And a good example is the MySQL database. They in fact, have a lot of documentation online it is available out there and they talk about the performance comparisons of different kinds of indexes. In particular, they consider two things, something called a B-Tree and the other thing called a Hash.

The B-Tree is basically like I said, a variant of the binary tree and is the most common kind of index. So, I would strongly recommend that anyone who is at least interested in building larger scale applications. If you are building a very small application probably most of this does not matter. But the moment you want to go even to a few 100 entries, these things start to matter. So, and anyway, it is good practice to sort of know the tools that you are working with. So, if you are using MySQL, where should you look?

(Refer Slide Time: 05:30)



Index-friendly query

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

And this is an example from that link that I just showed you. For a what we can see in MySQL, for example, or pretty much any index database is, let us say, I have a table in which I am searching, so I want to do select * and the search condition is key column like Patrick %.

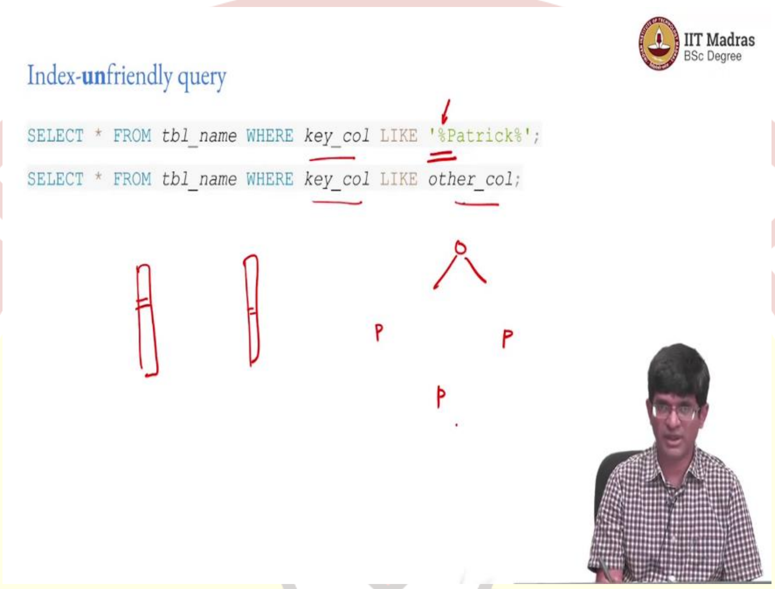
Now, what exactly does this search do? It search, it will search through the database for all entries, like let us say, Patrick X, Patrick YZA, all those entries will match. But let us say Pattrick will not match because it is a different name.

So, the point is, because of the fact that it is Patrick, I know the beginning part of the name, I can do this kind of binary search in order to narrow down the elements that I need to look at in more detail. Because I can say for sure that anything, which does not start with a P can straightaway be eliminated, among those that start with E, I can eliminate things that do not have A as an X later.

How do I do this? I basically go through the tree and narrow down the section within which I need to look. And because of the fact that it is starting from the first letter, I can actually take this entire tree and build up a specific sub sequence, which says, okay, this was the root, this was all things starting with P, this was all things at second letter A, this was all things with, T and so on. And essentially, eventually, I will come down and say, okay, this is the subtree that I really need to be looking at, and I can discard the rest of it.

Now, what happens if I have a query, which goes something like this Pat% something else? Now, this is a bit more tricky because what has happened over here is, I can go up to Pat, up to this point in the graph, after that, I need to start searching. So, all I can do is narrow it down Up to this point and from there, I then need to switch over and go into some other kind of string search where I basically look at all possible entries for the fourth character, and then see 5, 6, 7 do they match with underscore ck? At least this Pat part of it allowed me to use the index.

(Refer Slide Time: 8:06)



Index-unfriendly query

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

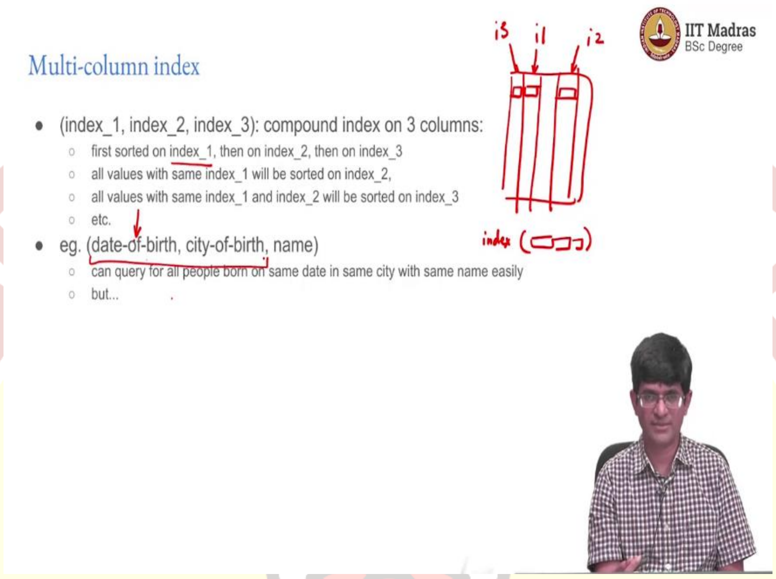
But what if my search was something like this? Here, the first letter itself is percent, which means that, which way should I go? I do not know. Maybe there is a P somewhere over here, maybe there is a P somewhere over here, maybe there is a P somewhere over here. I do not know how many letters can be present before the P. Which means, that I cannot use an index, I cannot use the tree that I have constructed at all.

Similarly, if I am trying to say, there is a key column, which needs to be like some other column. I do not know, because I will basically need to look at this column, I will need to look at this column and I will need to then search through all of them and find out whether there is a connection or not.

Which means that, this index which was created on the key column is of no use for answering a query of the sort. So, please keep this in mind. And the reason is simply because ultimately you are constructing queries. For the most part, at least in this course, we are trying to use an ORM like SQL alchemy, which takes care of a lot of this for you. The problem is, it is not going to tell you if the queries that you are constructing are poor.

So, if you do some kind of complicated query or something which does not make use of the indexes, SQL alchemy is not really going to tell you about it. There are ways of sort of profiling the database, which will finally come up with information that would be useful and tell you that, look, this is not really the way you should have done it.

(Refer Slide Time: 09:38)



Multi-column index

- (index_1, index_2, index_3): compound index on 3 columns:
  - first sorted on index_1, then on index_2, then on index_3
  - all values with same index_1 will be sorted on index_2,
  - all values with same index_1 and index_2 will be sorted on index_3
  - etc.
- eg. (date-of-birth, city-of-birth, name)
  - can query for all people born on same date in same city with same name easily
  - but...

You can also create, so called, multicolumn indexes. So, let us say that I have three columns in a table or rather I have many columns and I use this as the index 1, let us say this one as index 2 and this one as index 3. The order of the index has nothing to do with order of the columns in the table.

It is a compound index. So, what does that do? It basically takes an element from here, takes the corresponding element from here, takes the corresponding element from the i3 column and puts them together and creates this new string and this is what the index is created on.

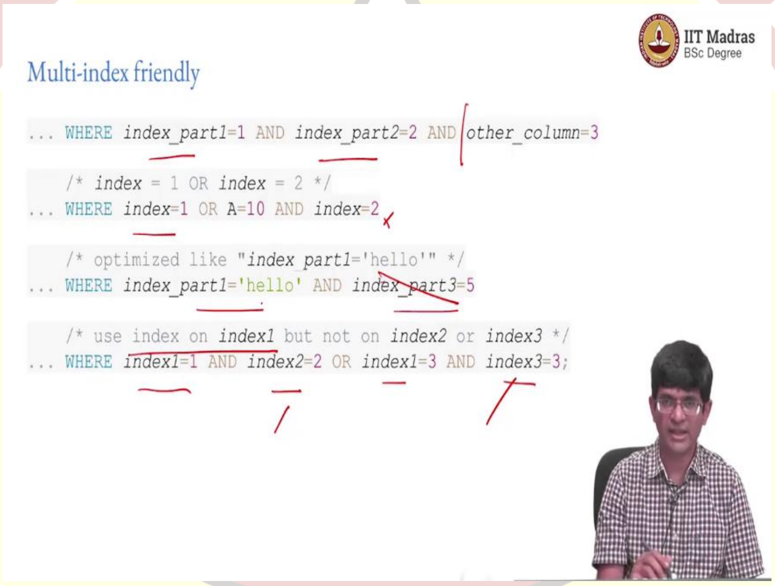
What do I mean by creating an index? Like we said, I mean, it basically creates a copy of all of them and sorts them with links back into the database. So, how this would work in practices effectively, this means that it is first sorted on index 1.

As an example, let us say that I want to create a compound index on date of birth, city of birth, and the name of a person and what it would do is, I would call create this compound index. First sort on index 1, which is the date of birth, then on index 2, the city of birth and finally on index 3, which is a name of the person. If I wanted to query for all people born on the same date, perfect, it just picks up the first entry and is able to narrow it down and tell me exactly who are all the people born on a given date.

Born on a given date in a given city, again, good. But born on a given date, and with the same name, but not necessarily in the same city not particularly good, because I cannot make use of this combined query across date of birth and city of birth, at this point. I will only be able to query for those born in the same date of birth, then I will have to narrow down the ones with the same name, I cannot use this index for that I might have another index, and then finally, go and narrow down the city of birth.

So, how you construct indexes has to be determined by the application developer. You need to sort of have an idea of what kind of queries you are likely to see in practice, and based on that, try and optimize.

(Refer Slide Time: 12:04)



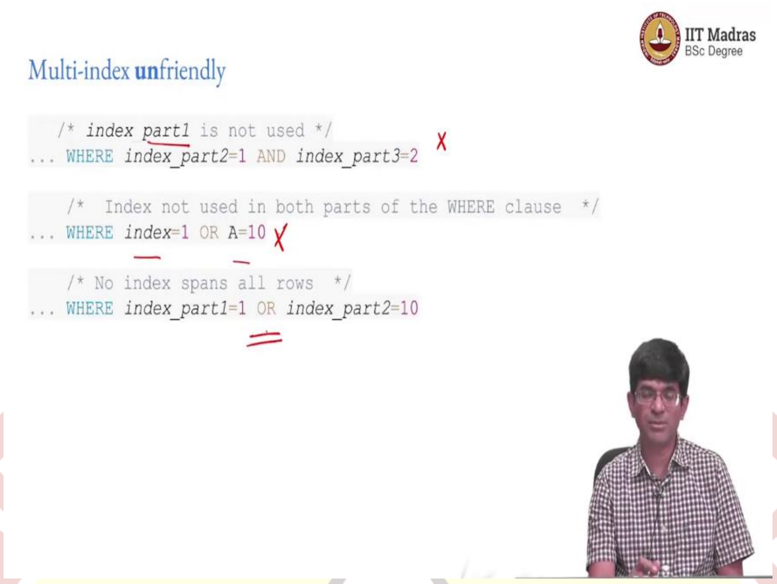
Multi-index friendly

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3

    /* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2

    /* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5

    /* use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

So, once again, to this reiterate, if we have multi-indexes, and I had a part which says index, part 1, index part 2, and this other part, it is not really using index part 3, but this is still good. At least it is making use of index part 1 and part 2. Or here, if I have index =1, at least this part of it is able to use, the index =2 thing, it is not really going to use.

What if I had index_part1=hello, and index_part3=5, but I am not specified what index_part2 is, it will sort of just ignore this and do the indexing based on index_part1. And let us say that, if I had something index=1 index=2 or index=1 and index=3, it can use the index on index 1, but it cannot really make use of the second, the 2 and 3 indexes. So, the search engine inside the database will automatically figure out what is the best that it can do and try and optimize for that.

(Refer Slide Time: 13:15)



Multi-index unfriendly

```
  /* index part1 is not used */
... WHERE index_part2=1 AND index_part3=2

  /*  Index not used in both parts of the WHERE clause  */
... WHERE index=1 OR A=10

  /* No index spans all rows  */
... WHERE index_part1=1 OR index_part2=10
```

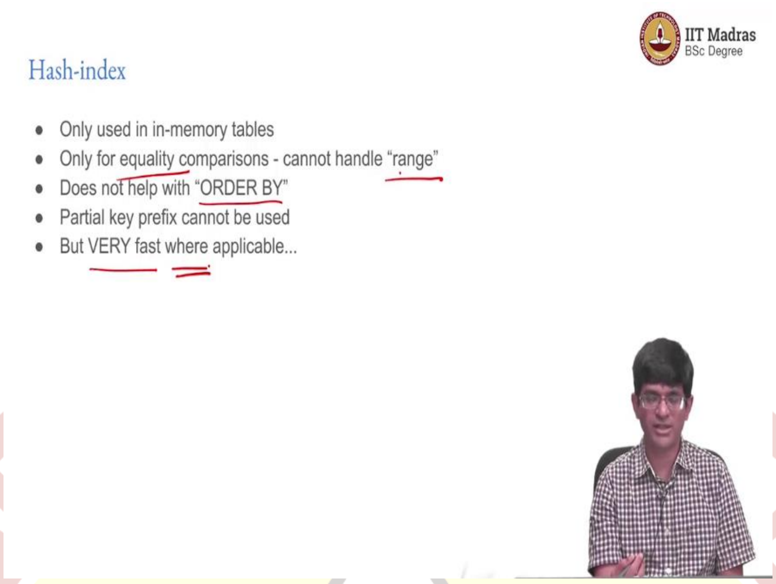
On the other hand, what if I ignore index part 1? This is bad, it basically cannot use that index at all. Over here, index=1 or =10 it is, I cannot really use the index at all, because the OR A =10 is like a completely different thing, which cannot really use an index at all.

And I have something else, which basically says. This is again, say the problem over here, it is an OR statement whereas an index is essentially an AND. It is taking index_part1= something and index_part2=something else an OR breaks that. So, there are many queries that you can easily create in this way, which are unfriendly to these kinds of multi-indexes.

So, should you even use multi-indexes? Should you use only single indexes? All that is essentially something, which you need to go much deeper into databases in order to understand them in detail. Why are they sort of important for an app developer to know? Because there is no unique answer that is going to come from the database side. Only you, as the application developer can say, in this particular case, I will need to access these entries more and therefore, I should create an index on this.

The database at best can give you a recommendation, saying, look, you probably should build an index on this. More importantly, there will be database engines that sort of keep track of the queries that you have been running, and then can sort of recommend options where you should be creating indexes to get better performance.

(Refer Slide Time: 14:52)



Hash-index

- Only used in in-memory tables
- Only for equality comparisons - cannot handle "range"
- Does not help with "ORDER BY"
- Partial key prefix cannot be used
- But VERY fast where applicable...

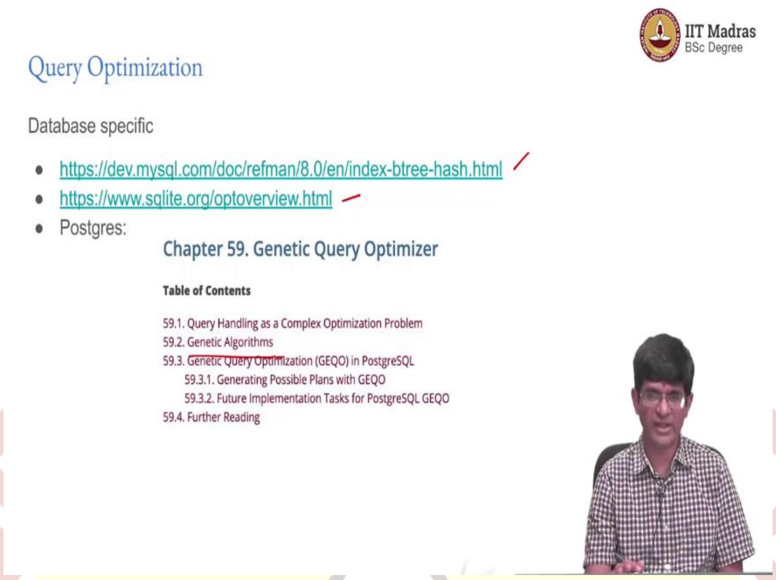
Now, finally, Hash indexes. This is something, which is again available in MySQL. But remember, like I said, the use of a Hash table is only for equality comparisons. In other words, what it is doing is, taking what you are searching for, let us say the name of a person and computing some function of that and directly saying, look, this is where the entry for this person is stored.

If I can create something of that sort, then fantastic. I mean, it is going to be faster than any of the B-Tree-based searching that I could do earlier, but it cannot, for example, handle a range of names. All peoples whose names begin with N; it cannot really do something of that sort. You give it an exact name it can search present or not present.

Similarly, if you need to search for all people whose name begins with N, and you know, order them by age or something of that sort, it does not help. So, where it is applicable, it can be very fast, but the question is, where is it applicable?

It is provided because there can be applications where you need to use something of this sort. MySQL, for example, does have something of the sort, so if you find that it is the right fit for your application you should use it.

(Refer Slide Time: 16:04)



Query Optimization

Database specific

- https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html
- https://www.sqlite.org/optoverview.html
- Postgres:

Chapter 59. Genetic Query Optimizer

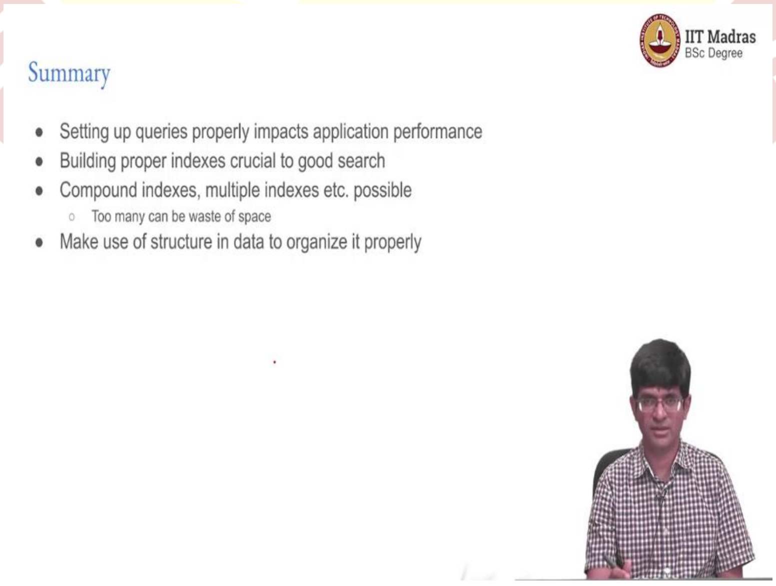
Table of Contents

59.1. Query Handling as a Complex Optimization Problem
59.2. Genetic Algorithms
59.3. Genetic Query Optimization (GEQO) in PostgreSQL
59.3.1. Generating Possible Plans with GEQO
59.3.2. Future Implementation Tasks for PostgreSQL GEQO
59.4. Further Reading

So, various databases have their own takes on query optimization. MySQL has a whole lot of information, like I said, what I just showed you, SQLite lite has something about optimization, overdue. Postgres, in fact, has even, it is sort of a research database in some sense and they even have things like, using genetic algorithms in order to optimize query structures.

So, these are all different possibilities that can be done. And in fact, is, to some extent is the topic of research even now. A large part of databases are very well studied so there is not too much new happening out here, but on the other hand, there are places where you might be able to make some fundamental changes as well.

(Refer Slide Time: 16:52)



Summary

- Setting up queries properly impacts application performance
- Building proper indexes crucial to good search
- Compound indexes, multiple indexes etc. possible
  - Too many can be waste of space
- Make use of structure in data to organize it properly

So, to summarize, setting up the queries properly can have a huge impact on the application performance, and building proper indexes is crucial to search. Now, you might think that, I am only going to build small apps, but you do not know at what point it stops being small and starts being significant. Because even something with a few hundred entries in a database can get really badly impacted if your indexes and your data store storage mechanisms are bad.

The moment you hit a few 1000 already log to base 2 of 1000 is 10, which means that either you are taking 1000 steps to find something or you are taking only 10 steps, so you have a factor of 100 difference in terms of what you are likely to spend in searching for something. That is a very rough estimate, but still you get the idea. So, do not sort of underestimate the value of creating the right kind of indexes.

Now, creating too many indexes, on the other hand is a bad idea. You will end up wasting space and it just confuses the database. It does not know how to search and or rather, which index is the right one to use. And ultimately, this whole idea of how to structure the data and put it properly the so called normalization step that is used in databases, which again, is completely out of scope of this course that is something, which you need to pay attention to especially I f you expect your application to grow larger with time.