10 DBMS Architecture: Managing Data

10.1 Storing Data: Disks and Files

10.1.1 Memory hierarchy

Memory in off-the-shelf computer systems is arranged in a hierarchy:

<table>
<thead>
<tr>
<th>Request</th>
<th>Storage Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>primary</td>
</tr>
<tr>
<td>CPU Cache (L1, L2)</td>
<td>primary</td>
</tr>
<tr>
<td>Main Memory (RAM)</td>
<td>secondary</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>secondary</td>
</tr>
<tr>
<td>Tape, CD-ROM, DVD</td>
<td>tertiary</td>
</tr>
</tbody>
</table>

Cost of primary memory \( \approx 100 \times \) cost of secondary storage space of the same size.

Size of address space in primary memory (e.g., \(2^{32}\) Byte = 4 GB) may not be sufficient to map the whole database (we might even have \(\gg 2^{32}\) records).

DBMS needs to make data persistent across DBMS (or host) shutdowns or crashes; only secondary/tertiary storage is nonvolatile.

DBMS needs to bring in data from lower levels in memory hierarchy as needed for processing.
10.1.1.1 Magnetic disks

- Tapes store vast amounts of data ($\gg 20$ GB; more for roboter tape farms) but they are sequential devices.
- Magnetic disks (hard disks) allow direct access to any desired location; hard disks dominate database system scenarios by far.

1. Data on a hard disk is arranged in concentric rings (tracks) on one or more platters,
2. tracks can be recorded on one or both surfaces of a platter,
3. set of tracks with same diameter form a cylinder,
4. an array (disk arm) of disk heads, one per recorded surface, is moved as a unit,
5. a stepper motor moves the disk heads from track to track, the platters steadily rotate.
Each track is divided into arc-shaped sectors (a characteristic of the disk’s hardware),

2. data is written to and read from disk block by block (the block size is set to a multiple of the sector size when the disk is formatted),

3. typical disk block sizes are 4 KB or 8 KB.

### 10.1.1.2 Performance implications of disk structure

Data blocks can only be written and read if disk heads and platters are positioned accordingly.

- This has implications on the disk access time:
  1. Disk heads have to be moved to desired track (seek time),
  2. disk controller waits for desired block to rotate under disk head (rotational delay),
  3. disk block data has to be actually written/read (transfer time).

\[
\text{access time} = 1 + 2 + 3
\]
Access time for the IBM Deskstar 14GPX

- 3.5 inch hard disk, 14.4 GB capacity
- 5 platters of 3.35 GB of user data each, platters rotate at 7200/min
- average seek time 9.1 ms (min: 2.2 ms [track-to-track], max: 15.5 ms)
- average rotational delay 4.17 ms
- data transfer rate 13 MB/s

\[
\text{access time}_{8 \text{ KB block}} \approx 9.1 \text{ ms} + 4.17 \text{ ms} + \frac{1 \text{ s}}{13 \text{ MB}/8 \text{ KB}} \approx 13.87 \text{ ms}
\]

N.B. Accessing a main memory location typically takes \(< 60\) ns.
The unit of a data transfer between disk and main memory is a block,
if a single item (e.g., record, attribute) is needed, the whole containing block must be transferred:

Reading or writing a disk block is called an I/O operation.

The time for I/O operations dominates the time taken for database operations.

DBMSs take the geometry and mechanics of hard disks into account.

- Current disk designs can transfer a whole track in one platter revolution, active disk head can be switched after each revolution.
- This implies a closeness measure for data records \( r_1, r_2 \) on disk:
  1. Place \( r_1 \) and \( r_2 \) inside the same block (single I/O operation!),
  2. place \( r_2 \) inside a block adjacent to \( r_1 \)'s block on the same track,
  3. place \( r_2 \) in a block somewhere on \( r_1 \)'s track,
  4. place \( r_2 \) in a track of the same cylinder than \( r_1 \)'s track,
  5. place \( r_2 \) in a cylinder adjacent to \( r_1 \)'s cylinder.
10.1.2 Disk space management

- The **disk space manager** encapsulates the gory details of hard disk access for the DBMS,
- the disk space manager talks to the disk controller and initiates I/O operations,
- once a block has been brought in from disk it is referred to as a **page**.
- Sequences of data pages are mapped onto contiguous sequences of blocks by the disk space manager.
- The DBMS issues allocate/deallocate and read/write commands to the disk space manager,
- which, internally, uses a

  \[
  \text{block-#} \leftrightarrow \text{page-#}
  \]

mapping to keep track of page locations and block usage.

---

1Disk blocks and pages are of the same size.
10.1.2.1 Keeping track of free blocks

- During database (or table) creation it is likely that blocks indeed can be arranged contiguously on disk.
- Subsequent deallocations and new allocations however will, in general, create holes.
- To reclaim space that has been freed, the disk space manager either uses
  - a free block list:
    1. keep a pointer to the first free block in a known location on disk,
    2. when a block is no longer needed, append/prepend this block to the free block list for future use,
    3. next pointers may be stored in disk blocks themselves,
  - or free block bitmap:
    1. reserve a block whose bytes are interpreted bit-wise (bit $n = 0$: block $n$ is free),
    2. toggle bit $n$ whenever block $n$ is (de-)allocated.
- Free block bitmaps allow for fast identification of contiguous sequences of free blocks.
10.1.3 Buffer manager

▶ Size of the database on secondary storage

\[ \geq \]

size of avail. primary mem. to hold user data.

▶ To scan the entire pages of a 20 GB table (SELECT * FROM ...), the DBMS needs to

1. bring in pages as they are needed for in-memory processing,
2. overwrite (replace) such pages when these become obsolete for query processing and new pages require in-memory space.

▶ The buffer manager manages a collection of pages in a designated main memory area, the buffer pool,

▶ once all slots (frames) in this pool have been occupied, the buffer manager uses a replacement policy to decide which frame to overwrite when a new page needs to be brought in.
**N.B.** Simply overwriting a page in the buffer pool is *not* sufficient if this page has been modified after it has been brought in (i.e., the page is so-called **dirty**).
Simple interface for a typical buffer manager:

- Indicate that page $p$ is needed for further processing:

  $$\text{pinPage}(p):$$
  ```
  \text{if buffer pool contains } p \text{ already then}
  \begin{align*}
  \quad & \text{pinCount}(p) \leftarrow \text{pinCount}(p) + 1; \\
  \quad & \text{return address of frame for } p;
  \end{align*}
  \text{select a victim frame } p' \text{ to be replaced using the replacement policy;}
  \text{if dirty}(p') \text{ then}
  \begin{align*}
  \quad & \text{write } p' \text{ to disk;}
  \end{align*}
  \text{read page } p \text{ from disk into selected frame;}
  \text{pinCount}(p) \leftarrow 1;
  \text{dirty}(p) \leftarrow \text{false;}
  ```

- Indicate that page $p$ is no longer needed as well as whether $p$ has been modified by a transaction ($d$):

  $$\text{unpinPage}(p, d):$$
  ```
  \text{pinCount}(p) \leftarrow \text{pinCount}(p) - 1;
  \text{dirty}(p) \leftarrow d;
  ```
The `pinCount` of a page indicates how many "users" (e.g., transactions) are working with that page,

- "clean" victim pages are not written back to disk,
- a call to `unpinPage` does not trigger any I/O operation, even if the `pinCount` for that page goes down to 0 (the page might become a suitable victim, though),
- a database transaction is required to properly "bracket" any page operation using `pinPage` and `unpinPage`, i.e.

\[
\begin{align*}
& a \leftarrow \text{pinPage}(p); \\
& \ldots \\
& \quad \text{read data (records) on page at address } a; \\
& \quad \ldots \\
& \text{unpinPage}(p, \text{false}); \\
& \text{or}
\end{align*}
\]
\[
a \leftarrow \text{pinPage}(p); \\
\]

\[
\quad \quad \text{\ldots}
\quad \quad \text{read and modify data (records) on page at address } a; \\
\quad \quad \text{\ldots}
\]

\[
\text{unpinPage}(p, \text{true});
\]
10.1.3.1 Buffer replacement policies

The choice of **victim frame selection** (or buffer replacement) policy can considerably affect DBMS performance.

Two policies found in a number of DBMSs:

1. **LRU** ("least recently used")
   - Keep a *queue* (often described as a *stack*) of pointers to frames.
   - In `unpinPage(p, d)`, append `p` to the *tail* of queue, if `pinCount(p)` is decremented to 0.
   - To find the next victim, search through the queue from its *head* and find the first page `p` with `pinCount(p) = 0`.

2. **Clock** ("second chance")
   - Number the `N` frames in buffer pool `0...N - 1`, initialize counter `current ← 0`, and maintain a bit array `referenced[0...N - 1]`, initialized to all 0.
   - In `pinPage(p)`, do `reference[p] ← 1`.
   - To find the next victim, consider page `current`.
     - If `pinCount(current) = 0` and `referenced[current] = 0`, `current` is the victim.
     - Otherwise, `referenced[current] ← 0`, `current ← (current + 1) mod N`, repeat.
**N.B.** LRU as well as Clock are **heuristics** only. Any heuristic can fail miserably in certain scenarios:

**A challenge for LRU**

A number of transactions want to scan the same sequence of pages (e.g., SELECT * FROM R) one after the other. Assume a buffer pool with a capacity of 10 pages.

1. Let the size of relation $R$ be 10 or less pages. How many I/Os do you expect?
2. Let the size of relation $R$ be 11 pages. What about the number of I/O operations in this case?

Other well-known replacement policies are, *e.g.*,

- **FIFO** ("first in, first out"),
- **LFU** ("least frequently used"),
- **MRU** ("most recently used"),
- **GCLOCK** ("generalized glock"),
- **WS, HS** ("working set", "hot set"),
- **Random**.
10.1.3.2 Buffer management in DBMSs vs. OSs

- Buffer management for a DBMS curiously “tastes” like the **virtual memory**\(^2\) concept of modern operating systems.
- Both techniques provide access to more data than will fit into primary memory.

**So: why, then, don’t we use OS virtual memory facilities to implement DBMSs?**

- A DBMS can predict certain **reference patterns** for pages in a buffer *a lot better* than a general purpose OS.
- This is mainly because page references in a DBMS are initiated by **higher-level operations** (sequential scans, relational operators) the DBMS itself knows about.

<table>
<thead>
<tr>
<th>Reference pattern examples in a DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Sequential scans</strong> call for <strong>prefetching</strong>.</td>
</tr>
<tr>
<td><strong>2. Nested-loop joins</strong> call for page <strong>fixing</strong> and <strong>hating</strong>.</td>
</tr>
</tbody>
</table>

- Finally, concurrency control is based on protocols which **prescribe the order** in which pages have to be written back to disk. Operating systems usually do not provide hooks for that.

\(^2\)Generally implemented using a hardware interrupt mechanism called **page faulting**.
10.1.4 File and record organization

We will now turn away from page management and will instead focus on page usage in a DBMS.

On the conceptual level, a relational DBMS manages tables of tuples, e.g.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>42</td>
<td>true</td>
<td>'foo'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

On the physical level, such tables are represented as files of records (tuple = record), each page holds one or more records (in general, |record| \ll |page|).

A file is a collection of records that may reside on several pages.

---

3More precisely, table actually means bag here (set of elements with multiplicity $\geq 0$).
10.1.4.1 Heap files

- The most simple file structure is the **heap file** which represents an unordered collection of records.
- As in any file structure, each record in a heap file has a **unique record identifier** (*rid*).
- A typical heap file interface supports the following operations:
  - **create/destroy** heap file *f* named *n*:
    ```plaintext
    createFile(n) / deleteFile(f)
    ```
  - **insert** record *r* and return its *rid*:
    ```plaintext
    insertRecord(f, r)
    ```
  - **delete** a record with a given *rid*:
    ```plaintext
    deleteRecord(f, rid)
    ```
  - **get** a record with a given *rid*:
    ```plaintext
    getRecord(f, rid)
    ```
  - initiate a sequential **scan** over the whole heap file:
    ```plaintext
    openScan(f)
    ```

- **N.B.** Record ids (*rids*) are used like **record addresses** (or pointers). Internally, the heap file structure must be able to map a given *rid* to the page containing the record.
To support openScan($f$), the heap file structure has to **keep track of all pages in file $f$**; to support insertRecord($f, r$) efficiently, we need to **keep track of all pages with free space in file $f$**.

Let us have a look at two simple structures which can offer this support.

### 10.1.4.2 Linked list of pages

When `createFile($n$)` is called,

1. the DBMS allocates a free page (the file *header*) and writes entry $\langle n, \text{header page} \rangle$ to a known location on disk;
2. the *header page* is initialized to point to **two doubly linked lists of pages**:

   ![Linked list of pages diagram](image)

   - Initially, both lists are empty.
Remarks:

- For `insertRecord(f, r)`,
  1. try to find a page `p` in the free list with free space > `|r|`; should this fail, ask the disk space manager to allocate a new page `p`;
  2. record `r` is written to page `p`;
  3. since generally `|r| \ll |p|`, `p` will belong to the list of pages with free space;
  4. a unique `rid` for `r` is computed and returned to the caller.

- For `openScan(f)`,
  1. both page lists have to be traversed.

- A call to `deleteRecord(f, rid)`
  1. may result in moving the containing page from the full to the free page list,
  2. or even lead to page deallocation if the page is completely free after deletion.
10.1.4.3 Directory of pages

▶ An alternative to the linked list approach is to maintain a directory of pages in a file.
▶ The *header page* contains the first page of a chain of directory pages; each entry in a directory page identifies a page of the file:

![Diagram of directory of pages]

Remarks:

▶ |page directory| ≪ |data pages|
▶ Free space management is also done via the directory:
  
  ■ each directory entry is actually of the form \( \langle \text{page addr } p, \text{nfree} \rangle \), where \( \text{nfree} \) indicates the **actual amount of free space** (e.g. in bytes) on page \( p \).
A heap file provides just enough structure to maintain a collection of records (of a table).

The heap file supports **sequential scans** (openScan) over the collection, *e.g.*

```
SELECT  A, B
FROM    R
```

No further operations receive specific support from the heap file.

For queries like

```
SELECT  A, B  
FROM    R  
WHERE   C > 42
```

```
SELECT  A, B
FROM    R
ORDER BY C ASC
```

it would definitely be helpful if the SQL query processor could rely on a particular **organization** of the records in the file for table \( R \).

---

**File organization for table \( R \)**

Which organization of records in the file for table \( R \) could speed up the evaluation of the two queries above?
This section ... presents a comparison of 3 file organizations:

1. files of randomly ordered records (heap files)
2. files sorted on some record field(s)
3. files hashed on some record field(s).

... introduces the index concept:

- A file organization is tuned to make a certain query (class) efficient, but if we have to support more than one query class, we may be in trouble. Consider:

\[
Q \equiv \text{SELECT } A, B, C \text{ FROM } R \text{ WHERE } A > 0 \text{ AND } A < 100
\]

If the file for table \( R \) is sorted on \( C \), this does not buy us anything for query \( Q \).

- If \( Q \) is an important query but is *not* supported by \( R \)’s file organization, we can build a support data structure, an index, to speed up (queries similar to) \( Q \).
10.2.1 Comparison of file organizations

▶ We will now enter a competition in which 3 file organizations are assessed in 5 disciplines:

① **Scan**: fetch all records in a given file.

② **Search with equality test**: needed to implement SQL queries like

\[
\text{SELECT } * \\
\text{FROM } R \\
\text{WHERE } [C = 42].
\]

③ **Search with range selection**: needed to implement SQL queries like (upper or lower bound might be unspecified)

\[
\text{SELECT } * \\
\text{FROM } R \\
\text{WHERE } [A > 0 \text{ AND } A < 100].
\]

④ **Insert** a given record in the file, respecting the file’s organization.

⑤ **Delete** a record (identified by its *rid*), fix up the file’s organization if needed.
10.2.1.1 Cost model

- Performing these 5 database operations clearly involves block I/O, the major cost factor.
- However, we have to additionally pay for CPU time used to search inside a page, compare a record field to a selection constant, etc.
- To analyze cost more accurately, we introduce the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td># of pages in the file</td>
</tr>
<tr>
<td>$r$</td>
<td># of records on a page</td>
</tr>
<tr>
<td>$D$</td>
<td>time needed to read/write a disk page</td>
</tr>
<tr>
<td>$C$</td>
<td>CPU time needed to process a record (e.g., compare a field value)</td>
</tr>
<tr>
<td>$H$</td>
<td>CPU time taken to apply a hash function to a record</td>
</tr>
</tbody>
</table>

Remarks:

- $D \approx 15$ ms
- $C \approx H \approx 0.1 \mu s$
- This is a coarse model to estimate the actual execution time (we do not model network access, cache effects, burst I/O, ...).
Aside: Hashing

A **hashed file** uses a **hash function** $h$ to map a given record onto a specific page of the file.

**Example:** $h$ uses the lower 3 bits of the first field (of type `INTEGER`) of the record to compute the corresponding page number:

- $h (\langle 42, true, "foo" \rangle) \rightarrow 2 \quad (42 = 101010_2)$
- $h (\langle 14, true, "bar" \rangle) \rightarrow 6 \quad (14 = 1110_2)$
- $h (\langle 26, false, "baz" \rangle) \rightarrow 2 \quad (26 = 11010_2)$

- The hash function determines the page number only; record placement inside a page is not prescribed by the hashed file.

- If a page $p$ is filled to capacity, a chain of **overflow** pages is maintained (hanging off page $p$) to store additional records with $h (\langle \ldots \rangle) = p$.

- To avoid immediate overflowing when a new record is inserted into a hashed file, pages are typically filled to 80% only when a heap file is initially (re)organized into a hashed file.
10.2.1.2 Scan

1 Heap file
Scanning the records of a file involves reading all \( b \) pages as well as processing each of the \( r \) records on each page:

\[
\text{Scan}_{\text{heap}} = b \cdot (D + r \cdot C)
\]

2 Sorted file
The sort order does not help much here. However, the scan retrieves the records in sorted order (which can be big plus):

\[
\text{Scan}_{\text{sort}} = b \cdot (D + r \cdot C)
\]

3 Hashed file
Again, the hash function does not help. We simply scan from the beginning (skipping over the spare free space typically found in hashed files):

\[
\text{Scan}_{\text{hash}} = \left(\frac{100}{80}\right) \cdot b \cdot (D + r \cdot C)
\]

In which order does a scan of a hashed file retrieve its records?

© M. Scholl, 2005/06 – Informationssysteme: 10. DBMS Architecture: Managing Data
10.2.1.3 Search with equality test \((A = \text{const})\)

1. **Heap file**
   (a) The equality test is on a *primary key*, (b) the equality test is *not* on a *primary key*:
   
   (a) \(\text{Search}_{\text{heap}} = \frac{1}{2} \cdot b \cdot (D + r \cdot C)\)
   
   (b) \(\text{Search}_{\text{heap}} = b \cdot (D + r \cdot C)\)

2. **Sorted file** (sorted on \(A\))
   We assume the equality test to be on the field determining the sort order. The sort order enables us to use *binary search*:
   
   \[
   \text{Search}_{\text{sort}} = \log_2 b \cdot D + \log_2 r \cdot C
   \]
   
   (If more than one record qualifies, all other matches are stored right after the first hit.)

3. **Hashed file** (hashed on \(A\))
   Hashed files support equality searching best. The hash function directly leads us to the page containing the hit (overflow chains ignored here):
   
   (a) \(\text{Search}_{\text{hash}} = H + D + \frac{1}{2} \cdot r \cdot C\)
   
   (b) \(\text{Search}_{\text{hash}} = H + D + r \cdot C\)

   (All qualifying records live on the same page or, if present, in its overflow chain.)
10.2.1.4 Search with range selection \((A \geq \text{lower} \ \text{AND} \ \ A \leq \text{upper})\)

1. **Heap file**
   Qualifying records can appear anywhere in the file:
   \[
   \text{Range}_{\text{heap}} = b \cdot (D + r \cdot C)
   \]

2. **Sorted file** (sorted on \(A\))
   Use equality search (with \(A = \text{lower}\)), then sequentially scan the file until a record with \(A > \text{upper}\) is found:
   \[
   \text{Range}_{\text{sort}} = \log_2 b \cdot D + \log_2 r \cdot C + \left\lfloor \frac{n}{r} \right\rfloor \cdot D + n \cdot C
   \]
   \((n\) denotes the number of hits in the range\)

3. **Hashed file** (sorted on \(A\))
   Hashing offers no help here as hash functions are designed to scatter records all over the hashed file (e.g., \(h(\langle 7, \ldots \rangle) = 7, \ h(\langle 8, \ldots \rangle) = 0)\):
   \[
   \text{Range}_{\text{hash}} = b \cdot (D + r \cdot C)
   \]
10.2.1.5 Insert

1 Heap file
We can add the record to some arbitrary page (e.g., the last page). This involves reading and writing the page:

\[ \text{Insert}_{\text{heap}} = 2 \cdot D + C \]

2 Sorted file
On average, the new record will belong in the middle of the file. After insertion, we have to shift all subsequent records (in the latter half of the file):

\[ \text{Insert}_{\text{sort}} = \log_2 b \cdot D + \log_2 r \cdot C + \frac{1}{2} \cdot b \cdot (D + r \cdot C) \]

3 Hashed file
We pretend to search for the record, then read and write the page determined by the hash function (we assume the spare 20% space on the page is sufficient to hold the new record):

\[ \text{Insert}_{\text{hash}} = H + \underbrace{D + C + D}_{\text{search}} \]
10.2.1.6 Delete (record specified by its \( rid \))

1. **Heap file**
   If we do not try to compact the file (because the file uses free space management) after we have found and removed the record, the cost is:
   
   \[
   \text{Delete}_{\text{heap}} = \sqrt[2]{D} + C + D
   \]

2. **Sorted file**
   Again, we access the record’s page and then (on average) shift the latter half the file to compact the file:
   
   \[
   \text{Delete}_{\text{sort}} = D + \frac{1}{2} \cdot b \cdot (D + r \cdot C)
   \]

3. **Hashed file**
   Accessing the page using the \( rid \) is even faster than the hash function, so the hashed file behaves like the heap file:
   
   \[
   \text{Delete}_{\text{hash}} = D + C + D
   \]
There is no single file organization that responds equally fast to all 5 operations. This is a dilemma, because more advanced file organizations can really make a difference in speed!

- Performance of **range selections** for files of increasing size 
  \[ (D = 15 \text{ ms}, C = 0.1 \mu s, r = 100, n = 10): \]

- Performance of **deletions** for files of increasing size 
  \[ (D = 15 \text{ ms}, C = 0.1 \mu s, r = 100): \]

There exist **index structures** which offer all the advantages of a sorted file and support insertions/deletions efficiently\(^\text{4}\): B\(^+\) trees.

\(^4\)At the cost of a modest space overhead.
10.2.2 Overview of indexes

If the basic organization of a file does not support a particular operation, we can additionally maintain an auxiliary structure, an index, which adds the needed support.

We will use indexes like guides. Each guide is specialized to accelerate searches on a specific attribute $A$ (or a combination of attributes) of the records in its associated file:

1. Query the index for the location of a record with $A = k$ ($k$ is the search key),
2. The index responds with an associated index entry $k^*$ ($k^*$ contains enough information to access the actual record in the file),
3. Read the actual record by using the guiding information in $k^*$; the record will have an $A$-field with value $k$.\(^5\)

\[^5\text{This is true for so-called “exact match” indexes. In the more general case with “similarity” indexes, the records are not guaranteed to contain the value } k, \text{ they are only “candidates” for having this value.}\]
We can design the **index entries**, i.e., the $k*$, in various ways:

<table>
<thead>
<tr>
<th>Variant</th>
<th>Index entry $k*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$\langle k, \langle \ldots, A = k, \ldots \rangle \rangle$</td>
</tr>
<tr>
<td>b</td>
<td>$\langle k, \text{rid} \rangle$</td>
</tr>
<tr>
<td>c</td>
<td>$\langle k, [\text{rid}_1, \text{rid}_2, \ldots] \rangle$</td>
</tr>
</tbody>
</table>

**Remarks:**

- With variant a, there is *no* need to store the data records in addition the index—the index itself is a special file organization.
- If we build multiple indexes for a file, at most one of these should use variant a to avoid redundant storage of records.
- Variants b and c use rid(s) to point into the actual data file.
- Variant c leads to less index entries if multiple records match a search key $k$, but index entries are of variable length.
The data file contains \( \langle \text{name, age, sal} \rangle \) records, the file itself (index entry variant ③) is hashed on field \( \text{age} \) (hash function \( h1 \)).

The index file contains \( \langle \text{sal, rid} \rangle \) index entries (variant ⑤), pointing into the data file.

This file organization + index efficiently supports equality searches on the \( \text{age and sal} \) keys.

\[ h1(\text{age}) = \begin{cases} 0 & \text{if age} = 0 \\ 1 & \text{if age} = 1 \\ 2 & \text{if age} = 2 \end{cases} \]

\[ h2(\text{sal}) = \begin{cases} 0 & \text{if sal} = 0 \\ 3 & \text{if sal} = 4 \\ \text{otherwise} & \end{cases} \]

---

\(^{6}\)① - ③ refer to the index lookup scheme at the beginning of this section.
10.2.3 Properties of indexes

10.2.3.1 Clustered vs. unclustered indexes

- Suppose, we have to support **range selections** on records such that \( \text{lower} \leq A \leq \text{upper} \) for field \( A \).
- If we maintain an index on the \( A \)-field, we can
  1. query the index once for a record with \( A = \text{lower} \), and then
  2. **sequentially scan the data file** from there until we encounter a record with field \( A > \text{upper} \).
- This will work provided that the **data file is sorted on the field** \( A \):

![Diagram showing B+ tree index file, data file, and index entries](image-url)
If the *data file* associated with an index *is sorted on the index search key*, the index is said to be *clustered*.

In general, the cost for a range selection grows tremendously if the index on $A$ is *unclustered*. In this case, proximity of index entries does *not* imply proximity in the data file.

- As before, we can query the index for a record with $A = \text{lower}$. To continue the scan, however, we have to *revisit the index entries* which point us to *data pages scattered* all over the data file:

![Diagram of a B+ tree index and its index entries pointing to data records scattered in the data file.]

**Remarks:**

- If the index entries ($k^*$) are of variant @, the index is obviously clustered by definition.
- A data file can have *at most one* clustered index (but any number of unclustered indexes).
10.2.3.2 Dense vs. sparse indexes

▶ A **clustered index** comes with more advantages than the improved speed for range selections presented above. We can additionally design the index to be **space efficient**:

- To keep the size of the index file small, we maintain *one index entry \( k^* \) per data file page* (not one index entry per data record). Key \( k \) is the smallest search key on that page.
- Indexes of this kind are called **sparse** (otherwise indexes are **dense**).

▶ To search a record with field \( A = k \) in a sparse \( A \)-index, we

1. locate the largest index entry \( k' \) such that \( k' \leq k \), then
2. access the page pointed to by \( k' \), and
3. scan this page (and the following pages, if needed) to find records with \( \langle \ldots, A = k, \ldots \rangle \).

Since the data file is clustered (i.e., sorted) on field \( A \), we are guaranteed to find matching records in the proximity.
Example:

Again, the data file contains \( \langle name, age, sal \rangle \) records. We maintain a **clustered sparse index** on field \( name \) and an **unclustered dense index** on field \( age \). Both use index entry variant \( \mathcal{O} \) to point into the data file:

\[
\begin{align*}
\text{Sparse Index} & \quad \text{Dense Index} \\
\text{on Name} & \quad \text{on Age}
\end{align*}
\]

- Ashby, 25, 3000
- Basu, 33, 4003
- Cass, 50, 5004
- Daniels, 22, 6003
- Jones, 40, 6003
- Smith, 44, 3000
- Tracy, 44, 5004

**Remarks:**

- Sparse indexes need 2–3 orders of magnitude less space than dense indexes.
- We cannot build a sparse index that is unclustered (i.e., there is at most one sparse index per file).

**SQL queries and index exploitation**

How do you propose to evaluate query \( \text{SELECT MAX(age) FROM employees} \)? How about \( \text{SELECT MAX(name) FROM employees} \)?
10.3  Tree-Structured Indexing

This section discusses two index structures which especially shine if we need to support range selections (and thus sorted file scans): ISAM files and \(B^+\) trees.

Both indexes are based on the same simple idea which naturally leads to a tree-structured organization of the indexes. (Hash indexes are covered later in this chapter.)

\(B^+\) trees refine the idea underlying the rather static ISAM scheme and add efficient support for insertions and deletions.

10.3.1  Indexed sequential access method (ISAM)

Remember: range selections on sorted files may use binary search to locate the lower range limit as a starting point for a sequential scan of the file (until the upper range limit is reached).

ISAM considerably improves on binary search to support navigation in the data file.
To support range selections\(^7\) on a field \(A\):

1. In addition to the \(A\)-sorted data file, maintain an **index file** with the following entries:

![Index Entry Diagram](image)

2. The index contains **index entries (records)** \(k_i^*\), key \(k_i\) is the first (\(i.e.,\) the minimal) \(A\) value on data file page \(p_i\):

\[
k_i^* = \langle k_i, \text{pointer to } p_i \rangle
\]

**N.B.**:

- In the index file, the \(k_i\) serve as **separators** between the contents of page \(p_{i-1}\) and \(p_i\).
- We additionally take care that \(k_{i-1} \lt k_i\) \((i = 2 \ldots m)\).

We obtain a **one-level ISAM structure**:

![One-Level ISAM Structure Diagram](image)

\(^7\)ISAM can speed up equality searches as well, see below.
To support a **range selection** like

```sql
SELECT  *
FROM    R
WHERE   lower ≤ A ≤ upper
```

count a binary search *on the index file* for a key of value `lower`, then start a *sequential scan* of the data file from the page pointed to by the index entry (scan until the `A`-field exceeds `upper`).

**Equality searches** are implemented accordingly.

- The size of the index file is likely to be *much* smaller than the data file size.
- For really large data files, however, even the index file might be too large to quickly search in.

**Main idea of the ISAM structure**

*Recursively* apply the index creation step, *i.e.*, treat the index level like the data file and add an additional index layer on top. Repeat, until the top-most index layer fits on a single page (the *root page*).
This recursive index creation scheme leads to a **tree-structured** hierarchy of index levels:

![Diagram of tree-structured hierarchy]

**Remarks:**

- Each tree node corresponds to one file page (disk block).
- To create the ISAM tree structure, proceed **bottom up**:
  1. Sort the data file (*i.e.*, the leaf pages) on the search key value,
  2. then create the index leaf level and proceed upwards.
- The upper index levels of the ISAM tree remain **static**: insertions and deletions in the data file do *not* affect the upper tree layers.
- If an index leaf page overflows, maintain a chain of **overflow pages** for that **primary leaf page**.
- Search performance in the ISAM tree degrades over time if insertions (deletions) occur frequently.
Example (ISAM index pages shown only, each page can hold two index entries):

1. After insertion of data records with keys 23, 48, 41, and 42:
After deletion of data records with keys 42, 51, and 97:

![ISAM Structure Diagram]

**NB:**
- The non-leaf levels of the ISAM structure have *not* been touched at all by the data file updates.
  - This may lead to index key entries which do not appear in the index leaf level (key value 51 above).

💡 **Orphaned index key entries…**

Does an index key entry like 51 above lead to problems during index searches?

- To preserve the *separator property* of the index key entries, we have to maintain overflow chains.
- As a result, the ISAM structure may **lose balance** after heavy updating.
Regardless of these deficiencies of the ISAM tree, ISAM-based searching is the most efficient \textit{order-aware} index structure we have met so far:

- Let $N$ be the number of pages in the data file, and let $F$ denote the \textbf{fan-out} of the ISAM tree, \textit{i.e.} the maximum number of children per index node (the fan-out in the previous example is 3).
- When index searching starts, the search space is of size $N$. With the help of the root page we are guided into a index subtree of size

$$N \cdot \frac{1}{F}.$$ 

- As we step down the tree, we repeatedly reduce the search space by a factor of $F$:

$$N \cdot \frac{1}{F} \cdot \frac{1}{F} \cdot \frac{1}{F} \cdots.$$ 

- Index searching ends after $s$ steps when the search space has been reduced to size 1 (\textit{i.e.} we have reached the leaf level and found a pointer to the page which contains the wanted record):

$$N \cdot \left(\frac{1}{F}\right)^s \overset{!}{=} 1 \iff s = \log_F N.$$  

\begin{itemize}
  \item Since $F \gg 2$ (typically $F \approx 1000$) this is significantly faster than access via binary search ($\log_2 N$).
\end{itemize}
Example:
With \( F = 1000 \), an ISAM tree of height 3 can index a file of one billion (\( = 10^9 \)) pages \( (i.e., \) 3 page I/O operations are sufficient to locate the data file page wanted).

The presence of overflow pages, however, can easily spoil this impressive I/O behavior.
Unbalanced insertions in the data file can ultimately lead to linear search in long overflow chains.

10.3.2 \( \text{B}^+ \) trees: a dynamic index structure

- The \( \text{B}^+ \) tree index structure is derived from the ISAM idea and represents the successful attempt to eat the cake and have it, too:
  1. Search performance is only dependant on the height of the \( \text{B}^+ \) tree (because of high fan-out \( F \), the height of \( \text{B}^+ \) trees is rarely \( > 3 \)).
  2. No overflow chains develop, the \( \text{B}^+ \) tree remains balanced all the time,
  3. \( \text{B}^+ \) trees offer efficient insert/delete procedures, the underlying data file can grow/shrink dynamically,
  4. \( \text{B}^+ \) tree nodes (despite the root page) are guaranteed to have a minimum occupancy of 50% (typically 67%).
10.3.2.1 Format of a $B^+$ tree node

- In a $B^+$ tree, **non-leaf nodes** use the same internal layout as in the ISAM case:

```
  index entry
  p_0  k_1  p_1  k_2  p_2  \ldots  \ldots  k_m  p_m
```

- The minimum and maximum number of entries in a node is bounded by the **order** $d$ of the $B^+$ tree:

\[
d \leq m \leq 2 \cdot d
\]

(*i.e.*, no node has less than 50\% entries actually occupied—with the exception of the root node which is allowed to have $1 \leq m \leq 2 \cdot d$ entries).

- Non-leaf nodes with $m$ entries contain $m+1$ pointers to child nodes. Pointer $p_i$ ($i = 1 \ldots m-1$) points to a subtree in which all key values $k$ are such that

\[
k_i \leq k < k_{i+1}
\]

.pointer $p_0$ points to a subtree with key values $< k_1$, $p_m$ points to a subtree with key values $\geq k_m$).
Unlike the ISAM case, B+ tree entries in leaf nodes point to data records, not data pages. A leaf node entry with key value $k$ is denoted as $k^*$ as usual.

Note that we can use all index entry variants (a) . . . (c) to implement the leaf entries:

- for variant (a), the B+ tree represents the index as well as the data file itself (i.e., in a leaf node, the $p_i$ are the actual data records):
  \[ k_i^* = \langle k_i, \langle \ldots \rangle \rangle. \]

- for variants (b) and (c), the B+ tree lives in a file distinct from the actual data file; the $p_i$ are rids pointing into the data file:
  \[ k_i^* = \langle k_i, \text{rid} \rangle. \]

Since B+ trees are dynamic structures whose leaf level may grow/shrink over lifetime, leaf level nodes are chained together in a doubly linked list, the so-called sequence set, to support range queries efficiently:
We will now proceed and discuss the 3 basic operations on B+ trees, **searching**, **insertion**, and **deletion** in considerable detail.

To keep matters simple, we defer the treatment of **duplicate** key values in the data file: for the next few sections, we assume key values to be **unique**.

### 10.3.2.2 Search

Searching a record with key \( k \) in a B+ tree is no different from the ISAM case. Rough sketch:

1. start the search on the B+ tree root page,
2. if the current page is a leaf page we are done (we have found the page containing \( k^* \)),
3. else, for the current page, determine \( i \), such that \( k_i \leq k < k_{i+1} \), descend into subtree pointed to by \( p_i \), goto ②.

**Example** (B+ tree of order \( d = 2 \)):
Algorithm: search \((k)\)
Input: search key value \(k\)
Output: pointer to \(B^+\) tree page containing potential hit(s)

\[
\text{return } \text{tree_search}(\text{root}, k); \quad \text{// root denotes the root page of the } B^+ \text{ tree}
\]

Algorithm: tree_search \((p, k)\)
Input: current page \(p\), search key value \(k\)
Output: pointer to \(B^+\) tree page containing potential hit(s)

\[
\text{if } \text{leaf}(p) \text{ then } \\
\quad \text{return } p; \quad \text{// layout of } p:\n\]

\[
\text{else } \\
\quad \text{if } k < k_1 \text{ then } \\
\quad \quad \text{return } \text{tree_search}(p_0, k); \\
\quad \text{else } \\
\quad \quad \text{if } k \geq k_m \text{ then } \\
\quad \quad \quad \text{return } \text{tree_search}(p_m, k); \\
\quad \quad \text{else } \\
\quad \quad \quad \text{find } i \text{ such that } k_i \leq k < k_{i+1}; \\
\quad \quad \quad \text{return } \text{tree_search}(p_i, k);
\]

\[\text{NB: To complete the search, we have to locate } k^* \text{ on the page returned by search}(k); \text{ this might fail.}\]
10.3.2.3 Insert

Remember that B\(^+\) trees remain **balanced\(^8\)** no matter which update operations we perform. Insertions and deletions have to preserve this invariant.

The basic principle of B\(^+\) tree insertion is simple:

1. To insert a record with key \(k\), call \(search(k)\) to find the page \(p\) to hold the new record. Let \(m\) denote the number of entries on \(p\).
2. If \(m < 2 \cdot d\) (i.e., there is capacity left on \(p\)), store \(k^*\) in page \(p\).
   
   **Otherwise** . . . ?

- We must *not* start an overflow chain hanging off \(p\): this would violate the balancing property.
- We want the cost for \(search(k)\) to be dependant on tree height only, so placing \(k^*\) somewhere else (even near \(p\)) is *no* option either.

\(^8\)All paths from the B\(^+\) tree root to any leaf are of equal length.
The $B^+$ tree approach:

1. To insert a record with key $k$, call $\text{search}(k)$ to find the page $p$ to hold the new record. Let $m$ denote the number of entries on $p$.
2. If $m < 2 \cdot d$ (i.e., there is capacity left on $p$), store $k*$ in page $p$.
   Otherwise, split $p$ into pages $p$ and $p'$ and distribute the $2 \cdot d$ entries evenly between $p$ and $p'$.
   Then adjust the upper tree layers to incorporate the new page $p'$.

Example:

1. Insert record with key $k = 8$ into the following $B^+$ tree:

   ![B+ Tree Diagram]

2. The left-most leaf page $p$ has to be split. Entries 2*, 3* remain on $p$, entries 5*, 7*, and 8* (new) go on new page $p'$. 
3. Pages $p$ and $p'$ are shown below. Key $k' = 5$, the **new separator** between pages $p$ and $p'$, has to be **inserted into the parent** of $p$ and $p'$ **recursively**:

```
13 17 24 30
2* 3* 5* 7* 8*
```

- Note that, after such a **leaf split**, the new separator key $k' = 5$ is **copied up** the tree: the entry $5*$ itself has to remain in its leaf page.

4. The insertion process is propagated upwards the tree: inserting key $k' = 5$ into the parent leads to a **non-leaf node split** (the $2 \cdot d + 1$ keys and $2 \cdot d + 2$ pointers make for two new non-leaf nodes and a **middle key** which we propagate further up for insertion):

```
17
5 13
```

- Note that, for a **non-leaf node split**, we can simply **push up** the middle key (17). Contrast this with a leaf node split.
Since the split node was the root node, we create a new root node which holds the pushed up middle key only:

Splitting the old root and creating a new root node is the only situation in which the B\(^+\) tree height increases. The B\(^+\) tree thus remains balanced. We cannot guarantee the minimum occupancy of \(d\) entries for the new root, though.

Of course, node insertion propagation stops as soon as a node with sufficient capacity (i.e., < \(2 \cdot d\) entries) is encountered.

**Further key insertions**

How does the insertion of records with keys \(k = 23\) and \(k = 40\) alter the B\(^+\) tree?
Algorithm: $\text{insert} \left( p, k^* \right)$

Input: current page $p$, entry $k^*$ to be inserted

Output: entry propagated upwards the tree (NULL if no further propagation)

$m \leftarrow \#\text{entries}(p)$;
if $\neg \text{leaf}(p)$ then
  on $p$, find $i$ such that $k_i \leq k < k_{i+1}$;
  $n \leftarrow \text{insert}(p_i, k^*)$;
  if $n = \text{NULL}$ then
    return NULL;
  else
    if $m < 2 \cdot d$ then
      insert $n$ into $p$;
      return NULL;
    else
      // $m + 1 = d + \left( \frac{1}{k^*} + d \right)$
      split $p$ into $p'$ and new page $p''$, first $d$ keys and $d + 1$ pointers stay on $p$, last $d$ keys and $d + 1$ pointers go to $p'$;
      $n \leftarrow \langle \text{middle key } k', \text{addr}(p') \rangle$;
      if $\text{root}(p)$ then
        $r \leftarrow \text{new empty node}$;
        $\text{root}(r) \leftarrow \text{true}$;
        insert $\text{addr}(p)$ into $r$; // as $p_0$
        insert $n$ into $r$;
        return NULL;
      else
        return $n$;
    else
      // $p$ is a leaf node
      if $m < 2 \cdot d$ then
        insert $k^*$ into $p$;
        return NULL;
      else
        // $m + 1 = 2 \cdot d + 1$
        split $p$ into $p'$ and new page $p''$, first $d$ entries stay on $p$, last $d + 1$ entries go to $p'$;
        $n \leftarrow \langle \text{smallest value } k', \text{addr}(p') \rangle$;
        return $n$;
We can further **improve the average occupancy** of B+ tree using a technique called **redistribution**:

- Suppose we are trying to insert a record with key $k = 6$ into the B+ tree below:

![B+ tree diagram]

- The left-most leaf is full already, its right **sibling** still has capacity, however.
- Here, we can avoid growing the tree by **redistributing** entries between siblings (entry 7* moved into right sibling):

![Updated B+ tree diagram]

- **NB**: we have to update the parent node (new separator 7) to reflect the redistribution.

- Inspecting one or both neighbor(s) of a B+ tree node involves additional I/O operations.
- Actual implementations often use redistribution on the leaf level only (because the sequence set page chaining gives direct access to both sibling pages).
Redistribution makes a difference
Insert a record with key \( k = 30 \)

1. without redistribution,
2. using leaf level redistribution

into the \( B^+ \) tree shown below. How does the tree change?

10.3.2.4 Delete

The principal idea to implement \( B^+ \) tree deletion comes as no surprise:

1. To delete a record with key \( k \), use \( \text{search}(k) \) to locate the leaf page \( p \) containing the record. Let \( m \) denote the number of entries on \( p \).
2. If \( m > d \) then \( p \) has sufficient occupancy: simply delete \( k^* \) from \( p \) (if \( k^* \) is present on \( p \) at all).
   Otherwise ... ?
① Delete record with key \( k = 19 \) (i.e., entry 19*) from the following \( \mathbb{B}^+ \) tree:

![B+ tree diagram](image)

② A call to `search(19)` leads us to leaf page \( p \) containing entries 19*, 20*, and 22*. We can safely remove 19* since \( m = 3 > 2 \) (no page underflow in \( p \) after removal).

③ Subsequent deletion of 20*, however, lets \( p \) underflow (\( p \) has minimal occupancy of \( d = 2 \) already).

We now use redistribution and borrow entry 24* from the right sibling \( p' \) of \( p \) (since \( p' \) hosts 3 > 2 entries, redistribution won’t let \( p' \) underflow).

The smallest key value on \( p' \) (27) is the new separator of \( p \) and \( p' \) in their common parent:

![Updated B+ tree diagram](image)
④ We continue and delete entry 24* from \( p \). Redistribution is no option now (sibling \( p' \) only has minimal occupancy of \( d = 2 \)).

\[ \text{Wonever: we now have}\, m_p + m_{p'} = 1 + 2 < 2 \cdot d. \]

\( B^+ \) tree deletion thus **merges leaf nodes** \( p \) and \( p' \).

Move entries 27*, 29* from \( p' \) to \( p \), then delete page \( p' \):

\[ \text{NB: the separator 27 between } p \text{ and } p' \text{ is no longer needed and thus discarded (recursively deleted)} \text{ from the parent.} \]

⑤ The parent of \( p \) experiences underflow. Redistribution is no option, so we **merge with left non-leaf sibling**.

After merging we have \( d + (d - 1) \) keys and \( d + 1 + d \) pointers on the merged page:
The missing key value, namely the separator of the two nodes (17), is **pulled down** (and thus deleted) from the parent to form the complete merged node. The → pointer in the parent node is no longer needed and thrown away. Contrast this with a leaf node merge.

Since we have now deleted the last remaining entry in the root, we discard the root (and make the merged node the new root):

This is the *only* situation in which the B⁺ tree height decreases. The B⁺ tree thus remains balanced.
We have now seen leaf node merging and redistribution as well as non-leaf node merging. The remaining case of non-leaf node redistribution is straightforward:

- Suppose during deletion we encounter the following intermediary B⁺ tree:

![Diagram of a B⁺ tree with a non-leaf node underflowed](image)

- The non-leaf node with entry 30 underflowed. Its left sibling has two entries (17 and 20) to spare.
- We redistribute entry 20 by "rotating it through" the parent (and push down the former parent entry 22):

![Diagram of the tree after redistribution](image)
Algorithm: \( \text{delete} \ (p, k) \)

Input: current page \( p \), key value \( k \) to be deleted

Output: key value to be deleted in \( p \)’s parent
\[(\text{NULL} \text{ if no further deletion in parent})\]

\[m \leftarrow \# \text{entries}(p);\]

\[\text{if} \ \neg \text{leaf}(p) \text{ then}\]

\[// \ p \text{ is a non-leaf node}\]

\[\text{find } i \text{ such that } k_i \leq k < k_{i+1};\]

\[n \leftarrow \text{delete}(p_i, k);\]

\[\text{if } n = \text{NULL} \text{ then}\]

\[\quad \text{return } \text{NULL};\]

\[\text{else}\]

\[\quad \text{remove entry with key } n \text{ from } p;\]

\[\quad \text{if } \text{root}(p) \land m = 1 \text{ then} \]

\[\quad \quad // \text{last entry in root deleted, determine new root}\]

\[\quad \quad \text{root(} \text{addr}(p_0) \text{)} \leftarrow \text{true};\]

\[\quad \quad \text{delete } p;\]

\[\quad \quad \text{return } \text{NULL};\]

\[\text{if } m > d \text{ then} \]

\[\quad \text{return } \text{NULL};\]

\[\text{else}\]

\[\quad // \text{underflow in } p\]

\[\quad p' \leftarrow \text{sibling}(p); \quad // \text{wlog: } p' \text{ right sibling of } p\]

\[\quad \text{if } \# \text{entries}(p') > d \text{ then} \]

\[\quad \quad // \text{non-leaf node redistribution}\]

\[\quad \quad \text{move smallest entry of } p' \ (= \langle k', p_{0}' \rangle) \text{ into } p;\]

\[\quad \quad \text{swap(} \text{separator}(p, p'), \text{key value } k' \text{ in } p);\]

\[\quad \quad \text{return } \text{NULL};\]

\[\quad \text{else}\]

\[\quad \quad // \text{merge non-leaf nodes } p, p'\]

\[\quad \quad \text{insert } \text{separator}(p, p') \text{ into } p;\]

\[\quad \quad \text{move all entries from } p' \text{ to } p;\]

\[\quad \quad \text{delete } p';\]

\[\quad \quad \text{return } \text{separator}(p, p');\]

\[\text{else}\]

\[\quad \text{...}\]

\[\quad // \text{leaf node handling: see next slide}\]

\[\quad \text{...}\]
Algorithm: \( \text{delete} \left( p, k \right) \)

Input: current page \( p \), key value \( k \) to be deleted

Output: key value to be deleted in \( p \)'s parent

\((\text{NULL if no further deletion in parent})\)

\[
m \leftarrow \#\text{entries}(p);
\]

if \(-\text{leaf}(p)\) then

\[
\ldots
\]

// non-leaf node handling: see previous slide

\[
\ldots
\]

else

// \( p \) is a leaf node

\[
\text{if } k^* \text{ found on } p \text{ then}
\]

\[
\quad \text{remove } k^* \text{ from } p;
\]

\[
\text{else}
\]

\[
\quad \text{return NULL;}
\]

\[
\text{if } m > d \text{ then}
\]

\[
\quad \text{return NULL;}
\]

\[
\text{else}
\]

// underflow in \( p \)

\[
p' \leftarrow \text{sibling}(p); \quad \text{\( wlog: \) } p' \text{ right sibling of } p
\]

\[
\text{if } \#\text{entries}(p') > d \text{ then}
\]

// leaf node redistribution

\[
\quad \text{move entry from } p' \text{ to } p;
\]

\[
\quad \text{separator}(p, p') \leftarrow \text{smallest key value on } p';
\]

\[
\quad \text{return NULL;}
\]

\[
\text{else}
\]

// merge leaf nodes \( p, p' \)

\[
\quad \text{move all entries from } p' \text{ to } p;
\]

\[
\quad \text{delete } p';
\]

\[
\quad \text{return separator}(p, p');
\]

\( \text{separator}(p, p') \) represents the separating key value \( k \) in the common parent node of siblings \( p \) and \( p' \).

\( \#\text{entries}(p) \) computes the number of actually occupied entries in \( \B^+ \) tree node \( p \).

\( \text{swap}(x, y) \) exchanges the values of \( x \) and \( y \).
10.3.2.5 Duplicates

As discussed here, the B+ tree search, insert, and delete procedures ignore the presence of duplicate key values.

Often this is a reasonable assumption:

- If the key field is a **primary key** for the data file (i.e., for the associated relation), the search keys $k$ are unique by definition.

Other approaches to make B+ trees aware of duplicates are:

1. Use variant © (see Section 10.2) to represent the index data entries $k*$:

   $$ k* = \langle k, [rid_1, rid_2, \ldots] \rangle $$

   - Each duplicate record with key field $k$ makes the list of rids grow.
   - B+ tree search and maintenance routines largely unaffected. Index data entry size varies, however (this affects the B+ tree **order** concept).

2. Treat duplicate key values like any other value in insert and delete. This affects the search procedure.
We now turn to a different family of index structures: hash indexes. Hash indexes are unbeatable when it comes to equality selections, e.g.

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad R \\
\text{WHERE} & \quad A = k
\end{align*}
\]

If we carefully maintain the hash index while the underlying data file (for relation \( R \)) grows or shrinks, we can answer such an equality query using a single I/O operation. (More precisely: it is rather easy to achieve an average of 1.2 I/Os.)

Other query types, like joins, internally initiate a whole flood of such equality tests.

Hash indexes provide no support for range searches, however (hash indexes are also known as scatter storage).

In a typical DBMS, you will find support for B\(^+\) trees as well as hash-based indexing structures.
In a $B^+$ tree world, to locate a record with key $k$ means to compare $k$ with other keys organized in a (tree-shaped) search data structure.

Hash indexes use the bits of $k$ itself (independent of all other stored records and their keys) to find (i.e., compute) the location of the associated record.

We will only look into static hashing to illustrate the basic ideas behind hashing.

- Static hashing does not handle updates well (much like ISAM).
- Later, dynamic hashing schemes have been proposed, e.g. extendible and linear hashing, which refine the hashing principle and adapt well to record insertions and deletions.

To build a static hash index for an attribute $A$ we need to

1. allocate an area of $N$ (successive) disk pages, the so-called primary buckets (or the hash table),
2. in each bucket, install a pointer to a chain of overflow pages (initially, set this pointer to nil),
3. define a hash function $h$ with range $[0 \ldots N - 1]$ (the domain of $h$ is the type of $A$, e.g. $h : \text{INTEGER} \rightarrow [0 \ldots N - 1]$ if $A$ has the SQL type INTEGER).
The resulting setup looks like this:

- A primary bucket and its associated chain of overflow pages is referred to as a bucket (above).
- Each bucket contains data entries \( k^* \) (implemented using any of the variants \( a \ldots c \), see Section 10.2).

To perform \( h\text{search}(k) \) (or \( h\text{insert}(k)/h\text{delete}(k) \)) for a record with key \( A = k \),

1. apply hash function \( h \) to the key value, i.e., compute \( h(k) \),
2. access the primary bucket page with number \( h(k) \),
3. then search (insert/delete) the record on this page or, if necessary, access the overflow chain of bucket \( h(k) \).
If we are lucky or (somehow) avoid chains of overflow pages altogether,
- \( hsearch(k) \) needs one I/O operation,
- \( hinsert(k) \) and \( hdelete(k) \) need two I/O operations.

At least for static hashing, **overflow chain management** is important:
- Generally, we do not want hash function \( h \) to avoid collisions, i.e.,
  \[
  h(k) = h(k') \quad \text{even if} \quad k \neq k'
  \]
  (otherwise we would need as many primary bucket pages as different key values in the data file, or even in \( A \)'s domain).
- However, we do want \( h \) to scatter the domain of the key attribute **evenly** across \([0 \ldots N - 1]\) (to avoid the development of extremely long overflow chains for few buckets).
- Such “good” hash functions are hard to discover, unfortunately (see next slide).
The birthday paradox

If you consider the people in a group as the domain and use their birthday as hash function \( h \) \((i.e., h : Person \rightarrow [0 \ldots 364])\), chances are already > 50% that two people share the same birthday (collision) if the group has \( \geq 23 \) people.

Check yourself:

1. Compute the probability that \( n \) people all have different birthdays:

\[
different\_birthday(n):
\begin{align*}
\text{if } n = 1 & \text{ then}\n\quad \text{return } 1 \\
\text{else} & \\
\quad \text{return } different\_birthday(n - 1) \times \frac{365 - (n - 1)}{365} \\
\end{align*}
\]

- probability that \( n - 1 \) persons have different birthdays
- probability that \( n^{th} \) person has birthday different from first \( n - 1 \) persons

2. ... or try to find birthday mates at the next larger party.
If key values would be purely **random** we could arbitrarily extract a few bits and use these for the hash function. Real key distributions found in DBMS are far from random, though.

Fairly good hash functions may be found using the following two simple approaches:

1. **By division.** Simply define

   \[ h(k) = k \mod N \, . \]

   This guarantees the range of \( h(k) \) to be \([0 \ldots N - 1]\). **N.B.** If you choose \( N = 2^d \) for some \( d \) you effectively consider the least \( d \) bits of \( k \) only. **Prime numbers** were found to work best for \( N \).

2. **By multiplication.** Extract the fractional part of \( Z \cdot k \) (for a specific \( Z \))\(^9\) and multiply by hash table size \( N \) (\( N \) is arbitrary here):

   \[ h(k) = \left\lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \right\rfloor \, . \]

   However, for \( Z = Z'/2^w \) and \( N = 2^d \) (\( w \): number of bits in a CPU word) we simply have

   \[ h(k) = \text{msb}_d(Z' \cdot k) \]

   where \( \text{msb}_d(x) \) denotes the \( d \) most significant (leading) bits of \( x \) (e.g., \( \text{msb}_3(42) = 5 \)).

**Non-numeric key domains?**

How would you hash a non-numeric key domain (e.g., hashing over a CHAR(·) attribute)?

\(^9\) \( Z = (\sqrt{5} - 1)/2 \approx 0.6180339887 \) is a good choice. See Don E. Knuth, “*Sorting and Searching*”.
Clearly, if the underlying data file grows, the development of overflow chains spoils the otherwise predictable hash I/O behaviour (1–2 I/Os).

Similarly, if the file shrinks significantly, the static hash table may be a waste of space (data entry slots in the primary buckets remain unallocated).

In the worst case, a hash table can degrade into a linear list (one long chain of overflow buckets).

Dynamic hashing schemes have been devised to overcome this problem by adapting the hash function and by combining the use of hash functions and directories guarding the way to the data records.
Execution of database queries issued in a declarative language (such as SQL) proceeds in a number of steps through different DBMS components:

1. **SQL query** → syntactic and semantic analysis → parse tree/operator tree → query optimization → execution plan → code generation → “executable” code → run-time query processor → query result

- Algebraic optimization;
- Selection of join algorithms;
- Access path selection

- Immediate execution;
- Deferred execution (store “access module”)
Remarks:

- **Syntactic and semantic analysis:**
  - which relations/attributes are mentioned in the query?
  - what operators to apply?

- **Query optimization:**
  (cf. next slide)

- **Code generation:**
  - executable machine code, or
  - executable code intermixed with interpreted parts, or
  - “executable operator tree” → interpreted execution model

- **Run-time query processor:**
  - for deferred execution:\(^\text{10}\)
    - check whether access module is still valid:\(^\text{11}\) reject or dynamically recompile query, if necessary (→ DB2)
    - generic routines for accessing catalog data, synchronization, recovery, ...
  - for immediate execution:
    - see above, but check for access module validity is not necessary here.
    - interpreter for intermediate code (“executable operator tree”).

\(^\text{10}\) for repeated execution (“repetitive/canned queries”) compilation more efficient than pure interpretation.

\(^\text{11}\) “early binding”, i.e., compile-time binding, w.r.t. DB schema and available access paths as opposed to “late binding”, i.e., run-time binding, in the case of interpretation.
Query Optimization

Optimization on two distinct levels:

■ **“algebraic”**: Query is translated into (extended) relational algebra, optimization using equivalence rules (cf. Chapter 2) yielding a “simpler” algebraic expression:
  ▶ especially important: elimination of redundant joins, selection “push-down”
  ▶ additional goals: “push-down” of projections (why?)

■ **“non-algebraic”**: depending on currently available storage structures, sort orders, indexes, and statistical data, perform cost estimation for different execution plans and select cheapest.

■ **important ingredient**: adequate cost model

Optimization plays a crucial role, particularly in relational DBMSs

■ performance

■ guarantees data independence of application programs, by separating logical and physical schemas (cf. ANIS 3-level schema approach). → DBMS has to do the optimization well, because otherwise users/application programmers would insist on “tuning” knobs!
10.6 Literature


