Architecture and Implementation of Database Systems (Winter 2013/14)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2013/14
Part III

Indexing
**SELECT** * 
**FROM** CUSTOMERS 
**WHERE** ZIPCODE BETWEEN 8800 AND 8999 

**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.

\[ k^* \text{ denotes the full data record with search key } k. \]
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.)
Observations:
- Make rather **far jumps initially**.
  → For each step read **full page**, but inspect only **one record**.
- After $\mathcal{O}(\log_2 \text{pagesize})$, search stays **within one page**.
  → I/O cost is used much more efficiently here.
Idea: “Cache” those records that might be needed for the first phase.

→ If we can keep the cache in memory, we can find any record with just a single I/O.

Is this assumption reasonable?
What if my data set is really large?

- “Cache” will span many pages, too.
  (In practice, we'll organize the cache just like any other database object.)
- Thus: “cache the cache” $\rightarrow$ hierarchical “cache”
**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: $\log_{\text{fanout}}(\#\text{ tuples})$
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 \( \approx 20\% \) 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.
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**:\(^2\)

leaves may contain **actual data** (like the ISAM index) or just **references** to data pages (**e.g.**, rids). ↗ slides 76 and 79

- 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 some systems implement it.
Searching a B\textsuperscript{+}-tree

1 **Function:** `search(k)`
2 `return tree_search(k, root);`

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

1 **Function:** `tree_search(k, 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 ≤ k < k_{i+1}`
8    `  return tree_search(k, p_i);`
9  **case** `k_{2d} ≤ k`
10 `  return tree_search(k, p_{2d});`

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 new entry with key **4222**.

→ Enough space in node 3, simply insert.

→ Keep entries **sorted within nodes**.
Insert key 6330.

→ Must **split** node 4.

→ **New separator** goes into node 1 (including pointer to new page).
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 \((k, rid, node)\)

2 if node is a leaf then
3 \[\text{return leaf_insert}(k, rid, node);\]

4 else

5 switch \(k\) do
6 \[\text{case } k < k_1\]
7 \[\langle sep, ptr \rangle \leftarrow \text{tree_insert}(k, rid, p_0);\]
8 \[\text{case } k_i \leq k < k_{i+1}\]
9 \[\langle sep, ptr \rangle \leftarrow \text{tree_insert}(k, rid, p_i);\]
10 \[\text{case } k_{2d} \leq k\]
11 \[\langle sep, ptr \rangle \leftarrow \text{tree_insert}(k, rid, p_{2d});\]

12 if \(sep\) is null then
13 \[\text{return } \langle \text{null}, \text{null} \rangle;\]
14 else
15 \[\text{return } \text{split}(sep, ptr, node);\]

© Jens Teubner · Architectur & Implementation of DBMS · Winter 2013/14 70
Function: leaf_insert \((k, rid, node)\)

if another entry fits into \(node\) then

insert \(\langle k, rid \rangle\) into \(node\);
return \(\langle null, null \rangle\);

else

allocate new leaf page \(p\);

\[
\text{take} \ \{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} := \text{entries from} \ \node \cup \ \{ \langle k, ptr \rangle \} \\
\text{leave entries} \ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle \ \text{in} \ \node; \\
\text{move entries} \ \langle k_{d+1}^+, p_{d+1}^+ \rangle, \ldots, \langle k_{2d}^+, p_{2d}^+ \rangle \ \text{to} \ \node; \\
\text{return} \ \langle k_{d+1}^+, p^+ \rangle;
\]

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

if another entry fits into \(node\) then

insert \(\langle k, ptr \rangle\) into \(node\);
return \(\langle null, null \rangle\);

else

allocate new leaf page \(p\);

\[
\text{take} \ \{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} := \text{entries from} \ \node \cup \ \{ \langle k, ptr \rangle \} \\
\text{leave entries} \ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle \ \text{in} \ \node; \\
\text{move entries} \ \langle k_{d+2}^+, p_{d+1}^+ \rangle, \ldots, \langle k_{2d}^+, p_{2d}^+ \rangle \ \text{to} \ \node; \\
\text{set} \ p_0 \leftarrow p_{d+1}^+ \ \text{in} \ \node; \\
\text{return} \ \langle k_{d+1}^+, p^+ \rangle;
\]
Insertion Algorithm

1 Function: insert \((k, rid)\)

2 \langle key, ptr \rangle \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 \hspace{1cm} \hspace{1cm} p_0 \leftarrow root;

7 \hspace{1cm} \hspace{1cm} k_1 \leftarrow key;

8 \hspace{1cm} \hspace{1cm} p_1 \leftarrow ptr;

9 \hspace{1cm} \hspace{1cm} root \leftarrow r ;

- \text{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"): 

```
4222
5012
6423
8280
3460
6423
8500
4222
5012
6423
8280
3460
8500
```

- "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
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 63.)

- 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** $k^*$. (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**.

→ slides 45 ff.

Alternative **2** seems to be the most common one.
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
```

?
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?**
Alternative 1 (slide 76) is a special case of a clustered index.

- index file ≡ data file
- Such a file is often called an **index organized table**.

E.g., Oracle8i

```sql
CREATE TABLE ( ... 
    ..., 
    PRIMARY KEY ( ... ))
ORGANIZATION INDEX;
```
### Address book of Berlin, anno 1858:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Popławsky, E.</td>
<td>Drehäuer in Horn und Holz, Schornsteinfegergasse 1.</td>
</tr>
<tr>
<td>Popławsky, M.</td>
<td>Wirthbrunn, Liepzig, Straße 43.</td>
</tr>
<tr>
<td>Popowitsch, A.</td>
<td>Kürschner, Ziegelstraße 11, 12.</td>
</tr>
<tr>
<td>Popowska, M.</td>
<td>Haushofmeisterin, Reichenstr. 43.</td>
</tr>
<tr>
<td>Popp, C.</td>
<td>Schlosser, Schlesische 7.</td>
</tr>
<tr>
<td></td>
<td>M. Schneider, Kommandantenstr. 31.</td>
</tr>
<tr>
<td></td>
<td>A. C. Seidenwirker, Prinzen-Allee 74.</td>
</tr>
<tr>
<td>Poppe, C. T.</td>
<td>Diätar, R. Wilhelmstraße 92.</td>
</tr>
<tr>
<td></td>
<td>W. Cafetier, Schüttergasse 1.</td>
</tr>
<tr>
<td></td>
<td>E. Holz- und Hornbrecher, Wallstraße 54.</td>
</tr>
<tr>
<td></td>
<td>Eduard, Filz- und Filzschuhhandel, Borsbucher, Oberhöfen, Plüschband und aller Arten Filzwaren, Friedrichstr. 109. E.</td>
</tr>
<tr>
<td>Porath, H.</td>
<td>Oberseweermann, Landwehrstraße 3.</td>
</tr>
<tr>
<td></td>
<td>W. Gärtners und Blumenhändler, Dranenburgerstraße 57.</td>
</tr>
<tr>
<td></td>
<td>F. Buchmacher, Weberstraße 34.</td>
</tr>
<tr>
<td>Parawsky, G.</td>
<td>Eisenbahn-Zugführer, Lübbenau 12.</td>
</tr>
<tr>
<td>Porepp, C.</td>
<td>Zimmer-Berliner, Unter den Linden 47.</td>
</tr>
<tr>
<td>Pormerter, F. W.</td>
<td>Buchdruckerei, Kommandantenstr. 7.</td>
</tr>
<tr>
<td>Porsch, H.</td>
<td>Kunstfahler, Seeligstrasse 25.</td>
</tr>
<tr>
<td>Porschien, C.</td>
<td>Schneider, Friedgracht 59.</td>
</tr>
<tr>
<td>Pott, C.</td>
<td>Tapesrier, Fruchtstraße 58.</td>
</tr>
<tr>
<td>Pottfett, H.</td>
<td>Polizei - Wachtmeister, Charlottenstr. 37.</td>
</tr>
<tr>
<td>Porth, C.</td>
<td>Hoffhäuser, Friedrichstraße 195.</td>
</tr>
<tr>
<td></td>
<td>S. Tischler, Markgrafstrasse 18.</td>
</tr>
<tr>
<td>Porthum, G.</td>
<td>Klempner, Rosenstr. 8.</td>
</tr>
<tr>
<td>Portier, L.</td>
<td>Dicke, Linienstr. 18.</td>
</tr>
<tr>
<td>Possart, F.</td>
<td>Intendantur - Applicant, Schumannstr. 9.</td>
</tr>
<tr>
<td></td>
<td>E. Kaufmann, Inhaber des landwirtschaftlichen Etablissements, Seeligstr. 3. F. Eugen Possart. C.</td>
</tr>
<tr>
<td></td>
<td>J. C. Kaufmann, Schumannstr. 9.</td>
</tr>
<tr>
<td>Posse, C.</td>
<td>Barbier, Mauerstraße 33.</td>
</tr>
<tr>
<td>Pospel, F.</td>
<td>Handelsmann, Dresdnerstraße 97.</td>
</tr>
<tr>
<td></td>
<td>F. Muth, Lindenstr. 56.</td>
</tr>
<tr>
<td>Possfeldt, H.</td>
<td>Kanzleidienst, Leipzigerstraße 5.</td>
</tr>
<tr>
<td></td>
<td>E. Mode, Stralsunder Str. 4.</td>
</tr>
<tr>
<td></td>
<td>E. Vorzeilmaler, Alte Jakobsstraße 60.</td>
</tr>
<tr>
<td></td>
<td>C. Koch und Restaurantier, Charitéstrasse 5.</td>
</tr>
<tr>
<td></td>
<td>L. Restaurantier, Mittelstr. 57.</td>
</tr>
</tbody>
</table>

---

4 http://adressbuch.zlb.de/
Prefix Truncation

Address book:

- To save space, common last names are printed only once.

Such **prefix truncation** can also be applied to B-trees:\(^5\)

<table>
<thead>
<tr>
<th>Prefix: Smith, J</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack</td>
</tr>
</tbody>
</table>

The advantage is **two-fold**:

1. save space → more keys fit on one page → higher fanout
2. need **fewer comparisons**

---

**Suffix Truncation**

- **Prefix truncation** is most effective near or in leaf pages. Why?
- Elsewhere, by contrast, the leading key parts are most discriminative.
- In fact, a key’s suffix might not be needed to guide navigation at all.

This motivates **suffix truncation**:

- Store keys only as far as needed to guide search.
- Remember: key values in inner tree nodes do **not** have to be contained in the actual data set.
Suffix Truncation

Example:

- Daniel
- David
- Jason
- Jennifer
- John
- Marc
- Michael
- Oliver
- Paul
- Peter
- Rachel
- Robert
- Ron
- Steve
- Thomas
Suffix truncation beyond the bottom-most level is difficult/dangerous.

→ Shortening ‘Pe’ to ‘P’ would be incorrect!
Composite Keys

B⁺-trees can (in theory⁶) 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.


⁶Some implementations won’t allow you to index, e.g., large character fields.
CREATE INDEX ON TABLE STUDENTS (SEMESTER, ZIPCODE);

What types of queries could this index support?
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?**
In the foregoing we described the **B⁺-tree**.

Bayer and McCreight originally proposed the **B-tree**:
- Inner nodes contain data entries, too. 📜 **Pros/cons?**

There is also a **B*-tree**:
- Keep non-root nodes at least \( \frac{2}{3} \) full (instead of \( \frac{1}{2} \)).
- 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 **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 \( \rightarrow \) space wasted, poor space locality
- \( n \) too small \( \rightarrow \) 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 \( \rightarrow \) **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.