Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm (每步最好的，不一定overall最好)

E.g., merge-join may be costlier than hash-join, but may provide a sorted output which could be useful later (the sorted result may be benefit to the later operation)

- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses **heuristics** to choose a plan. (看历史数据我是如何处理的)

Systems may use heuristics to **reduce the number of choices** that must be made in a cost-based fashion (because cost-based optimization is expensive); However, history rules and assumptions could become too simple for varied data and query structures, so historical plan cannot adapt to changing data distribution and lead to suboptimal performance.

Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

1. Perform selections early (reduces the number of tuples) (变短)
2. Perform projections early (reduces the number of attributes) (变窄)
3. Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations: This strategy prioritizes performing selection and join operations that filter out the largest number of rows as early as possible in the query execution process.

- Adaptive Plans - Wait for one/some parts of a plan to execute first, then choose the next best alternative

Troubleshooting to manage costs (e.g., Query Store in Microsoft)

- Identify ‘regressed queries’ - Pinpoint the queries for which execution metrics have recently regressed (for example, changed to worse).

- Track specific queries - Track the execution of the most important queries in real time. (e.g., most frequently asked queries)

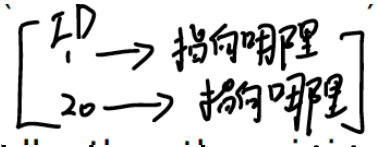
When you identify a query with suboptimal performance: 1.Force a query plan instead of the plan chosen by the optimizer 2.Do we need an **index**? --- quickly find the data in the query 3.Enforce statistic recompilation (重新编译) 4.Query rewriting with parameters for execution plan reuse 5.Memory optimised table (if there are some tables that are frequently used, we can create M.O.T, but if we apply the M.O.T on whole database, we cannot say any improvement because there is no enough space in memory) 6.limit the scope of selection before join

To further lower query cost: 1.**Store derived data:** When you frequently need derived values if Original data do not change frequently. 2. **Use pre-joined tables:** When tables need to be joined frequently;Regularly check and update pre-joined table for updates in the original table;May still return some ‘outdated’ result (pre-joined tables are not updated)

Indexing

1.**Search Key** - attribute or set of attributes used to look up records/rows in a system like an ID of a person (The ID, name, gender… will take us to get the entire record)

2.An **index file** consists of records (called index entries) of the form search-key, pointer to where data is



3.Index files are typically much smaller than the original data files and many parts of it are already in main memory (main memory is faster than disk)

Indexing makes Disk Access faster through (帮助我们和disk交互更快,原本不知道内容在disk哪里,在disk上瞎找)

1. records with a **specified value** in the attribute accessed with minimal disk accesses (e.g., Student ID = 101, the index file tells us which data block in disk to go)
2. records with an attribute value falling in a **specified range** of values can be retrieved with a single seek and then consecutive sequential reads (e.g., ID:100-200, find where is the first record (single seek of starting block), then read sequentially) **Note:** not all DB stores sequential data sequentially, so this method may not useful for some DB (to achieve 2, we need our data to be organized by the order of search key)

Criteria to choose index – always tradeoff

- Insertion time to index is also important
- Deletion time is important as well
- No big index rearrangement after insertion and deletion
- Space overhead needs to be considered for the index itself
- No single indexing technique is the best. Rather, each technique is best suited to particular applications.

频繁修改的数据不适合加index,
因为index也需要频繁修改-高overhead;

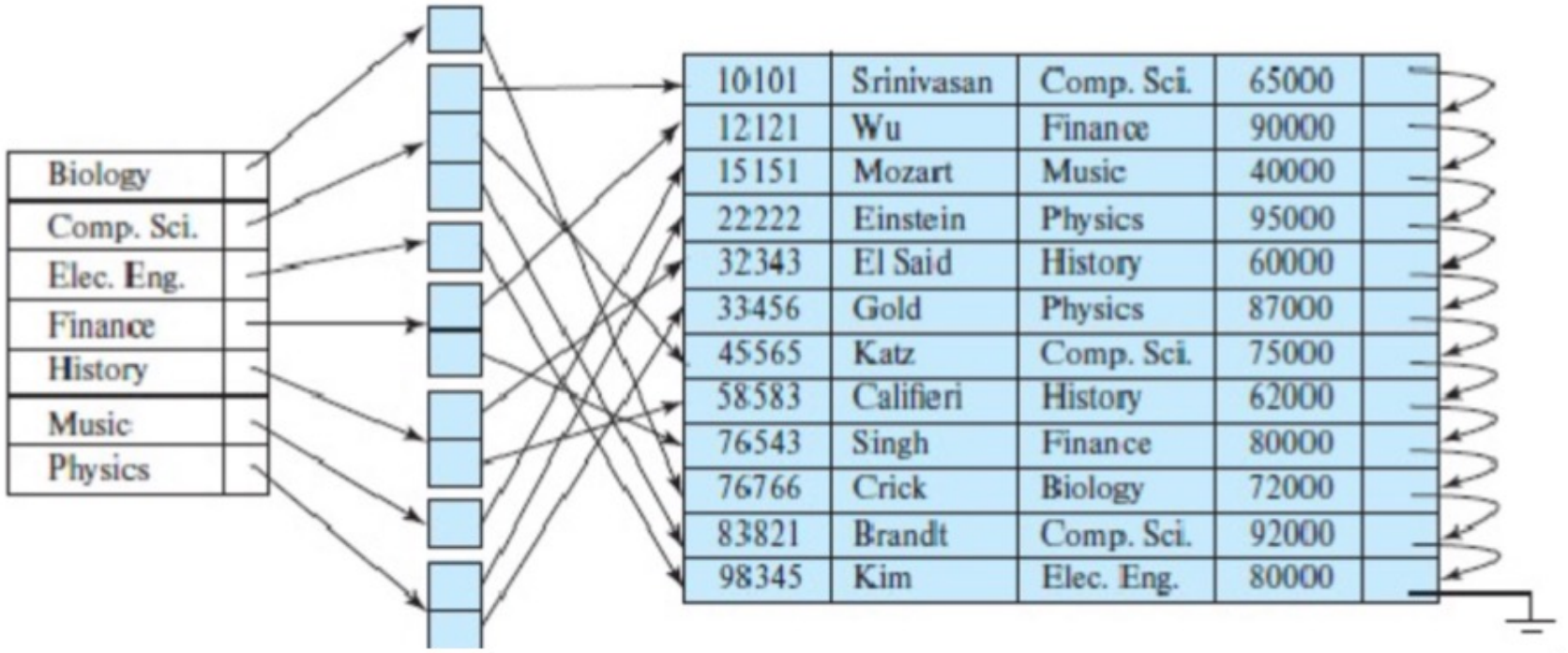
Types of indices based on search keys

- **Ordered indices** – Search keys are stored in some order

1.Clustering index / primary index: In a sequentially ordered file, the index whose search key specifies the sequential order of the file (数据本身是有顺序的, 比如ID 是按照顺序排好的，从小到大不重复) (The search key of a primary index is usually but not necessarily the primary key)

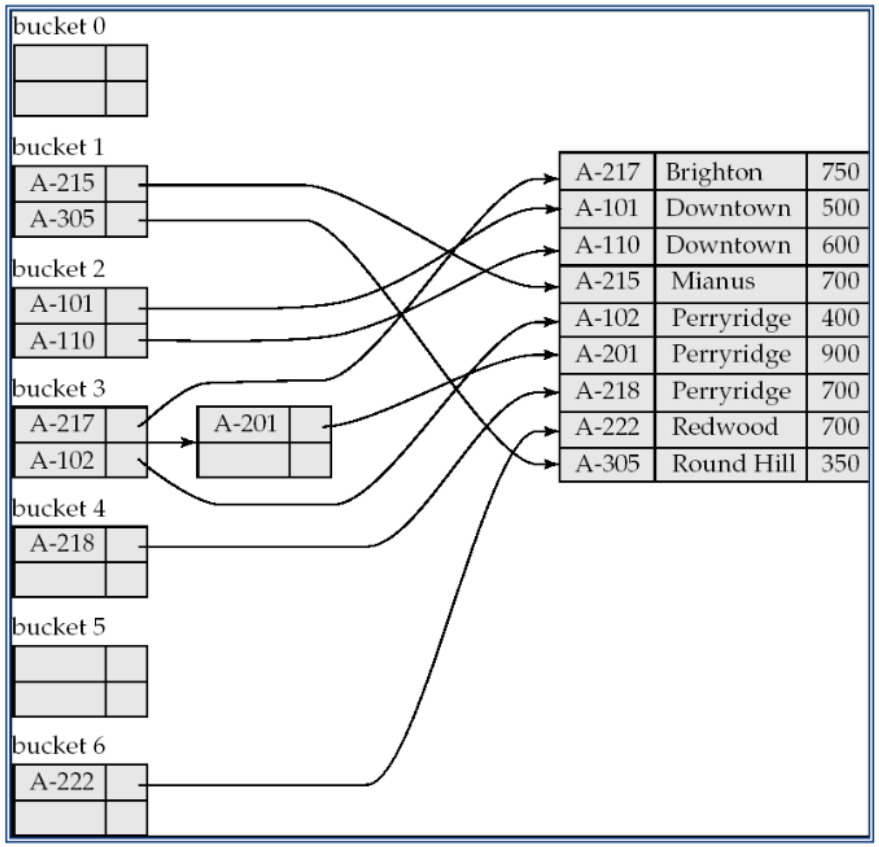
2.Non-clustering index / secondary index: An index whose search key specifies an order different from the sequential order of the file (Secondary indices improve the performance of queries that use keys other than the search key of the clustering index.) (与数据本身的顺序是不一样的)

按照department来分类



Hash index – suitable for uniform data distribution, if data distribution is not uniform, certain buckets may become overfilled, leading to higher collision rates and slower access times

Bitmap index- for the attribute that has small number of distinct values (for categorical data)



record number	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

B+ tree Definition / Rules It is similar to a binary tree in concept but with a fan out that is defined through a number n

1.All paths from root to leaf are of the same length (depth) 2.Each node that is not a root or a leaf has between $\lceil \frac{n}{2} \rceil$ and n children; 3.A leaf node has between $\lceil \frac{n-1}{2} \rceil$ and n-1 values. Special cases: 1.If the root is not a leaf, it has at least 2 children.(只有一个node时, 又是儿子又是爹) 2. If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n-1) values.

Q What are properties of an ideal hash function, why these properties are important for has indices?

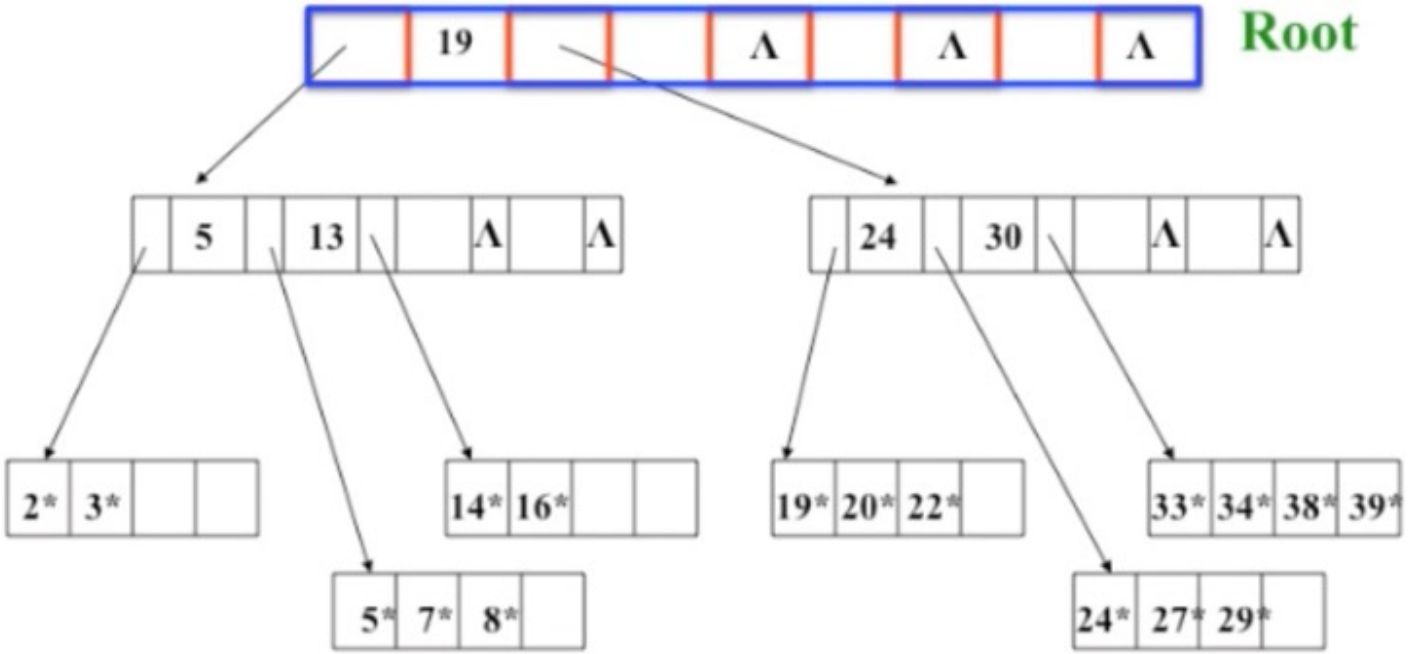
1.Uniform data dist, so that each bucket can store equally balanced number of pointers of records, which helps to find target query fast, rather then fail into a particular bucket with large number of values, which cost long time.

2.Make sure the each search key has only one unique mapping in hash index, each time it will point to the same record given a search key.

(range query & value query 都擅长)B+ tree Typical Node, search-keys in a node are ordered
n=4:p1k1p2k2p3k3p4 (k1<k2<k3<…) P: pointer; K: search key value

NOTE: Most of the higher level nodes of a B+ tree would be in main memory already! (leaf node is record and in disk)

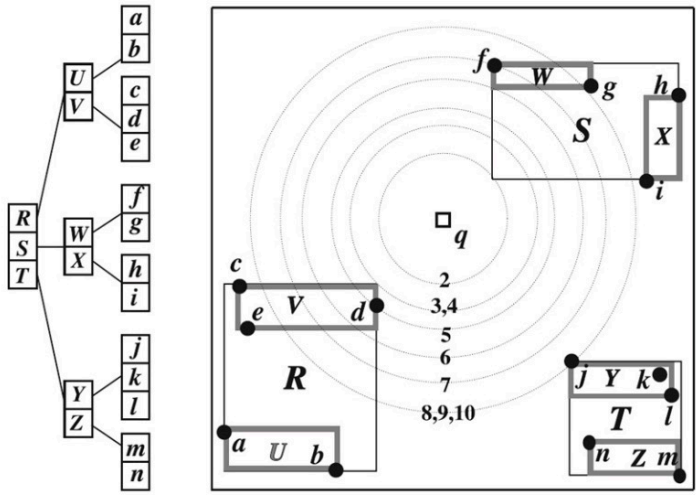
逐层比较大小



Pros:1.Automatically reorganizes itself with small, local changes in the face of insertions and deletions.2.Reorganization of entire file is not required to maintain performance. 3.Similar to Binary tree in many aspects but the fan out is much higher (height of B+ tree is smaller than binary tree)
Cons:Extra insertion and deletion over head and space overhead (reorganisation)

Advantages of B+-trees outweigh disadvantages for DBMSs

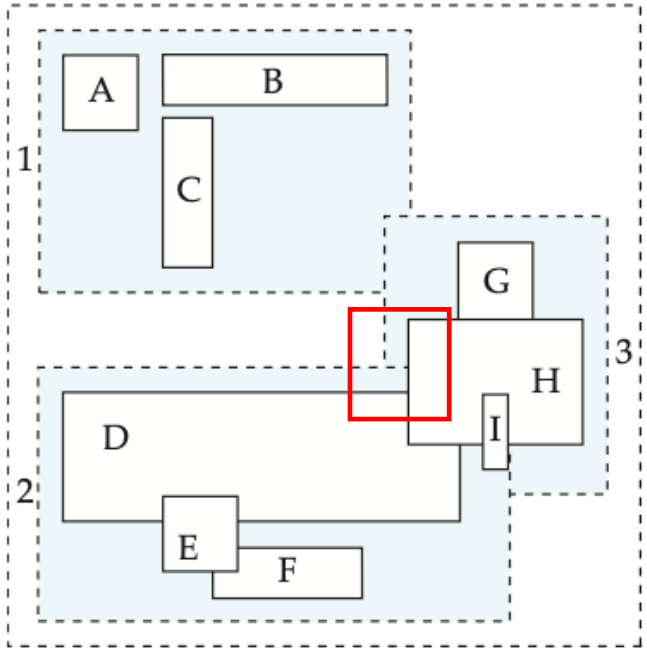
- **Quadtree (k-d tree)**: recursively divide a 2D space into four quadrants, subdividing until each node meets a specified condition (like a maximum number of points). Quadtrees are best suited for fixed spatial data, where the data points have a known, predictable spatial distribution (such as in images, terrain data, or spatial grids). Quadtrees are fast for point-based data and for finding data within rectangular, evenly distributed grids.
- **R-tree** group spatial objects based on their minimum bounding rectangles, arranging them in a hierarchical, tree-like structure that minimizes overlap and groups objects together. R-trees are best suited for spatial data with irregular boundaries, particularly when objects vary in size and shape (e.g., polygons, roads, and complex geographic features). R-trees are efficient for nearest-neighbor queries. **However, There might be multiple overlapping bounding boxes, which may lead to explore multiple parts or like going through many branches of tree to find the actual tree, can be computationally expensive in the worst case since multiple paths may need to be searched due to this overlaps.**



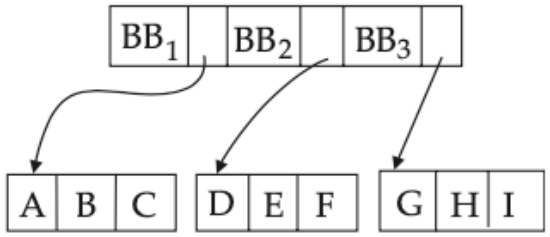
Step	Priority queue	Retrieved NN(s)
1	⟨S, R, T⟩	⟨⟩
2	⟨R, W, T, X⟩	⟨⟩
3	⟨V, W, T, X, U⟩	⟨⟩
4	⟨d, W, T, X, c, e, U⟩	⟨⟩

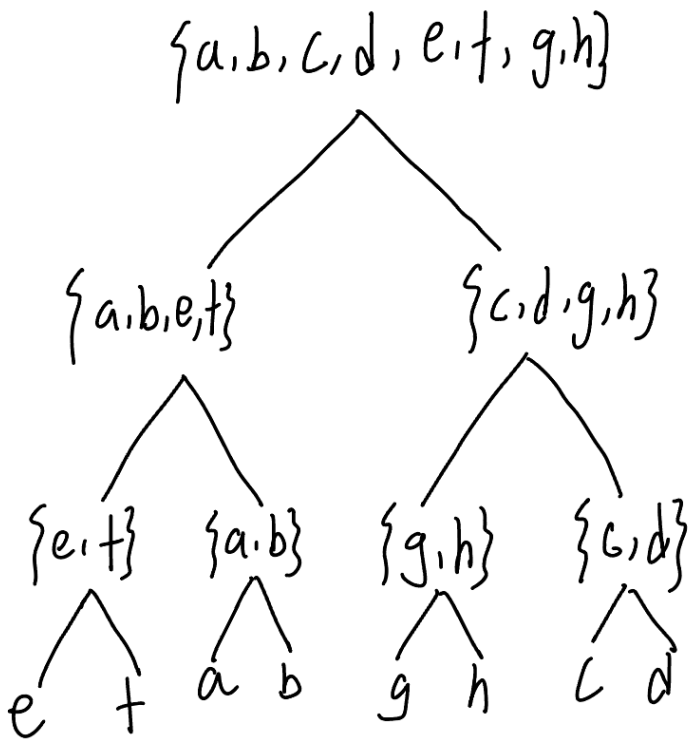
Step 5 finds d as the first NN (using Best First search)

1. <S,R,T> expand S (W,X,R,T)
2. <R, W, T, X> expand R (V, U, W, T, X)
3. <V, W, T, X, U> expand V (C, e, d, W, T, X, U)
4. <d, W, T, X, C, e, U> expand W <d, f, g, T, X, C, e, U>
5. <d, f, T, g, X, Ge, U> expand T (d, f, Y, Z, g, X, C, e, U)
6. <d, f, Y, g, X, C, e, U, Z> expand Y (d, f, j, K, Z, g, X, C, e, U)
7. <d, f, j, g, X, C, e, K, U, Z>



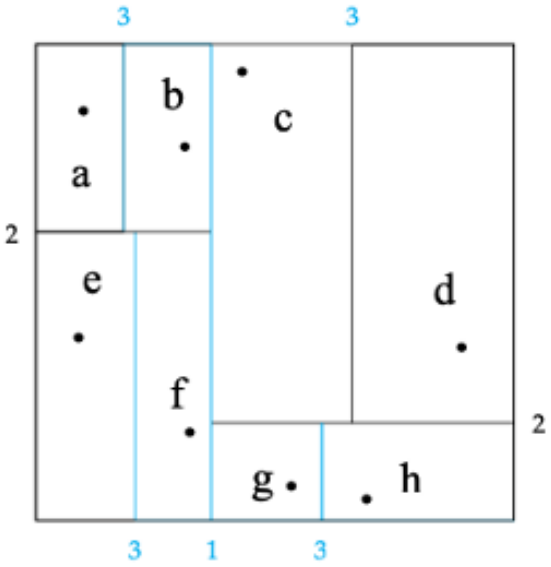
与BB1不相交，与BB2相交，与BB3相交，最后返回的是D和H





L → Left

L



Q1.Find at least on person whose salary is greater than 80000

Ans: Cost is search through all the records in this table, making comparison one by one, hence having all those index is not helpful for such huge volume of comparison.

Q1. Which one of the indexes is a suitable choice? Specify search key for that index. The data is the records of one million families, where each family is a row in a table. A unique ID and the total household income per year of each family are stored in the database. The total household income of any family can be between 0 to 2 million, and it is an integer number (i.e., a whole number without any decimal value). The most common queries in this database look like – (i) find the IDs of the families where the total household income is less than 70,000; (ii) find the IDs of the families where the total household income is between 70,000 and 100,000; (iii) find the IDs of the families where the total household income exactly equals to 72,050; Ans: 1.Not ordered index: there's no naturally sequentially ordered search key, we have to create by ourselves, which is time consuming; 2.Not bitmap, which is suitable for the attribute that has small number of distinct values, here 2 million is too huge; 3.Not hash index, which is suitable for specific value but not for specified range, also we don't have any information about distribution of data (i.e., if is it not uniform but a skew distribution that may lead to a overfilling bucket); **Therefore,** we need to use B+tree as index, and select search key as 'total household income per year', so that we can speedup these frequent queries for selecting specified ranges or values.

Q2. A unique ID, the location of the family's current residence, and the total household income per year of each family are stored in the database.

1. Find the IDs of the families where the total household income is less than 70,000. 2. Find the household income of the families who live within 5km of Melbourne central station. 3. Find the average household income of the families.

Ans: 1. A specified range query so it is not suitable for hash index; Too much distinctive values from 0 to 2million, so it not suitable for bit map; There is no naturally sequentially ordered search key, we have to create by ourselves which is time consuming and might not be help for other future queries. So we use B+tree index for this specified range query and select income as search key. 2.spatial related query location: R-trees are efficient for nearest-neighbor queries. 3.No index is suitable: for calculation of average value, we have to calculate the summation of income firstly, hence we have to access income data for all records, but index can only help us to quickly locate the specific records or subset of data. (Here index does not provide any speedup for selecting all income values and may even add overhead due to the need to reference the index structure.)

Q. We have seen the B+tree concept for indexing in class. These trees are deemed as an improvement over binary trees with their large fan out e.g., in the range of 100s. i) What is the benefit of making such a decision with fan outs, especially in the context of DBMSs? Briefly explain. ii) What are the benefits and disadvantages of using a B+tree in comparison to using hashing? Briefly explain.

Ans i)shallow depth can help to reduce the number of searching depth, reducing the I/O operations, and speedup the efficiency. Since the actual data records are stored in the leave nodes, if depth of tree is deeper, then we need more steps to reach leave nodes, resulting in high I\O costs for fetching data from disk. ii) **Pros:** Efficient range queries: B+tree supports efficient range queries, as the leaf nodes are linked sequentially allowing for ordered traversal. 2) Dynamic inserts and deletes: B+tree handle dynamic inserts and deletions without significant reconstructing, maintaining balanced tree tree structure. 3) B+tree store the data in ordered way, which is helpful for the queries requiring sorted output. **Cons:** Higher memory usage, B+tree requires additional pointers and nodes to maintain its structure, such structure using more memory than hash tables.