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

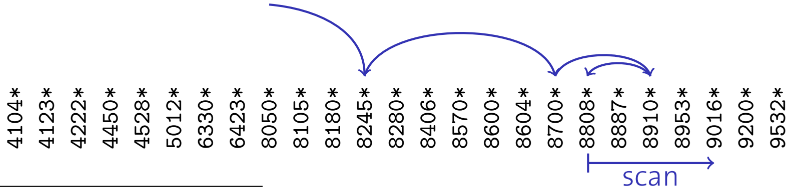
- Ich bedanke mich für die Bereitstellung des Materials

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

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

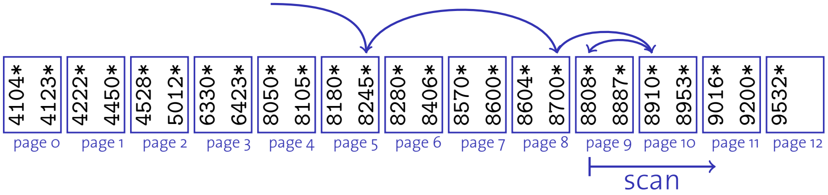
We could

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



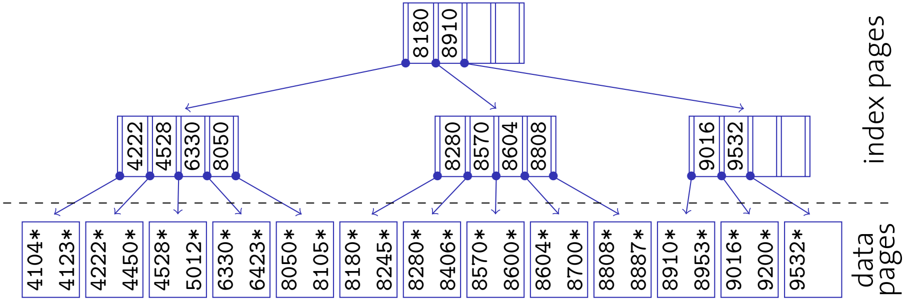
✔ We get **sequential access** during the **scan phase**.

We need to read $\log_2(\#\, tuples)$ tuples during the **search phase**.

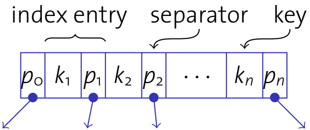
✗ We need to read about as many **pages** for this.
(The whole point of binary search is that we make far, unpredictable jumps. This largely defeats prefetching.)

ISAM—Indexed Sequential Access Method

Idea: Accelerate the search phase using an index.



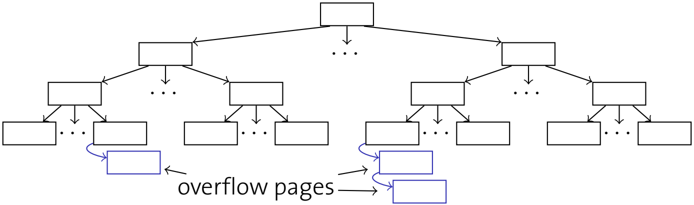
- All nodes are the size of a page
  - → hundreds of entries per page
  - → large fanout, low depth
- Search effort: $\log_{fanout}(\#\,tuples)$



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.



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.

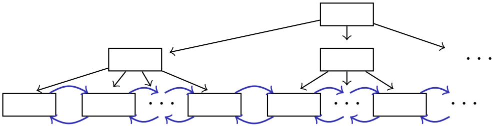
- **No overflow chains**; B$^+$-trees remain **balanced** at all times
- Gracefully adjusts to **inserts** and **deletes**.
- **Minimum occupancy** for all B$^+$-tree nodes (except the root): **50 %** (typically: 67 %).

- 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 67 and 70
  - 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)

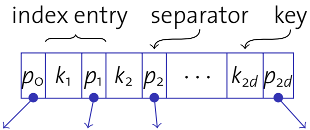
[2] This is not really a B$^+$-tree requirement, but most systems implement it.

Searching a B$^+$-tree

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

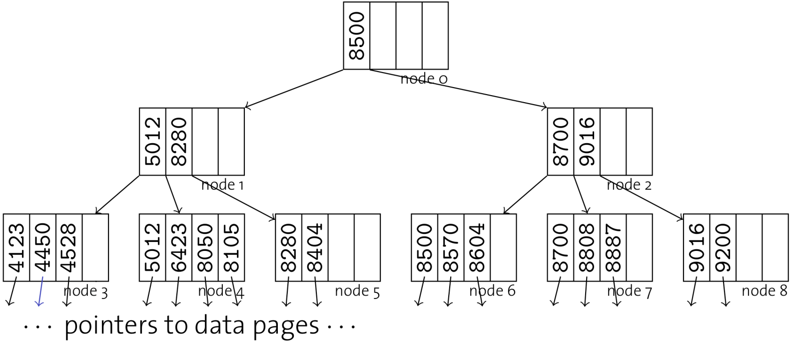
- The B^+-tree needs to remain **balanced** after every update.[3]
  $\longrightarrow$ We **cannot** create overflow pages.
- Sketch of the insertion procedure for entry $\langle k, p \rangle$ (key value k pointing to data page p):

  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)

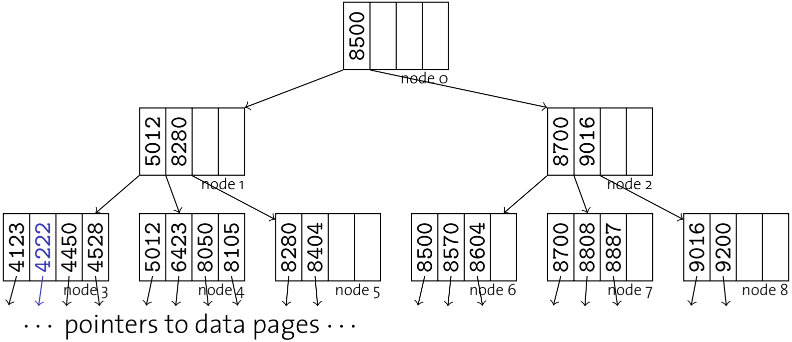


Insert new entry with key 4222.

→ Enough space in node 3, simply insert.

→ Keep entries **sorted within nodes**.

Insert: Examples (Insert without Split)

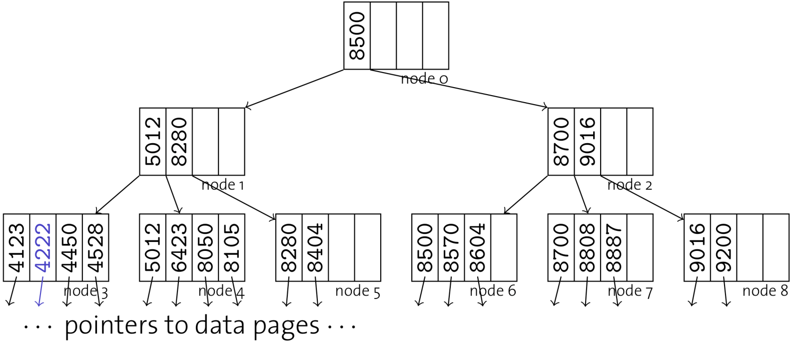


Insert new entry with key 4222.

→ Enough space in node 3, simply insert.

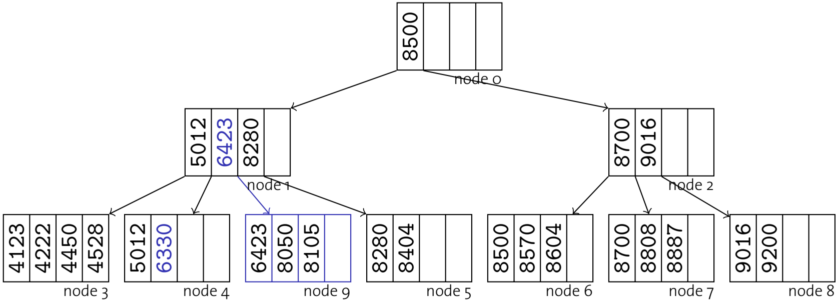
→ Keep entries **sorted within nodes**.

Insert: Examples (Insert without Split)



Insert new entry with key 6330.

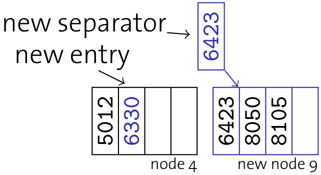
Insert: Examples (Insert with Leaf Split)



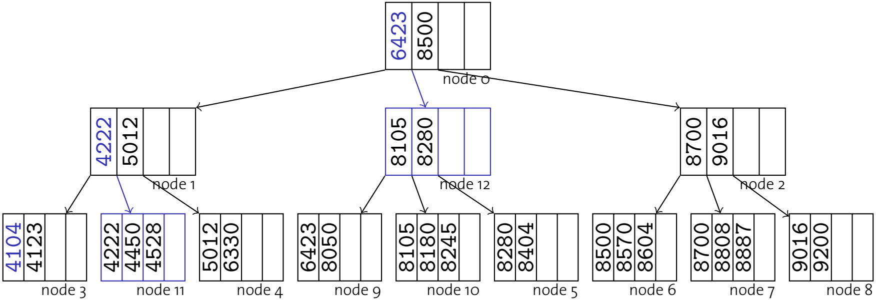
Insert key 6330.

→ Must **split** node 4.

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



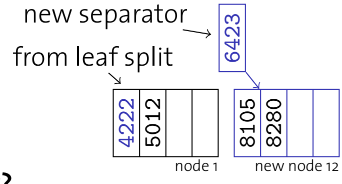
Insert: Examples (Insert with Inner Node Split)



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

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

if *node* is a leaf **then**
    return `leaf_insert` (k, rid, $node$);
else
    switch k **do**
        case $k < k_1$
            $\langle sep, ptr \rangle \leftarrow$ `tree_insert` (k, rid, p_o) ;
        case $k_i \leq k < k_{i+1}$
            $\langle sep, ptr \rangle \leftarrow$ `tree_insert` (k, rid, p_i) ;
        case $k_{2d} \leq k$
            $\langle sep, ptr \rangle \leftarrow$ `tree_insert` (k, rid, p_{2d}) ;

    see `tree_search` ()

    if *sep* is **null then**
        return $\langle$**null**, **null**$\rangle$;
    else
        return `split` (sep, ptr, $node$);

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

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

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

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

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 $\langle \textbf{null}, \textbf{null} \rangle$;

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

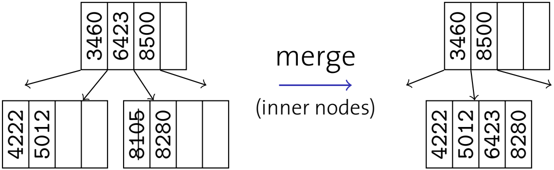
Function: `insert` (k, rid)

$\langle key, ptr \rangle \leftarrow$ `tree_insert` $(k, rid, root)$;
if key is not **null then**
  allocate new root page r;
  populate n **with**
    $p_0 \leftarrow root$;
    $k_1 \leftarrow key$;
    $p_1 \leftarrow ptr$;
  $root \leftarrow r$;

- `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

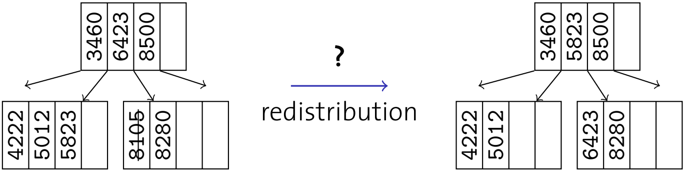
- If a node is sufficiently full (*i.e.*, contains at least $d+1$ entries, we may simply remove the entry from the node.
  - Note: Afterward, **inner nodes** may contain keys that no longer exist in the database. This is perfectly legal.
- **Merge** nodes in case of an **underflow** (“undo a split”):



  - “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")

partition change tracking (PCT) i

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 $\langle k, \{rid_1, rid_2, \dots\} \rangle$ pair. The items in the **rid list** rid_i are record ids of data entries with search key value k.

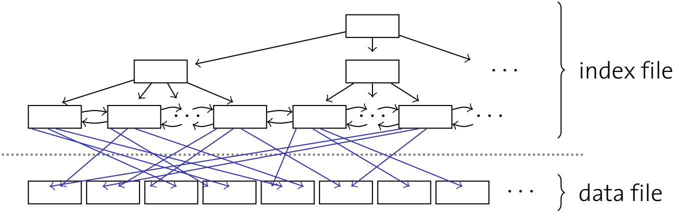
Options 2 and 3 are reasons why want record ids to be **stable**.

→ slides 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:

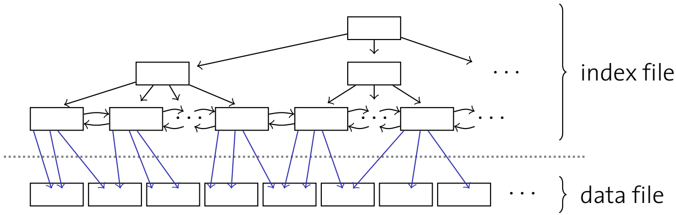


✎ **What are the implications when we want to execute**

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

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**.

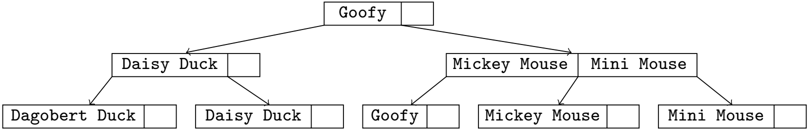
E.g., Oracle8i

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

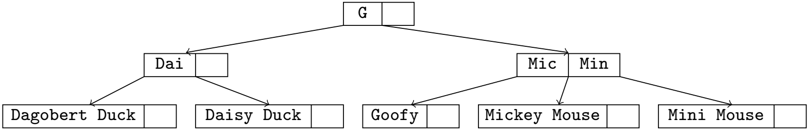
Prefix and Suffix Truncation

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**.



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



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.*, as "k_0").
- Keys have become highly discriminative now.

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

↗ R. Bayer, K. Unterauer: Prefix B-Trees. *ACM TODS* 2(1), March 1977

Composite Keys

B^+-trees can (in theory[4]) 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**:

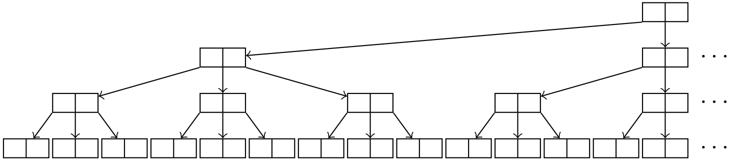
- Leading index attributes effectively **partition** the resulting B^+-tree.

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

[4] Some 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 **B^+-tree**.

Bayer and McCreight originally proposed the **B-tree**:

- Inner nodes contain data entries, too. ✎ **Pros/cons?**

There is also a **B^*-tree**:

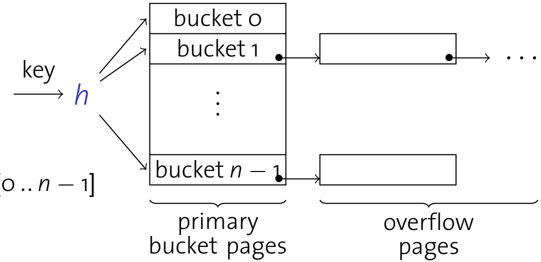
- Keep non-root nodes at least 2/3 full (instead of 1/2).
- Need to **redistribute on inserts** to achieve this. (Whenever **two** nodes are full, split them into **three**.)

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**.



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

- 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.