Einführung in Datenbanksysteme

Prof. Dr. Ralf Möller

TUHH

Indexierung
Danksagung

• Diese Vorlesung basiert auf dem Kurs *Architecture and Implementation of Database Systems* von Jens Teubner, ETH Zürich

• Ich bedanke mich für die Bereitstellung des Materials
How could we prepare for such queries and evaluate them efficiently?

We could

1. **sort** the table on disk (in ZIPCODE order).
2. To answer queries, then use **binary search** to find first qualifying tuple, and **scan** as long as ZIPCODE < 8999.

$k*$ denotes the full data record with search key $k$. 

```
SELECT *  
FROM CUSTOMERS  
WHERE ZIPCODE BETWEEN 8800 AND 8999
```
Ordered Files and Binary Search

We get **sequential access** during the **scan phase**.

We need to read \( \log_2(\# \text{ tuples}) \) tuples during the **search phase**.

❌ We need to read about as many **pages** for this.

(The whole point of binary search is that we make far, unpredictable jumps. This largely defeats prefetching.)
ISAM—Indexed Sequential Access Method

**Idea:** Accelerate the search phase using an index.

- All nodes are the size of a page
  - hundreds of entries per page
  - large fanout, low depth
- Search effort: $\log_{\text{fanout}}(\# \text{ tuples})$

- index entry
- separator
- key

$p_0 \quad k_1 \quad p_1 \quad k_2 \quad p_2 \quad \cdots \quad k_n \quad p_n$
ISAM Index: Updates

ISAM indexes are inherently static.

- **Deletion** is not a problem: delete record from data page.
- **Inserting** data can cause more effort:
  - If space is left on respective leaf page, insert record there (e.g., after a preceding deletion).
  - Otherwise, **overflow pages** need to be added. (Note that these will violate the sequential order.)
- ISAM indexes **degrade** after some time.
Remarks

- Leaving some free space during index creation reduces the insertion problem (typically ≈ 20 % free space).
- Since ISAM indexes are static, pages need not be locked during index access.
  - Locking can be a serious bottleneck in dynamic tree indexes (particularly near the root node).
- ISAM may be the index of choice for relatively static data.
**B+*-trees: A Dynamic Index Structure**

The **B+*-tree** is derived from the ISAM index, but is fully dynamic with respect to updates.

- No overflow chains; **B+*-trees remain balanced** at all times
- Gracefully adjusts to **inserts** and **deletes**.
- Minimum occupancy for all **B+*-tree nodes** (except the root): 50% (typically: 67%).

B$^+$-trees: Basics

B$^+$-trees look like ISAM indexes, where

- leaf nodes are, generally, not in sequential order on disk,
- leaves are connected to form a double-linked list:

- leaves may contain actual data (like the ISAM index) or just references to data pages (e.g., rids). We assume the latter case in the following, since it is the more common one.
- each B$^+$-tree node contains between $d$ and $2d$ entries ($d$ is the order of the B$^+$-tree; the root is the only exception)

$^2$This is not really a B$^+$-tree requirement, but most systems implement it.
Searching a B$^+$-tree

1 Function: search ($k$)
2 return tree_search ($k, \text{root}$);

1 Function: tree_search ($k, \text{node}$)
2 if node is a leaf then
3 return node;
4 switch $k$ do
5 case $k < k_1$
6 return tree_search ($k, p_0$);
7 case $k_i \leq k < k_{i+1}$
8 return tree_search ($k, p_i$);
9 case $k_{2d} \leq k$
10 return tree_search ($k, p_{2d}$);

Function search ($k$) returns a pointer to the leaf node that contains potential hits for search key $k$. 

index entry separator key

node page layout
Insert: Overview

- The B⁺-tree needs to remain **balanced** after every update.³
  → We **cannot** create overflow pages.
- Sketch of the insertion procedure for entry \(\langle k, p \rangle\)
  (key value \(k\) pointing to data page \(p\)):
  1. **Find leaf page** \(n\) where we would expect the entry for \(k\).
  2. If \(n\) has **enough space** to hold the new entry (\(i.e., \) at most \(2d - 1\) entries in \(n\)), **simply insert** \(\langle k, p \rangle\) into \(n\).
  3. Otherwise node \(n\) must be **split** into \(n\) and \(n'\) and a new **separator** has to be inserted into the parent of \(n\).

Splitting happens recursively and may eventually lead to a split of the root node (increasing the tree height).

³\(i.e., \) every root-to-leaf path must have the same length.
Insert: Examples (Insert without Split)

\[ \begin{array}{c}
\text{8500} \\
\text{5012} \\
\text{4123} \\
\text{4450} \\
\text{4528} \\
\text{8280} \\
\text{8404} \\
\text{8500} \\
\text{8570} \\
\text{8608} \\
\text{8887} \\
\text{9016} \\
\text{9200} \\
\end{array} \]

\[ \begin{array}{c}
\text{node 0} \\
\text{node 1} \\
\text{node 3} \\
\text{node 4} \\
\text{node 5} \\
\text{node 6} \\
\text{node 7} \\
\text{node 8} \\
\end{array} \]

\[ \begin{array}{c}
\text{\cdots pointers to data pages\cdots} \\
\end{array} \]

Insert new entry with key 4222.

\[ \begin{array}{c}
\rightarrow \text{Enough space in node 3, simply insert.} \\
\rightarrow \text{Keep entries sorted within nodes.} \\
\end{array} \]
Insert new entry with key 4222.

→ Enough space in node 3, simply insert.

→ Keep entries sorted within nodes.
Insert: Examples (Insert without Split)

... pointers to data pages ...

Insert new entry with key 6330.
Insert: Examples (Insert with Leaf Split)

Insert key 6330.

→ Must **split** node 4.

→ **New separator** goes into node 1 (including pointer to new page).
Insert: Examples (Insert with Inner Node Split)

After 8180, 8245, insert key 4104.

→ Must split node 3.
→ Node 1 overflows → split it
→ New separator goes into root

Unlike during leaf split, separator key does not remain in inner node. 🧐 Why?
Insert: Root Node Split

- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied.
- Eventually, this can lead to a split of the root node:
  - Split like any other inner node.
  - Use the separator to create a new root.
- The root node is the only node that may have an occupancy of less than 50%.
- This is the only situation where the tree height increases.

💡 How often do you expect a root split to happen?
Insertion Algorithm

1 Function: tree_insert \((k, \text{rid}, \text{node})\)

2 if node is a leaf then

3 \hspace{1em} return leaf_insert \((k, \text{rid}, \text{node})\);

4 else

5 \hspace{1em} switch \(k\) do

6 \hspace{2em} case \(k < k_1\)

7 \hspace{3em} \langle \text{sep}, \text{ptr} \rangle \leftarrow \text{tree_insert} \((k, \text{rid}, p_0)\);

8 \hspace{2em} case \(k_i \leq k < k_{i+1}\)

9 \hspace{3em} \langle \text{sep}, \text{ptr} \rangle \leftarrow \text{tree_insert} \((k, \text{rid}, p_i)\);

10 \hspace{2em} case \(k_{2d} \leq k\)

11 \hspace{3em} \langle \text{sep}, \text{ptr} \rangle \leftarrow \text{tree_insert} \((k, \text{rid}, p_{2d})\);

12 if \(sep\) is null then

13 \hspace{1em} return \langle \text{null}, \text{null} \rangle;

14 else

15 \hspace{1em} return split \((sep, \text{ptr}, \text{node})\);
Function: leaf_insert \((k, rid, node)\)

1. if another entry fits into node then
   1.1. insert \(\langle k, rid \rangle\) into node ;
   1.2. return \(\langle \text{null, null} \rangle\);

2. else
   2.1. allocate new leaf page \(p\);
   2.2. take \(\{\langle k^+_1, p^+_1 \rangle, \ldots, \langle k^+_{2d+1}, p^+_{2d+1} \rangle\} := \text{entries from node} \cup \{\langle k, \text{ptr} \rangle\}\)
   2.3. leave entries \(\langle k^+_1, p^+_1 \rangle, \ldots, \langle k^+_d, p^+_d \rangle\) in node ;
   2.4. move entries \(\langle k^+_d+1, p^+_{d+1} \rangle, \ldots, \langle k^+_{2d}, p^+_{2d} \rangle\) to \(p\);
   2.5. return \(\langle k^+_{d+1}, p \rangle\);

Function: split \((k, ptr, node)\)

1. if another entry fits into node then
   1.1. insert \(\langle k, ptr \rangle\) into node ;
   1.2. return \(\langle \text{null, null} \rangle\);

2. else
   2.1. allocate new leaf page \(p\);
   2.2. take \(\{\langle k^+_1, p^+_1 \rangle, \ldots, \langle k^+_{2d+1}, p^+_{2d+1} \rangle\} := \text{entries from node} \cup \{\langle k, \text{ptr} \rangle\}\)
   2.3. leave entries \(\langle k^+_1, p^+_1 \rangle, \ldots, \langle k^+_d, p^+_d \rangle\) in node ;
   2.4. move entries \(\langle k^+_d+2, p^+_{d+1} \rangle, \ldots, \langle k^+_{2d}, p^+_{2d} \rangle\) to \(p\);
   2.5. set \(p_0 \leftarrow p^+_{d+1} \) in node;
   2.6. return \(\langle k^+_{d+1}, p \rangle\);
Insertion Algorithm

1 Function: insert \( (k, rid) \)
2 \((key, ptr) \leftarrow \text{tree_insert} (k, rid, root);\)
3 if key is not null then
4 allocate new root page \(r;\)
5 populate \(n\) with
6 \(p_0 \leftarrow root;\)
7 \(k_1 \leftarrow key;\)
8 \(p_1 \leftarrow ptr;\)
9 \(root \leftarrow r;\)

- insert \((k, rid)\) is called from outside.
- Note how leaf node entries point to rids, while inner nodes contain pointers to other \(B^+\)-tree nodes.
Deletion

- If a node is sufficiently full (i.e., contains at least $d + 1$ entries, we may simply remove the entry from the node.
  - Note: Afterward, **inner nodes** may contain keys that no longer exist in the database. This is perfectly legal.
- **Merge** nodes in case of an **underflow** (“undo a split”):
  - “Pull” separator into merged node.
Deletion

It’s not quite that easy...

- Merging only works if **two** neighboring nodes were 50% full.
- Otherwise, we have to **re-distribute**:
  - “rotate” entry through parent
B⁺-trees in Real Systems

- Actual systems often avoid the cost of merging and/or redistribution, but relax the minimum occupancy rule.

- *E.g.*, IBM DB2 UDB:
  - The MINPCTUSED parameter controls when the system should try a leaf node merge ("on-line index reorg").
    (This is particularly simple because of the pointers between adjacent leaf nodes, slide 54.)
  - Inner nodes are never merged
    (→ need to do full table reorg for that).

- To improve *concurrency*, systems sometimes only mark index entries as deleted and physically remove them later
  (*e.g.*, IBM DB2 UDB "type-2 indexes")
What’s Stored Inside the Leaves?

Basically three alternatives:

1. The **full data entry** $k\star$.
   (Such an index is inherently **clustered**. See next slides.)
2. A $\langle k, \text{rid} \rangle$ pair, where $\text{rid}$ is the **record id** of the data entry.
3. A $\langle k, \{\text{rid}_1, \text{rid}_2, \ldots \} \rangle$ pair. The items in the **rid list** $\text{rid}_i$ are record ids of data entries with search key value $k$.

Options 2 and 3 are reasons why want record ids to be **stable**.

$\rightarrow$ slides 42 ff.

Alternative 2 seems to be the most common one.
**B**⁺-trees and Sorting

A typical situation according to alternative 2 looks like this:

What are the implications when we want to execute

```
SELECT * FROM CUSTOMERS ORDER BY ZIPCODE
```

- "Random" access to data pages when we scan the B⁺-tree.
- Page I/Os needed: \( \approx \) number of tuples in \( \text{CUSTOMERS} \).
- For comparison: Using external sorting, we could sort the entire file with 3–5 sequential file reads.
Clustered B⁺-trees

If the data file was sorted, the scenario would look different:

We call such an index a clustered index.

- Scanning the index now leads to sequential access.
- This is particularly good for range queries.

Why don’t we make all indexes clustered?
Index Organized Tables

Alternative 1 (slide 67) is a special case of a clustered index.

- index file $\equiv$ data file
- Such a file is often called an index organized table.

E.g., Oracle8i

```sql
CREATE TABLE (...) 
    ..., 
    PRIMARY KEY ( ... )

    ORGANIZATION INDEX;
```
Prefix and Suffix Truncation

$B^+$-tree fanout is proportional to the number of index entries per page, i.e., inversely proportional to the key size.

→ Reduce key size, particularly for variable-length strings.

Prefix:

$\text{Goofy}$

$\text{Daisy Duck}$

$\text{Dagobert Duck}$

$\text{Mickey Mouse}$

$\text{Mini Mouse}$

Suffix:

Suffix truncation: Make separator keys only as long as necessary:

$\text{Dagobert Duck}$

$\text{Daisy Duck}$

$\text{Goofy}$

$\text{Mickey Mouse}$

$\text{Mini Mouse}$

Note that separators need not be actual data values.
Prefix Truncation

Keys within a node often share a common prefix.

Prefix truncation:

▶ Store common prefix only once \( (e.g., \text{as } "k_0") \).
▶ Keys have become highly discriminative now.

Violating the “50 % occupation” rule can help improve the effectiveness of prefix truncation.

\( \uparrow \text{R. Bayer, K. Unterauer: Prefix B-Trees. ACM TODS 2(1), March 1977} \)
Composite Keys

B⁺-trees can (in theory⁴) be used to index everything with a defined total order, e.g.:

- integers, strings, dates, …, and
- concatenations thereof (based on lexicographical order).

E.g., in most SQL dialects:

```
CREATE INDEX ON TABLE CUSTOMERS (LASTNAME, FIRSTNAME);
```

A useful application are, e.g., partitioned B-trees:

- Leading index attributes effectively partition the resulting B⁺-tree.


⁴Some implementations won’t allow you to index, e.g., large character fields.
Bulk-Loading $B^+$-trees

Building a $B^+$-tree is particularly easy when the input is sorted.

- Build $B^+$-tree **bottom-up** and **left-to-right**.
- Create a parent for every $2d + 1$ unparented nodes. (Actual implementations typically leave some space for future updates. e.g., DB2’s PCTFREE parameter)

✍️ What use cases could you think of for bulk-loading?
Stars, Pluses, …

In the foregoing we described the $\mathbf{B^+}$-tree.

Bayer and McCreight originally proposed the $\mathbf{B}$-tree:

▶ Inner nodes contain data entries, too. ☝️ Pros/cons?

There is also a $\mathbf{B^*}$-tree:

▶ Keep non-root nodes at least $2/3$ full (instead of $1/2$).
▶ Need to redistribute on inserts to achieve this.
(Whenever two nodes are full, split them into three.)

Most people say “B-tree” and mean any of these variations. Real systems typically implement $\mathbf{B^+}$-trees.

“B-trees” are also used outside the database domain, e.g., in modern file systems (ReiserFS, HFS, NTFS, …).
B⁺-trees are by far the predominant type of indices in databases. An alternative is hash-based indexing.

- Hash indices can only be used to answer equality predicates.
- Particularly good for strings (even for very long ones).
Dynamic Hashing

Problem: How do we choose $n$ (the number of buckets)?

- $n$ too large → space wasted, poor space locality
- $n$ too small → many overflow pages, degrades to linked list

Database systems, therefore, use **dynamic hashing** techniques:

- extendible hashing,
- linear hashing.

Few systems support true hash indices (e.g., PostgreSQL).

More popular uses of hashing are:

- support for $\mathbf{B}^+$-trees over hash values (e.g., SQL Server)
- the use of hashing during query processing → hash join.
Recap

Indexed Sequential Access Method (ISAM)
A static, tree-based index structure.

B$^+$-trees
The database index structure; indexing based on any kind of (linear) order; adapts dynamically to inserts and deletes; low tree heights ($\sim 3–4$) guarantee fast lookups.

Clustered vs. Unclustered Indices
An index is clustered if its underlying data pages are ordered according to the index; fast sequential access for clustered B$^+$-trees.

Hash-Based Indices
Extendible hashing and linear hashing adapt dynamically to the number of data entries.