

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 29
Indexing and Hashing/4 : Hashing

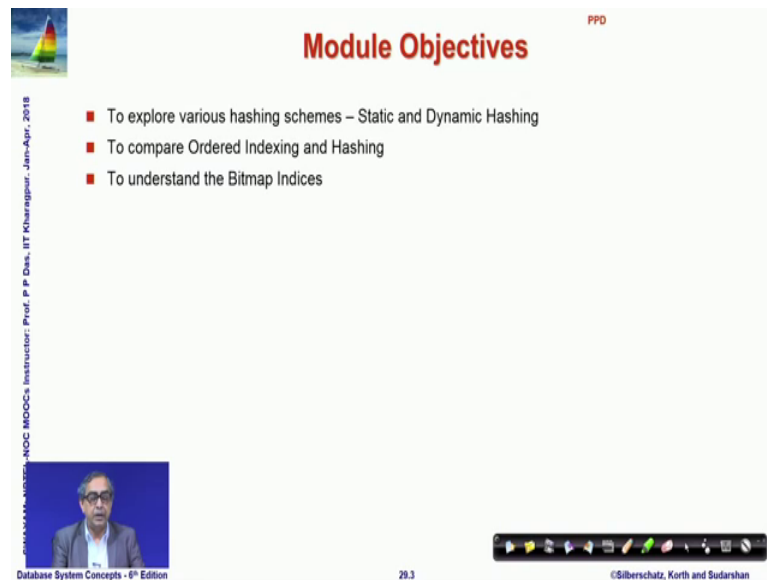
Welcome to module 29 of Database Management Systems; we have been talking about indexing and hashing and this is a fourth in the series.

(Refer Slide Time: 00:26)

The slide is titled "Module Recap" in red text. It features a small sailboat icon in the top left corner. On the left side, there is vertical text: "SWAYAM: NPTEL-NOC MOOC's Instructor: Prof. P. P. Das, IIT Kharagpur, Jan-Apr, 2018". The main content is a list of two items, both preceded by a red square bullet point: "B*-Tree Index Files" and "B-Tree Index Files". At the bottom of the slide, there is a navigation bar with several icons. The footer contains the text "Database System Concepts - 6th Edition", "29.2", and "©Silberschatz, Korth and Sudarshan".

In the previous 3 we have talked about different aspects of indexing and specifically in the last module, we have introduced the most powerful data structure B plus tree for index files.

(Refer Slide Time: 00:39)



PPD

Module Objectives

- To explore various hashing schemes – Static and Dynamic Hashing
- To compare Ordered Indexing and Hashing
- To understand the Bitmap Indices

Database System Concepts - 6th Edition

©Silberschatz, Korth and Sudarshan

29.3

Database System Concepts - 6th Edition

©Silberschatz, Korth and Sudarshan

29.3

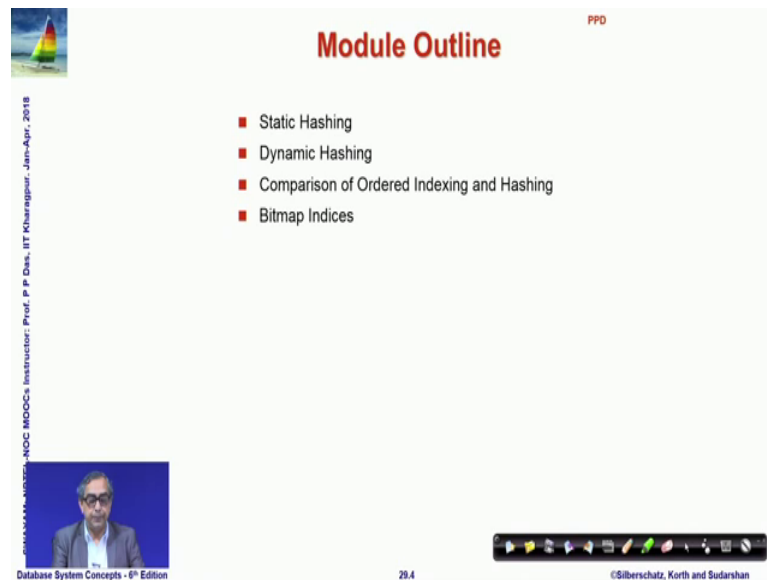
Database System Concepts - 6th Edition

©Silberschatz, Korth and Sudarshan

29.3

In this module, we will take a look into a explore into various hashing schemes for achieving the similar targets we will look at static and dynamic hashing. And we will then compare it between the ordered indexing that we have discussed already and hashing and we will also understand about what are called bitmap indices.

(Refer Slide Time: 01:03)



PPD

Module Outline

- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices

Database System Concepts - 6th Edition

©Silberschatz, Korth and Sudarshan

29.4

Database System Concepts - 6th Edition

©Silberschatz, Korth and Sudarshan

29.4

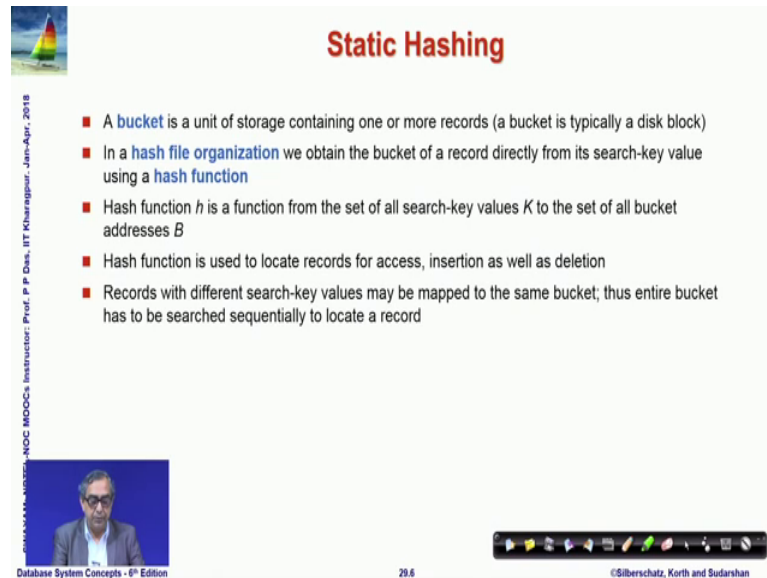
Database System Concepts - 6th Edition

©Silberschatz, Korth and Sudarshan

29.4

So, these are the module outline.

(Refer Slide Time: 01:07)



Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block)
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B
- Hash function is used to locate records for access, insertion as well as deletion
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record

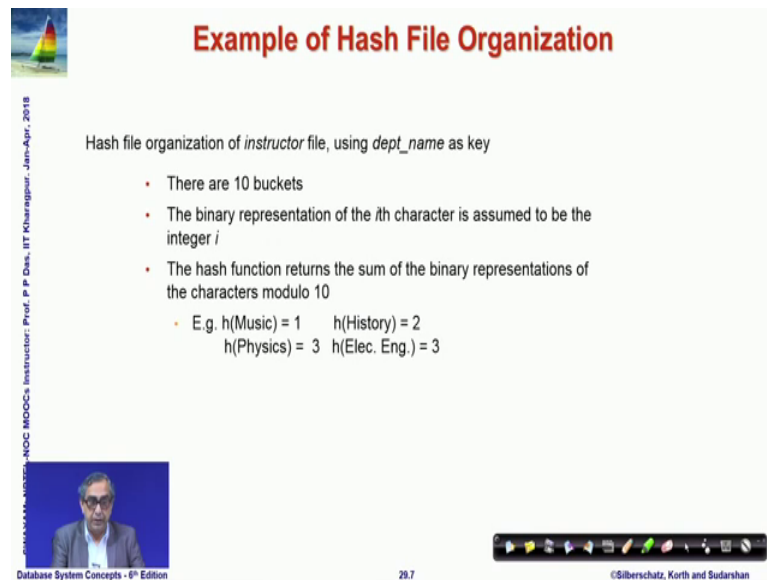
Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

So, static hashing I am assuming that all of you know about basic concept of hashing. So, it what it does a there is a bucket is a unit of storage containing one or more records. So, that is the basic logical concept typically in physical terms a bucket can be a disk block. So, a hash file organization obtains we in a hash file organization; we attempt to obtain a bucket for a record directly from its search key value using a hash function.

So, this is where it becomes very different compared to the ordered indexing for which we saw all this I some method and the B plus tree where we went through different index structure, but here we want to make use of a mathematical hash function. So, that given the key ideally I should be able to get the bucket in which that particular record containing the key exists that is the requirement.

So, hash function h is a function from the set of all search key values K to the set of all bucket addresses B . So, it is a mathematical function and it is used to locate the records for access insert as well as delete records with different search key values may be mapped to the same bucket right this is not what ideally we wanted, but it is possible there is a entire bucket has to be searched sequentially once you reach there to look at a record, we can make use of other techniques there we will come to that.

(Refer Slide Time: 02:33)



Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

- There are 10 buckets
- The binary representation of the *i*th character is assumed to be the integer *i*
- The hash function returns the sum of the binary representations of the characters modulo 10
 - E.g. $h(\text{Music}) = 1$ $h(\text{History}) = 2$
 $h(\text{Physics}) = 3$ $h(\text{Elec. Eng.}) = 3$

Database System Concepts - 6th Edition
©Silberschatz, Korth and Sudarshan

So, let us take a quick example hash file organization of an instructor file using say department named as key. So, we need to design a hash function. So, let us assume that on the address space B we have 10 buckets. So, every bucket is designated by a by a serial number bucket 0 to bucket 9 and we take department name is a key. So, it is a character string.

So, we take the binary representation of the *i*th character and assume it to be the integer *I* simply every character you take its binary representation and think as if it is an integer. And then as a hash function we add these integer values of binary representations modulo 10. So, *M* hash value of music we take the binary representation of *m* which is the as key code of *M* capital *M*; then add the as key code of *u* the lower case *u* and so, on and do that modulo 10 and we get a value which is 1.

So, naturally since we are doing modulo 10 which is the number of buckets here. So, we will get a result for the hash function which is between 0 to 9 which, is a bucket address where it is expected.

(Refer Slide Time: 03:51)

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califfieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key

Database System Concepts - 6th Edition 29.8 ©Silberschatz, Korth and Sudarshan

So, here we are showing an example. So, you can see in the earlier slide we are showing music is 1 history is 2 physics and electrical engineering both are hash value 3.

So, you can see here in bucket 2 since history has value 2. So, those records El Said and Califfieri records go to bucket 2 whereas, physics and electrical engineering both have hash value 3. So, that Einstein golden came all go to the bucket 3 similarly it happens with the other buckets as well not all buckets are shown here shown only 8 buckets are shown, but in this way we can actually directly map them to the buckets.

(Refer Slide Time: 04:33)

Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file
- Typical hash functions perform computation on the internal binary representation of the search-key
 - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned

Database System Concepts - 6th Edition 29.9 ©Silberschatz, Korth and Sudarshan

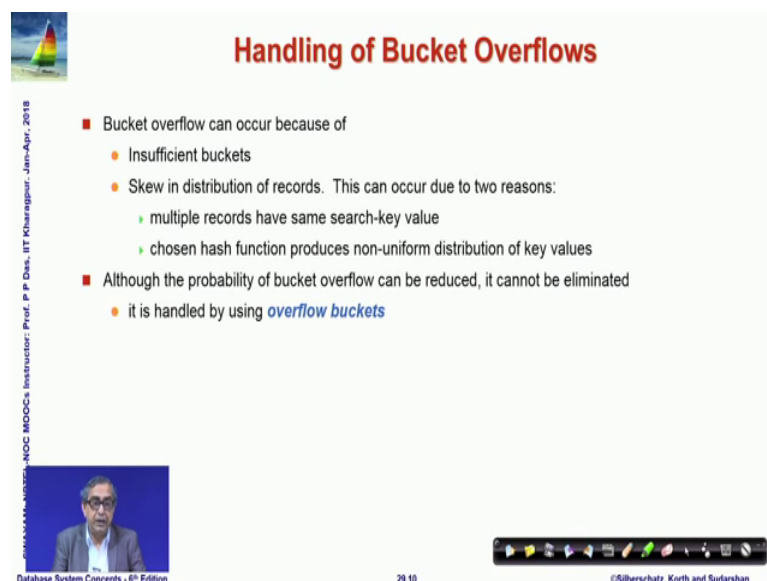
And; so, such a hash function would be really useful now the question is a it is a mathematical function; so, how good or how bad it is. So, we will say that the worst possible hash function is one which maps all key values to the same bucket. So, that everything will have to within them serially; so, that is of no use.

So, the ideal one would be which will distribute the different search keys values in different buckets in an uniform manner to the from the set of all possible values. So, that would be that will be nice to have and ideal would be that if the hash function is random which means that. So, that each bucket will I mean it will generate from the key value it will generate the bucket number, it will generate the bucket address in kind of a random manner.

So, that in a random phenomena; so, that irrespective of what kind of actual distribution the search keys may have the buckets over which the distribute will be more or less the same. So, every bucket will have same number of records things will be balanced.

A typical hash function performs computation on the internal binary representation of the search key that is the basic that that is the one that you have just seen. So, if it is a string then you treat the characters as they are binary representations as integer do some modulo a number exactly what we did in the last case.

(Refer Slide Time: 05:11)



Handling of Bucket Overflows

- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - ▶ multiple records have same search-key value
 - ▶ chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated
 - it is handled by using *overflow buckets*

Database System Concepts - 8th Edition 29.10 ©Silberschatz, Korth and Sudarshan

Now the question is the buckets have a certain size we said that a bucket is a disk block. So, a bucket can overflow because there may not be enough sufficient buckets to keep all the records. So, it will not fit in or your distribution could be skewed. So, there may be many buckets where there are lot of space left, but some buckets may have a too many records coming on to it because of the behavior of the hash function. So, that multiple records have the same key value or chosen hash function produces non uniform distribution and so, on.

So, if that happens then the probability of bucket flow; bucket overflow will happen and we can try to reduce that, but it cannot be eliminated. So, all that you do is to have overflow bucket which is nothing, but having other buckets connected to this target bucket in a chain.

(Refer Slide Time: 07:04)

The slide, titled "Handling of Bucket Overflows (Cont.)", features a small image of a sailboat in the top left corner. The main content includes a bulleted list and a diagram. The list defines "Overflow chaining" as overflow buckets of a given bucket being chained together in a linked list, notes that this scheme is called "closed hashing", and mentions an alternative called "open hashing" which does not use overflow buckets and is not suitable for database applications. The diagram shows four buckets labeled "bucket 0", "bucket 1", "bucket 2", and "bucket 3" arranged vertically. From the right side of "bucket 1", an arrow points to a chain of two additional boxes labeled "overflow buckets for bucket 1". A small video inset of a man is in the bottom left, and a navigation bar is at the bottom right. Text at the bottom left reads "Database System Concepts - 6th Edition" and "©Silberschatz, Korth and Sudarshan". The slide number "29.11" is at the bottom center.

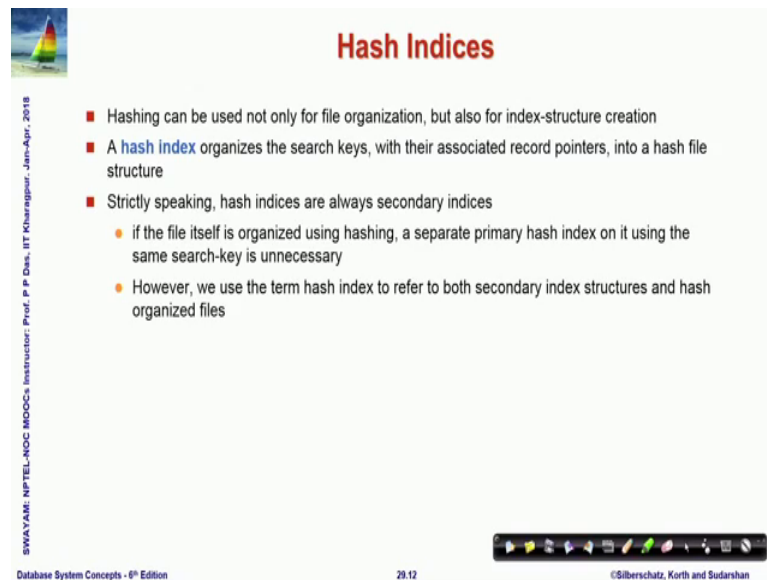
- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list
- Above scheme is called **closed hashing**
- An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications

bucket 0
bucket 1
bucket 2
bucket 3

overflow buckets for bucket 1

So, this is called a overflow chaining as you can see there are 4 buckets shown here and bucket 1 we are saying showing are connected with other two buckets which are the overflow buckets for bucket 1. So, that this kind of a scheme is called closed hashing there is an alternate scheme called open hashing, which does not use a bucket overflow and, but it is not therefore, suitable for database applications and we will not discuss it here.

(Refer Slide Time: 07:31)



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure
- Strictly speaking, hash indices are always secondary indices
 - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
 - However, we use the term hash index to refer to both secondary index structures and hash organized files

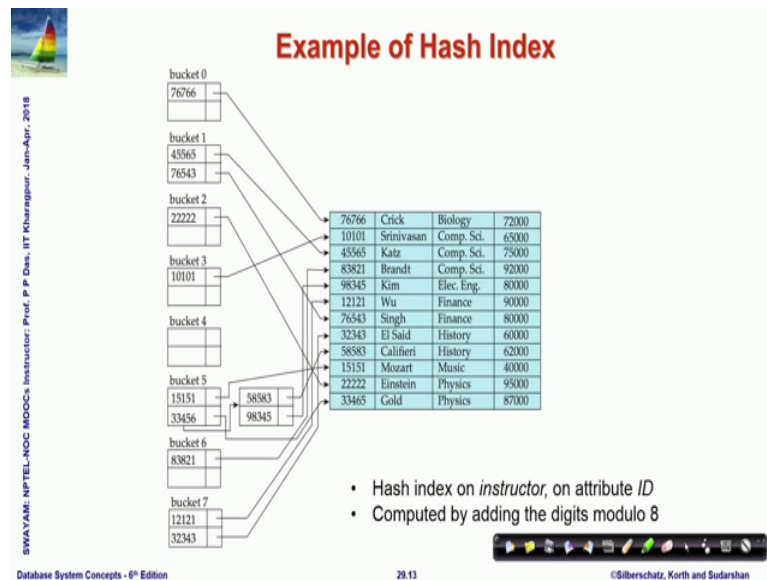
SWAYAM: NPTEL-NOC MDOCS Instructor: Prof. P. P. Das, IIT Khargpur, Jan-April, 2018

Database System Concepts - 8th Edition 29.12 ©Silberschatz, Korth and Sudarshan

So, hash indices can be used not only for file organization, but they can also be used for indexed structure creation like we did for B plus tree we can use the hash indices for index structure also. So, hash index organizes the search keys with their associated record pointers into a hash file structure exactly in the same way its hashing otherwise.

So, but the you can note that the hash indices are always kind of secondary indices because if a file itself is organized using hashing; then a separate primary hash index on it using the same search keys are necessary. Because if if you are talking about primary hash indexing then it will mean that you are taking the primary search key and creating a hash index on that, but if the file is hash created by hash indexing then that already exists. So, anything that you create in terms of indexing is basically a secondary indexing structure in a hash organized file.

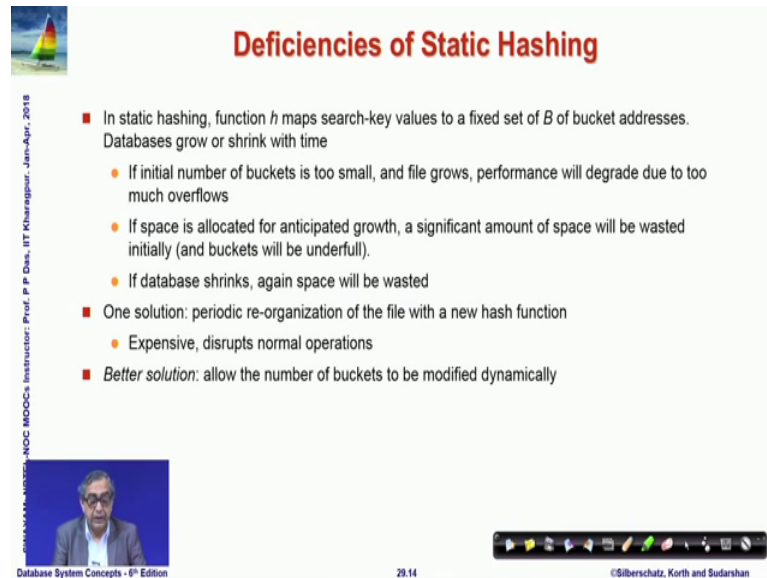
(Refer Slide Time: 08:38)



So, this is kind of hash indexing example. So, here I am I am showing the hash indexing with the ID of this table and the index is computed by adding the digits modulo 8 assuming that there are 8 buckets. So, if you take; so, if you look at bucket 0 then the key that has gone there is 76766 which is 7 plus 6; 13, 20 plus 6; 26 plus 6 32 modulo 8 is 0.

So, it goes into bucket 0 it happens that way if, but if we look into bucket 4 you will find that the 4 IDs actually all have this value 5 under the hash function. So, they all need to go to this bucket and therefore, but the bucket size assumed here is just 2. So, after the 2 indices have gone in there a overflow chain is created and another overflow bucket is used to keep the next two IDs there. So, this is how a hash index can be created.

(Refer Slide Time: 09:45)



Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows
 - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
 - If database shrinks, again space will be wasted
- One solution: periodic re-organization of the file with a new hash function
 - Expensive, disrupts normal operations
- *Better solution*: allow the number of buckets to be modified dynamically

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

Now, this is this kind of a scheme where you start with a fixed number of buckets and then you design a hashing function which maps the search key values to this fixed set of buckets is known as a static hashing it is static because you start with a fixed number of buckets.

So, yeah naturally the question is what should be this value of B the number of buckets. Now if it is initially too small then the file keeps on growing the performance will degrade because you will have too many overflow chains and if the all advantages of having done the hashing will get lost. On the other hand if you take a too large a B then you will unnecessarily allocate a lot of space anticipating growth, but it may take a very significant amount of time to utilize that that space or also it is possible that it the database at certain point of time grew to a large size and then it started shrinking and then again space will get wasted.

So, static hashing has these limitations. So, naturally what you will have to do is to periodically reorganize the file with a new hash function which is certainly very expensive because it changes the positions of all records. So, it disrupts the normal operation; so, it would be better if we could allow to change the number of buckets to be changed dynamically at the as the database grows. So, if the database grows it can use more and more buckets and if we could adjust this in the hashing scheme inherently; then

it will certainly be better as a solution. So, that gives rise to what is known as dynamic hashing.

(Refer Slide Time: 11:30)

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
 - Hash function generates values over a large range — typically b -bit integers, with $b = 32$
 - At any time use only a prefix of the hash function to index into a table of bucket addresses
 - Let the length of the prefix be i bits, $0 \leq i \leq 32$
 - Bucket address table size = 2^i . Initially $i = 0$
 - Value of i grows and shrinks as the size of the database grows and shrinks
 - Multiple entries in the bucket address table may point to a bucket (why?)
 - Thus, actual number of buckets is $< 2^i$
 - The number of buckets also changes dynamically due to coalescing and splitting of buckets

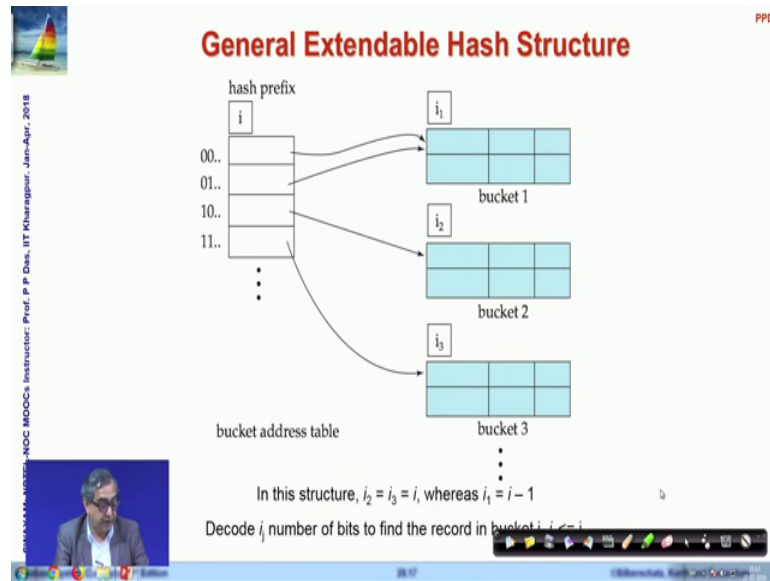
So, it is certainly good for databases that regularly grows and shrinks in size, allows the hash function to be modified dynamically. Of the different dynamic hashing schemes I will discuss the extendable hashing which is a very popular scheme. So, let us see what how it works. So, it at the hash function will generate the value over a large range say typically a B bit integer say 32 bit integer now.

So, what you have is you have generated a value which is hash value which is say over 32 bits, but what you do at any time you use only a prefix of that; you only use a part frontal part of that to index the table to the bucket address and the length of that prefix is i bits; then naturally it could be at least theoretically it could be 0 that is you do not use any prefix and it could be up to that you use all the prefixes.

And so, therefore, if you are using i bits then the bucket address table the possible you know bucket addresses that you could have is 2 to the power i initially you keep that as 0.

So, then the address table will actually point to different buckets let us start moving to an example and see what is happening.

(Refer Slide Time: 12:57)

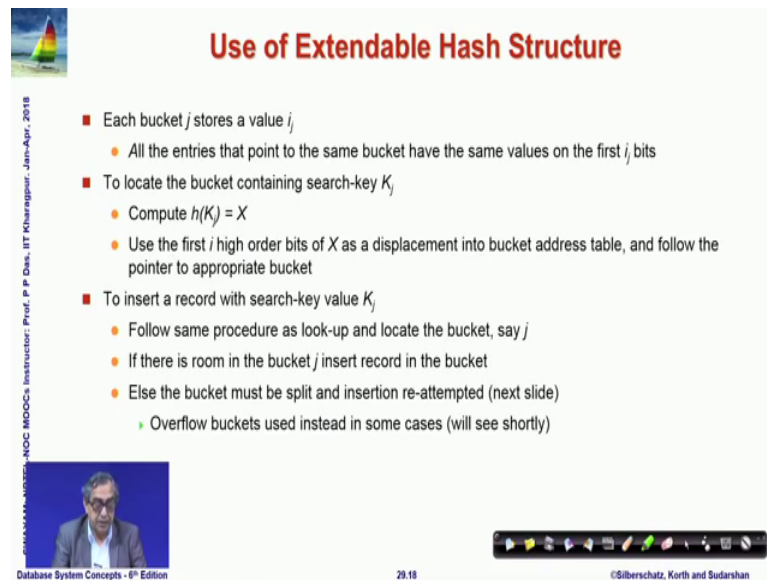


So, this is the general scheme. So, you have a hash prefix which is using i number of bits and therefore, different values of i number of bits. So, there will be 2 to the power i entries naturally you have different buckets here, but you may not actually have all 2 to the power i buckets you may have less than that as it is shown here that bucket 2 and bucket 3 exist, but bucket 1 is a holder for both this prefix 0 0 as well as prefix 0 1.

So, and on top of every bucket you have a kind of bucket depth given. So, it is a number of bits that you need to explore in the in the representation in the; so, that you can distinguish the different records of that bucket.

Naturally the maximum value of any of these i is the i here, but it could be less than that. So, I am I am sure this is this probably is not making much sense immediately. So, let me move to my detailed discussion.

(Refer Slide Time: 14:19)



Use of Extendable Hash Structure

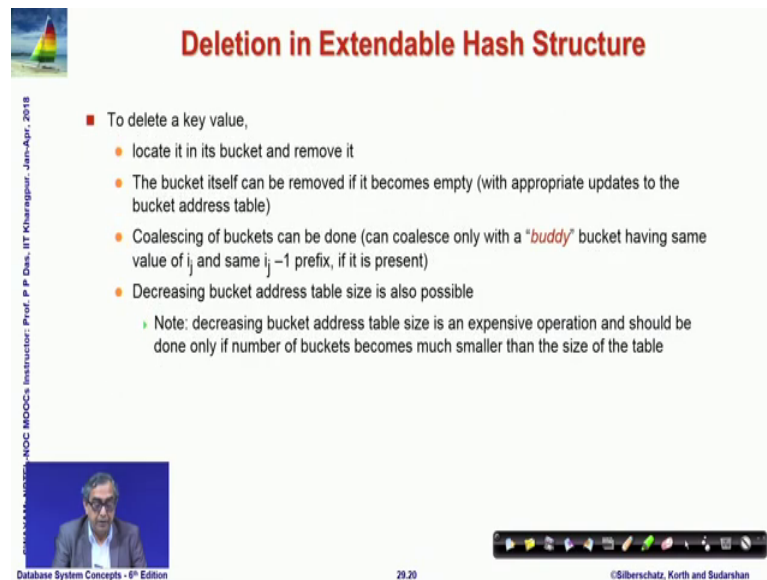
- Each bucket j stores a value i_j
 - All the entries that point to the same bucket have the same values on the first i_j bits
- To locate the bucket containing search-key K_j
 - Compute $h(K_j) = X$
 - Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
 - Follow same procedure as look-up and locate the bucket, say j
 - If there is room in the bucket j insert record in the bucket
 - Else the bucket must be split and insertion re-attempted (next slide)
 - ▶ Overflow buckets used instead in some cases (will see shortly)

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

So, what I was saying that each bucket j stores a value i_j . So, this is all the entries that point to this bucket has the same value on the first i_j bits. So, these i_j bits are identical. So, all of them have come to this bucket. So, how do you look at the bucket that contains the search key K_j ? So, it computes the hash function which is X and uses the prefix i_j bits of X as a displacement into the bucket address table and follows the pointer to the appropriate bucket.

Now if I have to insert a record with a search key K_j ; you will follow that same procedure as a lookup and look at the bucket j and then you will have to look for making some space. So, let me do something.

(Refer Slide Time: 15:15)



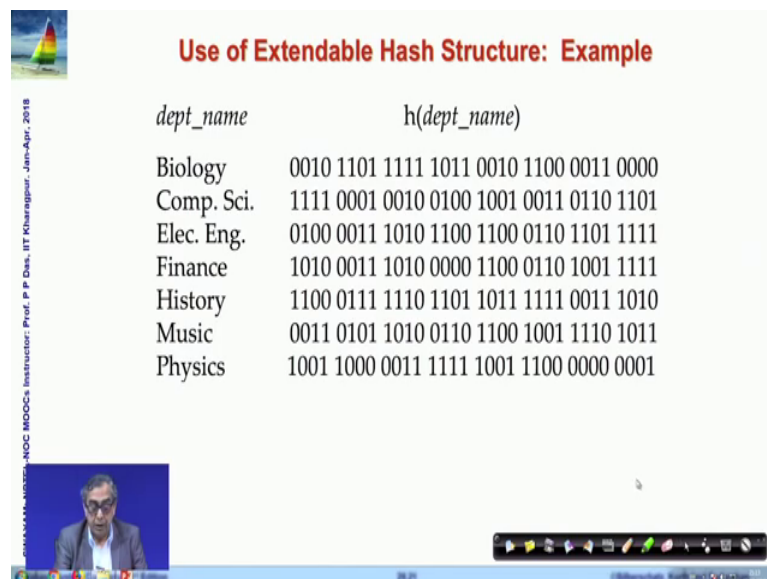
Deletion in Extendable Hash Structure

- To delete a key value,
 - locate it in its bucket and remove it
 - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table)
 - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of i_j and same i_{j-1} prefix, if it is present)
 - Decreasing bucket address table size is also possible
 - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Database System Concepts - 8th Edition ©Silberschatz, Korth and Sudarshan

Let me before going through this statement of the algorithm.

(Refer Slide Time: 15:20)



Use of Extendable Hash Structure: Example

<i>dept_name</i>	<i>h(dept_name)</i>
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Database System Concepts - 8th Edition ©Silberschatz, Korth and Sudarshan

Let me just go through an example first and we can come back to this formal statement.

So, what we are trying to do is there is the department names which we are using as a key to do this hashing index and they are represented in terms of the binary representation. So, this is this is the hash of that department name and hashed into you can you can easily see this is 1, 2, 3, 4, 5, 6, 7, 8. So, this is hashed into 32 bit number.

(Refer Slide Time: 16:01)

Example (Cont.)

Initial Hash structure; bucket size = 2

hash prefix: 0

bucket address table: []

bucket 1: [] []

Insert "Mozart", "Srinivasan", and "Wu" records

dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
43565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	IJ Saad	History	60000
58583	Callieri	History	62000
13110	Moran	Music	42000

Now, what do we do? So, initially we start with. So, this is this is all the different hash values that you can see I am sorry this is all the different hash values and this is the table that I need to actually represent. So, initially there is nothing. So, I try to I will try to insert Mozart Srinivasan and these 3 records here. So, let me try that.

(Refer Slide Time: 16:33)

Example (Cont.)

Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records

hash prefix: 1

bucket address table: [] []

bucket 1: [15151 | Mozart | Music | 40000]
[10101 | Srinivasan | Comp. Sci. | 90000]
[12121 | Wu | Finance | 90000]

stein record

dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
43565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	IJ Saad	History	60000
58583	Callieri	History	62000
13110	Moran	Music	42000

So, if I look at Mozart then Mozart is from the department of music. So, and Srinivasan is from computer science Wu is from finance. So, let us look at this. So, Mozart is from music Srinivasan is from computer science and Wu is from finance. So, these are the 3

now if we look into the prefixes of their hash values; you can see that their hash values are 1 1 and for music it is 0 right.

So, if I use a hash prefix which has just one bit and naturally therefore, I have two entries 0 and 1 then music with the value 0 maps to this bucket where I entered the record for music. And computer science and finance the records corresponding to them has a hash value prefix 1. So, they both map to discipline this is how it can get started. So, you you find out while inserting you find out where is Mozart and based on that you create this.

Now, let us try to insert Einstein.

(Refer Slide Time: 18:09)

Example (Cont.)

dept_name h(dept_name)

Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

■ Hash structure after insertion of Einstein record

hash prefix

bucket address table

1	15151	Mozart	Music	40000
2	12121	Wu	Finance	90000
2	22222	Einstein	Physics	95000
2	10101	Srinivasan	Comp.Sci.	65000

and El Said records

76786	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Caliberi	History	62000
11101	Mozart	Music	40000

Database System Concepts - 9th Edition 29.24 ©Silberschatz, Korth and Sudarshan

So, to insert Einstein what do we find? So, what all we already have? We have music, we have computer science, we have finance and now Einstein comes in Einstein is from physics. So, what would happen when you try to insert Einstein? You already had computer science with 1 as 1 prefix and finance with 1 prefix.

So, you had in bucket two corresponding to 1 you already have 2 records that bucket is full assuming that it can take only 2 records. So, now, you get another which is value 1; so, its value is 1. So, what I need to do? I need to actually expand this now how do I do that? I cannot expand this because there is no more space left. So, all that I need to do is to actually expand the bucket address table.

So, earlier if I if I just if I just go back. So, if I just go back earlier we had only two entries because we are using only 1 bit in the prefix and with that I could not have inserted Einstein oh is from department of physics which also has a 1 bit prefix which is 1 it was. So, I need more space; so, I have increased the prefix level to 2 going here and now naturally I have if I have increases to 2; now I have let me erase this these entries and now I look at for music I look at 2 for physics 1 0, for finance 1 0, for computer science 1 1.

So, now, I find that after I have moved from looking at 1 bit of prefix to 2 bits of prefix now finance and computer science which was earlier together because I was looking at 1 bit now becomes different, but finance and physics both come to the same 1 0.

So, in the hash bucket address table 1 0 you have financial physics coming in with Wu and Einstein records and computer science which has got 1 1 the Srinivasan record goes to a new bucket which comes from 1 1 here. Now the interesting fact is what happens to Mozart who which was there if you if you remember the earlier structure this is we had only 1 here. So, this was going to Mozart this was 1 and Mozart was here because music had a prefix 0; now music has a prefix 0 0.

So, what he would have expected? You would have expected that therefore, since 0 0 has come and 0 1 has also come in. So, you would have expected that to have another bucket here which 0 1 is, but then you observe that actually that would be a quite a wasteful to do because you do not actually have a record which has a prefix 0 1.

Out of these two which are we are looking at two prefixes, but you do not really right now need to look at both the prefixes you can still resolve just by based on the first prefix 1. So, you do a simple trick you do not change the prefix level of the particular bucket you say it is 1. Because it is you just need to look at one bit to be able to come to the records in this bucket and the globally it has changed to 2 bits prefix, but locally you keep it as 1.

And with that what you have? You have 0 1 which has a bucket address table entry actually does not have a bucket because there is no records for that. So, you let that point to the same bucket. So, this is a very critical observation that these numbers are basically the local depth; the local information of how many bits in the prefix you need to look at to be able to resolve for coming to this bucket for the records that you currently have.

Whereas this is the global one this is a global maximum that you have. So, naturally local depth cannot exceed the global depth, but if it is equal then you have a unique mapping from the bucket address table entry to the bucket, but if the local depth as in here is less than the global depth; in terms of the number of prefix bits you are looking at then multiple bucket address table pointers actually end up in the same bucket and that is the main principle of this algorithm we can just continue further inserting gold and said into this.

(Refer Slide Time: 23:47)

Example (Cont.)

dept_name h(dept_name)

Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

■ Hash structure after insertion of Gold and El Said records

hash prefix

3	15151	Mozart	Music	40000
3	22222	Einstein	Physics	95000
3	33456	Gold	Physics	87000
3	12121	Wu	Finance	90000
2	10101	Srinivasan	Comp. Sci.	65000
2	32343	El Said	History	60000

■ Insert Katz record

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98545	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
38983	Callifert	History	62000
11111	Morgan	History	60000

So, as you try to insert gold and said gold is also from physics which we already had. So, physics and said is from history which we did not have. So, history finance computers let me let me just mark them by the side. So, you have now computer science, finance, history, music and physics.

Now, you will find that you need to you now have physics is 1 0 and you have two records for that and music is continues to be 0 0 ah; obviously, history is 1 1 same as computer science. So, that has to go on this and finances on 1 0 now, but what happens is when you try to do this; you could not have inserted more records because you have run out of space in the buckets. So, again you have run out of that; so, you need to expand in terms of the number of bits that you look at.

So, you increase that to 3 and now you have 0 0 0 to 1 1 1, but as I have explained not all buckets really need to look into. So, many bits Mozart this bucket continues to B with a

local depth of 1 because if you look into all these 4 different cases; then music is the only one which has a prefix 0, everyone else has a prefix 1. So, if I know that it is 0 then it comes to only this bucket and nowhere else consequently all these 4 bucket address pointers actually go to this bucket table.

Whereas these two for physics I have 1 0 and for finance we have 1 0 here and these come to. So, physics now is looking into 3; so, it is 1 0 0 finance is into 3 it is 1 0 1. So, both physics and finance go to different buckets; now coming to computer science it is 1 1 1 and there is no. So, computer science is 1 1 1 and there is no 1 1 0. So, the 1 1 0 bucket address table pointer continues to point to the same bucket and the local depth value is just 2 less than 3 in the global table.

So, this is the basic process of doing dynamic hashing. So, I will not ah; so, the whole example in terms of this table I have given here worked out.

(Refer Slide Time: 26:49)

Example (Cont.)

■ Hash structure after insertion of Katz record

hash prefix	bucket	dept_name	h(dept_name)	
1	15151	Mozart	Music	40000
3	22222	Einstein	Physics	95000
3	33456	Gold	Physics	87000
3	12121	Wu	Finance	90000
3	32343	El Said	History	60000
3	10101	Srinivasan	Comp. Sci.	65000
3	45565	Katz	Comp. Sci.	75000

■ Insert Singh, Califieri, Crick, Brandt record

dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

76796	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98343	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
75101	Cricket	History	751000

Database System Concepts - 9th Edition 29.26 ©Silberschatz, Korth and Sudarshan

So, you can just go through every step and try to convince yourself.

(Refer Slide Time: 26:54)

Example (Cont.)

dept_name h(dept_name)

Biology	0010 1101 1111 1011 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0111 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

■ Hash structure after insertion of Singh, Califieri, Crick, Brandt records

■ Insert Kim record

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98545	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000

This is an interesting case that happens here where again you come to computer science professors to be entered. So, at level 3 of prefix you have all of them have prefix 1 1 1. So, you would have required to split or increase the prefix level globally the prefix level to 4, but assuming that there is an upper bound on the number of prefix levels; you can do which decides the size of the bucket address table. If that is given to be 3 you certainly cannot increase it further; so, all that you will have to do is actually do a kind of an overflow chain here as well.

So, all of them are 1 1 1 here which brings you to this you cannot find it you go to this and all 1 1 1 ones in future will have to be. So, it is a its kind of a tradeoff between what is the size of the global depth, how many prefixes globally you would like to look at what is the size of every bucket that you will have to maintain and what is the kind of chaining that you will have to accept because of that. So, this is what happens particularly.

(Refer Slide Time: 28:05)

Example (Cont.)

hash prefix

bucket address table

Hash structure after insertion of Kim record

dept_name	h(dept_name)
Biology	0010 1101 1111 1011 1010 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

15151	Mozart	Music	4000
76766	Crick	Biology	7200
98345	Kim	Elec. Eng.	8000
22222	Einstein	Physics	9000
33456	Gold	Physics	8700
12121	Wu	Finance	9000
76543	Singh	Finance	8000
32343	El Said	History	6000
58583	Califeri	History	6200
10101	Srinivasan	Comp. Sci.	6900
45565	Katz	Comp. Sci.	7900
83821	Brandt	Comp. Sci.	9200

76766	Crick	Biology	7200
10101	Srinivasan	Comp. Sci.	6900
45565	Katz	Comp. Sci.	7900
83821	Brandt	Comp. Sci.	9200
98345	Kim	Elec. Eng.	8000
12121	Wu	Finance	9000
76543	Singh	Finance	8000
32343	El Said	History	6000
58583	Califeri	History	6200
15151	Mozart	Music	4000

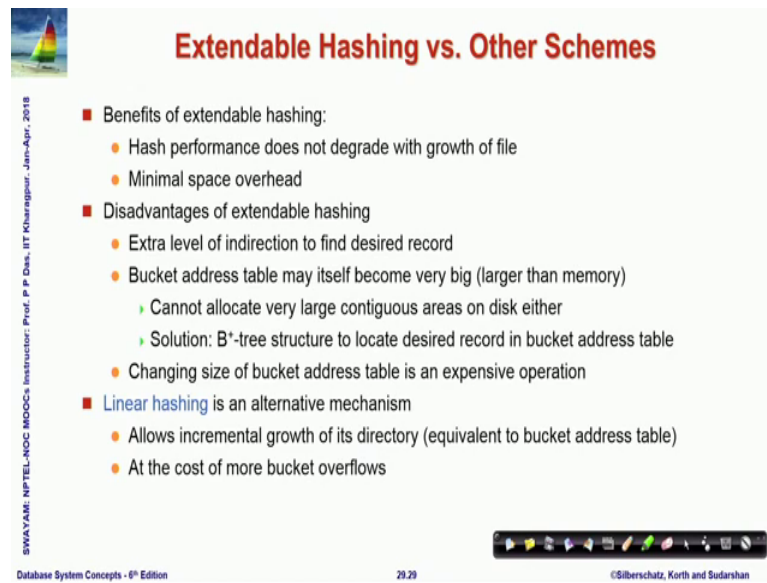
SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P. P. Das, IIT Khargpur, Jan-Apr, 2018

Database System Concepts - 8th Edition 29.28 ©Silberschatz, Korth and Sudarshan

So, you can continue in this way and this is a final table where all things have been hashed well. So, this is the basic extendable hashing scheme it has in this the performance does not degrade with the growth of the file and there is very minimal overhead of the space. But it does have disadvantages for example, there is an extra level of indirection to find the desired record because if the hash then comes to the hash bucket address table and then go to the bucket address table itself may be very big because it is exponential in the size of the number of buckets.

So, it could be larger than memory if that. So, much of you know a contiguous allocation may not be possible. So, you will need to have another possibly a B plus tree structure to locate the desired record in the bucket address table first. And then changing the bucket address table will become a quite an expensive operation. So, the growth will become.

(Refer Slide Time: 28:13)



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
 - Hash performance does not degrade with growth of file
 - Minimal space overhead
- Disadvantages of extendable hashing
 - Extra level of indirection to find desired record
 - Bucket address table may itself become very big (larger than memory)
 - ↳ Cannot allocate very large contiguous areas on disk either
 - ↳ Solution: B*-tree structure to locate desired record in bucket address table
 - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
 - Allows incremental growth of its directory (equivalent to bucket address table)
 - At the cost of more bucket overflows

Database System Concepts - 6th Edition 29.29 ©Silberschatz, Korth and Sudarshan

So, there are several disadvantages that also this scheme has. So, another alternate is to use a linear hashing allows incremental growth of his directory at the cost of more bucket overflows of course,.

(Refer Slide Time: 29:25)



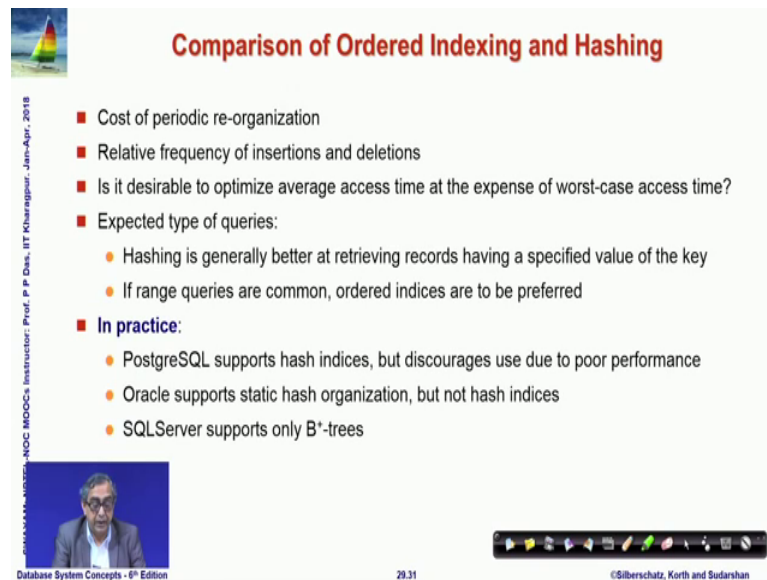
COMPARATIVE SCHEMES

- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Bitmap Indices

Database System Concepts - 6th Edition 29.30 ©Silberschatz, Korth and Sudarshan

I would quickly try to compare the two major schemes that we have discussed.

(Refer Slide Time: 29:30)



Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
 - Hashing is generally better at retrieving records having a specified value of the key
 - If range queries are common, ordered indices are to be preferred
- **In practice:**
 - PostgreSQL supports hash indices, but discourages use due to poor performance
 - Oracle supports static hash organization, but not hash indices
 - SQLServer supports only B⁺-trees

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

The ordered indexing and the hashing now naturally ordered indexing has suffers from the cost of periodic reorganization. And because the indexing will have to be maintained the hashing is better in terms of that relative you will have to look at the relative frequency of insertion deletion that decides much of the cost between going between these two schemes.

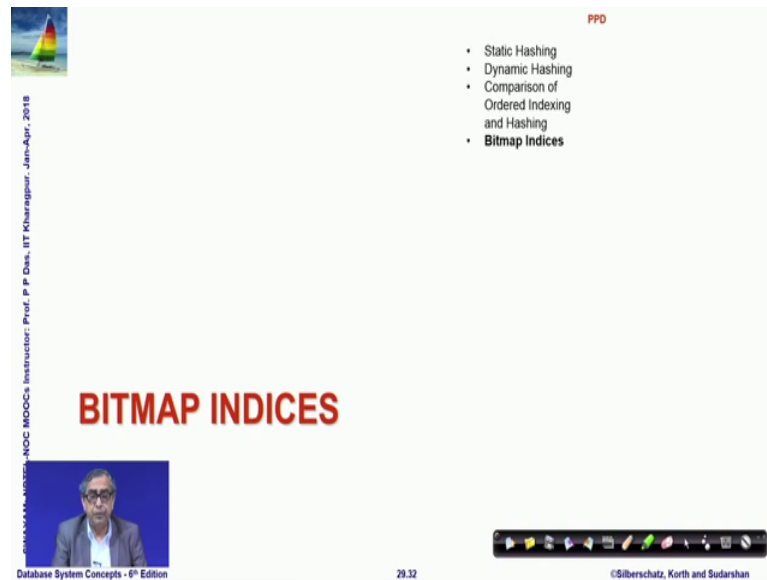
You will have to see is it desirable to optimize average access time at the expense of worst case access time. For example there could be several ways to organize; so, that your average become your worst case may be really really bad, but as long as your averages is very good you should be happy about it. So, those kind of hashing schemes should be more preferred. So, you also depends on the kind of expected type of query.

So, for example, hashing is better in terms of retrieving records which have a specific value of the key because you can directly map from that key to the bucket. And if range queries are common then as we have seen ordered indices would make it make much better sense because in terms of the ordering you can quickly get all the required records at the same physical location nearby physical location.

If you would like to understand as to what the industry practices are it is very mixed. And if you just look into 3 of the very common database systems PostgreSQL does support hash index, but recommends does not recommend it because of the poor performance oracle supports static hash organization, but not hash indices SQL server

supports only B plus trees no hash index space scheme. So, of course, you can see that there as the community is mixed in terms of it is a reaction to whether its indexing or hashing, but hashing powerful at least in limited ways is a powerful technique to go with.

(Refer Slide Time: 31:35)



PPD

- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- **Bitmap Indices**

BITMAP INDICES

Database System Concepts - 6th Edition

29.32

©Silberschatz, Korth and Sudarshan

CHANDAN K. MISRA, NOC MOOCs Instructor, Prof. P. P. Das, IIT Kharagpur, Jan-Apr, 2018

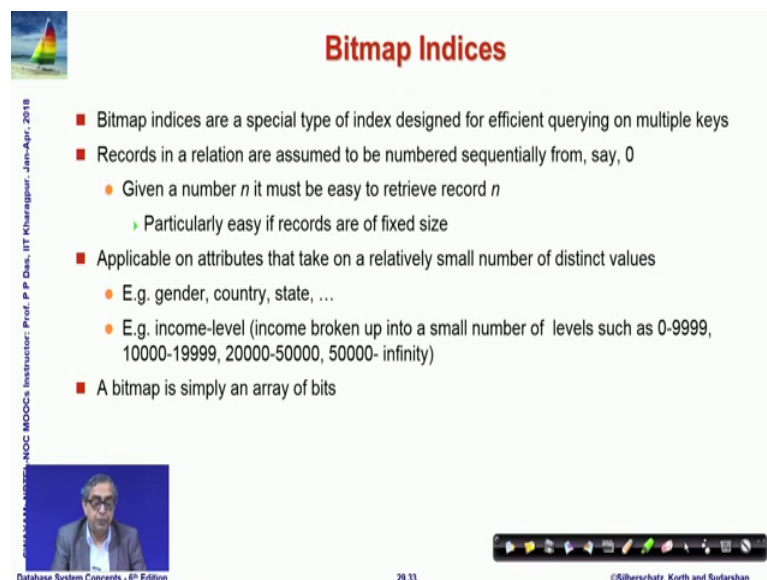
Database System Concepts - 6th Edition

29.32

©Silberschatz, Korth and Sudarshan

The last two that I would like to just quickly remind I mean take you through is what is known as bitmap indexes.

(Refer Slide Time: 31:43)



Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - ▶ Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - E.g. gender, country, state, ...
 - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Database System Concepts - 6th Edition

29.33

©Silberschatz, Korth and Sudarshan

CHANDAN K. MISRA, NOC MOOCs Instructor, Prof. P. P. Das, IIT Kharagpur, Jan-Apr, 2018

Database System Concepts - 6th Edition

29.33

©Silberschatz, Korth and Sudarshan

Bitmap indexing is a very simple idea. So, what you it is a special type of indexing which is designed for querying on multiple keys; the basic idea is that if let us assume

that all records in a relation are numbered from 0 to n and let us say that you are talking about attributes which can take very small number of distinct values.

So, bitmap indexing is not for any attribute. So, consider attributes such a very small number of distinct value say gender which has two possible values or few possible values the country state. So, take those or maybe you can you can nominally bucket a range of numbers source income level 5, 6, 10 income levels. So, small range of possibilities and bitmap is simply array of bits. So, take an array of possible array for the records and for the possible values you mark 1 or 2 0.

(Refer Slide Time: 32:39)

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
- Bitmap has as many bits as records
- In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	ID	gender	income_level
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for gender

m: 10010
f: 01101

Bitmaps for income_level

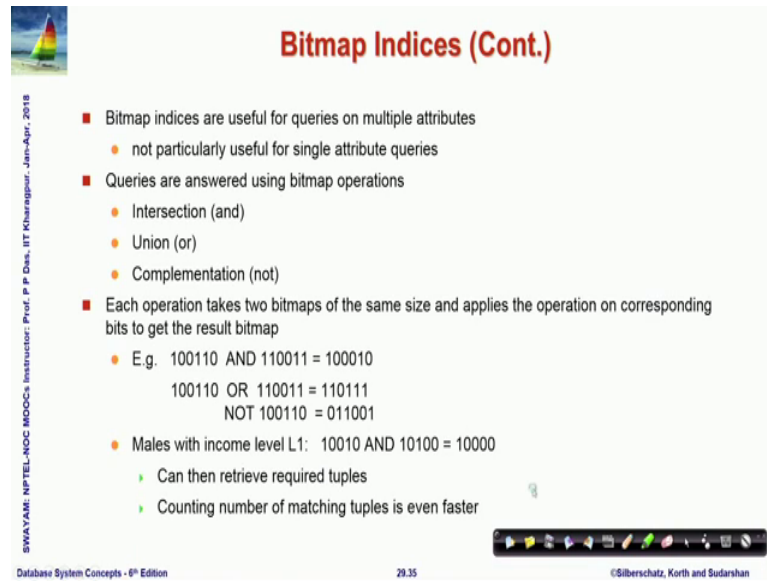
L1: 10100
L2: 01000
L3: 00001
L4: 00010
L5: 00000

Database System Concepts - 9th Edition 29.34 ©Silberschatz, Korth and Sudarshan

So, this here is an example showing it. So, we are showing bitmap index for gender. So, you have a array for m the male gender and f female gender and if you look into the record 076766 has male under m gender m. And therefore, in the male gender bitmap index the first bit is 1 in f it is 0; so, actually m and f are complimentary.

Similarly, for the income levels you have 5 different bitmaps encoding; the 5 different possible levels in the income that you can have.

(Refer Slide Time: 33:21)



Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
 - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
 - Intersection (and)
 - Union (or)
 - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
 - E.g. $100110 \text{ AND } 110011 = 100010$
 $100110 \text{ OR } 110011 = 110111$
 $\text{NOT } 100110 = 011001$
 - Males with income level L1: $10010 \text{ AND } 10100 = 10000$
 - ▶ Can then retrieve required tuples
 - ▶ Counting number of matching tuples is even faster

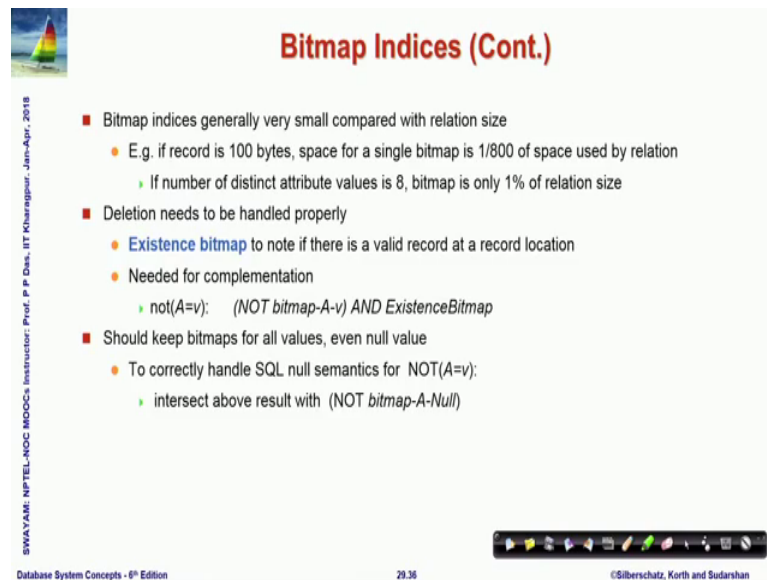
SWAYAM: NPTEL-NOC MDOCS Instructor: Prof. P. P. Das, IIT Khargpur, Jan-Apr, 2018

Database System Concepts - 6th Edition 29.35 ©Silberschatz, Korth and Sudarshan

Now the big advantage of bitmap indices are doing different queries on multiple attributes. And for example, the often queries have intersection union and they can be simply computed in terms of bitmapped operations. So, if you have two different values to be two conditions to check in terms of bitmap indices; then you can just make there and whatever satisfy.

So, say if you are looking at males at for example, here males at income level L 1, then you can you can just take the bitmap for gender and bitmap for income level and do the ending and you get that the first record has value 1.

(Refer Slide Time: 34:08)



Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
 - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation
 - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
 - Existence bitmap to note if there is a valid record at a record location
 - Needed for complementation
 - $\text{not}(A=v)$: $(\text{NOT bitmap-}A-v) \text{ AND ExistenceBitmap}$
- Should keep bitmaps for all values, even null value
 - To correctly handle SQL null semantics for $\text{NOT}(A=v)$:
 - intersect above result with $(\text{NOT bitmap-}A\text{-Null})$

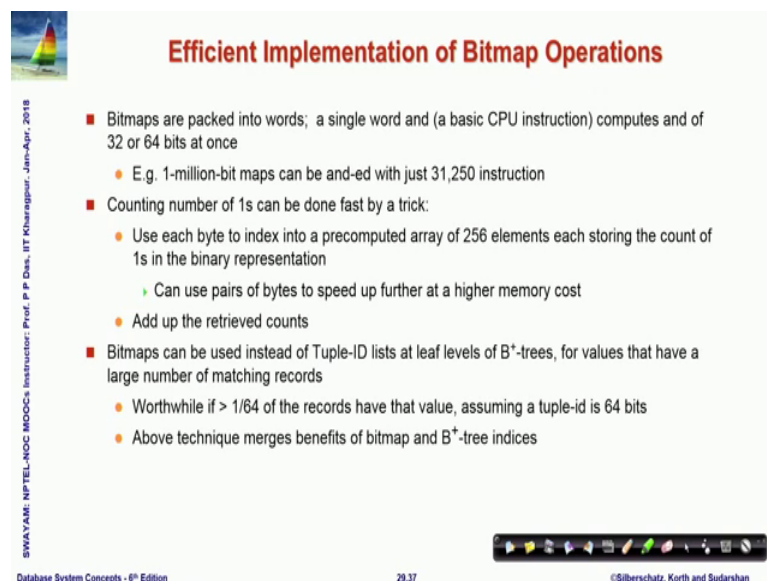
SWAYAM: NPTEL-NOC MOCs Instructor: Prof. P. P. Das, IIT Khargpur, Jan-Apr, 2018

Database System Concepts - 6th Edition 29.36 ©Silberschatz, Korth and Sudarshan

So, that is answer and you can quickly come to that. So, bitmap indices generally very I mean naturally they are they are they are small in compared to the relation size because you are doing bitmap indexing only if the attribute can take small number of distinct values.

Of course, the deletion has to be handled properly look at this and should keep bitmap for all values even if there are null values you must keep that otherwise you will lose track of that.

(Refer Slide Time: 34:33)



Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
 - E.g. 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
 - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
 - Can use pairs of bytes to speed up further at a higher memory cost
 - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B⁺-trees, for values that have a large number of matching records
 - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
 - Above technique merges benefits of bitmap and B⁺-tree indices

SWAYAM: NPTEL-NOC MOCs Instructor: Prof. P. P. Das, IIT Khargpur, Jan-Apr, 2018

Database System Concepts - 6th Edition 29.37 ©Silberschatz, Korth and Sudarshan

And there are several efficient implementations some information I have given, but is we do not want to go in much depth here.

(Refer Slide Time: 34:45)

Module Summary

- Explored various hashing schemes – Static and Dynamic Hashing
- Compared Ordered Indexing and Hashing
- Studies the use of Bitmap Indices for fast access of columns with limited number of distinct values

Database System Concepts - 6th Edition

29.38

©Silberschatz, Korth and Sudarshan

But several compression techniques are possible in terms of bitmaps; in the next module I will talk little bit more about how to use that. In this module to summarize we have talked about various hashing schemes static and dynamic hashing, compared the order indexing with hashing and introduced the notion of bitmap indices.