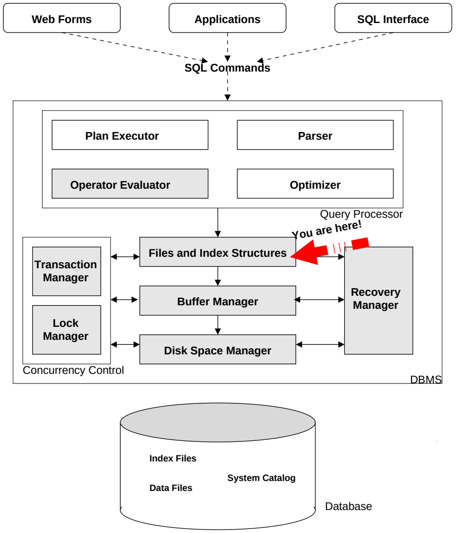
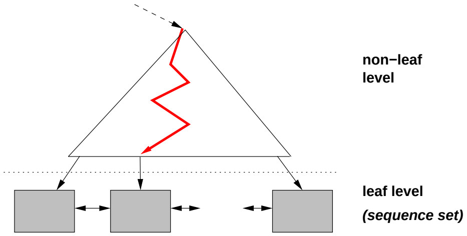
Module 4: Tree-Structured Indexing

Module Outline



4.1 B$^+$ trees

- Here we review an **index structure** which especially shines if we need to support **range selections** (and thus sorted file scans): **B$^+$ trees**.
- B$^+$ trees refine the idea underlying *binary search* on a sorted file by introducing a high fan-out, multi-level path selection mechanism.
- B$^+$ trees provide a **balanced** index structure that is resistant to data skew and automatically adapts to dynamic **inserts** and **deletes**.

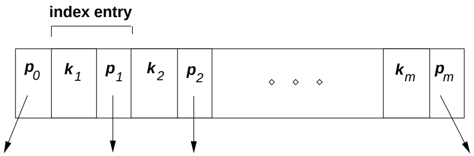


4.2 Structure of B$^+$ trees

- **Properties of B$^+$ trees:**
  - **Order:** d
  - **Occupancy:**
    - each non-leaf node holds at least d and at most $2d$ keys (exception: root may hold at least 1 key)
    - each leaf node holds between d and $2d$ index entries[1]
  - **Fan-out:** each non-leaf node holding m keys has $m + 1$ children (subtrees)
  - **Sorted order:**
    - all nodes contain entries in ascending key-order
    - child pointer $p_i (1 \leq i \leq m - 1)$ of an internal node with m keys $k_1 \dots k_m$ leads to a subtree where all keys k are $k_i \leq k < k_{i+1}$; p_0 points to a subtree with keys $k < k_1$ and p_m to a subtree with keys $k \geq k_m$
  - **Balance:** all leaf nodes are on the same level
  - **Height:** because of high fan-out, B$^+$ trees have a low height, which is $\log_F N$, for N . . . total number of index entries/records and F . . . average fan-out

[1] depending on the kind of index entries, we sometimes use a separate order d^* for leaf nodes

- Structure of non-leaf nodes:



- A leaf node entry with key value k is denoted as $k*$ as usual. Note that we can use all index entry variants ⓐ…ⓒ to implement the leaf entries:
  - for variant ⓐ, the B$^+$ tree represents the index as well as the data file itself (*i.e.*, a leaf node contains the actual data records):

$$k_i* = \Big\langle k_i, \langle \ldots \textit{data values} \ldots \rangle \Big\rangle \enspace .$$

  - for variants ⓑ and ⓒ, the B$^+$ tree lives in a file distinct from the actual data file; the p_i are (one or more) *rid*(s) pointing into the data file:

$$k_i* = \langle k_i, \mathit{rid} \rangle \quad \text{or} \quad k_i* = \langle k_i, \{\mathit{rid}\} \rangle \enspace .$$

- Leaf nodes are chained together in a doubly linked list, the so-called **sequence set**, to support range queries and sorted sequential access efficiently.

4.3 Operations on B$^+$ trees

- **Search:** Given search key k and a B$^+$ tree of height (= number of levels) h, we need h page accesses to find the index entry for k (successful search) or to determine that there is no corresponding record (unsuccessful search).

  Range selection starts the search with one of the interval boundaries and uses the sequence set to proceed.

- **Insert:** First, we find the right leaf node, insert the entry, if space is left and are done.

  If no space left, first try to move entries to neighbors, if they have space left. Otherwise, split the leaf node, redistribute entries, insert new separator one level higher.

  Split may propagate upwards, ultimately tree may grow in height by one.

- **Delete:** Search index entry and delete it.

  If underflow: first try to redistribute entries from neighbors. If they would also underflow: merge two neighbors and delete separator from one level higher.

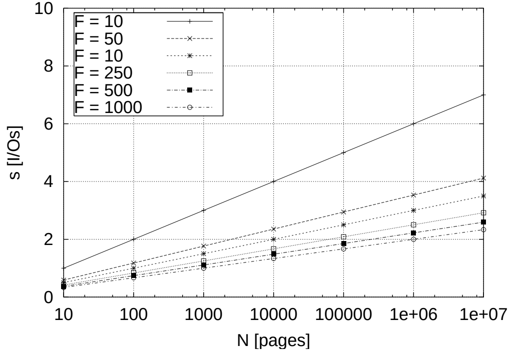
  Merge may propagate upwards, ultimately tree may shrink in height by one.

4.4 Extensions

4.4.1 Key compression

- Recall the analysis of the search I/O effort s in a B$^+$ tree for a file of N pages. The **fan-out** F plays a major role:

$$s = \log_F N \quad .$$



- It clearly pays off to invest effort and **try to maximize the fan-out** F of a given B$^+$ tree implementation.

- Index entries in inner (*i.e.*, non-leaf) B$^+$ tree nodes are pairs

$$\langle k_i, \mathit{pointer\ to\ } p_i \rangle$$

  - The representation of p_i is prescribed by the DBMS or hardware specifics, and
  - especially for key field types like `CHAR( · )` or `VARCHAR( · )`, we will have $|p_i| \ll |k_i|$.

- To **minimize the size of keys** observe that key values in inner index nodes are used only to direct traffic to the appropriate leaf page:

  During the `search` procedure, we need to find the smallest index i in each visited node such that $k_i \leq k < k_{i+1}$ and then we follow link p_i.

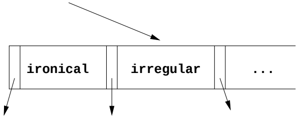
- We do *not* need the key values k_i in their entirety to use them as guides.

- Rather, we could arbitrarily chose any suitable value k_i' such that k_i' separates the values to its left from those to its right. In particular, we can chose as short a value as possible.

- For text attributes, a good choice can be **prefixes** of key values.

Example:

- To guide a search across this B$^+$ tree node

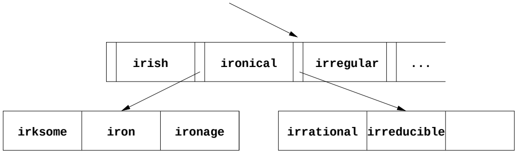


it is sufficient to store the **prefixes** `iro` and `irr`. We must preserve the B$^+$ tree semantics, though:

All index entries stored in the subtree left of `iro` *have keys* $k <$ `iro` *and index entries stored in the right subtree right of* `iro` *have keys* $k \geq$ `iro` *(and* $k <$ `irr`*).*

✎ Key prefix compression

How would a B$^+$ tree **key prefix compressor** alter the key entries in the inner node of this B$^+$ tree snippet?



4.4.2 Bulk loading a B$^+$ tree

- Consider the following database session log (this might as well be commands executed on behalf of a database transaction):

```
1  ⟨connect to database⟩
2
3  db2 => create table foo (A int, bar varchar(10))
4  DB20000I  The SQL command completed successfully.
5
6  ⟨insert 1 000 000 rows into table foo⟩
7
8  db2 => create index foo_A on foo (A asc)
```

- The SQL command in line 8 initiates 1 000 000 calls to *insert*(·)—a so-called **bulk-load**.
  - At least this is not as bad as swapping lines 6 and 8. **Why?**
  - Anyway, we are going to traverse the growing B$^+$ tree index from its root down to the leaf pages 1 000 000 times.
- Many DBMS provide a **B$^+$ tree bulk loading utility** to reduce the cost of operations like the above.

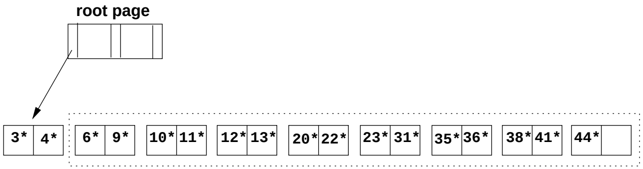
B$^+$ tree bulk-loading algorithm:

① For all records (call their keys k) in the data file, create a **sorted** list of pages of index leaf entries $k*$.

- **Note**: if we are using index entry variants ⓑ or ⓒ, this does *not* imply to sort the data file itself. (For variant ⓐ, we effectively create a clustered index.)

② Allocate an empty index root page and let its p_0 page pointer point to the first page of sorted $k*$ entries.

Example: State of bulk-load utility after step ② (order of B$^+$ tree $d = 1$):



(Index leaf pages not yet in B$^+$ tree are framed.)

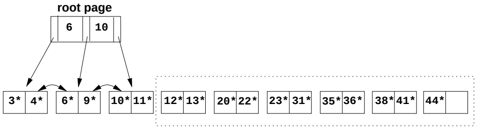
Bulk-loading continued
Can you anticipate how the bulk loading utility will proceed from this point on?

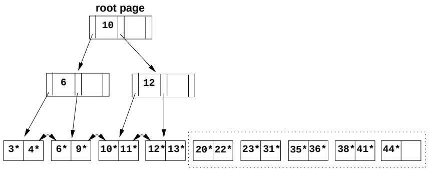
- We now use the fact that the $k*$ are **sorted**. Any insertion will thus hit the **right-most index node** (just above the leaf level).
- Use a specialized *bulk_insert* procedure that avoids B$^+$ tree root-to-leaf traversals altogether.

③ For each leaf level page p, insert the index entry $\langle$*minimum key on p*, *pointer to p*$\rangle$ into the right-most index node just above the leaf level.

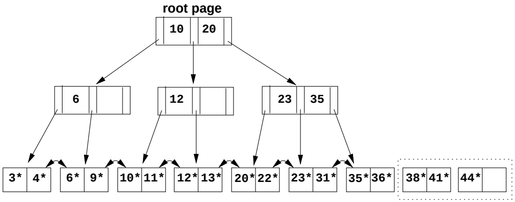
- The right-most node is filled **left-to-right**. Splits occur only on the **right-most path** from the leaf level up to the root.

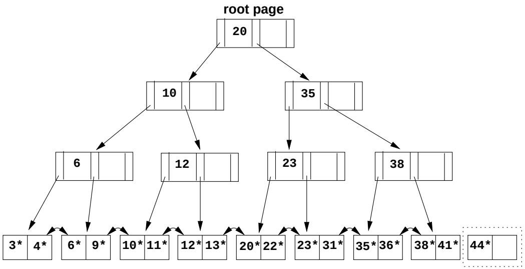
Example (continued):





Example (continued):





Observations

- Bulk-loading is more (time-) efficient this way, because tree traversals are saved.
- Furthermore, less page I/Os are necessary (or, in other words: the buffer pool is utilized more effectively).
- Finally, as seen in the example, bulk-loading is also more space-efficient: all the leaf nodes in the example have been filled up completely.

✎ **Space efficiency of bulk-loading**
How would the resulting tree in the above example look like, if you used the standard *insert*$(\cdot)$ routine on the sorted list of index entries $k*$?

- Inserting sorted data into a B$^+$ tree yields minimum occupancy of (only) d entries in all nodes.

4.4.3 A note on order

We defined B^+ trees using the concept of *order* (parameter d of a B^+ tree). This was useful for presenting the algorithms, but is hardly ever used in practical implementations, because

- key values may often be of a variable length datatype,
- duplicates may lead to variable numbers of *rid*s in an index entry $k*$ according to variant ©,
- leaf and non-leaf nodes may have different capacities due to index entries of variant ⓐ,
- key compression may introduce variable length separator values, ...

Therefore, in practice we relax the order concept and replace it with a physical space criterion, such as, each node needs to be at least half-full.

4.4.4 A note on clustered indexes

A clustered index stores the actual data records inside the index structure (variant ⓐ entries). If the index is a B$^+$ tree, splitting and merging leaf nodes *moves data records* from one page to another.

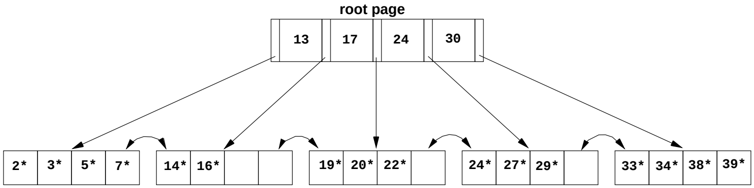
- Depending on the addressing scheme used, the *rid* of a record may change, if it is moved to another page!
- Even with the TID addressing scheme, that allows to move records within a page without any effect and that leaves proxies in case a records moves between pages, this may incur an intolerable performance overhead.
- To avoid having to update other indexes or to avoid many proxies, some systems use the search key of the clustered index as (location independent) record addresses for other, non-clustered indexes.

4.5 Generalized Access Path

A B$^+$ tree structure can also be used to index the records of multiple files at the same time, provided that those files share common attributes.

All we have to do is to use index entries that allow for two (or more) data records (variant ⓐ), *rids* (ⓑ), or *rid*-lists (ⓒ), resp'ly, one for each data file that is being indexed by this so-called "**generalized access path**".

Example: If the following B$^+$ tree would be built on top of two files, R and S,

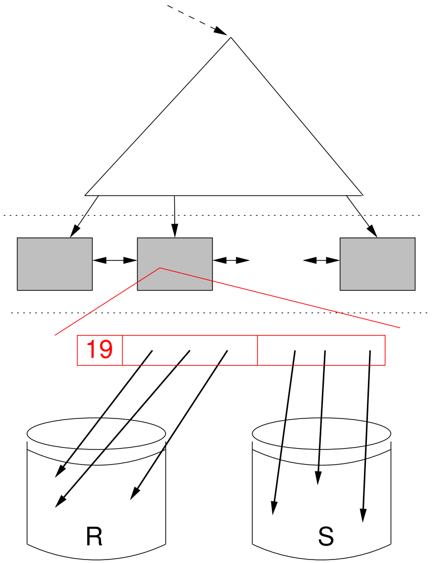


the leaf-level entries, *e.g.*, 19∗ will be of the form (using variant ⓒ here):

$$\left\langle 19, \{rid_1^R, rid_2^R, \dots\}, \{rid_1^S, rid_2^S, \dots\} \right\rangle ,$$

that is, for each key value, there would be two distinct address lists pointing to records of the two files R and S, resp'ly.

Generalized Access Path pointing into two files *R* and *S*:



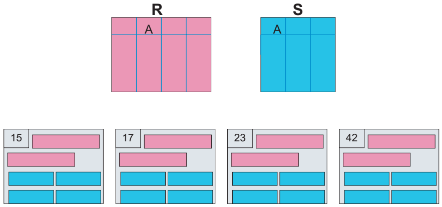
This structure has been proposed in (Härder, 1978)

4.6 ORACLE Clusters

Commercial systems sometimes also provide mechanisms to use one index for multiple data files. **ORACLE clusters** are one example of such "co-clustered" indexes. The following sequence of SQL commands (rough sketch) will create a cluster with an index and use it for the storage of records originating from both tables, `R` and `S`:

```
create cluster C ...
create index C-INDEX on C using A ...
create table R ( A, ... ) in cluster C
create table S ( A, ... ) in cluster C
```

Records from both tables will be placed into cluster pages according to their *A*-values, one page per *A*-value. *A*-values are part of the page header.



Properties of ORACLE clusters

- Selections on each table contained in the cluster are efficiently supported through the clustered index.
- Joins (equi-joins on A) between any of the clustered tables are already materialized.
- Sequential scans of any of the tables suffer from less compact storage.
- Index could also be a hash table.
- Up to 32 tables can be defined in a cluster.

Bibliography

Bayer, R. and McCreight, E. (1972). Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189.

Bayer, R. and Unterauer, K. (1977). Prefix b-trees. *ACM Transactions on Database Systems*, 2(1):11–26.

Comer, D. (1979). The ubiquituous b-tree. *ACM Computing Surveys*, 11(2):121–137.

Härder, T. (1978). Implementing a generalized access path structure for a relational database system. *ACM Transactions on Database Systems*, 3(3):285–298.

Knuth, D. E. (1973). *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

Ollmert, H. (1989). *Datenstrukturen und Datenorganisation*. Oldenbourg Verlag.

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, New York, 3 edition.

Teory, T. (1994). *Database Modelling & Design, The Fundamental Principles*. Morgan Kaufman.