7. Indexing

Contents:

- Single-Level Ordered Indexes
- Multi-Level Indexes
- B^+ Tree based Indexes
- Index Definition in SQL

Basic Concepts

- Indexing mechanisms are used to optimize certain accesses to data (records) managed in files. For example, the author catalog in a library is a type of index.
- *Search Key (definition):* attribute or combination of attributes used to look up records in a file.
- An *Index File* consists of records (called index entries) of the form

search key value	pointer to block in data file

- Index files are typically much smaller than the original file because only the values for search key and pointer are stored.
- There are two basic types of indexes:
  - *Ordered indexes*: Search keys are stored in a sorted order (main focus here in class).
  - *Hash indexes:* Search keys are distributed uniformly across "buckets" using a hash function.

Index Evaluation Criteria

Indexing techniques are evaluated on the basis of:

- Access types that are efficiently supported; for example,
  - search for records with specified values for an attribute
    (**select** $*$ **from** EMP **where** EmpNo $=$ 4711;)
  - search for records with an attribute value in a specified range
    (**select** $*$ **from** EMP **where** DeptNo **between** 20 **and** 50;)
- Access time (index entry $\longrightarrow$ record)
- Insertion time (record $\longrightarrow$ index entry)
- Deletion time (record $\longrightarrow$ index entry)
- Space and time overhead (for maintaining index)

Types of Single Level Ordered Indexes

- In an *ordered index file*, index entries are stored sorted by the search key value.
  - most versatile kind of index: supports lookup by search key value or by range of search key values
- *Primary Index:* in a sequentially ordered file (e.g., for a relation), the index whose search key specifies the sequential order of the file. For a relation, there can be at most one primary index. ($\leadsto$ index-sequential file)
- *Secondary Index:* an index whose search key is different from the sequential order of the file (i.e., records in the file are not ordered according to secondary index).
- If search key does not correspond to primary key (of a relation), then multiple records can have the same search key value
- *Dense Index Files:* index entry appears for every search key value in the record file.
- *Sparse Index Files:* only index entries for some search key values are recorded.
  - To locate a record with search key value K, first find index entry with largest search key value $< K$, then search file sequentially starting at the record the index entry points to
  - Less space and maintenance overhead for insertions and deletions
  - Generally slower than dense index for directly locating records

Secondary Indexes

- Often one wants to find all records whose values in a certain field (which is not the search key of the primary index) satisfy some condition
  - Example 1: In the EMPLOYEE database, records are stored sequentially by EmpNo, we want to find employees working in a particular department.
  - Example 2: as above, but we want to find all employees with a specified salary or range of salary

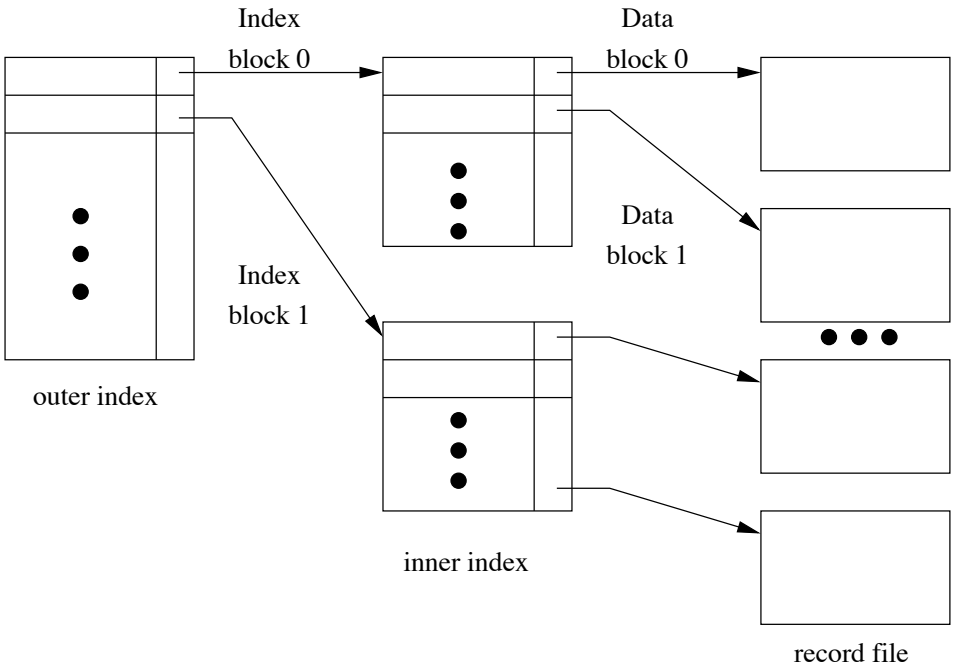
- One can specify a secondary index with an index entry for each search key value; index entry points to a *bucket*, which contains pointers to all the actual records with that particular search key.

Primary Indexes vs. Secondary Indexes

- Secondary indexes have to be dense
- Indexes offer substantial benefits when searching for records
- When a record file is modified (e.g., a relation), every index on that file must be updated. Updating indexes imposes overhead on database performance.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive (each record access may fetch a new block from disk)

Multi-Level Index

- If primary index does not fit in memory, access to records becomes expensive
- To reduce number of disk accesses to index entries, treat primary index on disk as sequential file and construct a sparse index on it.
  – outer index ⟶ a sparse index of primary index
  – inner index ⟶ the primary index file
- Multilevel Index structure



- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Note that indexes at all levels must be updated on insertions or deletions of records from a file.

Dynamic Multi-Level Indexes using B^+-Trees

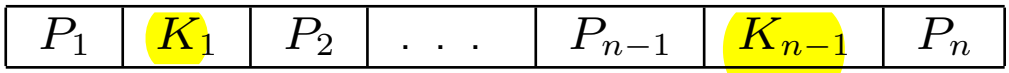
B^+-Tree indexes are an alternative to index sequential files.

- Disadvantage of index-sequential files: performance degrades as sequential file grows, because many overflow blocks are created. Periodic reorganization of entire file is required.
- Advantage of B^+-Tree index file: automatically reorganizes itself with small, local changes in the case of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B^+-Trees: extra insertions and deletion overhead, space overhead.
- Advantages of B^+-Trees outweigh disadvantages, and B^+-Trees are used extensively in all DBMS.

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

- All paths from the root to leaf have the same length ($\Longrightarrow$ a B$^+$ tree is a balanced tree).
- Each node that is not the root or a leaf node has between $\lceil n/2 \rceil$ and n children (where n is fixed for a particular tree).
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values.
- Special case: if the root is not a leaf, it has at least 2 children. If the root is a leaf, it can have between 0 and $n-1$ values.

- Typical structure of a node:

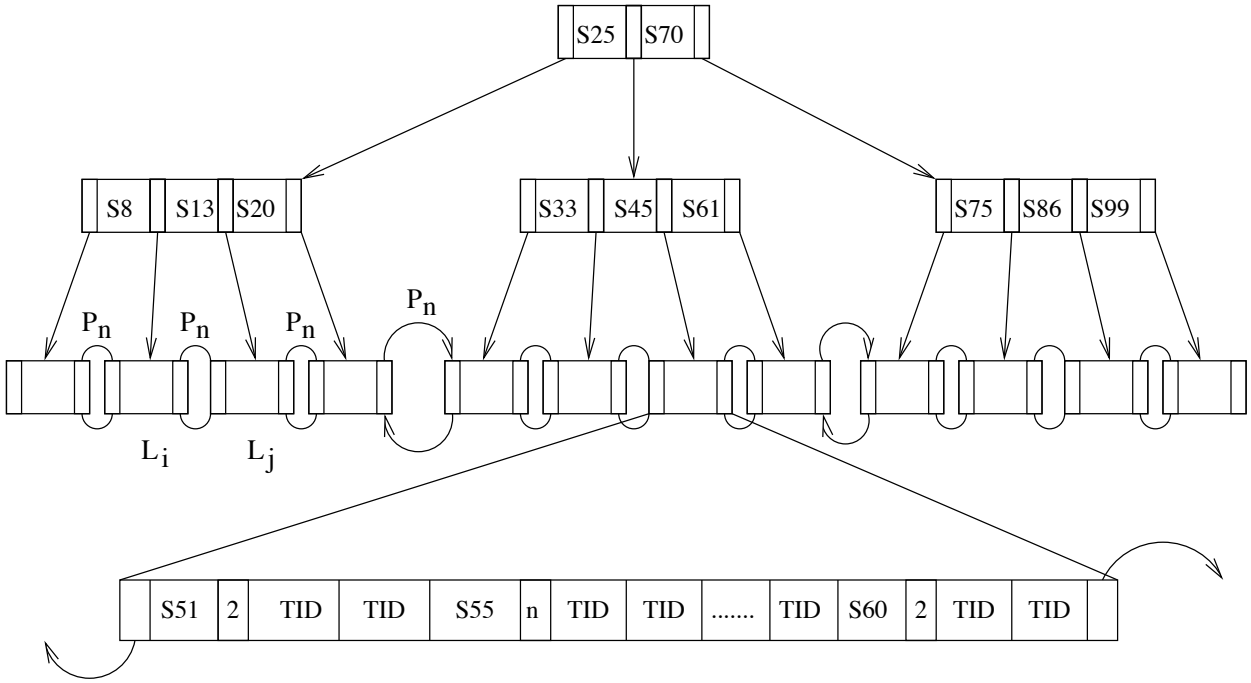


  - K_i are the search key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

- The search keys in a node are ordered, i.e,

  $K_1 < K_2 < K_3 \ldots < K_{n-1}$

Example of a B$^+$-Tree



Leaf Nodes in a B$^+$-Tree

- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search key value K_i (using the tuple identifier, tid), or to a bucket of pointers to file records, each record having search key value K_i.

  Note that one only needs bucket structure if search key does not correspond to primary key of relation the index is associated with.

- If L_i, L_j are leaf nodes and $i < j$, L_i's search key values are less than L_j's search key values.

- P_n points to next leaf node in search key order.

Non-Leaf Nodes in a B$^+$-Tree

- All the search keys in the subtree to which P_1 points are less than K_1; all search keys in the subtree to which P_m points are greater than or equal to K_{m-1}.

Observations about B$^+$-Trees

- Since the inter-node connections are done by pointers, there is no assumption that in the B$^+$-Tree logically close blocks are also "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 (logarithmic in size of the main file), thus searches can be done efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time. (☞ ECS ~~110~~) 6D

Queries on B$^+$-Trees

Find all records with a search key value of k

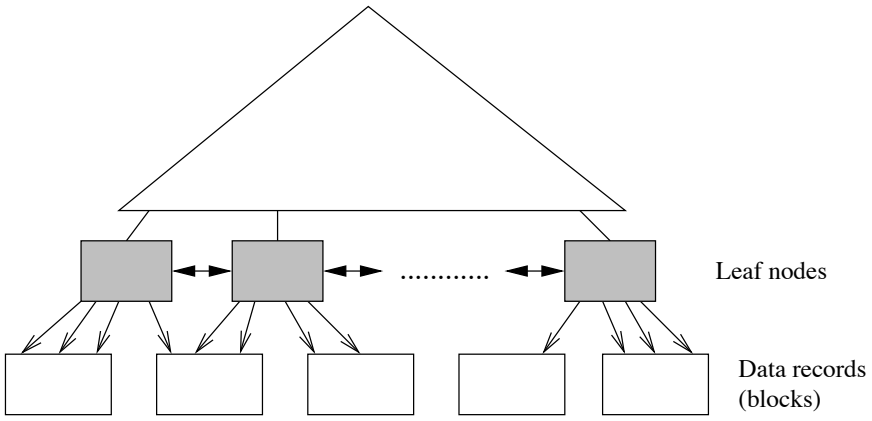
- Start with the root node
  - Examine the node for the smallest search key value $> k$.
  - If such a value exists, assume it is K_i. Then follow P_i to the child node.
  - Otherwise, $k \geq K_{m-1}$, where are m pointers in the node. Then follow P_m to the child node.
- If the node is reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
- Eventually reach a leaf node. Scan entries K_i in the leaf node. If $K_i = k$, follow pointer P_i to the desired record or bucket. Otherwise no record with search key value k exists.
- Further comments:
  - If there are V search key values in the file, the path from the root to a leaf node is no longer than $\lceil \log_{\lceil n/2 \rceil}(V) \rceil$.
  - In general a node has the same size as a disk block, typically 4KB, and $n \approx 100$ (40 bytes per index entry).
  - With $1,000,000$ search key values and $n = 100$, at most $\log_{50}(1,000,000) = 4$ nodes are accessed in the lookup!

B^+-Tree File Organization

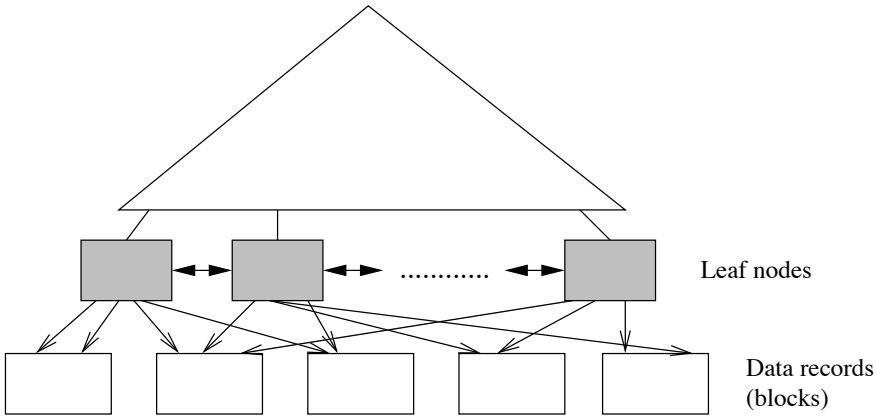
- Index file degradation problem is solved by using B^+-Tree indices. Data file degradation problem is solved by using a B^+-Tree file organization.
- Leaf nodes in a B^+-Tree file organization can store records instead of just pointers.

Clustered vs. Unclustered Indices

Clustered: Order of data records is the same as order of index entries.



Clustered Tree Index



Unclustered Tree Index

Index Definition in PostgreSQL

- Indexes are not part of SQL standard, but nearly all DBMS's support them via a syntax like the one below.
- PostgreSQL syntax:

  create [unique] index <index name> **on** <relation name> (<list of attributes>);

  drop index <index name>;

- Many more options available, including clauses to specify sort order, partial indexes, fill factor, tablespace, concurrent construction, index method, . . .

- By default, indexes are created in ascending order.

- With primary key in a relation, an index is associated.

- Information about indexes is stored in the system catalogs. Relevant tables are `pg_index` and `pg_class`.

  The system catalog table `pg_index`:

Column	Description
indexrelid	The OID of the `pg_class` entry for this index
indrelid	The OID of the `pg_class` entry for the table this index is for
indnatts	The number of columns in the index (duplicates `pg_class`.relnatts)
indisunique	If true, this is a unique index
indisprimary	If true, this index represents the primary key of the table

  . . .

- Example:

  create index city_name_idx **on** CITY(name);