Architecture and Implementation of Database Systems (Summer 2018)

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

Summer 2018

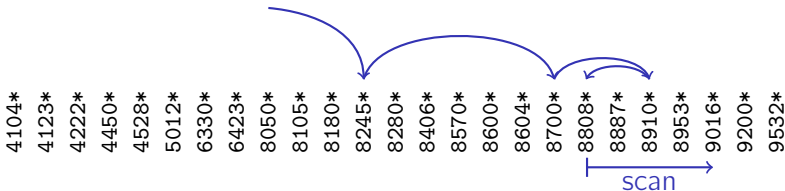# Part III

Indexing

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

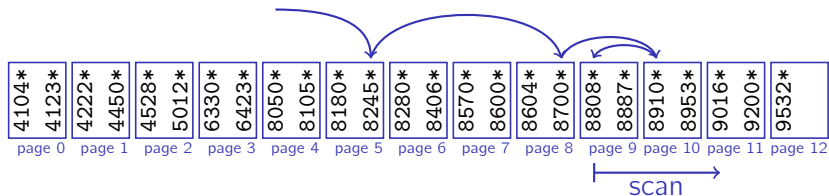**How could we prepare for such queries and evaluate them efficiently?**

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.

4104* 4123* 4222* 4450* 4528* 5012* 6330* 6423* 8050* 8105* 8180* 8245* 8280* 8406* 8570* 8600* 8604* 8700* 8808* 8887* 8910* 8953* 9016* 9200* 9532*

scan

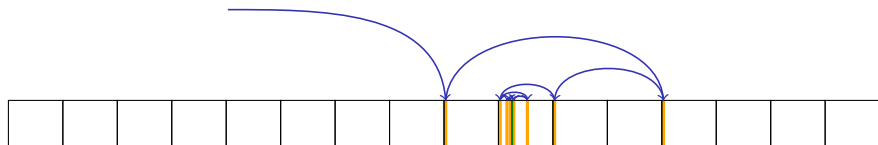$k*$ denotes the full data record with search key k.

Ordered Files and Binary Search

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

We need to read $\log_2(\# \textit{tuples})$ 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.)
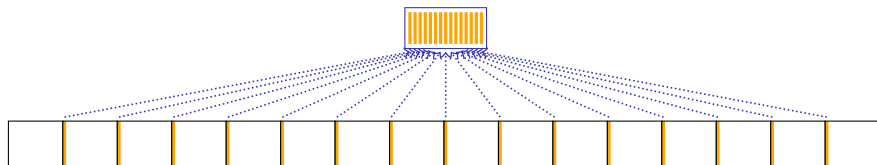
Observations:

- Make rather **far jumps initially**.
  - $\rightarrow$ For each step read **full page**, but inspect only **one record**.
- After $\mathcal{O}\left(\log_2 \mathit{pagesize}\right)$, search stays **within one page**.
  - $\rightarrow$ I/O cost is used much more efficiently here.

Binary Search and Database Pages

**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**.

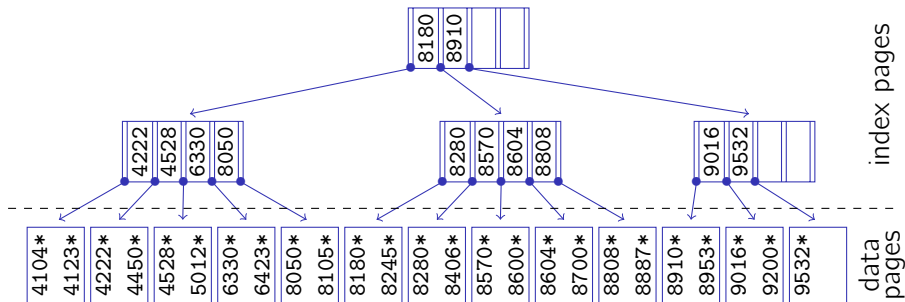✎ **Is this assumption reasonable?**

Large Data

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"** → hierarchical "cache"

inner nodes
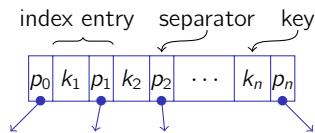
leave nodes (here: data pages)
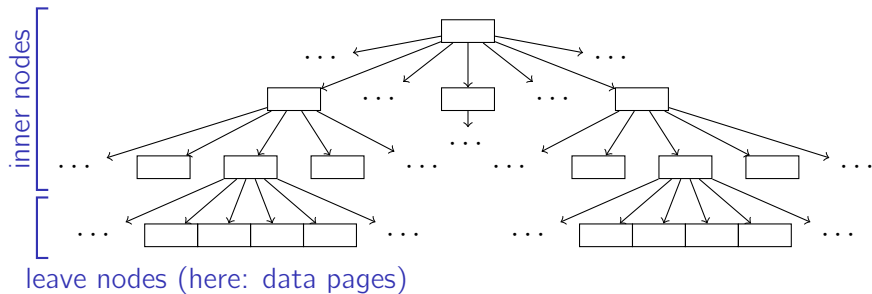
ISAM—Indexed Sequential Access Method

**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: $\log_{fanout}(\# \, tuples)$

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.

overflow pages

Remarks
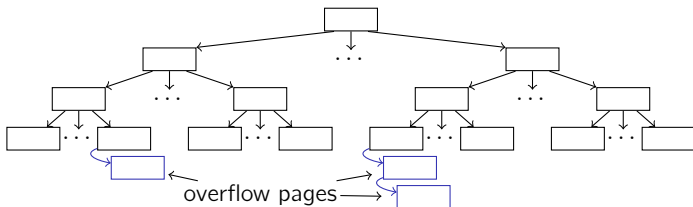
- Leaving some free space during index creation reduces the insertion problem (typically $\approx 20\,\%$ free space).
- Since ISAM indexes are static, pages need not be **locked** (database jargon: "latched") during index access.
    - Latching can be a serious bottleneck in dynamic tree indexes (particularly 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 %).

- Original version: **B-tree**: R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, vol. 1, no. 3, September 1972.

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]
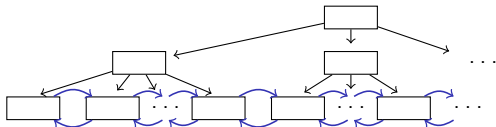
- leaves may contain **actual data** (like the ISAM index)
  or just **references** to data pages (*e.g.*, rids). ↗ slides 80 and 86
  - We assume the **latter** case in the following, since it is the more common one.
- each B$^+$-tree node contains between d and $2d$ entries (d is the **order** of the 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$^+$-tree

1 **Function:** search (k)

2 **return** tree_search (k, *root*);

_____

1 **Function:** tree_search (k, *node*)

2 **if** *node* is a leaf **then**

3   **return** *node*;

4 **switch** k **do**

5   **case** $k < k_1$ **do**

6   **return**
       tree_search (k, p_0);

7   **case** $k_i \leq k < k_{i+1}$ **do**

8   **return**
       tree_search (k, p_i);

9   **case** $k_{2d} \leq k$ **do**

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.

node page layout

Insert: Overview
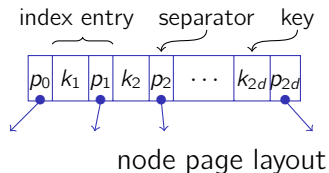
- The B$^+$-tree needs to remain **balanced** after every update.[3]
  - $\rightarrow$ We **cannot** create overflow pages.
- Sketch of the insertion procedure for entry $\langle k, p \rangle$
  (key value k pointing to data page p):

  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.

··· pointers to data pages ···
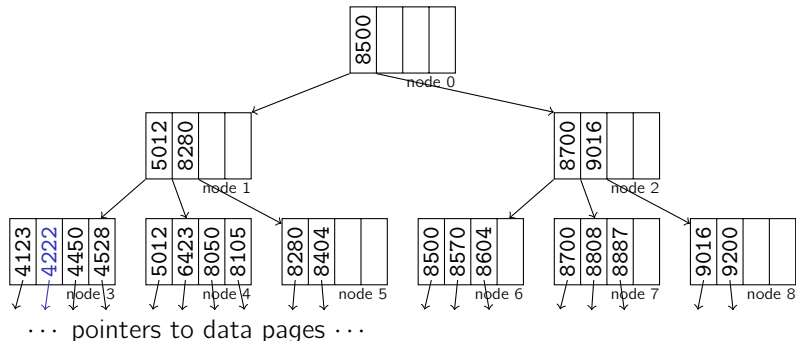
Insert new entry with key 4222.

→ Enough space in node 3, simply insert.

→ Keep entries **sorted within nodes**.

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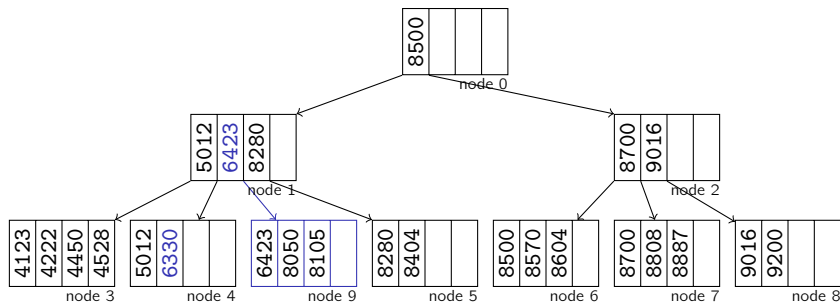
Insert key 6330.

→ Must **split** node 4.

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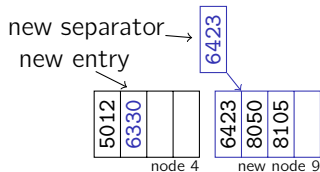
Insert: Examples (Insert with Inner Node Split)

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?**

new separator
from leaf split
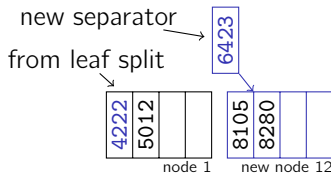
Insert: Root Node Split
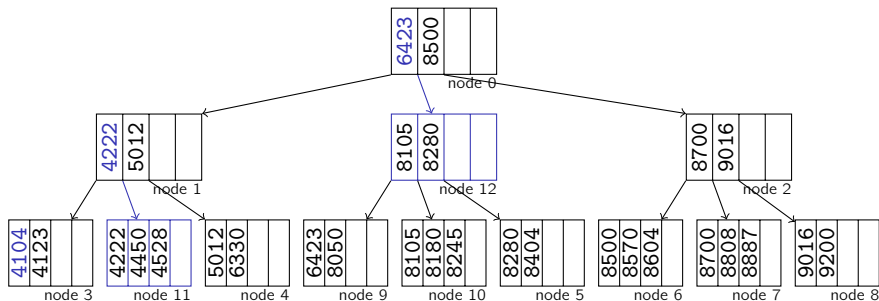
- Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied.
- Eventually, this can lead to a split of the **root node**:
  - Split like any other inner node.
  - Use the separator to create a **new root**.
- The root node is the **only** node that may have an 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 | **return** `leaf_insert` (k, rid, $node$);

4 **else**

5 | **switch** k **do**

6 | | **case** $k < k_1$ **do**

7 | | | $\langle sep, ptr \rangle \leftarrow$ `tree_insert` (k, rid, p_0);

8 | | **case** $k_i \leq k < k_{i+1}$ **do**

9 | | | $\langle sep, ptr \rangle \leftarrow$ `tree_insert` (k, rid, p_i);

                                                              see `tree_search` ()

10 | | **case** $k_{2d} \leq k$ **do**

11 | | | $\langle sep, ptr \rangle \leftarrow$ `tree_insert` (k, rid, p_{2d});

12 | **if** sep is **null then**

13 | | **return** $\langle$**null**, **null**$\rangle$;

14 | **else**

15 | | **return** `split` (sep, ptr, $node$);

1 **Function:** leaf_insert $(k, rid, node)$

2 **if** another entry fits into *node* **then**

3   insert $\langle k, rid \rangle$ into *node* ;

4   **return** $\langle \textbf{null}, \textbf{null} \rangle$;

5 **else**

6   allocate new leaf page p ;

7   **take** $\left\{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \right\} :=$ entries from *node* $\cup \; \{\langle k, ptr \rangle\}$

8   leave entries $\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle$ in *node* ;

9   move entries $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$ to p ;

10   **return** $\langle k_{d+1}^+, p \rangle$;

---

1 **Function:** split $(k, ptr, node)$

2 **if** another entry fits into *node* **then**

3   insert $\langle k, ptr \rangle$ into *node* ;

4   **return** $\langle \textbf{null}, \textbf{null} \rangle$;

5 **else**

6   allocate new leaf page p ;

7   **take** $\left\{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \right\} :=$ entries from *node* $\cup \; \{\langle k, ptr \rangle\}$

8   leave entries $\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle$ in *node* ;

9   move entries $\langle k_{d+2}^+, p_{d+2}^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$ to p ;

10   set $p_0 \leftarrow p_{d+1}^+$ in *node*;

11   **return** $\langle k_{d+1}^+, p \rangle$;

Insertion Algorithm

1 **Function:** `insert` (k, rid)
2 $\langle key, ptr \rangle \leftarrow$ `tree_insert` (k, rid, $root$);
3 **if** key is not **null then**
4   allocate new root page r;
5   **populate** r **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.

Index Pages

B-trees use slotted pages, too.

Inner Nodes:

- record ≡ ⟨*key*, *childPage*⟩ pairs.

- Additional key value to hold extra child pointer
    - *e.g.*, key value from reference in parent
    - "dummy key" for far-left or far-right end

- Similar to leaves, ⟨*key*, *childPage-list*⟩ might make sense, too.

Index Pages

Leaf Nodes: Three options:

1. Store full data records in B-tree leaf
    - $\rightarrow$ B-tree becomes a method to physically organize the table's data pages.
    - $\rightarrow$ "clustered index" or "index-organized table"

2. record $\equiv \langle key, \textit{rid-list} \rangle$
    - $\rightarrow$ There could be more than one tuple for same key.

3. record $\equiv \langle key, rid \rangle$
    - $\rightarrow$ Easier when keys are unique.   ✎ **Why?**

Options 2 and 3 are reasons why we want record ids to be **stable**.
  $\rightarrow$ slides 51 ff.

E.g., index on `VARCHAR` field with random content:

```
Hi Key 0:
   Offset Location = 668  (x29C)
   Record Length   = 455  (x1C7)
   Key Part 1:
       Variable Length Character String
       Actual Length = 0
   Child Pointer  =>  Page 24694
   Table RID: x(0000 03C6 0027) r(000003C6;0027) d(966;39)
   Child Pointer  =>  Page 24695
   Table RID: x(0000 0514 0018) r(00000514;0018) d(1300;24)
   ...
Hi Key 1:
   Offset Location = 1123  (x463)
   Record Length   = 31  (x1F)
   Key Part 1:
       Variable Length Character String
       Actual Length = 16
             2B2B357A 5169792F 31307556 73513D3D     ++5zQiy/10uVsQ==
   Child Pointer  =>  Page 24739
   Table RID: x(FFFF FFFF FFFF) r(FFFFFFFF;FFFF) d(4294967295;65535)
```

Slotted Pages for Data and Indexes

Data Pages:

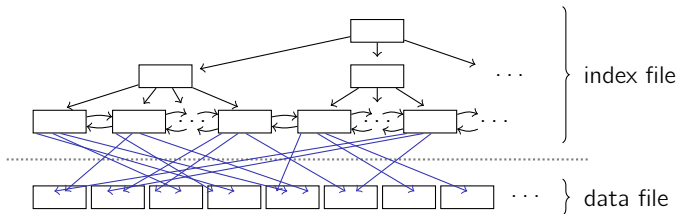- Move record without changing its slot/RID.

Index Pages:

- Also: change slots without moving data.

✎ **Huh?**

B$^+$-trees and Sorting

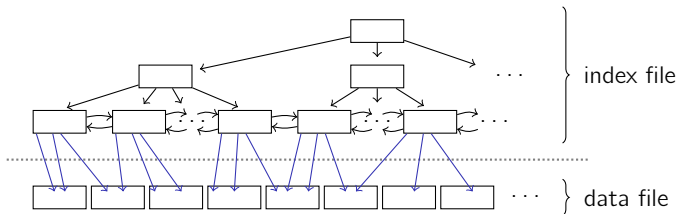A typical situation according to alternative 2 looks like this:

> ✎ **What are the implications when we want to execute**
>   `SELECT * FROM CUSTOMERS ORDER BY ZIPCODE`   **?**

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?**

📈 Indexes and Clustering in IBM DB2

DB2 does **not** offer clustered indexes in the sense discussed here.

But:

- Can declare one "clustering index" per table.

  ```
  CREATE INDEX IndexName
      ON TableName ( col₁ , col₂ , . . . , colₙ ) CLUSTER
  ```

- DB2 will attempt (!) to cluster the table's **data pages** according to the key of the index.
    - Table **re-organization** will re-establish clustering if necessary.
    - Use `ALTER TABLE` and `PCTFREE` to ease future inserts.

Index Organized Tables

Alternative 1 (slide 80) is a special case of a clustered index.

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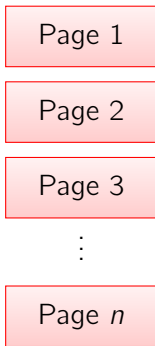- Such a file is often called an **index organized table**.

📈 E.g., Oracle8i

```
CREATE TABLE (...
              ...,
              PRIMARY KEY ( ... ))
  ORGANIZATION INDEX;
```
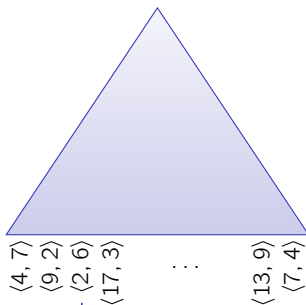

Option A: Heap file for data, indexes with RIDs

data pages
(heap file)

Page 1

Page 2

Page 3

⋮

Page n

index on k
(non-clustered)

$\langle 4, 7 \rangle$ $\langle 9, 2 \rangle$ $\langle 2, 6 \rangle$ $\langle 17, 3 \rangle$ $\cdots$ $\langle 13, 9 \rangle$ $\langle 7, 4 \rangle$
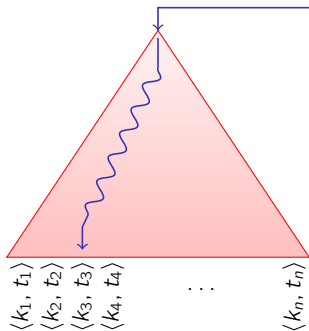
RIDs in
leaf nodes

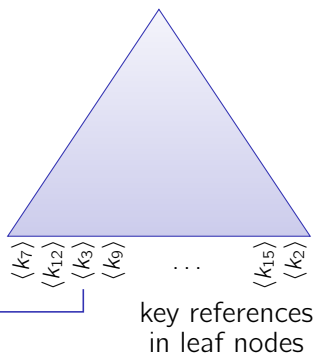$\rightarrow$ Can have arbitrarily many indexes of this kind.

Indexes on Tables

Option B: Data sits in clustered index

unique index on k
(clustered; contains data)
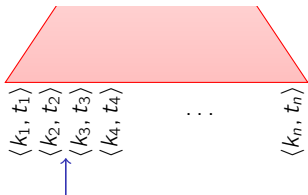
secondary index on a
(non-clustered)

$\langle k_1, t_1 \rangle$ $\langle k_2, t_2 \rangle$ $\langle k_3, t_3 \rangle$ $\langle k_4, t_4 \rangle$ ... $\langle k_n, t_n \rangle$

$\langle k_7 \rangle$ $\langle k_{12} \rangle$ $\langle k_3 \rangle$ $\langle k_9 \rangle$ ... $\langle k_{15} \rangle$ $\langle k_2 \rangle$

key references
in leaf nodes

■ Secondary indexes use key values to reference tuples.

✎ **What about this setup?**

unique index on k **secondary index on** a
(clustered; contains data)   (non-clustered)

RIDs in leaf nodes

Prefix Truncation
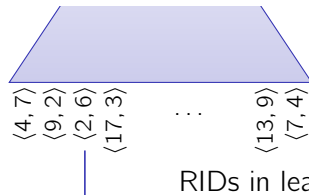
Address book of Berlin, anno 1858:[4]

Poplawsky — Präger. 353

Poplawsky, C., Drechsler in Horn und Holz, Schornsteinfegergergasse 1.
Poplowsky, W., Uhrmacher, Leipziger-straße 43.
Popowitsch, R., Kürschner, Ziegel-straße 11. 12.
Popowska, M., Haushofmeisterfrau, Kochstr. 43.
Popp, C., Schlosser, Schießgasse 7.
— M., Schneider f. D., Kommandan-tenstr. 31.
— A. C., Seidenwirker, Prinzen-Allee 74.
— F., geb. Schmid, Ww., Gutsbe-sitzerin, Kronenstr. 17.
Poppe, C. T., Diätar, J., Wilhelms-straße 92.
— B., Cafetier, Schustergasse 1.
— C., Holz- und Horndrechsler, Wall-straße 54.
— Eduard, Filz- und Filzschuhfabrik, Wolldruckerei u. Lager von Filzschuh-Oberstoff, Plüschband und aller Arten Filzwaaren, Friedrichsstr. 109. E.
— W., Fuhrherr, Neu-Kölln a. W. 21.
— J., Handelsmann, Sophienstr. 26.

Porath, H., Oberfeuermann, Land-wehrstr. 3.
— W., Gärtner und Blumenhändler, Oranienburgerstr. 57.
— F., Tuchmacher, Weberstr. 34.
Parawsky, S., Eisenbahn-Zugführer, Louisenplatz 12.
Porepp, C., Zimmer-Vermiether, Unter den Linden 47.
Pormetter, F. W., Buchdruckerei-besitzer, Kommandantenstr. 7.
Porsch, H., Kunstmaler, Heiligegeist-straße 25.
Porschien, G., Schneider, Friedrichs-gracht 59.
Port, C., Tapezirer, Fruchtstr. 58.
Portessel, J., Polizei-Wachtmeister, Charlottenstr. 37.
— M. u. J., Gerschw., Putz- und Mo-dehändlerinnen, Charlottenstr. 37.
Porth, C., Hofschauspieler, Friedrichs-straße 195.
— H., Tischler, Markgrafenstr. 18.
Porthum, G., Klempner, Rosenstr. 8.
Portier, L., Dlle., Lindenstr. 48.

Possart, F., Intendantur-Applicant, Schumannsstr. 9. 2—4.
— C., Kaufmann, Inhaber des land-wirthschaftlichen Etablissements, Hei-ligegeiststr. 3. F. Eugen Possart. Cp.
— J. E., Kaufmann, Schumannsstr. 9. 12—2.
— F., Schankwirth, Köpnickerstr. 129.
Posse, C., Barbier, Mauerstr. 33.
— L., Schneider f. H., Neuer Markt 9.
— A., Schneider f. H., Krausenstr. 4. 5.
Possel, F., Handelsmann, Dresdner-straße 97.
— F., Musikus, Lindenstr. 56.
Posseldt, J., Geh. Registratur-Assistent im Ministerium für Handel rc., Gra-benstr. 3. 4—5.
Posselt, P., Kanzleidiener, Leipziger-straße 5.
— C., Modelleur, Stralauerplatz 4.
— C., Porzellanmaler, Alte Jakobs-straße 60. E.
— L., Koch und Restaurateur, Charité-straße 5.
— L., Restaurateur, Mittelstr. 57.

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]

Prefix: `Smith, J`								
`ack`	`ane`	`ason`	`eremy`	`ill`	`ohn`	...	`une`	

The advantage is **two-fold**:

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

---

[5]R. Bayer and K. Unterauer. *Prefix B-Trees*. TODS 2(1), March 1972.
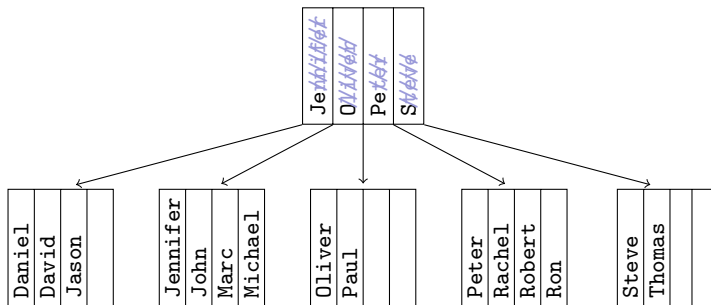
Suffix Truncation

- **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.
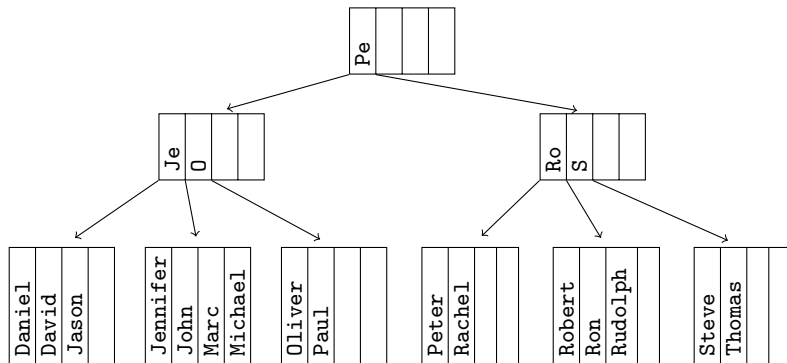
Suffix Truncation

Example:

Suffix Truncation

Suffix truncation beyond the bottom-most level is difficult/dangerous.

$\rightarrow$ Shortening 'Pe' to 'P' would be incorrect!

"Poor Man's Normalized Keys"

The effect of discriminative prefixes can also exploited as follows:

- Store a **fixed-length prefix** as an additional field in every entry of the slot directory.
- Need to follow the pointer only if the prefix is not enough to decide on the comparison outcome.
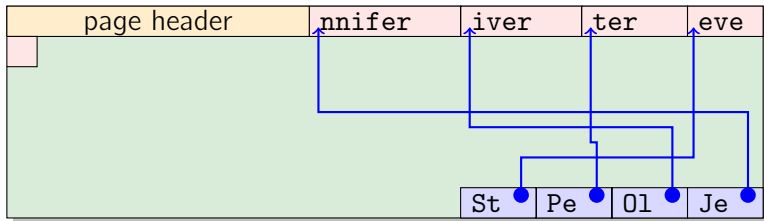
"Poor Man's Normalized Keys"

- Most accesses are to an array of fixed-length elements
  (Pointer chasing in memory is relatively expensive on modern hardware.)

- Can use, *e.g.*, integer comparisons to evaluate four-byte prefix comparisons.

  ⚠ May need to **re-order** bytes for this.

- **CPU cache efficient:** When a slot entry is read, likely the prefix is in the same cache line.

Key Normalization

In practice, key comparisons are not as simple as they look on slides:

- language-specific **collation**
- representations as different **character sets**
- NULL values

Plus, keys might be composed of **multiple columns**.

Thus:

- **Normalize** keys and represent any key as a **bit string**.
    - $\rightarrow$ All of the above issues only affect normalization,
      but not B-tree operations themselves.
- Can prepare, *e.g.*, for integer (rather than bit or byte) comparisons.

Key Normalization

Examples:

- Map upper and lower case letters to same bit string if collation is case insensitive.

- Use bit representations for characters according to collation
  E.g., ö < z in German; z < ö in Swedish.

- To sort NULL before any value: Prepend any valid value with a '1' bit and represent NULL as a '0' bit.

A	B	C	normalized key
2	'Smith'	'John'	$\underline{1}\,00\cdots00000010\,\underline{1}$ Smith'\0' $\underline{1}$ John'\0'
3	'Miller'	''	$\underline{1}\,00\cdots00000011\,\underline{1}$ Miller'\0' $\underline{1}$ '\0'
64	–	'Dave'	$\underline{1}\,00\cdots00100000\,\underline{0}\,\underline{1}$ Dave'\0'
–	''	–	$\underline{0}\,\underline{1}$ '\0' $\underline{0}$

Key Normalization

⚠ Information might get lost during normalization
(*e.g.*, capitalization)

→ Store normalized **and** original key (redundantly) in leaf nodes or
→ Use normalization only in inner nodes

Keys tend to become larger due to normalization.

→ **Order-preserving compression** might be useful.

Key normalization and prefix/suffix truncation go particularly well together.
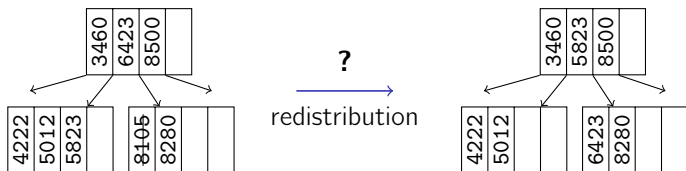
Deletion

- 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"):
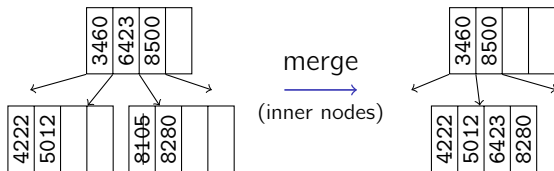
- "Pull" separator into merged node.

⚠ 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.
- To improve **concurrency**, systems sometimes only **mark** index entries as deleted and physically remove them later (*e.g.*, IBM DB2 UDB "type-2 indexes")
    - → **"Ghost bits" / "ghost records"**
    - → Often kept around for a while → re-use on next insert.

Ghost Records — 🗽 IBM DB2

Before key deletion:

```
Key 64:
 Offset Location = 3710  (xE7E)
 Record Length  = 40  (x28)

 Key Part 1:
     Variable Length Character String
     Actual Length = 28
          44516A6B 7A334650 76724471 534B7767    DQjkz3FPvrDqSKwg
     58432B59 345A7837 4852383D               XC+Y4Zx7HR8=
 Table RID: x(0000 1237 0001) r(00001237;0001) d(4663;1) ridFlags=x0
```

After key deletion:

```
Key 64:
 Offset Location = 3710  (xE7E)
 Record Length  = 40  (x28)

 Key Part 1:
     Variable Length Character String
     Actual Length = 28
          44516A6B 7A334650 76724471 534B7767    DQjkz3FPvrDqSKwg
     58432B59 345A7837 4852383D               XC+Y4Zx7HR8=
 Table RID: x(0000 1237 0001) r(00001237;0001) d(4663;1) ridFlags=x3 Punc Deleted
```

Key Deletion — 📕 IBM DB2

In IBM DB2, redistribution and merging are **only** applied if

- the page is a **leaf node** and
  (Remember the pointers between adjacent leaf nodes, ↗ slide 69.)

- the fill degree of the page falls below MINPCTUSED and
  (That also means that MINPCTUSED must have a value greater than
  its default, which is 0.)

- the transaction holds an **exclusive lock on the table**.

This is called **online index defragmentation** in DB2.

Otherwise, "clean-up" only happens during explicit index maintenance.
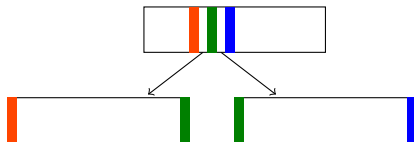
- Use REORG INDEX to trigger maintenance.
- Use REORGCHK to check whether index(es) need maintenance.

Ghost Records

Ghost records turn out to be useful for a number of purposes.

E.g., **fence keys**

- Keep a copy of parent's separator keys in every node

- Fence keys span range of **possible** key values in this node
  - $\rightarrow$ Avoids problems with **prefix truncation**.
- One key is an **exclusive bound**, thus **must** be a ghost record.
- The other one may or may not be a ghost record.
- Can be used, *e.g.*, to check **integrity** of B-tree.

Variable-Length Keys

With **variable-length keys**, the original B-tree property

$$d \leq \text{number of keys in a node} \leq 2d$$

is not practical any more.

$\rightarrow$ Real-world systems do not really care about this "50 % rule."

With truncation, the storage space for a key might even **change** during re-organizations.

- ✎ Will this cause any trouble during updates?

Composite Keys

B^+-trees can (in theory[6]) 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.

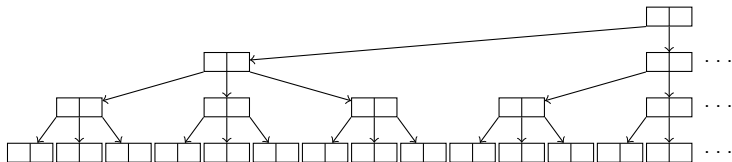↗ G. Graefe: Sorting And Indexing With Partitioned B-Trees. *CIDR 2003.*

---

[6]Some implementations won't allow you to index, *e.g.*, large character fields.

Partitioned B-trees

```
CREATE INDEX ON TABLE STUDENTS (SEMESTER, ZIPCODE);
```

✎ **What types of queries could this index support?**

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 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 $2/3$ 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 B^+-trees.

"B-trees" are also used outside the database domain, *e.g.*, in modern **file systems** (ReiserFS, HFS, NTFS, . . .).

Hash-Based Indexing

B^+-trees are **by far** the predominant type of indices in databases. An alternative is **hash-based indexing**.
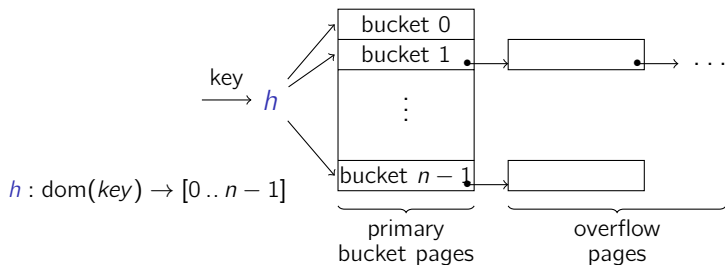
- Hash indices can only be used to answer **equality predicates**.
- Particularly good for strings (even for very long ones).

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 **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.