CS562: Database Management Systems II Notes

Steven DeFalco

Spring 2024

Contents

1 Indexing

Indexing methods are used to speed up access to desired data. A ***search key*** is an attribute used to look up records in a file. An ***index file*** consists of records (called **index entries**) of the form. Index files are typically much smaller than the original file. There are two basic kinds of indices:

- **Ordered indices**: search keys are stored in sorted order
- **Hash indices**: search keys are distributed uniformly across *buckets* using a *hash function*

Remark When *optimizing* in database queries, the goal is to have a small search space.

In an ***ordered index***, index entries are stored sorted on the search key value. In a sequentially ordered file, the ***primary index*** is the index whose search key specifies the sequential order of the file; the search key of a primary index is usually but not necessarily the primary key. The ***secondary index*** is an index whose key specifies an order different from the sequential order of the file (also called non-clustering index). An ***index-sequential file*** is an ordered sequential file with a primary index.
A ***dense index*** is where an index record appears for every search-key value in the file. ***Sparse index*** contains index records for only some search-key values. To locate a record with search-key value K we:

1. Find index record with largest search-key value $< K$
2. Search file sequentially starting at the record to which the index record points

Remark Sparse index must be primary index

Compared to dense indices, sparse indices have less space and less maintenance overhead for insertions and deletions. However, they are generally slower than dense index for locating records. a record with search-key value K we:

1. Find index record with largest search-key value $< K$
2. Search file sequentially starting at the record to which the index record points

Remark Sparse index must be primary index

Compared to dense indices, sparse indices have less space and less maintenance overhead for insertions and deletions. However, they are generally slower than dense index for locating records.

If primary index does not fit in memory, access becomes expensive. A solution to this is to treat primary index kept on disk as a sequential file and construct

a sparse index on it. The outer index is a sparse index of primary index. The inner index is the primary index file. If even out index is too large to fit in main memory, yet another level of index can be created and so on. Indices at all levels must be updated on insertion or deletion from the file.

1.1 Index Updates

1.1.1 Deletion

If the deleted record was the only record in the file with its particular search-key value, the seark-key is deleted from the index also. For singl e-level index deletion.

- Dense indices—deleetion of search-key
- Sparse indices—
  - If an entry for the search key exists in the index, it is deleted by replacing the entry in teh index with the next search-key value in the file
  - If the next search-key value already has an index entry, the entry is deleted instead of being replaced

1.1.2 Insertion

In single-level index insertion:

- Perform a lookup using the searh-key value appearing in the record to be inserted.
- Dense indices—if the search-key value does not appear in the index, insert it
- sparse indices—if index stores an entry for each block of the file, no chnage needs to be made to the index unless a new block is created. If a new block is created, the first search-key value appearing in the new block is inserted into the index.

In multilevel insertion, algorithms are simple extensiosn of the single-level algorithms.
Indices offer substantial benefits when searching for records, but updating indices imposes overhead on database modification—when a file is modified, every index on the file must be updated. Sequential scan using primary index is efficient, but a sequential scan using a second index is expensive.

2 B+ Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files. Some disadvantages of indexed-sequential files are that performance degrades as the file grows (since many overflow blocks get created) and that periodic reorganization of the entire file is required. Advantages of B^+-tree index files are:

- automatically reorganizes itself with small, local changes in the face of insertions and deletions
- Reorganization of entire file is not required to maintain performance

One minor disadvantage of B^+-trees is the extra insertion and deletion overhead (space overhead); however, the advantages outweigh the disadvantages and thus this structure is used extensively.

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases...
  - If the root is not a leaf, it has at least 2 children.
  - 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.

2.1 Types of nodes in B^+-trees

The following are properties of a **leaf node**:

- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_j's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order

Nonleaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

- All the search-keys in the subtree to which P_1 points are less than K_1
- For $2 \leq i \leq n-1$, all the search-keys in the substree to which P_i points have values greater than or equal to K_{i-1} and less than K_i

- All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

Since the inter-node connections are done by pointers, *logically* close blocks need not be *physically* close. The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices. The B$^+$-tree contains a relatively small number of levels; level below root has at least $2 \times \lceil n/2 \rceil$ values. If there are K search-key values in the file, the tree height is no more than $\left\lceil \log_{\lceil n/2 \rceil}(K) \right\rceil$. Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

2.2 Queries on a B$^+$-tree

Find all records of a search-key value of k.

1. $N =$ root
2. Repeat
   (a) Examine N for the smallest search-key value $> k$.
   (b) If such a value exists, assume it is K_i. THen set $N = P_i$
   (c) Otherwise $k \geq K_{n-1}$. Set $N = P_n$

   Until N is a leaf node
3. If for some i, key $K_i = k$ follow pointer P_i to the desired record or bucket.
4. Else no record with search-key value k exists

2.3 Updates on B$^+$-trees

2.3.1 Insertion

To perform insertion on a B$^+$-tree...

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   (a) Add record to the file
   (b) If necessary add a pointer to the bucket
3. If the search-key value is not present, then
   (a) Add the record to the main file (and create a bucket if necessary)
   (b) If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   (c) Otherwise, split the node (along with new (key-value, pointer) entry)

To split a leaf node...

- Take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
- let the new node by p, and let k be the least key value in p. Insert (k, p) in the parent of the node being split.
- If the parent is full, split it and **propagate** the split further up.

Splitting of nodes proceeds upwards until a node that is not full is found. In the wrost case, the root node may be split increasing the height of the tree by 1.

To split a non-leaf node: when inserting (k, p) into an already full internal node N

- Copy N to an in-memory area M with space for $n+1$ pointers and n keys
- Insert (k, p) into M
- Copy from M back into node N
- Copy from M into newly allocated node N'
- Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

2.3.2 Deletion

To perform a deletion in a B^+-tree...

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then *merge siblings*:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then *redistribute pointers*:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries

  - Update the corresponding search-key value in the parent of the node

- The node deletions may cascade upwards until a node which has $\lceil n/2 \rceil$ or more pointers is found

- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

2.3.3 Indexing

Variable lenght strings are used as the keys: use space utilization as criterion for splitting, not number of pointers. ***Prefix compression*** is that key values at internal nodes can be prefixes of the full key. For this, must keep enough chraracters to distinguish entries in the subtrees separated by the key value. Keys in the leaf nodes can be compressed by sharing common prefixes.

2.3.4 B-Tree Index Files

B-Tree index files are similar to B^+-trees, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys. Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included. Some advantages of B-tree indices are:

- May use less tree nodes than a corresponding B^+-tree

- Sometimes possible to find search-key value before reaching leaf node

Some disadvantages of B-tree indices are:

- Only small fraction of all search-key values are found correctly

- Non-leaf nodes are larger, so fan-out is reduced. Thus, B-trees typically have greater depth than corresponding B^+-tree

- Insertion and deletion are more complicated than in B^+-trees

- Implementation is harder than B^+-trees

Typically, the advantages of B-Trees do not out weigh the disadvantages.
Covering indices refers to adding extra attributes to an index so (some) queries can avoid fetching the actual records. If a record moves, all secondary indices that store record pointers have to be updated. Node splits in B^+-tree file organizations become very expensive; the solution is to use primary-index search keys instead of record pointers in secondary index.

3 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 a 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.

Remark 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.

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.

Bucket overflow can occur because of...

- Insufficient buckets
- Skew in distribution of records

Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*. In ***closed hashing*** (overflow chaining), the overflow buckets of a given bucket are chained together in a linked list (when bucket is full, just chain/link to a new bucket). An alternative, called ***closed hashing***, which does not use overflow buckets, is not suitable for database applications.

3.1 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. This is applicable on attributes that take on a relatively small number of distinct values. A bitmap is simply an array of bits.

In its simplest form, a bitmap index on an attribute has a bitmap for each value of th attribute. The 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.

Remark To find the number (and location) of records that have two specific attributes, then we **and** together the two bitmaps for each attribute and the resulting bit map will have ones in the locations of every record that shares both those attributes. E.g. all zeroes in the resulting bitmap would indicate that there are no records which have both of those attributes.

Remark Bitmaps work very well when the **cardinality is low** (i.e. each attribute only has a few possible values that it can take on). If attributes include gender, country, or similar values that don't have many unique values; then a bitmap is an appropriate option.

4 Query Processing

The basic stpes in query processing are **parsing and translation**, **optimization**, and **evaluation**.

4.1 Optimization

A relational algebra expression may have many equivalent expressions. Each relational algebra operation can be evaluated using one of several different algorithms. Correspondingly, a relational-algebra expression can be evaluated in many ways. Annotates expression specifying detailed evaluation strategy is called an ***evaluation-plan***.

Amongst all equivlaent evaluation plans, chosoe the one with the lowest cost. Cost is estimated using statistical information about the database.

4.1.1 Measures of Cost

Cost is generally measured as total elapsed time for answering query. Typically disk access is the predominant cost, and is also relatively easy to estimate. This is measured by taking into account

- Number of seeks times average seek-cost
- Number of blocks read times average block read cost
- Number of blocks written times average block write cost

Remark The cost to write a block is greater than the cost to read a block. Data is read back after being written to ensure that the write was successful.

Consider that t_T is the time to transfer one block and t_S is the time for one seek. Thus, the cost for one b block transfers plus S seeks is $b \times t_T + S \times t_S$.

4.2 Selection Operation

File scan are search algorithms that locate and retrieve records that fulfill a selection condition. Algorithm **A1** (linear search) will scan each file block and test all records to see whether they satisfy the selection condition. The cost estimate for linear search is b_r block transfers + 1 seek (b_r denotes the number of blocks contianing records from relation r). If selection is on a key attribute, can stop on finding record. Linear search can be applied regadless of selection

condition, ordering of records in the file, or availability of indices.

A2 (binary search) is applicable if an equality comparison on the attribute on which file is ordered. Assume that the blocks of a relation are stored contiguously. The cost estimate (number of blocks to be scanned) is the $\lceil \log_2(b_r) \rceil \times (t_T + t_S)$.

Index scan are search algorithms that use an index. The selection condition must be on search-key of index.

In **A3** (primary index on candidate key, equality), retrieve a single record that satisfies the corresponding equality condition

$$\text{Cost} = (h_i + 1) \times (t_T + t_S)$$

where h_i is the height of the index.

In **A4** (primary index on nonkey, equality), retrieve multiple records. Records will be on consecutive blocks. Let b = number of blocks containing matching records

$$\text{Cost} = h_i \times (t_T + t_S) + t_S + t_T \times b$$

In **A5** (equality on serach-key of secondary index), retrieve a single record if the sarch-key is a candidate key

$$\text{Cost} = (h_i + 1) \times (t_T + t_S)$$

Retrieve multiple records if search-key is not a candidate key. Each of n matching records may be on a different

$$\text{Cost} = (h_i + n) \times (t_T + t_S)$$

In **A6** (primary index, comparison) (Relation is sorted on A)... For $\sigma_{A \geq v}(r)$ use index to find first tuples $\geq v$ and scan relation sequentially from there. For $\sigma_{A \leq v}(r)$ just scan relation sequentially until the first tuple $> v$; do not use index.

In **A7** (secondary index, comparison)... For $\sigma_{A \geq v}(r)$ use index to find first index entry *geqv* and scan index sequentially from there, to find pointers to records. For $\sigma_{A \leq v}(r)$ just scan leaf pages of index finding pointers to records, until first entry $> v$. In either case, retrieve records that are pointed to

- requires an I/O for each record
- Linear file scan may be cheapter (use second index iff very few records are selected).

Conjunction is when there are multiple conditions to check for. In **A8** (conjunctive selection using one index), select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$. Test other conditions on tuple after fetching it into memory buffer.

In **A9** (conjunctive selection using multiple-key index), use appropriate composite (multiple-key) index if available.

A10 (conjunctive selection by intersection of identifiers) requires indices with record pointers. Use corresponding index for each condition and take intersection of all the obtained sets of record pointers. Then fetch records from file. If some conditions do not have appropriate indices, apply test in memory.

Disjunction is when there are multiple conditions *or*ed together. **A11** (disjunctive selection by union of identifiers) is applicable if all conditions have available indices, otherwise use linear scan. Use corresponding index for each condition, and take union of all the obtained sets of record pointers. Then fetch records from file. When **negation** is present, use linear scan on file.

4.3 Join Operations

There are several different algorithms that can be used to implement joins

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join

The choice should be made based on cost estimate.

4.3.1 Nested-Loop Join

In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$n_r \times b_s + b_r$$

block transfers, plus

$$n_r + b_r$$

seeks. If the smaller relation fits entirely in memory, use that as the inner relation. This reduces the cost to $b_r + b_s$ block transfers and 2 seeks.

4.3.2 Block Nested-Loop Join

Block Nested-Loop Join is a variant of nest-loop join in which every block of inner relation is paired with every block of outer relation. The worst case estimate is $b_r \times b_s + b_r$ block transfers $+2 \times b_r$ seeks. Each block in the inner relation is read once for each block in the outer relation (instead of once for each tuples in the outer relation). The best case is $b_r + b_s$ block transfers $+2$ seeks. For each tupel t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.

4.3.3 Index Nested-Loop Join

Index lookups can replace file scans if jion is an equi-jion or natural noin and an index is available on the inner relation's join attribute. For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r. The worst case is that the buffer has space for only one page of r, and, for ecah tuple in r, we perform an index lookup on s. The cost of the join is $b_r(t_T + t_s) + n_r \times c$ where c is the cost of traversing index and fetching all matching s tuples for one tuple of r. c can be estimated as the cost of a single selection on s using the join condition. If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

4.3.4 Hash-Join

Hash-join is applicable for equi-joins and natural joins. A hash function h is used to partition tuples of both relations. h maps *JoinAttrs* values to $\{0, 1, \ldots, n\}$ where *JoinAttrs* denotes the common attributes of t and s used on the natural join. r tuples in r_i need only to be compared with s tuples in s_i and need not be compared with s tuples in any other partition, since an r tuples and an s tuple that satisfy the join condition will have the same value for the join attributes.

4.3.5 Complex Joins

For joins with **conjunctive** conditions (*or*ed together), either use nested loops/block nested loops or conpute the resolt of one of the simpler joins.

For joins with a **disjunctive** condition, either use nested block loops/block nested loops or compute as the union of the records in individual joins.

4.4 Other Operations

Duplicate elimination can be implemented via hashing or sorting. On sorting, duplicates will come adjacent to each other, and all but one set of duplicates can be deleted. For optimization, duplicates can be deleted during run generation

as well as at intermediate merge steps in external sort-merge.

Aggregation can be implemented in a manner similar to duplicate elimination. Sorting or hasing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group. For optimization, combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values.

Set Operations can either use variant of merge-join after sorting, or variant of hash-join.

Outer join can be computed either as a join followed by addition of null-padded non-participating tuples or by modifying the join algorithms.

5 Materialization and Pipelining

There are some alternatives for evaluating an entire expression tree. In ***materialization***, we generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk (and repeat). In ***pipelining***, pass on tuples to parent operations even as an operation is being executed.

5.0.1 Materialization

Materialized evaluation will evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations. Materialized evaluation is always applicable. However, the costs of writing results to disk and reading them back can be quite high. ***Double buffering*** will use two output buffers for each operation, when one is full write it to disk while the other is getting filled. THis allows overlap of disk writes with computation and reduces execution time.

5.0.2 Pipelining

Pipelined evaluation evalues several operations simulatenously, passing the results of one operation on to the next. This is much cheaper than materialization: there is no need to store a temporary relation to the disk. However, pipelining may not always be possible. For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operations. Pipelines can be executed in two ways: **demand driven** and **producer driven**.

6 Query Optimization

An ***evaluation plan*** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated. The cost difference between evaluation plans for a query can be enormous. There are three steps in ***cost-based query optimization***:

1. Generate logically equivalent expressions using **equivalence rules**
2. Annotate resultant expressions to get alternative query plans
3. Choose the cheapest plan based on **estimated cost**

The estimation of plan cost is based on

- Statistical information about relations (e.g. number of tuples, number of distinct values for an attribute)
- Statistics estimation for intermediate results
- Cost formulae for algorithms, computed using statistics

Some **equivalence rules** include the following:

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
2. Selection oeprations are commutative.
3. Only the last in a sequence of projection operations is needed, the others can be omitted.
4. Selections can be combined with Cartesian products and theta joins.
5. Theta-join operations (and natural joins) are commutative.
6. Natural join operations are associative.
7. The selection operation distributes over the theta join operation under the following two conditions
   - When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.
   - When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.
8. The projection operation distributes over the theta join operation in some cases.
9. The set operations union and intersection are commutative.
10. Set union and intersection are associative.

11. The selection operations distributes over $\cup, \cap,$ and $-$.

12. The projection operation distributes over union.

Remark *Cost* based optimization selects the execution plan with the lowest cost. A *heuristic* approach to optimization uses logic to select the best execution plan; make the table as small as possible and do so as early as possible.

Three strategies used in ***heuristic optimization*** are as follows:

1. Push down selection operators (to reduce number of rows)

2. Push down projection operations (to reduce number of columns)

3. Do the most restrictive join first

Join associativity is the property that allows us to adjust the order of joins; however, testing to find the most effcient join is expensive and not typically done in practice.

Query optimizers use equivalence rules to ***systematically*** generate expressions equivalent to the given expression. Space requirements are reduced by sharing common sub-expressions. Time requirements are reduced by not generating all expressions. To determine the cost of each operator, we need the statistics of input relations; these statistics (number of columns/rows, number of distinct values: cardinality) are also stored for each data set.

We must consider the interaction of evaluation techniques when choose evaluation plans; choosing the cheapest algorithm for each operation independently may not yield the best overall algorithm. 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

To find the best join tree for a set of n relations... Consider all possible plans, recursively compute costs for joining subsets to find the cost of each plan, when a plan for any subset is computed, store it and reuse it when it is required again (instead of recomputing it): **Dynamic Programming**.

In ***left-deep join trees***, the right-hand-side input for each join is a relation, not the result of an intermediate join. Cost-based optimization is expensive, even with dynamic programming. Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion. Heuristic optimization transofrms the query-tree by using a set of rules that typically improve execution performance.

6.1 Statistics for Cost Estimation

Some important statistics to keep track of are...

- n_r: number of tuples in a relation r
- b_r: number of blocks containign tuples of r
- I_r: size of a tuple r
- f_r: blocking factor of r—i.e. the number of tuples of r that fit into one block
- $V(A, r)$: number of distinct values that appear in r for attribute A; same as the size of $\prod_A(r)$