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

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

Winter 2013/14

Part III

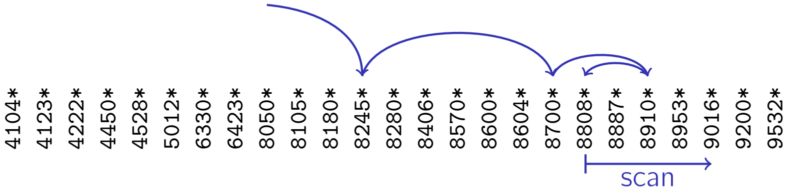
Indexing

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

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

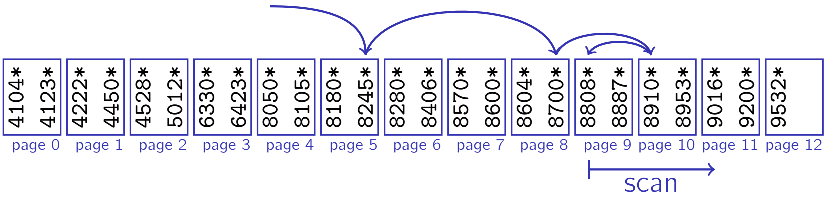
We could

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.



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

Ordered Files and Binary Search

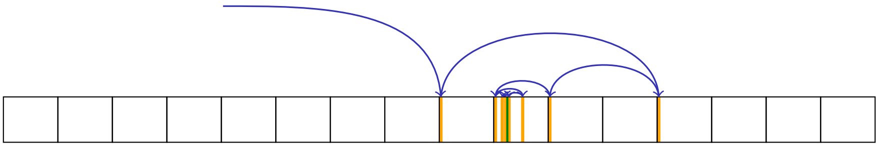


✔ We get **sequential access** during the **scan phase**.

We need to read $\log_2(\#\,tuples)$ 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.)

Binary Search and Database Pages

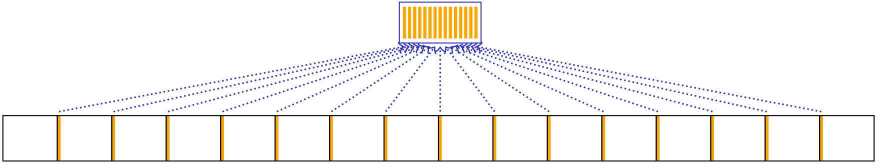


Observations:

- Make rather **far jumps initially**.
  - $\rightarrow$ For each step read **full page**, but inspect only **one record**.
- After $\mathcal{O}(\log_2 \textit{pagesize})$, 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.



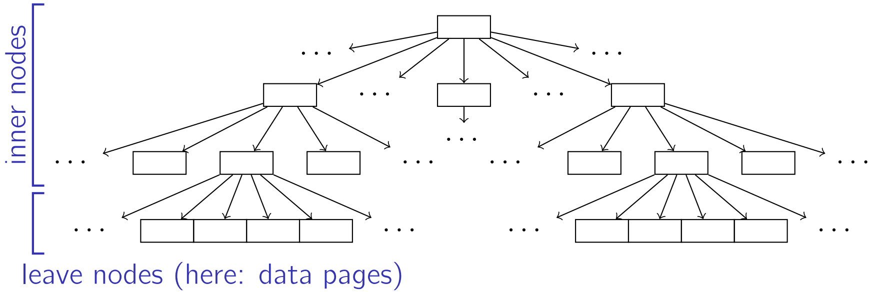
→ If we can keep the cache **in memory**, we can find **any** record with just a **single I/O**.

✎ **Is this assumption reasonable?**

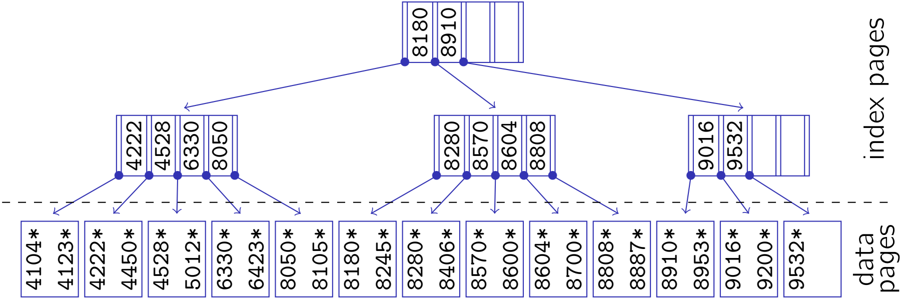
Large Data

What if my data set is really large?

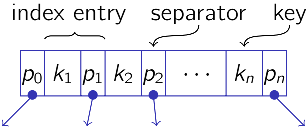
- "Cache" will span many pages, too.
  (In practice, we'll organize the cache just like any other database object.)
- Thus: **"cache the cache"** → hierarchical "cache"



Idea: Accelerate the search phase using an index.



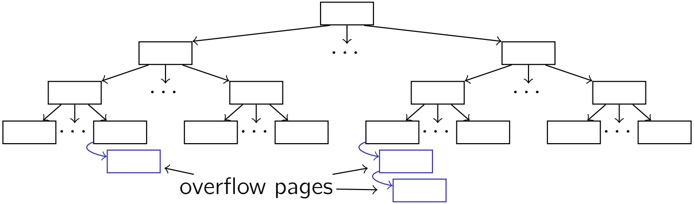
- All nodes are the size of a page
  - → hundreds of entries per page
  - → large fanout, low depth
- Search effort: $\log_{\mathit{fanout}}(\#\ \mathit{tuples})$



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.



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.

B^+-trees: A Dynamic Index Structure

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

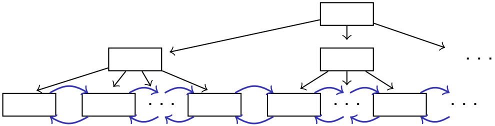
- **No overflow chains**; B^+-trees remain **balanced** at all times
- Gracefully adjusts to **inserts** and **deletes**.
- **Minimum occupancy** for all B^+-tree nodes (except the root): **50 %** (typically: 67 %).

- 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 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)

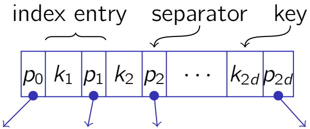
[2] This is not really a B^+-tree requirement, but some systems implement it.

Searching a B^+-tree

Function: `search` (k)
return `tree_search` $(k, root)$;

Function: `tree_search` $(k, node)$
if *node* is a leaf **then**
    return *node*;
switch k **do**
    case $k < k_1$
        return `tree_search` (k, p_0);
    case $k_i \leq k < k_{i+1}$
        return `tree_search` (k, p_i);
    case $k_{2d} \leq k$
        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

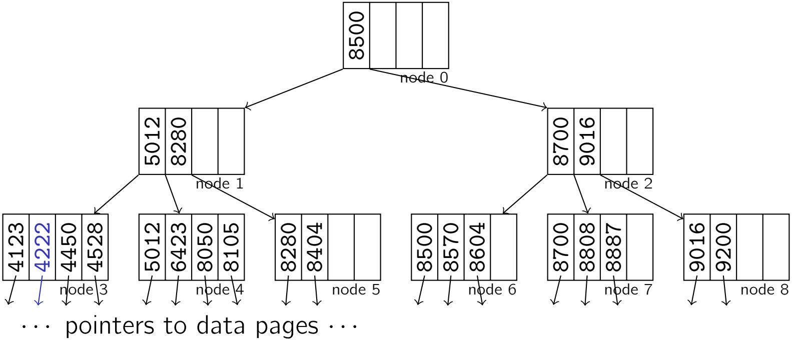
Insert: Overview

- 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.

Insert: Examples (Insert without Split)

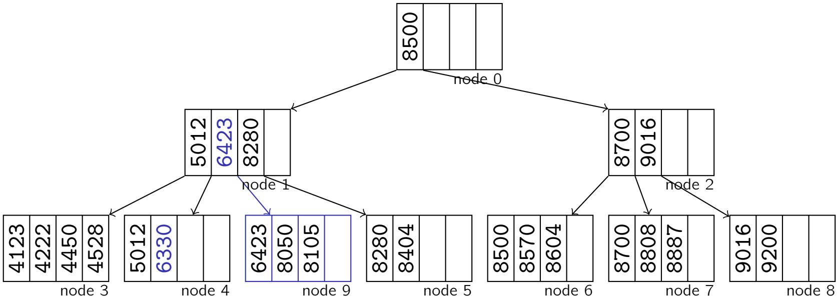


Insert new entry with key 4222.

→ Enough space in node 3, simply insert.

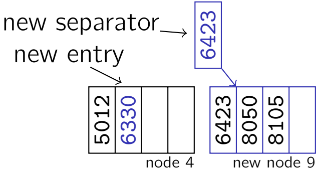
→ Keep entries **sorted within nodes**.

Insert: Examples (Insert with Leaf Split)

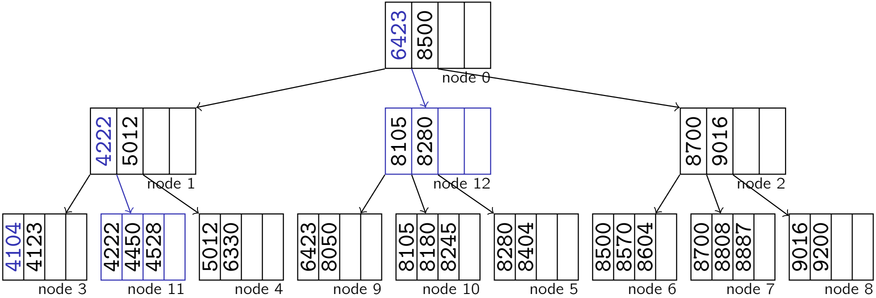


Insert key 6330.

- → Must **split** node 4.
- → **New separator** goes into node 1 (including pointer to new page).



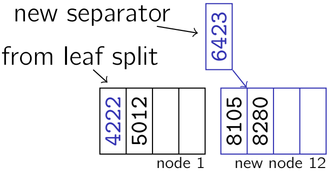
Insert: Examples (Insert with Inner Node Split)



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?**



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

```
Function: tree_insert (k, rid, node)
if node is a leaf then
    return leaf_insert (k, rid, node);
else
    switch k do
        case k < k_1
            ⟨sep, ptr⟩ ← tree_insert (k, rid, p_0);
        case k_i ≤ k < k_{i+1}
            ⟨sep, ptr⟩ ← tree_insert (k, rid, p_i);
        case k_{2d} ≤ k
            ⟨sep, ptr⟩ ← tree_insert (k, rid, p_{2d});
    if sep is null then
        return ⟨null, null⟩;
    else
        return split (sep, ptr, node);
```

see `tree_search ()`

Function: `leaf_insert` (k, rid, $node$)

if another entry fits into *node* **then**
  insert $\langle k, rid \rangle$ into *node* ;
  return $\langle \mathbf{null}, \mathbf{null} \rangle$;
else
  allocate new leaf page p ;
  take $\left\{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \right\} :=$ entries from *node* $\cup \; \{\langle k, ptr \rangle\}$
    leave entries $\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle$ in *node* ;
    move entries $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \ldots, \langle k_{2d}^+, p_{2d}^+ \rangle$ to p ;
  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 $\langle \mathbf{null}, \mathbf{null} \rangle$;
else
  allocate new leaf page p ;
  take $\left\{ \langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \right\} :=$ entries from *node* $\cup \; \{\langle k, ptr \rangle\}$
    leave entries $\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle$ in *node* ;
    move entries $\langle k_{d+2}^+, p_{d+1}^+ \rangle, \ldots, \langle k_{2d}^+, p_{2d}^+ \rangle$ to p ;
    set $p_0 \leftarrow p_{d+1}^+$ in *node*;
  return $\langle k_{d+1}^+, p \rangle$;

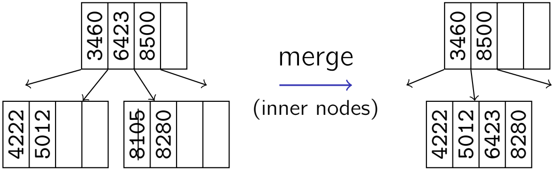
Insertion Algorithm

```
Function: insert (k, rid)
⟨key, ptr⟩ ← tree_insert (k, rid, root);
if key is not null then
    allocate new root page r;
    populate n with
        p0 ← root;
        k1 ← key;
        p1 ← ptr;
    root ← 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

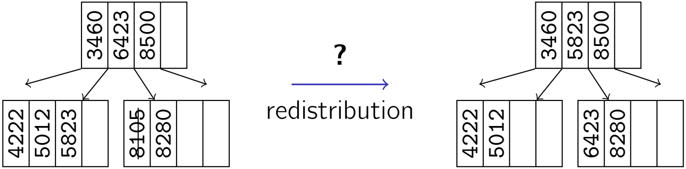
- If a node is sufficiently full (*i.e.*, contains at least $d + 1$ entries, we may simply remove the entry from the node.
    - Note: Afterward, **inner nodes** may contain keys that no longer exist in the database. This is perfectly legal.
- **Merge** nodes in case of an **underflow** ("undo a split"):



    - "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, $\nearrow$ slide 63.)
  - Inner nodes are never merged
    ($\rightarrow$ 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 $\langle k, rid \rangle$ pair, where *rid* is the **record id** of the data entry.
3. A $\langle k, \{rid_1, rid_2, \ldots\} \rangle$ pair. The items in the **rid list** rid_i are record ids of data entries with search key value k.

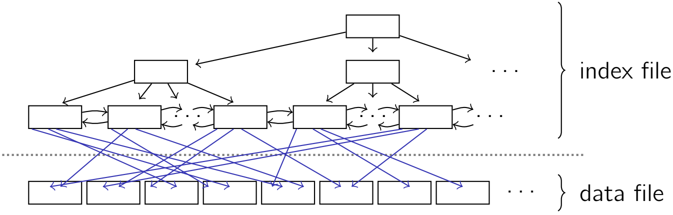
Options 2 and 3 are reasons why want record ids to be **stable**.

→ slides 45 ff.

Alternative 2 seems to be the most common one.

B^+-trees and Sorting

A typical situation according to alternative 2 looks like this:

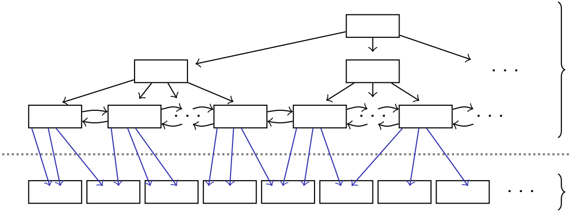


What are the implications when we want to execute

`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?**

Index Organized Tables

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

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

Prefix Truncation

Address book of Berlin, anno 1858:[4]

Poplawsky — Präger. 353

Poplawsky, C., Drechsler in Horn und Holz, Schornsteinfegergasse 1.
Poplowsky, W., Uhrmacher, Leipzigerstraße 43.
Popowitsch, R., Kürschner, Ziegelstraße 11. 12.
Popowska, M., Haushofmeisterfrau, Kochstr. 43.
Popp, C., Schlosser, Schießgasse 7.
— M., Schneider f. D., Kommandantenstr. 31.
— A. C., Seidenwirker, Prinzen-Allee 74.
— F., geb. Schmidt, Ww., Gutsbesitzerin, Kronenstr. 17.
Poppe, C. T., Diätar, J., Wilhelmsstraße 92.
— W., Cafetier, Schustergasse 1.
— E., Holz- und Horndrechsler, Wallstraße 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, Landwehrstr. 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., Buchdruckereibesitzer, Kommandantenstr. 7.
Porsch, H., Kunstmaler, Heiligegeiststraße 25.
Porschien, G., Schneider, Friedrichsgracht 59.
Port, C., Tapezirer, Fruchtstr. 58.
Portesset, J., Polizei-Wachtmeister, Charlottenstr. 37.
— M. u. J., Geschw., Putz- und Modehändlerinnen, Charlottenstr. 37.
Porth, C., Hofschauspieler, Friedrichsstraße 195.
— H., Tischler, Markgrafenstr. 18.
Porthun, G., Klempner, Rosenstr. 8.
Portier, L., Dlle., Lindenstr. 48.

Possart, F., Intendantur-Applicant, Schumannsstr. 9. 2—4.
— E., Kaufmann, Inhaber des landwirthschaftlichen Etablissements, Heiligegeiststr. 3. F. Eugen Possart. Cp.
— J. C., 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, Dresdnerstraße 97.
— F., Musikus, Lindenstr. 56.
Posseldt, J., Geh. Registratur-Assistent im Ministerium für Handel rc., Grabenstr. 3. 4—5.
Posselt, P., Kanzleidiener, Leipzigerstraße 5.
— C., Modelleur, Stralauerplatz 4.
C., Porzellanmaler, Alte Jakobsstraße 60. E.
— L., Koch und Restaurateur, Charitéstraße 5.
— L., Restaurateur, Mittelstr. 57.

[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]

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

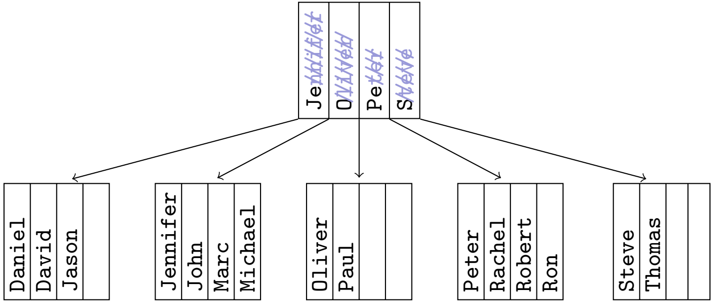
- **Prefix truncation** is most effective near or in leaf pages. ✎ **Why?**
- Elsewhere, by contrast, the leading key parts are most discriminative.
- In fact, a key's suffix might not be needed to guide navigation at all.

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.

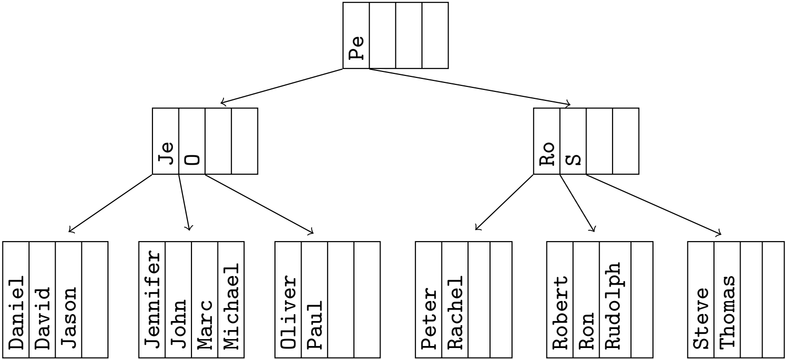
Suffix Truncation

Example:



Suffix Truncation

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



→ Shortening '`Pe`' to '`P`' would be incorrect!

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.

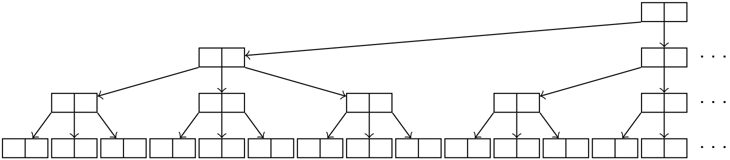
Partitioned B-trees

```
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**:

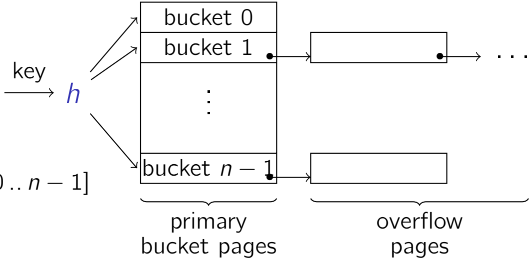
- Keep non-root nodes at least 2/3 full (instead of 1/2).
- Need to **redistribute on inserts** to achieve this.
  (Whenever **two** nodes are full, split them into **three**.)

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**.



$h : \mathrm{dom}(key) \rightarrow [0 \mathinner{..} n-1]$

- 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 **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.