Module - 4

~~Introduction:~~

Indexing

Imagine we want to search for a topic in a 1000 pages notebook, without proper indices and (page numbers). The topic could be anywhere in those 1000 pages and we will have to search for it in the whole 1000 pages book.

With Content table and indices, our search is narrowed down significantly. If they have specified page no. for all chapters, and a chapter is 100 pages long, we can just search in those 100 pages.

This can be further narrowed down with more indices too.

1) Term definitions:

a) ~~Ordered~~ Index:

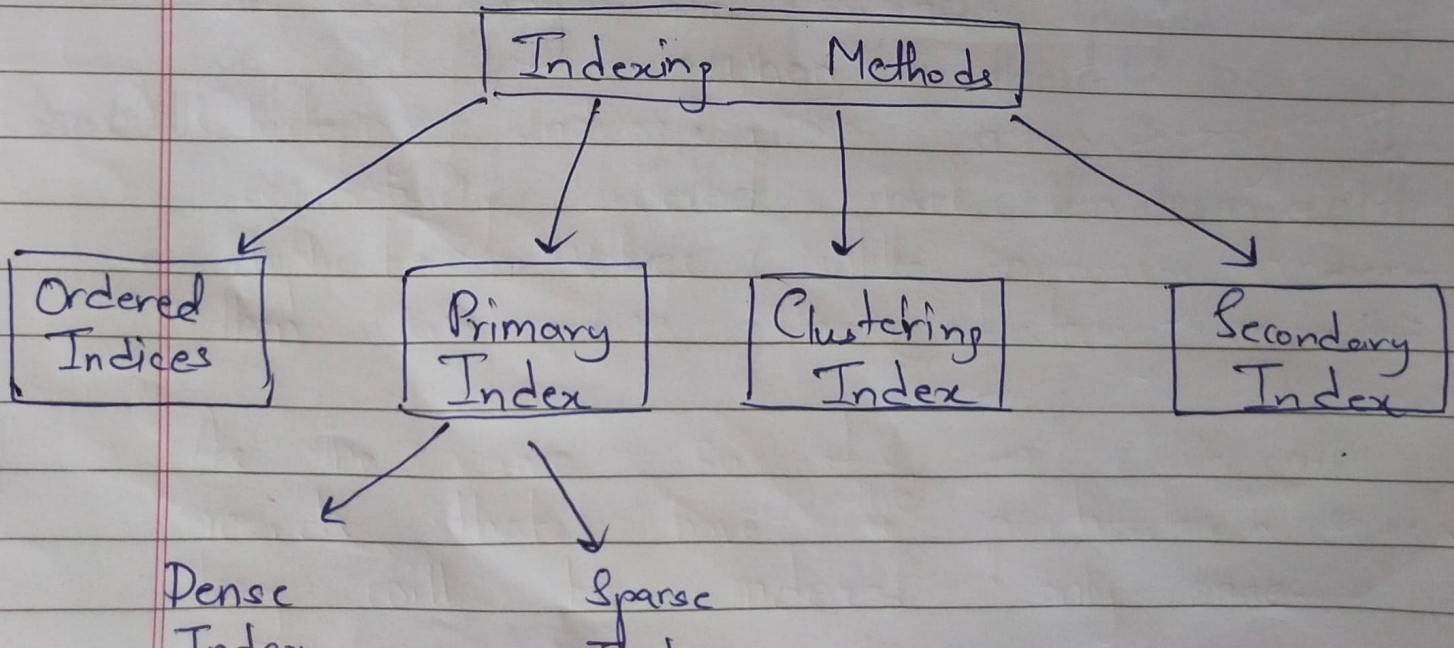
We create different types of indexing for our files based on the type of search key (search key can be type primary key, etc.).

b) → The index structure is:

Search key	Data Reference

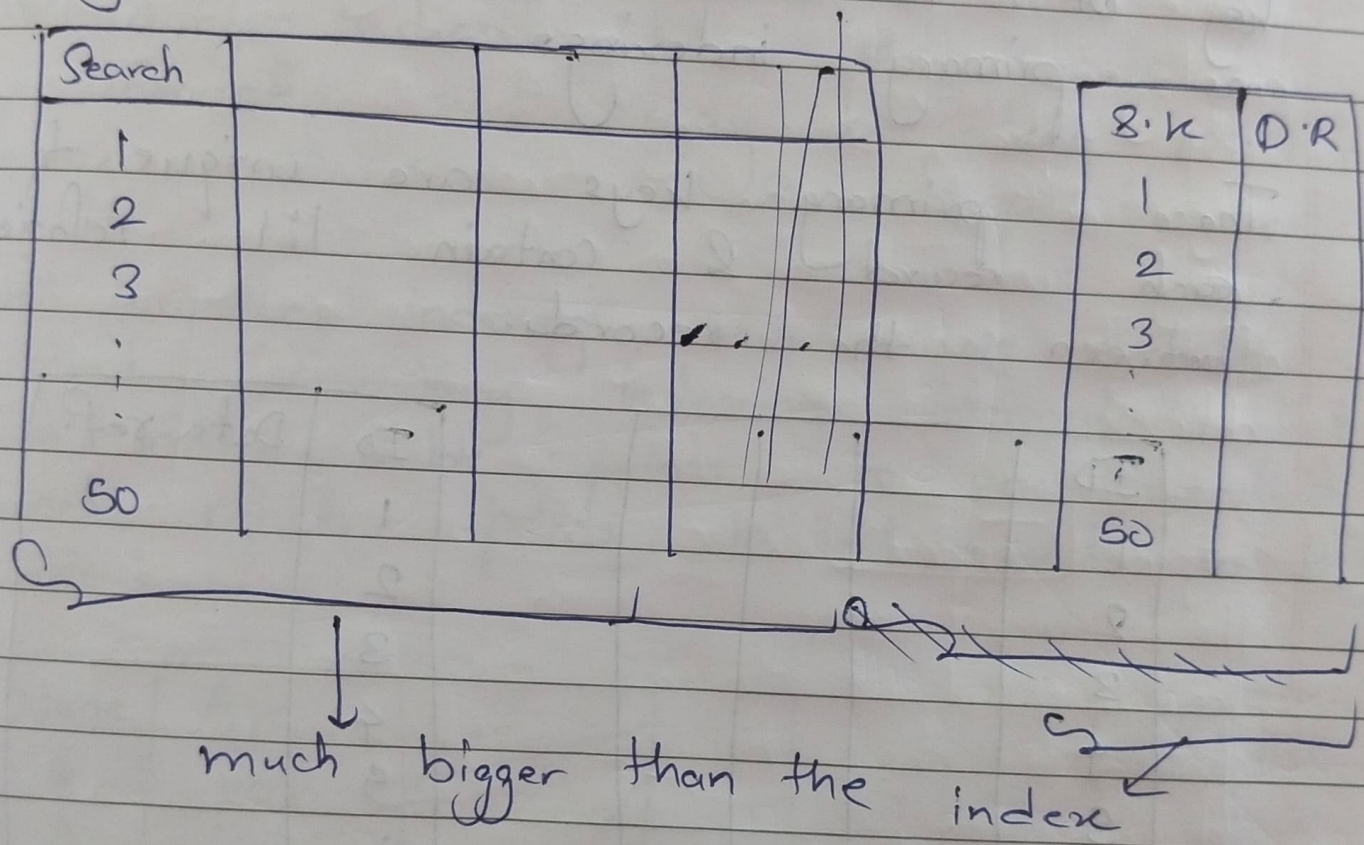
This is like a node in a linked list. The first column contains the search key value and the second column contains a set of pointers holding of the record.

2) Indexing Methods:



a) Ordered Indices:
These are indices which are simply sorted in an order. This makes searching quicker.

b) Primary Indices:



Searching for the 50th record without indexing will take much longer and will be much costlier because the size of record in the table is bigger than the index.

Thus, the ordered index can be used, the 50th record will be searched & the corresponding data reference will be studied.

when
b) Primary indices: Used ~~on~~ a search key that is also the primary key (or any unique key).

In other words, if the index is created on the basis of the primary key of the table, then it is known as primary indexing.

These primary keys are unique to each record & contain 1:1 relation between the records.

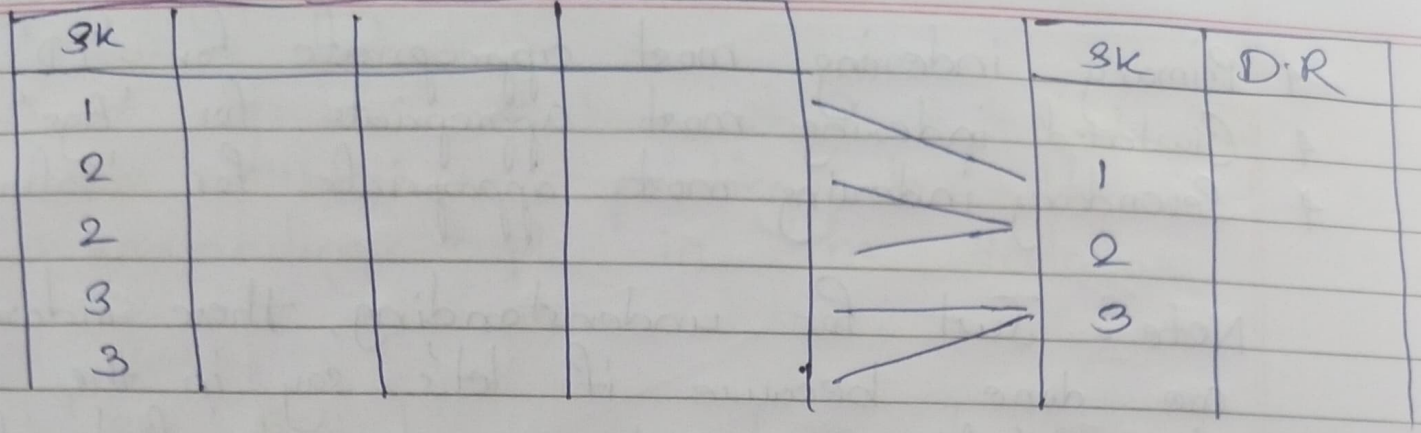
ID
1
2
3
4
5
:

ID	Data ref.
1	
2	
3	
4	
5	
:	

This is efficient because primary keys are already ~~ord~~ sorted.

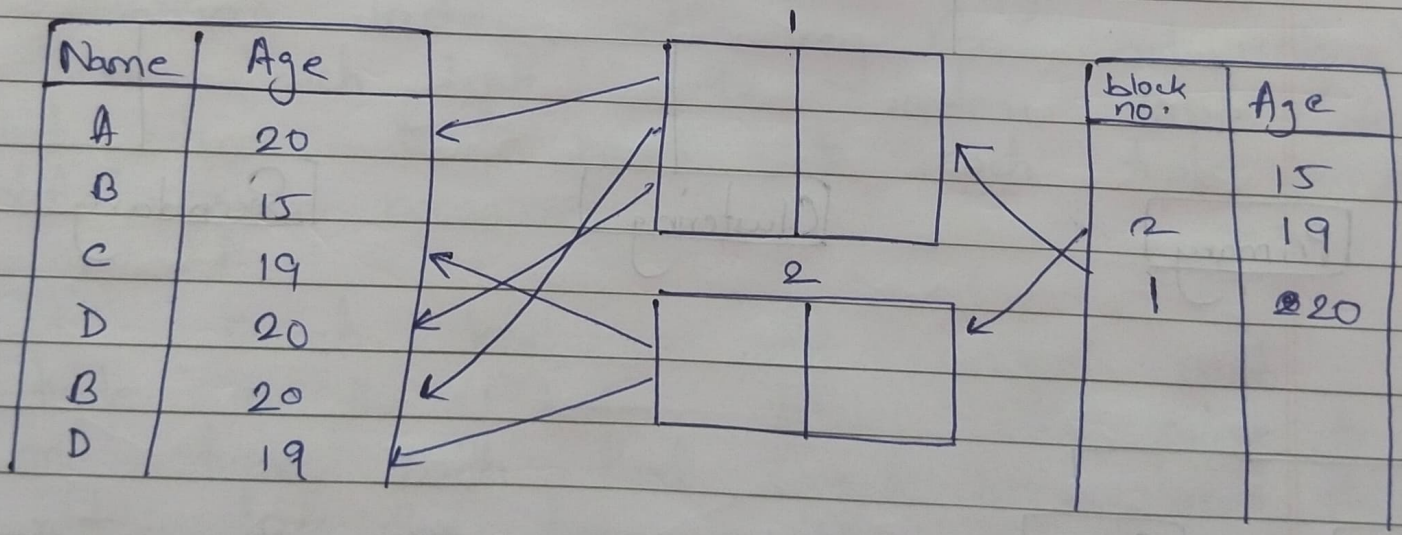
When
c) Clustering Indices: The search key is sequentially ordered, but can have duplicate values (unlike primary indexing) then we use the clustering indexing.

As the name suggests, a cluster of records that have the same particular search key will be indexed only once.



d) Secondary Indices: When the search key is unsorted and may be key or non-key, we use secondary indexing.

In secondary indexing, ~~we~~ we use something called ~~secondary indexing~~ intermediate blocks. The index table points to an intemediate block, which points to ~~al~~ records in the table:

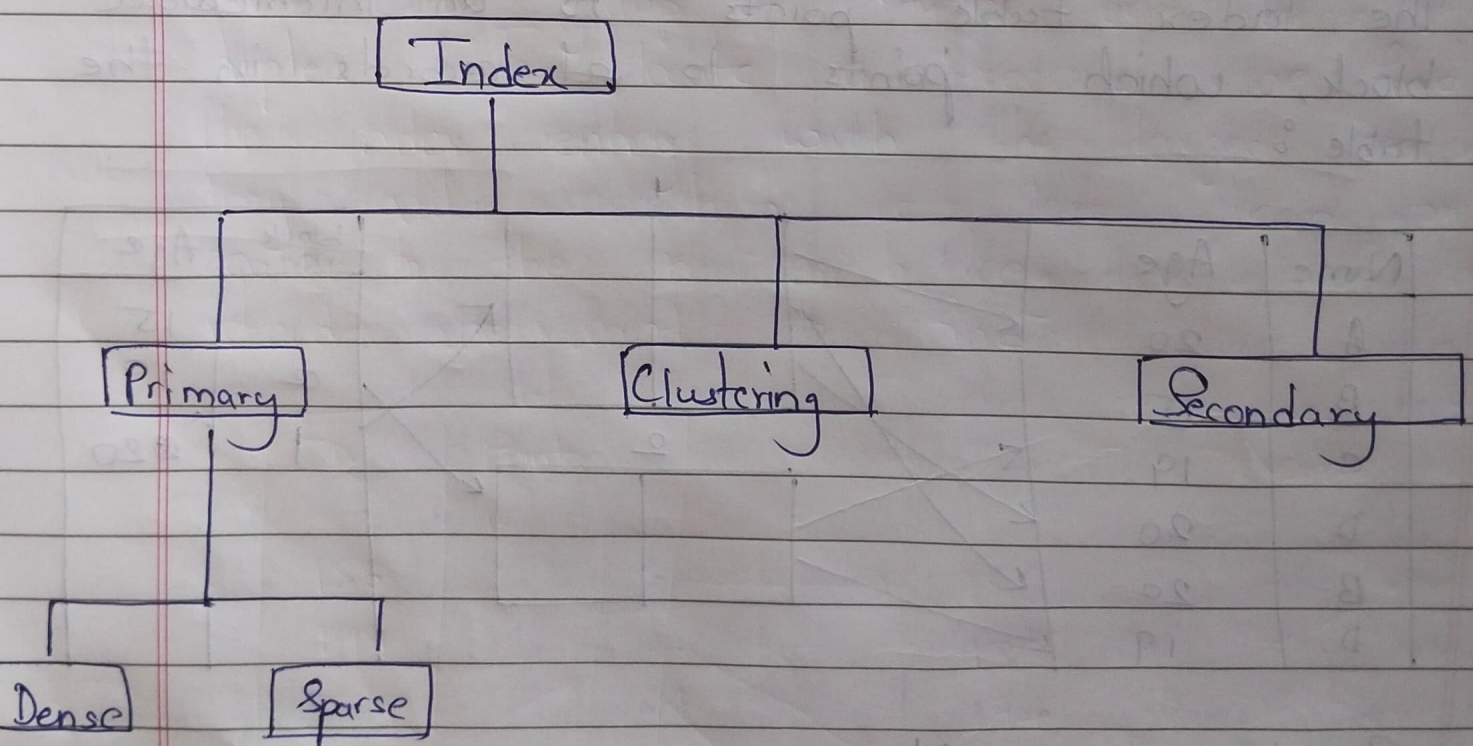


→ Consider the table:

ID	Name	Age	Salary
1	A	19	200
2	B	19	300
3	A	20	200
4	D	21	125

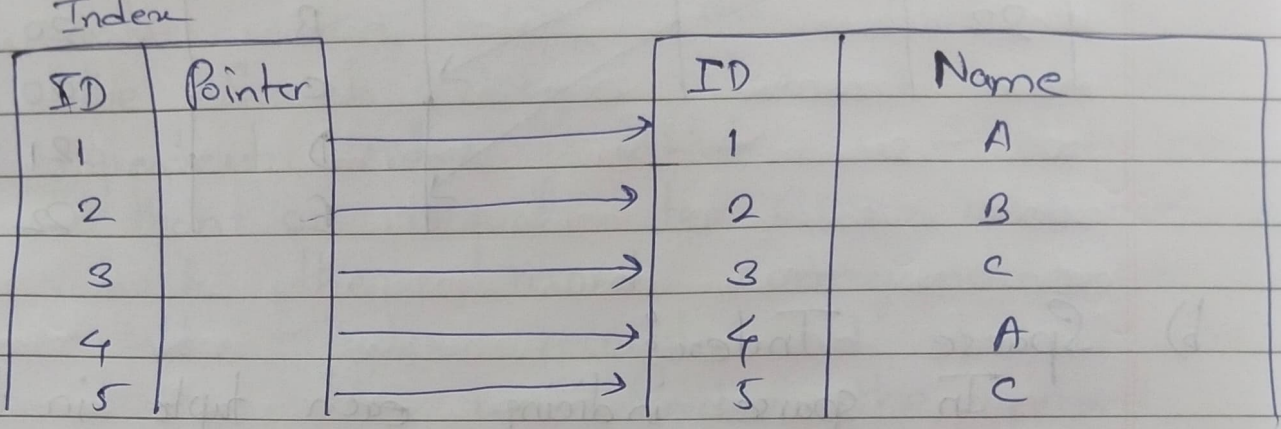
* Primary indexing most appropriate for 'ID'
* Clustered indexing most appropriate for 'Age'
* Secondary indexing most appropriate for 'Salary'

Note: Just for understanding, these indexing are done because if let's say in the above table we write a query to find all employees with a particular salary, it will ~~take~~ make searching for ~~that~~ those employees very difficult and time consuming. That's why we need to index every kind of attribute, so that it's quicker to find data.



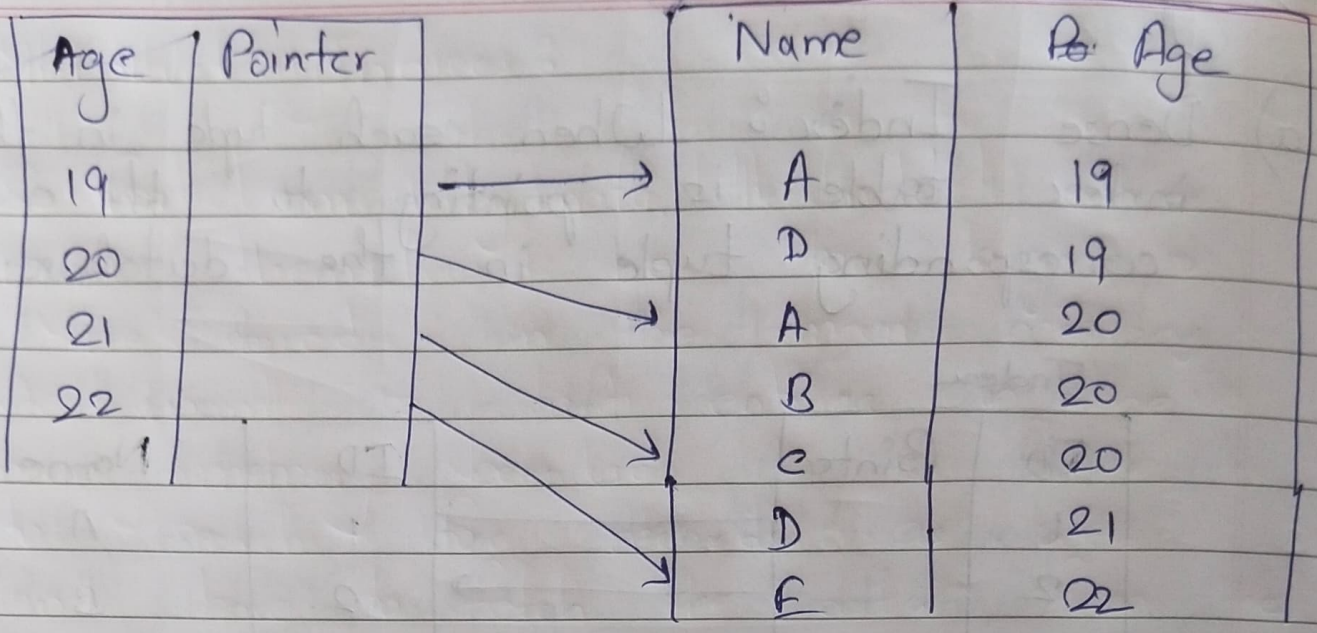
3) Types of Indices:

a) Dense Index: When each tuple in the index table is pointing to ~~the~~ a corresponding tuple in the database.



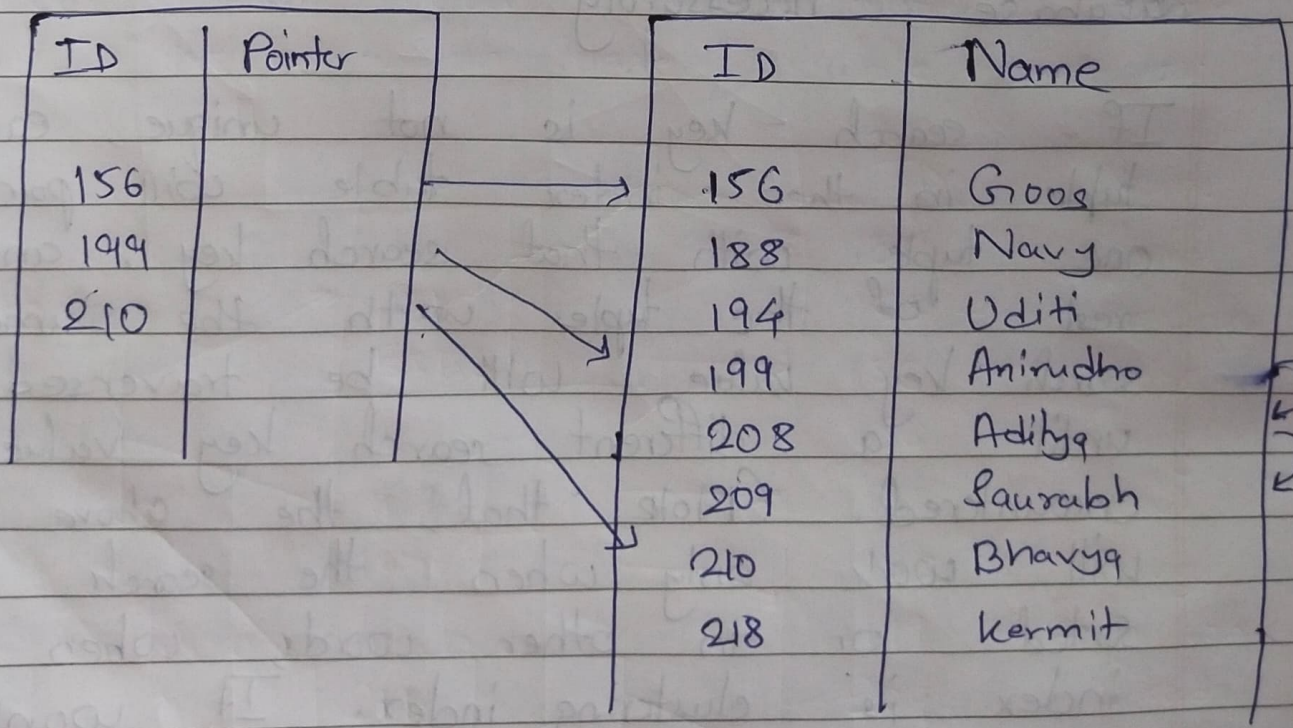
But the above definition is only true when the search key is the primary key (eg: ID). If the search key is ~~clustering index~~ not key (or not unique) then each tuple in the index table does not have to point to each tuple in the database necessarily.

If search key is not unique, each tuple in the index table will point to one tuple with that search key, and the rest of the tuples with the same search key value will be traversed linearly until a different search key value is encountered. (Note that, the above process will work only when the search key is sorted or in other words, when the index is clustering index. It won't work on non-clustering (unsorted) table.



b) Sparse Index:

In sparse indicing each tuple in the index table points to one tuple for every group of tuples or block of tuples. To locate a record, we find the index entry with the largest search key value which is less than or equal to the search key value of the record we want to find.



eg) If we want to find the record 209 with search key value 209, we see that the Search key value 199 is the greatest value which is still less than or equal to 209 and we check the record corresponding to that index record and traverse the table till we find the record with search key value = 209.

* It is faster to locate a record with the dense index but the sparse index table takes lesser space to be stored.

The system designer must take the decision depending on the access time & space overhead that the company values more.

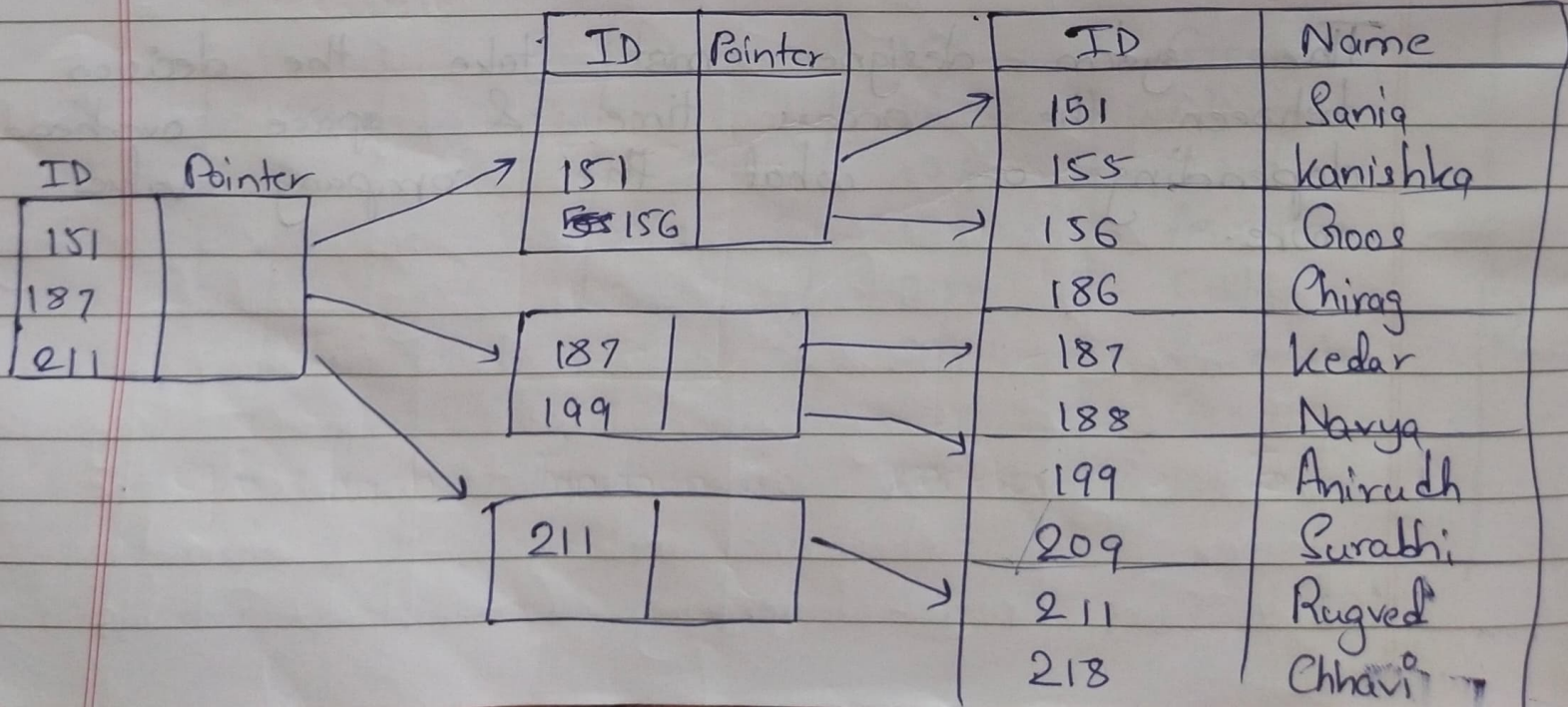
The system designer must take the decision between the access time & space overhead, depending on what the company values more.

4) Multilevel Indices:

So for some big data, the index entries might themselves be too large to be stored in the main memory. We want to store entries in the main memory because it is faster to search for records, as we don't have to fetch the data from the disk to the main memory.
Normal indexing may take only $\log_2 n$ time (using binary search) but this value would still be too high, for bigger data.

Above problems can be tackled using multilevel indexing.

In multilevel indexing, we treat the index table as any other table and create an index table for it.



The above is 2- level indexing. If the index table created after 2 level indexing is still too big to store in the memory, we can create yet another level of index. This is called multilevel indexing.

These multilevel indexes can be seen as in the form of a tree.

However, insertion & deletion of new index entries is a severe problem because every level of the index is an ordered file.

Hashing

1) Introduction: Hashing is a technique in which data is stored at data blocks whose address is generated by using the hashing function.
This makes searching much faster as we get the direct address of data using the hash function.
~~However, there is some extra mem~~

~~2) Definitions: Static Hashing (Definitions):~~
2) Static Hashing:
a) Bucket: A term used to denote a unit of storage that can store one or more records.

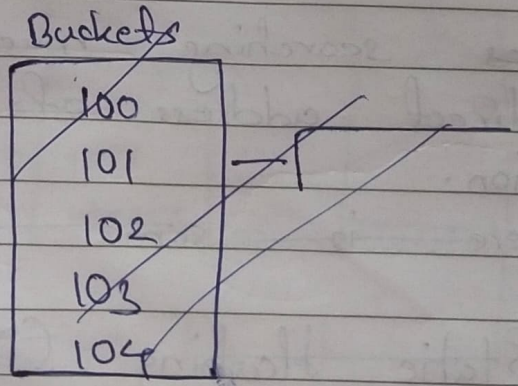
A bucket could be linked list of index entries or records.

b) Hash file Organization: In a hash file organization, instead of record pointers, buckets store the actual records.

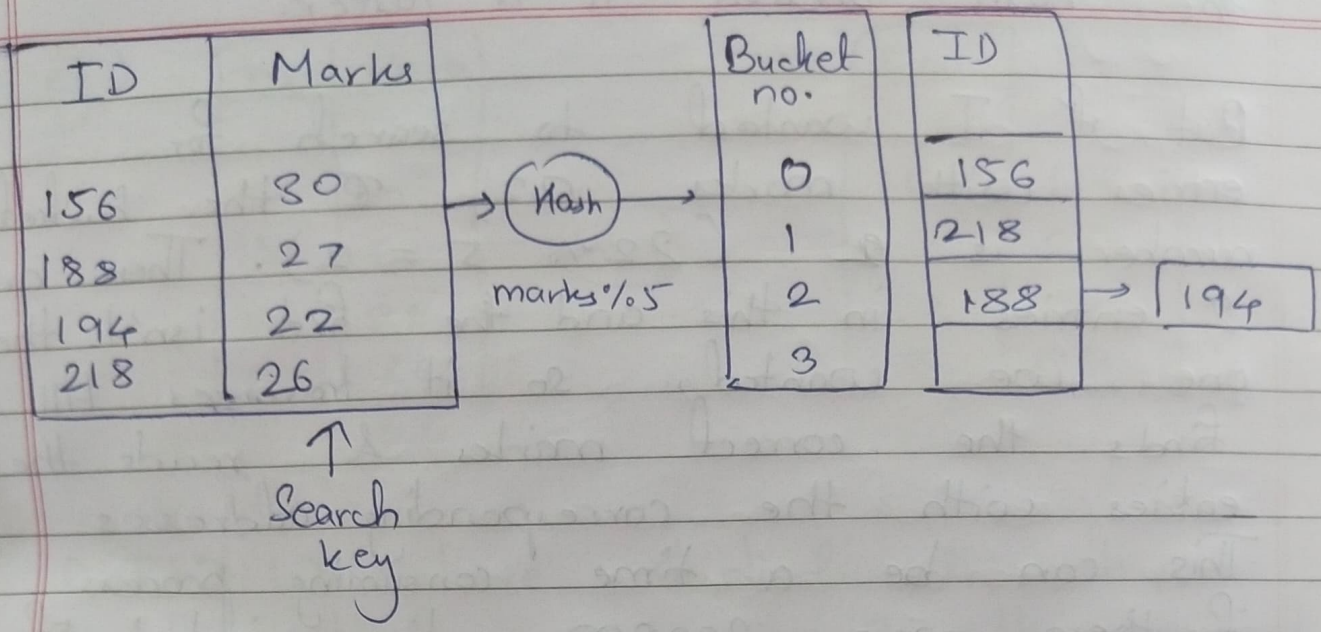
c) Hash functions: This is a function from the search key (k) to its corresponding bucket or address (B).

d) Overflow chaining: When buckets are full, a new bucket is allocated for the same hash function result and is linked after the previous one. (Like a linked list). The overflow buckets of a given bucket are chained together in a linked list. This is called Closed Hashing a coell.

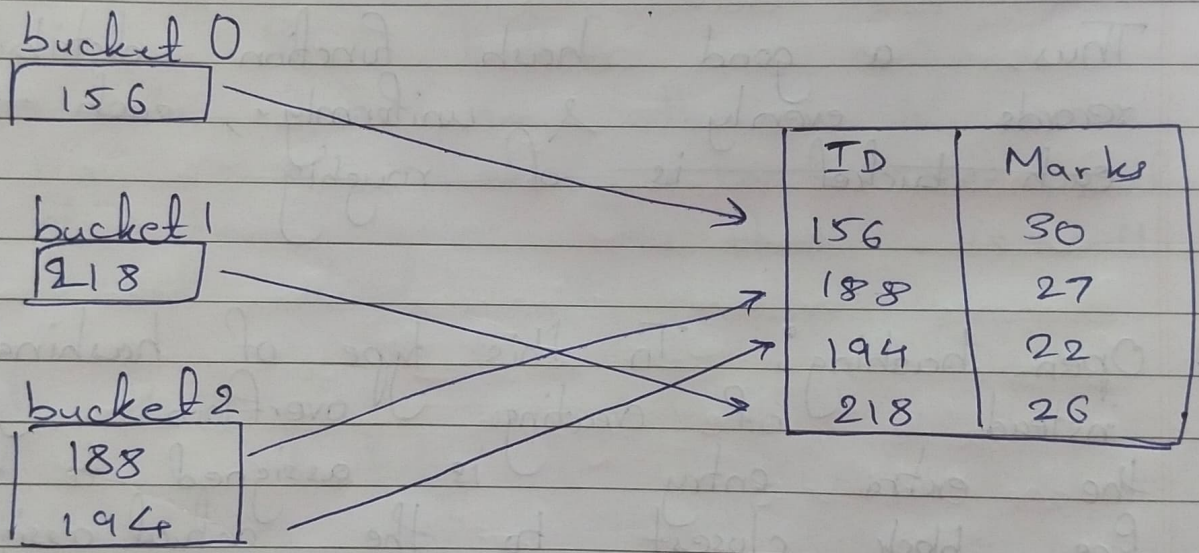
Or closed hashing



eg: hash function : marks % 5



Another representation:



If I want to search for all entries with marks = 26, the hash function will tell me that the address for that entry is in bucket 1 (26%5 = 1), and it will give me the ~~no~~ ID using the address that is stored in that bucket.
All this is done by the computer. Since it doesn't have to traverse the Marks table and can get the direct location using

the hash function, it is faster.

But if I wanted to search for entries with marks 22, the bucket number is 2 (22% 5 = 2). There are 2 entries in this and the first isn't the one we wanted, so it traverses till it finds the correct marks & reads the entries with the corresponding addresses. This can be a time consuming process if there were 200000 entries instead of 2 in bucket.

Thus, a good hash function distributes the records evenly & uniformly, ~~such that each bucket is of roughly~~

e) Open hashing: In this type of hashing instead of creating overflow buckets, the extra entry is assigned to the free block closest to the actual assigned bucket.

eg)

ID	Marks
156	30
188	27
194	22
218	26

hash

B.no	ID
0	156
1	218
2	188
3	194

194

Note: We have studied all this in data structure in sem 3, the concept is same.

f) Skew: Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket skew.

g) Static Hashing: In static hashing, the set of buckets is fixed. This is suitable to use when the size of ~~data~~ is known in advance.

h) Hash function properties: Since we do not know at design time precisely which search key values will be stored in the file, we want to choose a hash function that assigns search key values to buckets in such a way that distribution has these ~~qualities~~ qualities:

i) The distribution is uniform. Each bucket has

roughly same no. of entries.

ii) The distribution is random, meaning there is no obvious patter to the distribution.

I) Disadvantages of Static Hashing:

The set of buckets is fixed at the time the hashing index is created. If the relation goes far beyond the expected size, these indices will be inefficient due to long overflow chains.

3) Dynamic Hashing:

Dynamic hashing techniques allow the hash function to be modified dynamically to accomodate the growth or shrinkage of the database.

~~a) Ext~~

a) Extendable hashing:

i) Important definitions:

- Hash Values: A hash value, also known as a hash code, is the output of a hash function.

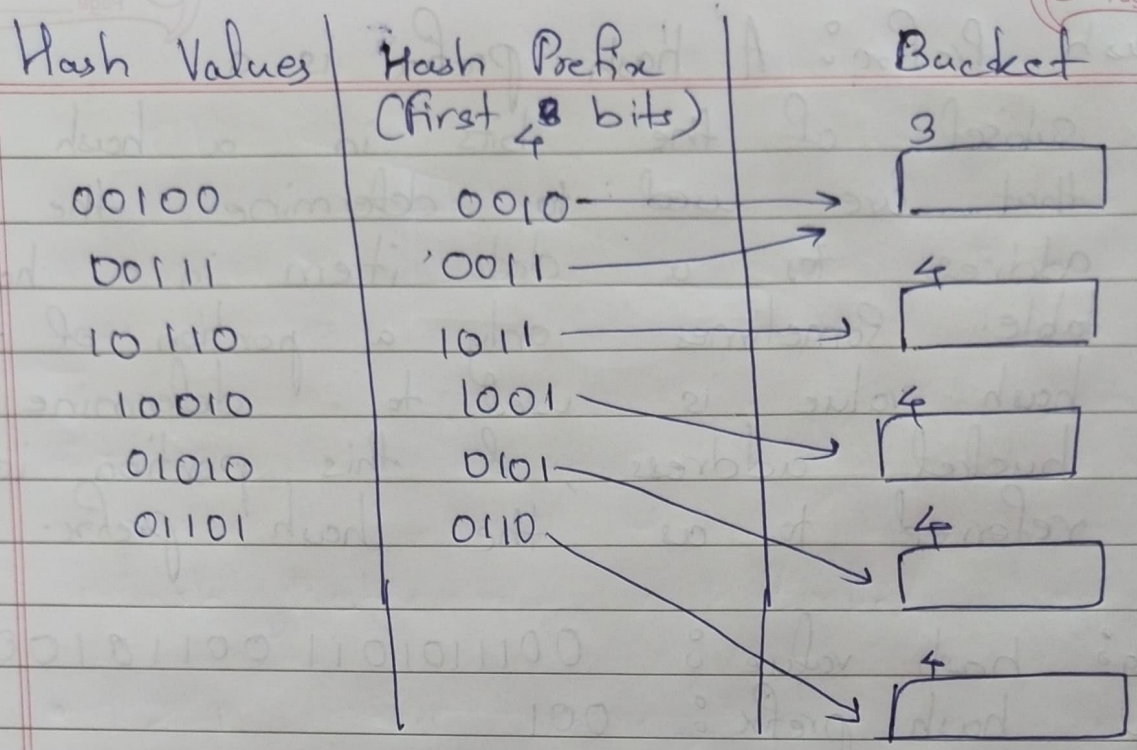
- Hash Prefix: A hash prefix is a subset of the bits in a hash value that are used to determine the bucket address for a data item in a hash table. Sometimes only a portion of the hash value is used to determine the bucket address, & this portion is referred to as the hash prefix.

eg: hash value : 00111010111 00111010011010
hash prefix : 001

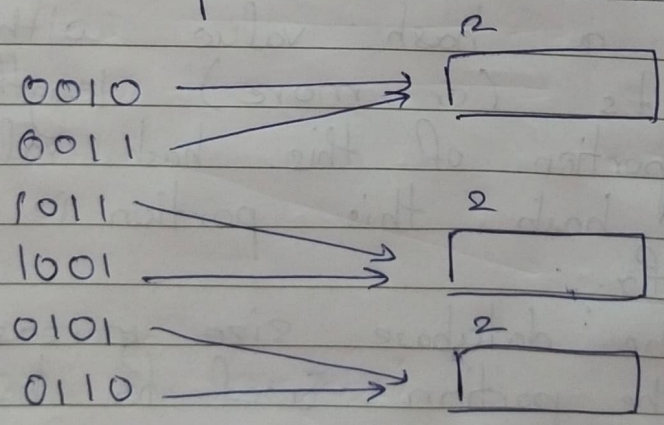
- ~~Common hash prefix:~~

ii) How does it work?

- The hash function in extendable hashing generates a hash value with 32 bits (or more) length.
- Only a portion of this hash value is used to hash, this portion is called the hash prefix.
- When the database size increases, the size of the portion used for hash prefix increases and vice-versa.
- Because the size of the hash prefix can be changed, this makes the hashing dynamic, point is explained below:



Here, the first two hash prefix values point to the same bucket because the first 3 bits are same. Note that if we looked at the first two bits, then 1011 & 1001 and 0101 & 0110 would point to same buckets.



The bits that are same in the hash prefixes are called as common hash prefix.

The common hash prefix is the portion of the hash prefix that is shared by multiple hash values.

iii) Size of the hash prefix:

Let's say that the hash value created is b - bit long.

At any point, we use 'i' bits, where $0 \leq i \leq b$, to determine the value of hash prefix.

The value of 'i' grows and shrinks with the size of the database.

As for the common hash prefix, its length may be less than 'i'. Thus we write the length of the common hash prefix on every single bucket. If there is only one hash prefix pointing to a bucket, then the length of the hash prefix is written.

iv) Extendability:

We can shrink or grow the value of 'i' with the database. If the size of the database increases, we can increase the number of bits 'i', which means we increase the length of the hash prefix.

Because of this increase (an extra bit), data that were previously overflowing a bucket will be mapped to a different bucket, making it efficient for dynamic sized databases (can be better understood under insertion).

v) Queries and updates

- Performing lookup : To locate the bucket containing search key value k, the system follows the first i bits of the hash value for that search key h(k) then looks at the corresponding table entry for this bit string, & follows the bucket pointer in the table entry.

- Deleting : To delete a record with search key value k, the system follows the same procedure for lookup as before. It then removes both, the search key and the record from the file.

- Insertion : To insert a record with search key value k, the system follows the same procedure for lookup as before. If the bucket ~~is~~ has space, the record will be inserted and ~~at~~ the pointer to that record will be stored in the bucket pointed at by the hash value h(k).

However, if the bucket is full, we will need an extra bucket. The system itself makes a duplicate bucket where when the Original bucket is overflowed. This is called splitting of the bucket,

because we get two buckets from one. It is important to know that, the capacity of the buckets created & the original bucket is same, it isn't halved.

There are 2 scenarios that can occur while splitting a bucket, depending on the value of i (hash prefix length), and i_j (common hash prefix length).

- $i > i_j$, this means that the hash prefix length is greater than the common hash prefix length, thus more than one entry in the bucket address table points to the bucket.

where we assume

A's search key value's hashed value = 1101110101

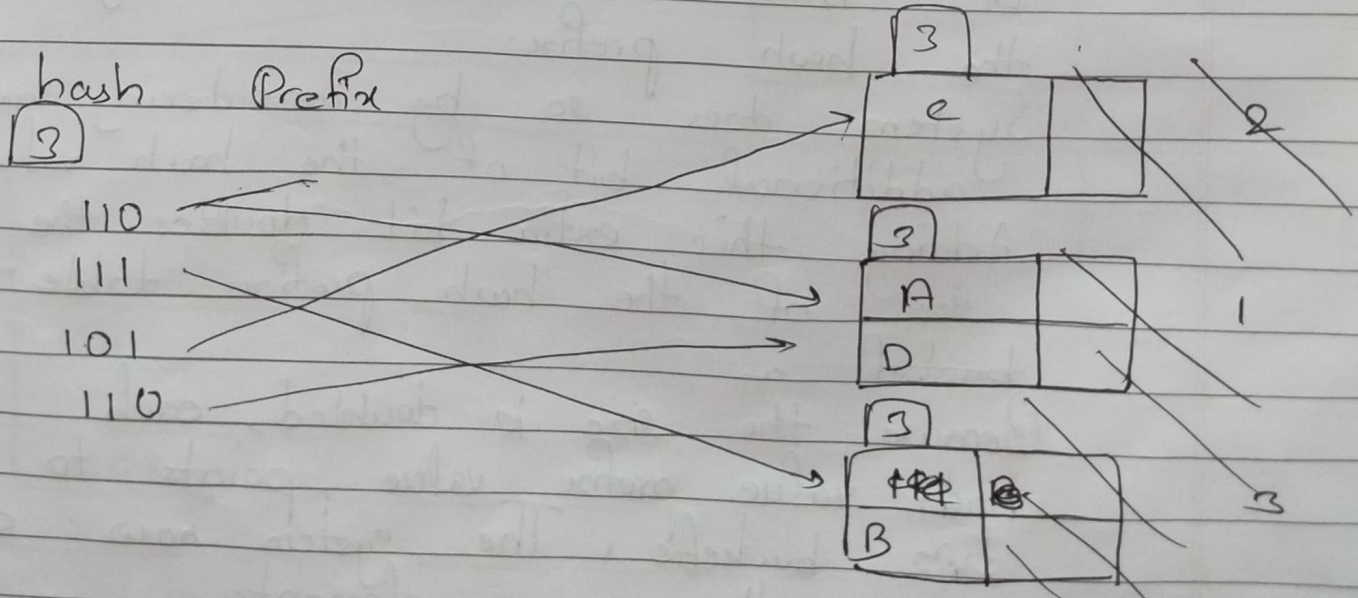
B's search key value's hashed value = 1111101011

C's search key value's, hashed value = 0010101101

prefix

Observe that the hash value lengths that we are taking is $i = 3$, and the length of common hash prefix value for bucket 1 is 2.

Now, if we had to insert another element whose search key value's hashed value is 1101001111 we will first see if $i > ij$, if yes, the system allocates a new bucket ~~and~~ (let's say bucket 3) and increases the bit length by 1, from 2 to 3.

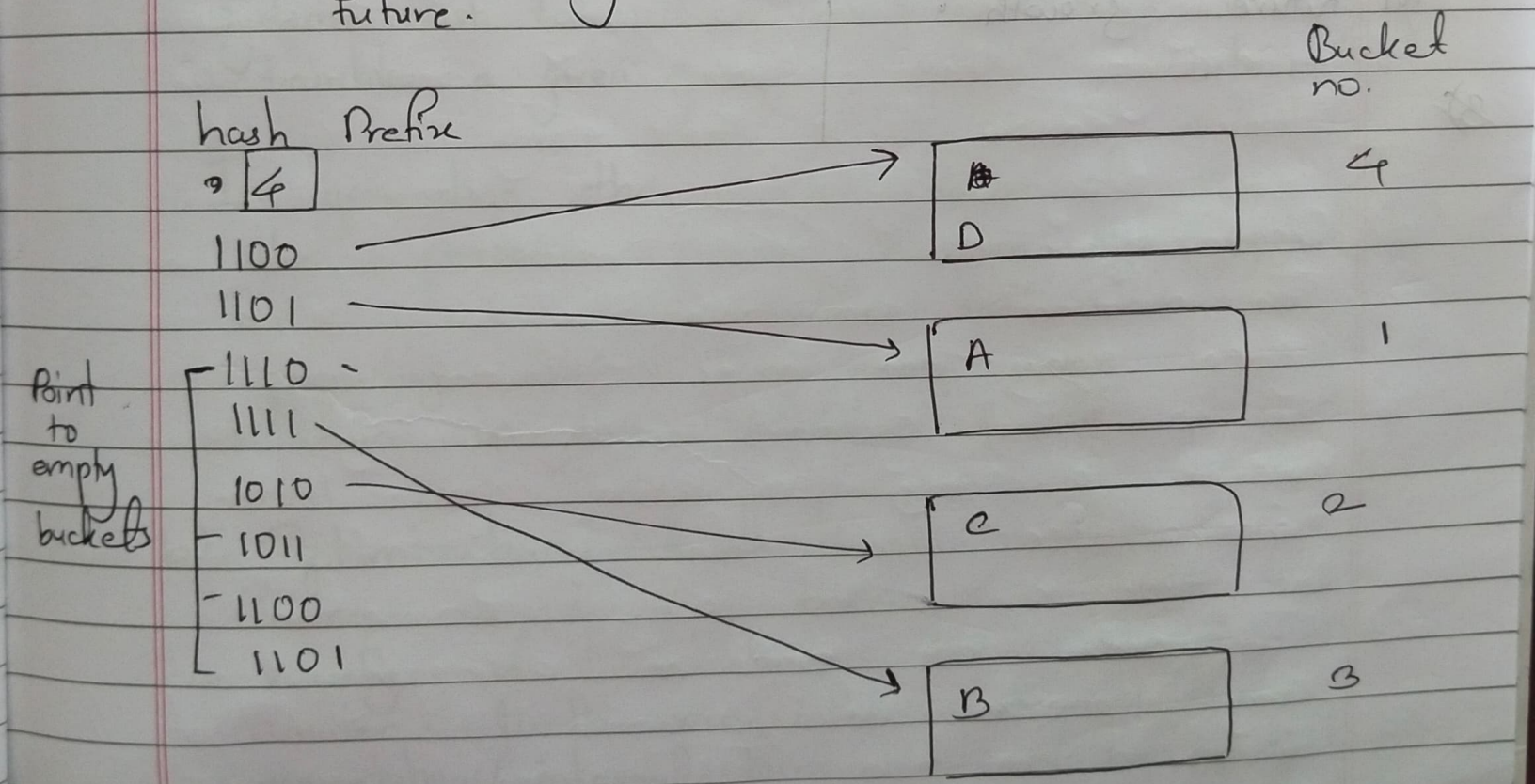


The system basically divides the bucket entries such that first half of all entries remain in the same bucket & the seond half goes in the new bucket. The system then rehashes the new entries (~~to~~ because 1 bit increase in common hash prefix) & allocates it to either of the 2 buckets.

In the above example if we want to insert yet another element with search key value's hash value = '11000011O', we cannot follow the above algorithm, because i is now equal to j; (3) and ~~there~~ prefix ~~is no are~~ every single hash value points to only one bucket (one to one).

- i = ij : The above situation can be handled if we increase the number of bits we are considering for the hash prefix. System does so by considering an additional bit of the hash value. Adding this extra bit doubles the size of the hash prefix table. ~~for bucket a~~ Because the size is doubled, each hash value prefix value points to two buckets. The system now can insert the new element.

Note: It is possible that adding this extra bit creates a hash prefix value that doesn't have any search key for it yet. The bucket thus created will be empty, it can be used if that search key value is inserted in the future.



Note that in ~~te~~ the above example we only took the bucket max capacity to be 2.

This insertion can be better understood using the example given in korth.

76766	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	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

These are the full hash values for all the departments. we will use these value for storing the bucket address

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

Figure 24.9 Hash function for *dept_name*.

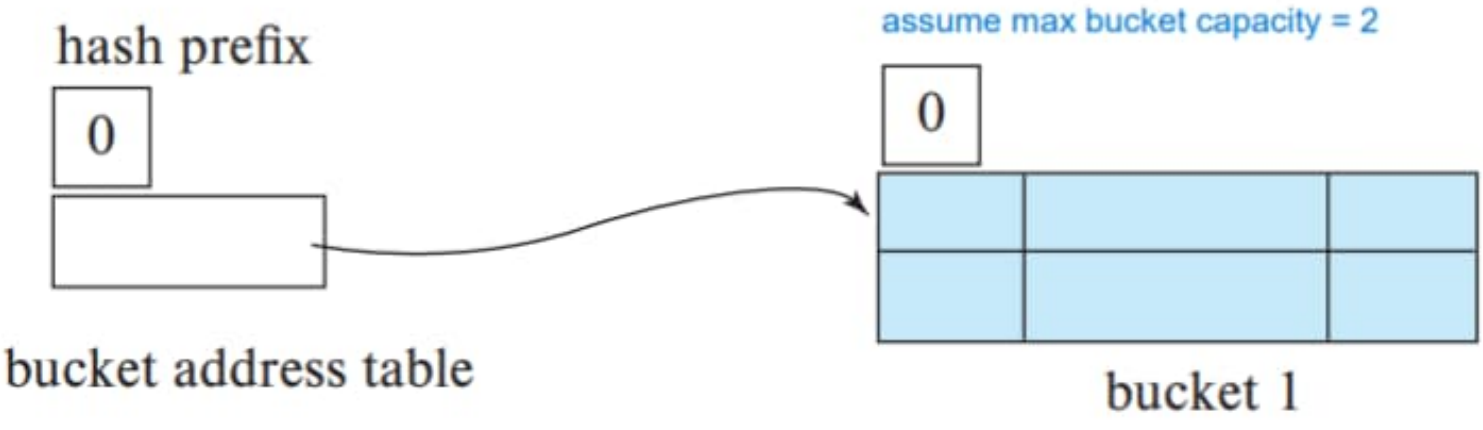
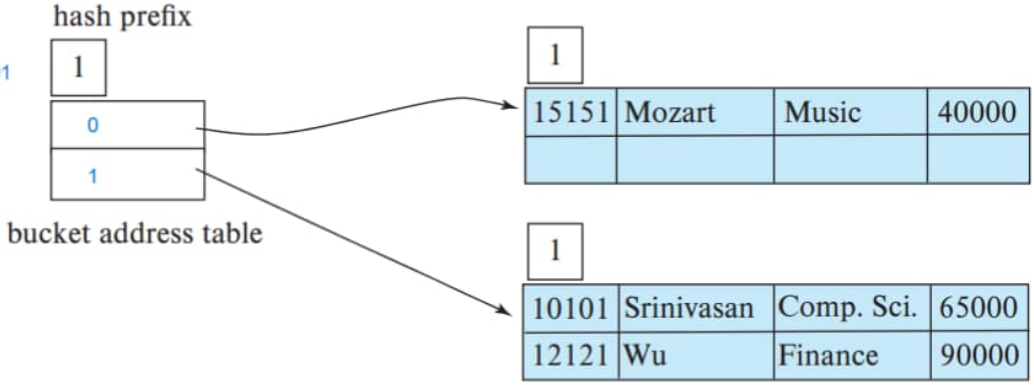


Figure 24.10 Initial extendable hash structure.

Comp. Sci. 1111 0001 0010 0100 1001 0011 0110 1101
Finance 1010 0011 1010 0000 1100 0110 1001 1111
Music 0011 0101 1010 0110 1100 1001 1110 1011



now we try to insert another entry with dept physics. this causes a bucket overflow because the hash prefix for physics is 1, and the bucket for hash prefix value 1 is full, and i = ij, thus we increase the hash prefix bit by 1.

Figure 24.11 Hash structure after three insertions.

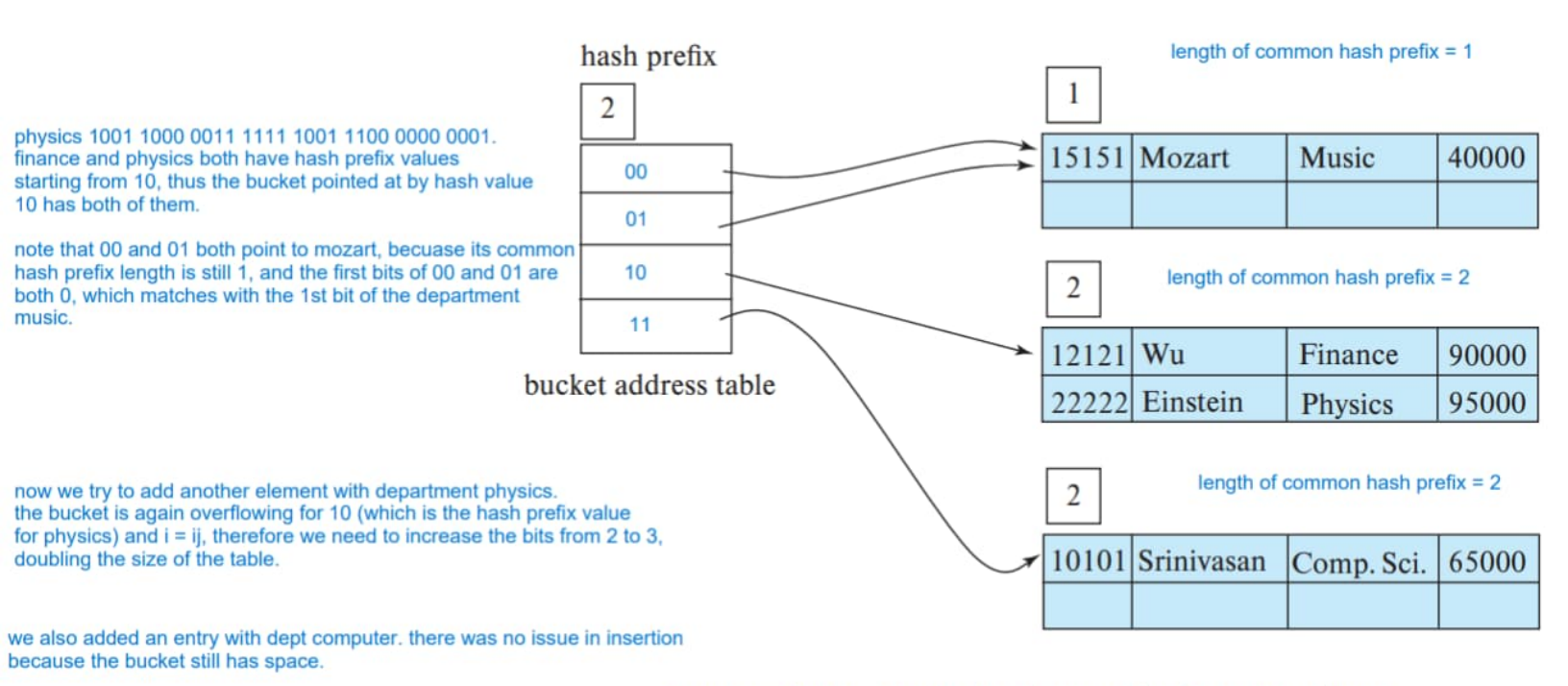
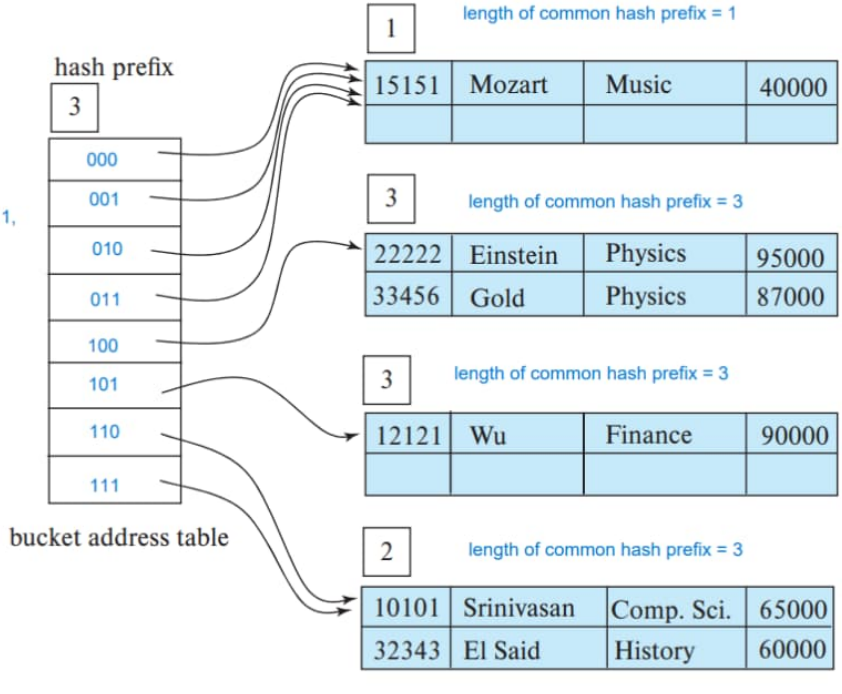


Figure 24.12 Hash structure after four insertions.

the size of the table has doubled, along with the max number of buckets supported. notice that 4 values point to mozart for the same reason as before. now 2 values point to the last bucket, becuase this bucket has ij = 2, and both the values in the table have 11, like the initial bits of the departments history and comp



case of i > ij, thus new entry katz can be entered without needing to increase the number of bits from 3 to 4

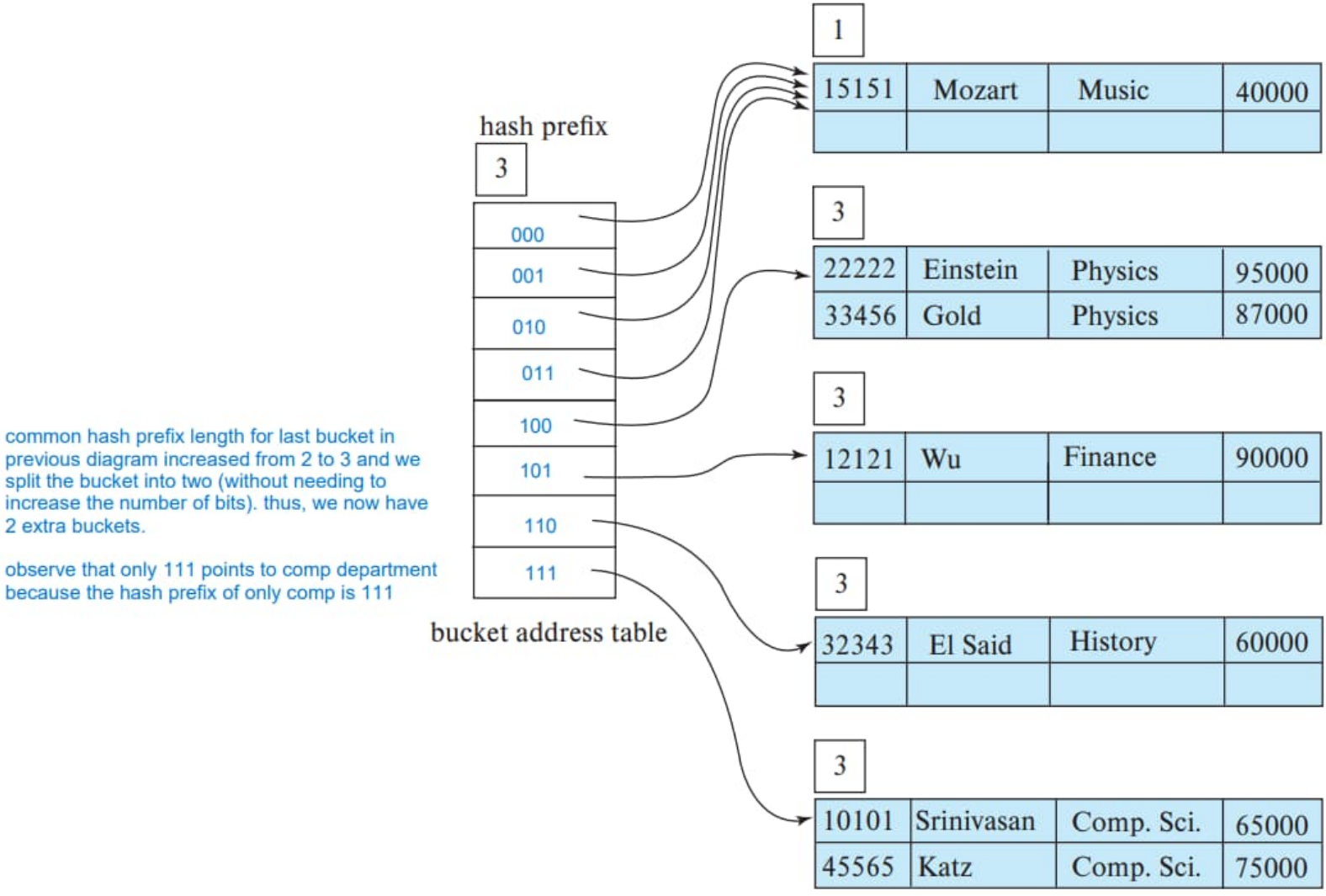


Figure 24.14 Hash structure after seven insertions.

we could add the element with dept biology becuase its hash value starts with 0, and there was empty space in the bucket for this hash prefix

inserting more elements in comp cannot be done by either of the methods because the search key value remains same (dept) making the hash value same for many elements. thus we now have to use overflow buckets, like in static hashing.

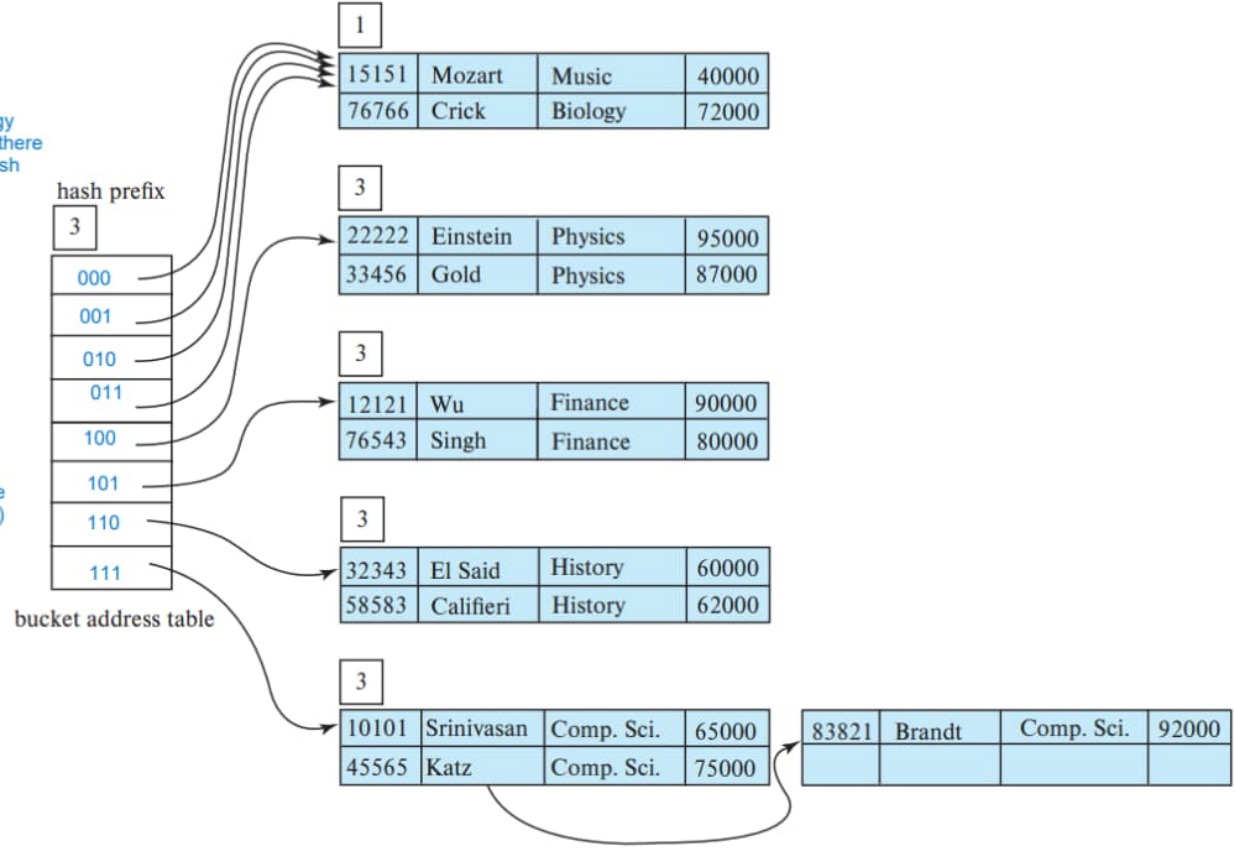


Figure 24.15 Hash structure after 11 insertions.

3) Static vs Dynamic Hashing

Static	Hashing
1) Performance degrades as the file grows.	1) Performance does not degrade as file grows.
~~2) More space overhead due to overflowing buckets.~~	2) Lesser space overhead.
2) More space overhead due to the fact that we have to reserve buckets for future growth.	

3) comparison of ordered indexing and hashing

Ordered Indexing Vs Hashing

Indexing	Hashing
It is a technique that allows to quickly retrieve records from database file.	It is a technique that allows to quickly retrieve records from database file.
It is generally used to optimize or increase performance of database simply by minimizing number of disk accesses that are required when a query is processed.	It is generally used to index and retrieve items in database as it is faster to search that specific item using shorter hashed key rather than using its original value.
It is not considered best for large databases and its good for small databases.	It is considered best for large databases.
It uses data reference to hold address of disk block.	It uses mathematical functions known as hash function to calculate direct location of records on disk.

4.3

Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high level database languages into expressions that can be used at the physical level of the file system, query optimization and evaluation of queries.

4.3.1 Steps involved in query processing:

1) The basic steps are:

a) Parsing and translation: Before query processing can begin, the system must translate the query into a usable form. Thus, the first action the system must take in query processing is to translate a given query into its internal form.

In generating the internal form of the query the ~~processor~~ parser checks the syntax of the user's query, verifies the relation names appearing in the query are names of the relation in the database & so on.

b) Optimization: Evaluation:
Given a query, there are generally a variety of methods for computing the answer. Furthermore, the relational Algebra representation (internal form understood by ~~the~~ system) specifies only partially how to evaluate a

query; there are usually several ways to evaluate relation algebra expressions.

To specify fully how to evaluate a query, we need to provide the relational algebra expression and also annotate the instructions specifying how to evaluate each operations with it.

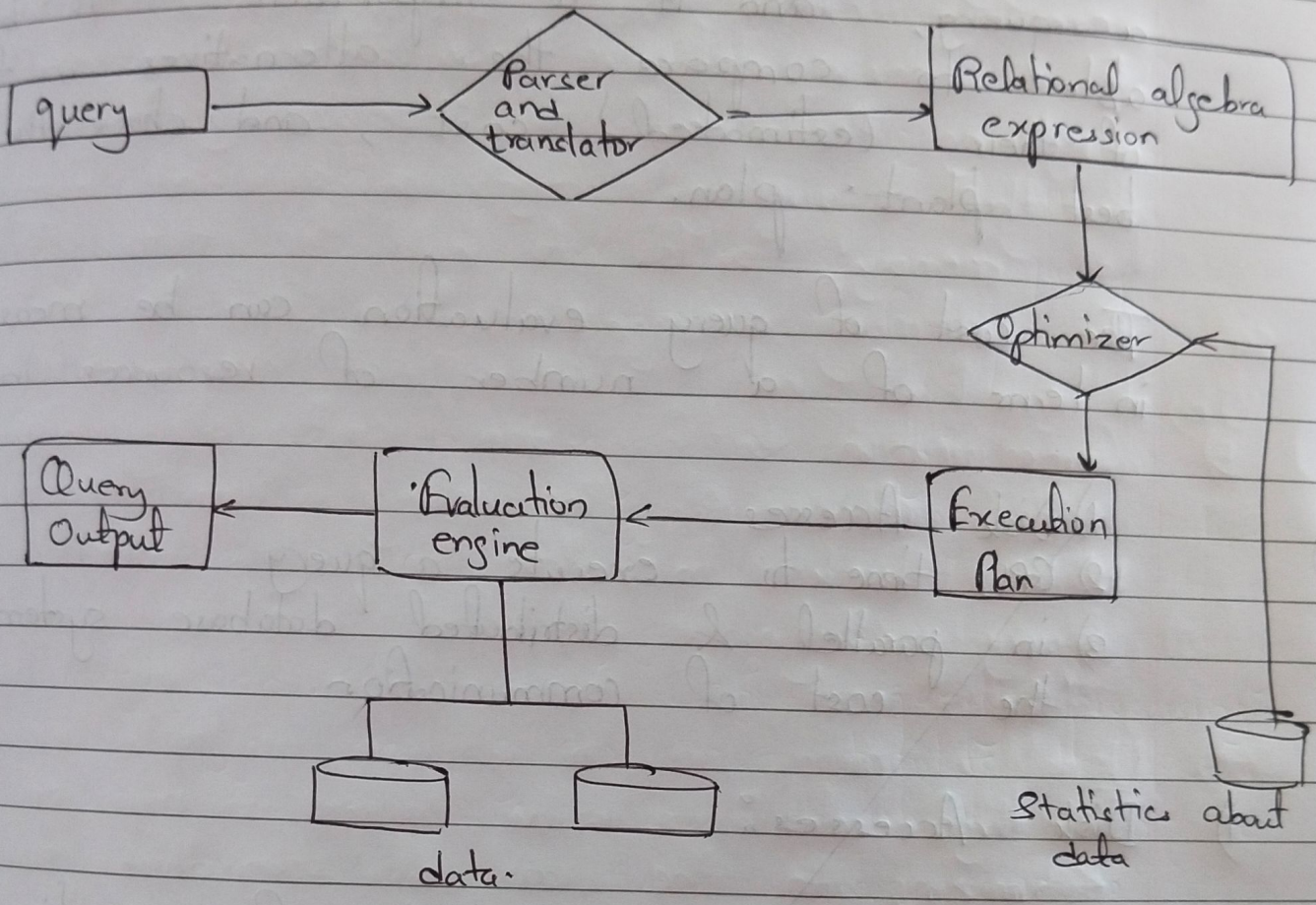
i) Evaluation primitive: A relational-Algebra operation annotated with instructions on how to evaluate it.

ii) Query execution plan: A sequence of primitive operations that can be used to evaluate a query.

iii) Query execution engine: The query execution engine takes a query evaluation plan, executes that plan, & returns the final answer to the query.

c) Optimization:
The different evaluation plans for a given query can have different costs. It is the system's responsibility to construct a query evaluation plan that minimizes the cost of query evaluation, this task is called query optimization.
In order to optimize a query, a query optimizer must know the cost of each operation. It depends on many parameters such as actual memory available to the operation.

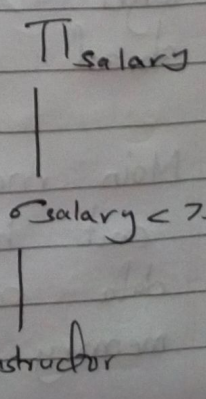
In 14.3 we will look into how to find the cost of execution plans, optimization will be studied in 14.4.



eg) select salary
from instructor
where salary < 75000;

translations: $\sigma_{salary < 75000}(\Pi_{salary}(instructor))$
Possible $\Pi_{salary}(\sigma_{salary < 75000}(instructor))$

It is also possible to pipeline the tasks:



4.3.2 Measures of query cost:

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and chose the best ~~plant~~ plan.

The cost of query evaluation can be measured in terms of a number of resources including:

1) Access cost to secondary storage.
2) Disk storage cost.
3) Computation Cost.
4) Memory wage cost.
5) Communication Cost.

1) Access cost to secondary storage: This is the cost of reading or writing data blocks between secondary storage and main memory.

The cost of searching for records from secondary storage depends:

a) On the type of access structure on the file (type of indexing & hashing)

b) On whether the data blocks are allocated contiguously or ~~sea~~ are they scattered on the disk.

2) Disk storage Cost: This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.

3) Computation Cost: This is the cost of performing in-memory operations on the records in the main memory during execution.
In-memory operations refer to the processing of data within the main memory (RAM) of a computer system.

Examples of in-memory operations include: searching, sorting, merging, etc.

This cost is also known as CPU cost.

4) Memory usage cost: To optimize the query execution cost, the system tries to keep as much data as possible in memory, reducing the need for secondary storage operations & this requires allocating a sufficient number of memory buffers to hold the data. This is the memory usage cost.

5) Communication Cost: The cost of sending the result of the query from the database site back to the site where query originated from.

In larger databases, the focus is more on optimizing to the cost of secondary storage access.

In smaller databases, the focus is more on minimizing the computation cost because most of the data is already in main memory.

* The method of optimizing the query by choosing a strategy that results in minimum cost is called cost-based query optimization.

4.33) Algorithm for SELECT operation:

The SELECT operation is basically a search operation to locate the records in a disk file that satisfy a certain condition. There are many algorithms for executing a SELECT operation.

Definitions:

- File scan: Scanning the records of a file directly to search for and retrieve records that satisfy a selection condition.
- Index Scan: When the search algorithm involves the use of an index, the index search is called an index scan.

For simple one condition queries like

→ SELECT * WHERE SSN = '123456789'
→ SELECT * WHERE Dno = 5

We can use the below 6 algorithms to implement a SELECT operation:

S1) Linear Search (Brute Force Algorithm): Retrieve every record in the file, & test whether its attribute values satisfy the SELECTION condition.

S2) Binary Search: If the attribute is ordered & the SELECT operation involves equality comparison,

we can use binary search as its quicker.

S3)a) Using a Primary index: If SELECT condition involves an equality comparison on ~~an~~ key attribute with primary indexing (eg: on ID), we can use the primary indexing to retrieve the record.

S3)b) Using hash key: If SELECT condition involves an equality comparizon on a key attribute with a hash key, we can use the hash key to retrieve the data.

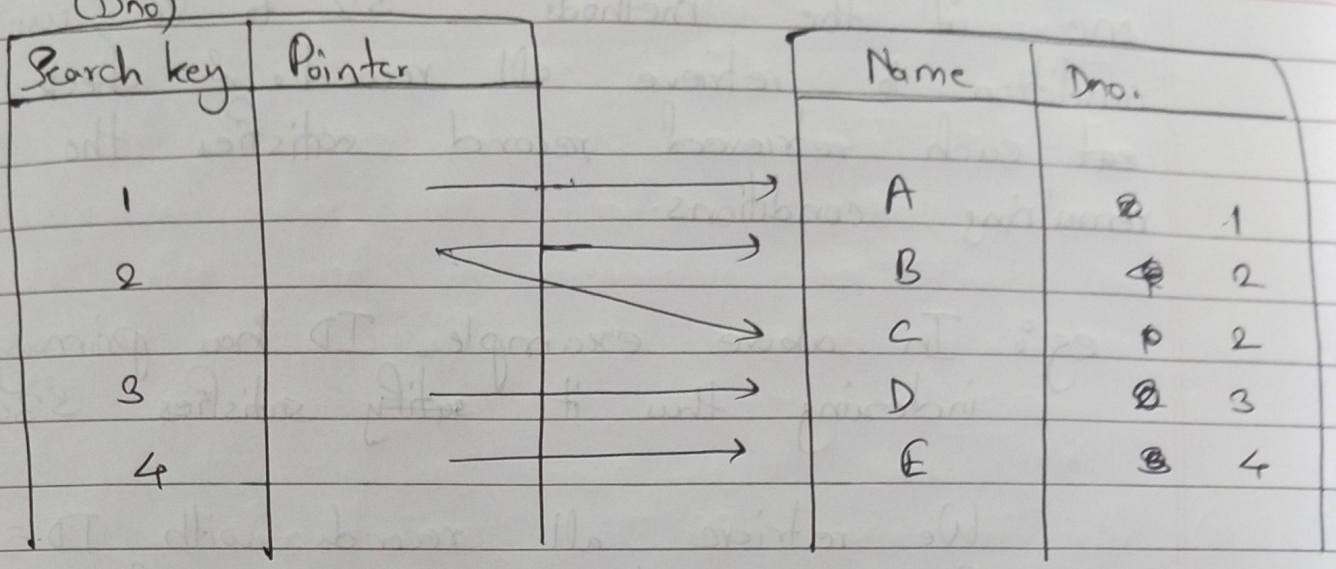
S4) Using a primary key to retrieve multiple records: If the select condition comparison is <, >, <=, >= on a key value ~~if~~ field with primary indexing

Eg: SELECT * WHERE salary > 75000

we use the primary index for the record that satisfies the corresponding equality condition (salary = 75000) & (in the example above) retrieve all records that come after it.

S5) Using a clustering index to retrieve multiple records: If the selection condition involves an equality comparison on a non-key attribute with a clustering index, we use the index to retrieve all the records satisfying the condition.

eg: SELECT * WHERE DNO. = 2



S6) Using Secondary index on an equality comparison:

This search ~~key~~ method can be used to retrieve any ~~kind of~~ ~~record~~ ~~f~~ record based on any kind of attribute (key/non-key) and any kind of comparison (equality/ inequality)

---

* S4 and S6 can thus, be used to find range ~~qure~~ queries.

If the SELECT query involves more than 1 condition & they are eg: Conjunctive condition (simple conditions connected with an AND)

eg: SELECT * WHERE age = '19' AND ID = '194'.

we can use the below algorithms:

S7) Conjunctive selection using ~~an~~ an individual index:

If an attribute involved in any single simple condition in the conjunctive SELECT condition has an

access path that permits the use of one of the methods S2 to S6, use that condition to retrieve all records & check whether each retrieved record satisfies the remaining conditions.

eg: In above example, ID has primary indexing thus it satisfies S3a).

We retriere all records with ID = '194' (only one here) & check if that record satisfies the other condition age = '19'.

S8) Conjunctive selection using a composite index:
If two or more attributes are involved in equality conditions in the conjunctive SELECT condition And the attributes form a composite key which has an indexing or hashing, we directly use the indexing/hashing to retrieve the data.

eg: SELECT * WHERE name = 'Uditi'
AND age = '19'
(Assuming name & age form a composite key which is indexed).

S9) Conjunctive selection by intersection of record pointers:
If the conditions connected by AND have secondary indexing for the attributes in the condition, the we can

we the secondary indexing to retrieve all the record pointers for each attribute & in the end, use intersection to find the final records. If only some attributes have secondary indexing, then we need to further check each record with remaining conditions.

~~S = Sq~~ Sq

* Note that, for ~~con~~ ~~Sq~~, we assumed secondary indexing because it is used for non-key values. If the ~~ind~~ indexing is primary (key value) indexing then we would directly get the desired record as only one record satisfies the equality condition.

* Query optimization is needed mostly for conjunctive select conditions whenever more than one of the attributes involved in the conditions have an access path. This can be done using the selectivity ratio, which is defined as $\frac{\text{no. of tuples retrieved}}{\text{~~no.~~ of total no. of tuples}}$.

The smaller the selectivity ratio, the higher desirability of using that condition first to retrieve records.

---

for disjunctive SELECT conditions

eg: SELECT * WHERE name = 'Chhavi'
OR marks = '25'

It is difficult to process & optimize the operation.
This is because we want a UNION of the records satisfying each condition. So, if even one attribute is not indexed/hashed, we will be forced to find all the desired records using the Brute force (linear) approach.

4.3.3) Algorithm for PROJECT operation:

- Definition: The PROJECT operation displays the specific columns of a table. It is denoted by pic (Π). It is a vertical subset of the original relation and it ~~eliminitat~~ eliminates duplicate tuples.

→ If the PROJECT operation is done on an attribute that is key attribute for a relation R ($\Pi_{<attribute>}(R)$) then it is straightforward algorithm because there are no duplicates. The result will have same no. of tuples as the parent relation (R).

→ If the attribute is not key, the duplicate tuples must be eliminated.

This can be done by

a) Sorting: sorting the resultant table according to the ~~the~~ attribute value & eliminate duplicate values as they are arranged consecutively.

b) Hashing: Since each attribute will have a hashed value after hashing, duplicate attribute values will have the same hash value. First particular We insert the record with a hash value in a bucket. If that hash value is encountered again, we will that the ~~recor~~ bucket already has a record & the duplicate will not be inserted.

4.4

Query Optimization:

Query optimization is the process of selecting the most efficient query evaluation plan from among the many strategies usually possible for processing a given query.

4.4.1) Overview:

1) Definitions of symbols:

- σ → Selection
- $\sigma_{dept\_name = "physics"}(instructor)$
  Selecting all tuples in relation 'instructor' with dept_name = "physics".
- $\wedge$ → and
- $\vee$ → OR

- $\neg$ → not
- Projection → Π
  - $\Pi_{ID, name, salary}(instructor)$
    Displays non duplicate tuples of the Relation instructors with attributes ID, name and salary only.
- $\Pi_{name}(\sigma_{dept\_name = "physics"}(instructor))$

  Find all names of the instructors whose dept_name = "physics" from the relation instructor.
- $\times$ → cartesian product : combine each tuple of one relation with every tuple of another relation.
- $\bowtie_\theta$ → Join operation : Combines related tuples from different relations, if & only if a given join condition is satisfied.

  Here θ is the condition on the attributes.
  - $\sigma_{instructor.ID = teaches.ID}(Instructor \times teaches)$
    $= \bowtie_{instructor.ID = teaches.ID}$
  - teaches $\bowtie \Pi_{course\_ID, title}(course)$: joins the relations 'teaches' and 'course' according to the value of course_ID.

2) There are 2 main techniques that are employed during query optimization:

a) Heuristic Query Optimization: A query can be represented as a tree data structure.

Operations are at the interior nodes & data items (tables, columns) are at the leaves. (Inorder)

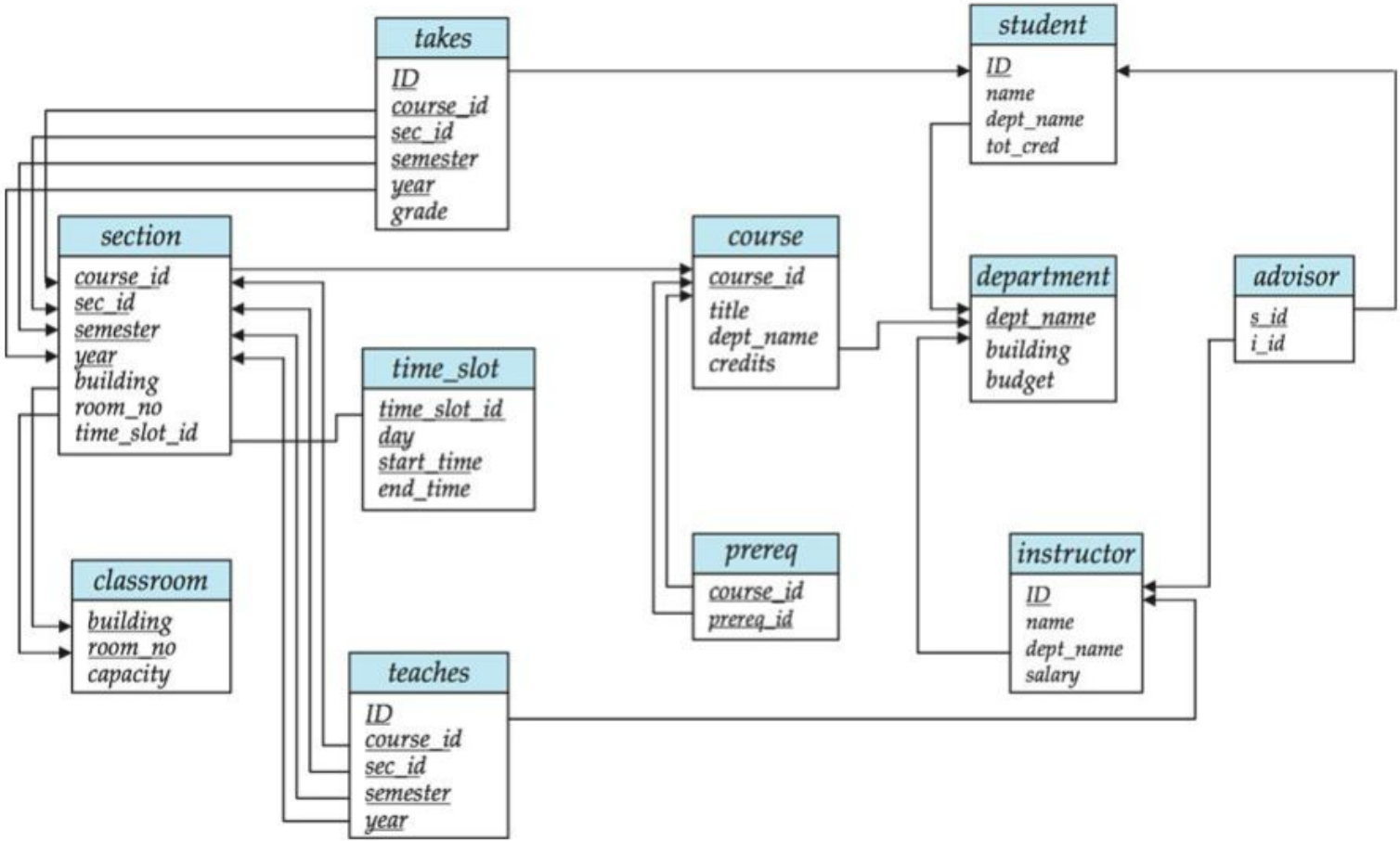
b) Systematically estimating the cost of different execution strategies & choosing the execution plan with the lowest cost estimate.

We use heuristic query optimization technique.

3) Expressions with symbols mentioned above are called Relational - algebra expression. Any query can be converted into a relational algebra expression.

4) Consider the following relationship algebra expression, for the query "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach".

Schema Diagram for University Database



$\Pi_{name, title}(\sigma_{dept\_name="Music"}(instructor \bowtie (teaches \bowtie \Pi_{course\_ID, title}(course))))$

a) In above expression, first ~~of~~ a table is created containing the columns for course_ID and title only, $(\Pi_{course\_ID, title}(course))$ of the relation course.

b) The above table is then joined with the relation teaches using course_ID as the common column.

c) The joined table is joined again with the instructor relation using ID as the common column.

d) From the table in c) all tuples with dept_name "Music" are selected.

e) ~~These~~ The name & title of these selected tuples are copied into a new table, & we get our final result.

5) In above step a), a very big relation is formed with 9 different columns, while we are only interest in a few tuples & a few columns.

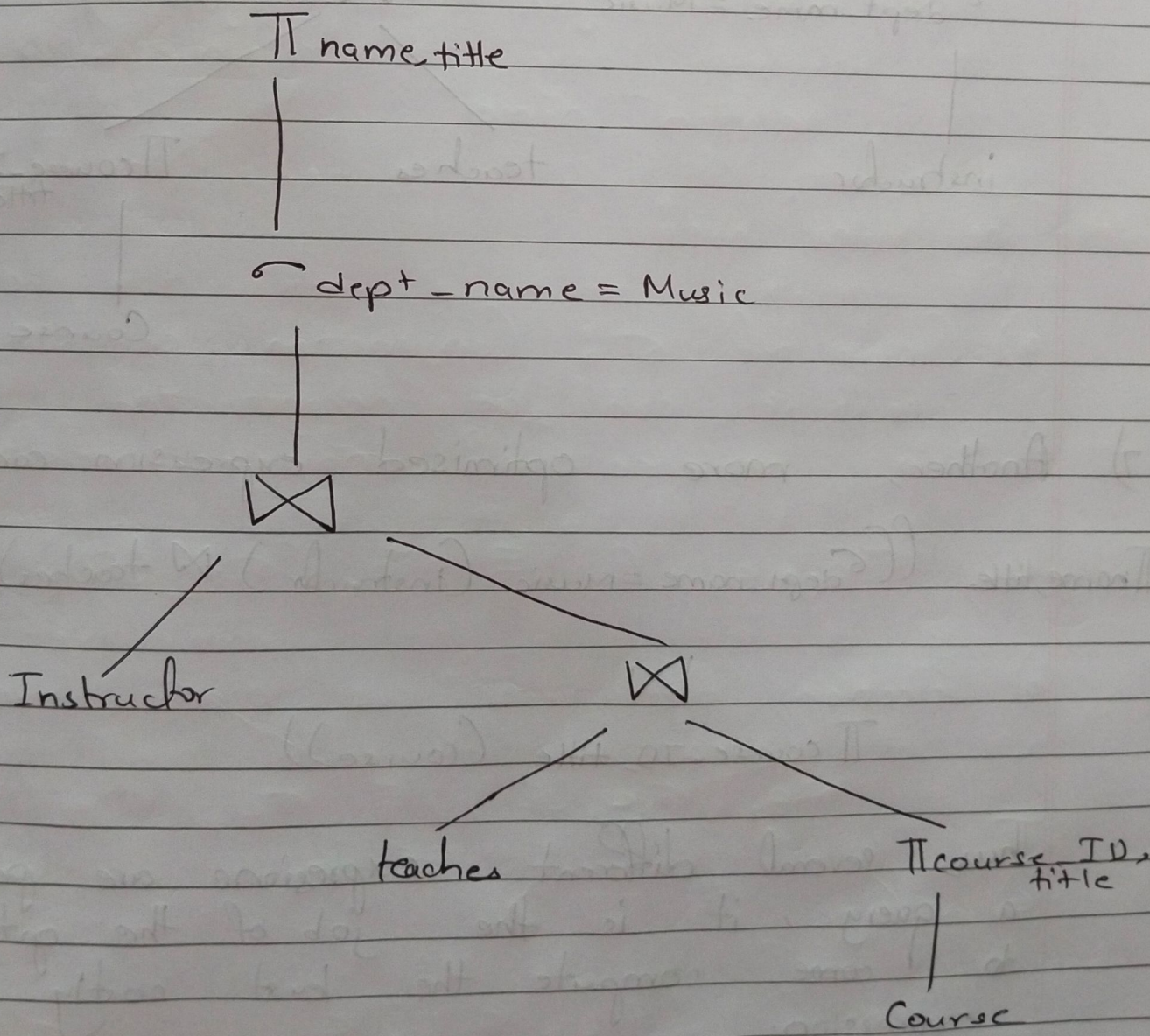
To optimize our expression we can

change it to :

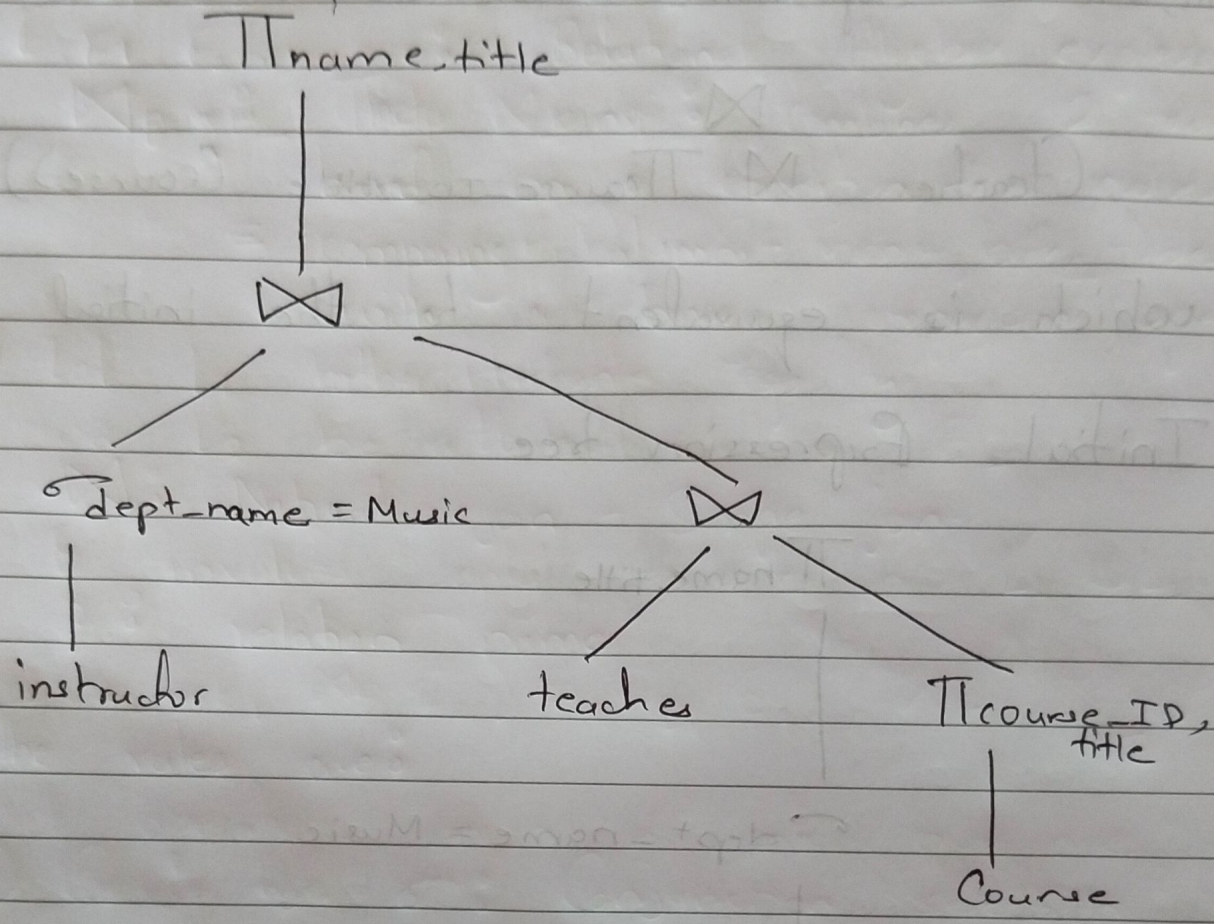
$\Pi_{name, title}((\sigma_{dept\_name = "Music"}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_ID, title}(course)))$

which is equivalent to the initial query.

6) Initial Expression tree



Transformed expression tree:



7) Another, more optimised expression can be:

$$\Pi_{name, title}\left(\left(\sigma_{dept\_name = Music}(instructor) \bowtie teaches\right) \bowtie \Pi_{course\_ID, title}(course)\right)$$

Thus, several different expressions are possible for a query, it is the job of the optimizer to compute the least costly expression.

8) To find the most optimized way, the optimizer needs to follow the following steps:

a) generate the expressions that are logically equivalent to the given expression.

b) Annotate the resultant expressions in alternative ways to generate alternate query-evaluation plans.

c) estimating the cost ~~& chosing~~ of each evaluation plan & chosing the least cost plan.

We will look into these three steps & the module will be over.

4.4.2) Transformation of Relational Expressions:

1) To implement the first step of the optimizer, it must generate all the expressions equivalent to given expression.

It does so by means of equivalence rules that specify how to transform an expression into a logically equivalent one.

2) Two relational-algebra expressions are said to be equivalent if the two expressions generate the same set of tuples.
Note that the order of the tuples is irrelevant.

3) Definitions:

$\theta, \theta_1, \theta_2, \ldots$: Conditions

$L, L_1, L_2, \ldots$: list of Attributes

$E, E1, E_2, \ldots$: Relational Algebra expressions

$\bowtie_\theta$: Theta joins : combines 2 tables based on a condition that involves comparing columns from both tables with a binary comparison operator other than equality.

$\bowtie$: Natural join - Simple join.

r : relation name

4) The rules are:

a) Cascade of σ : Conjunction selection operations can be deconstructed into a sequence of individual selections

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

b) Selection operations are commutative

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

c) Cascade of Π :

when $L_1 \subseteq L_2 \subseteq L_3 \subseteq L_4 \cdots \subseteq L_n$

$$\Pi_{L_1}(\Pi_{L_2}(\cdots(\Pi_{L_n}(E))\cdots)) \equiv \Pi_{L_1}(E)$$

Only final operations in a sequence of

projection operations are needed.

d) Selections can be combined with cartesian products & theta joins

i) $\sigma_\theta (E_1 \times E_2) \equiv E_1 \bowtie_\theta E_2$

ii) $\sigma_{\theta_1} (E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

e) Theta - Join operations are commutative

i) $E_1 \bowtie_\theta E_2 \equiv E_2 \bowtie_\theta E_1$

ii) $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$

f) i) Natural join operations are associative

$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$

ii) Theta joins are associative in the following manner:

$(E_1 \bowtie_\theta E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$

Cartesian Product is also associative.

g) i) Condition: Out of the 2 expressions only one expression's attributes should be involved in condition θ_i

$\sigma_{\theta_1} (E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta E_2$

ii) Condition: θ_1 should involve attributes of E_1 & θ_2 should involve attributes of E_2

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2)$$

$$\equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_1}(E_2))$$

h) Let L_1 be attribute list of E_1 and L_2 be attribute list of E_2

θ involves only attributes in $L_1 \cup L_2$, then

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

i) Set operations union & intersection are commutative:

i) $E_1 \cup E_2 \equiv E_2 \cup E_1$

ii) $E_1 \cap E_2 \equiv E_2 \cap E_1$

Set difference is not commutative.

j) Set union & intersection are associative

i) $(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$

ii) $(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$

k) Selection operation distributes over the union, intersection & set difference operations

i) $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta (E_2)$

ii) $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta (E_2)$

iii) $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta (E_2)$

iv) $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap E_2$

v) $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - E_2$

l) The projection operation distributes over the union operation

$$\Pi_L (E_1 \cup E_2) \equiv (\Pi_L (E_1)) \cup (\Pi_L (E_2))$$

eg) Earlier, we optimized the expression:

$\Pi_{name, title} (\sigma_{dept\_name = Music} (instructor \bowtie (teaches \bowtie \Pi_{course\_ID, title} (course))))$

to

$\Pi_{name, title} ((\sigma_{dept\_name = Music} (instructor)) \bowtie (teaches \bowtie \Pi_{course\_ID, title} (course)))$

Rule g(i) was used to perform the selection operation as early as possible, reducing the size of the relation to be joined.