Universität
Konstanz

Database System Architecture and Implementation

Module 2

File Organizations and Indexing

November 6, 2018

Orientation

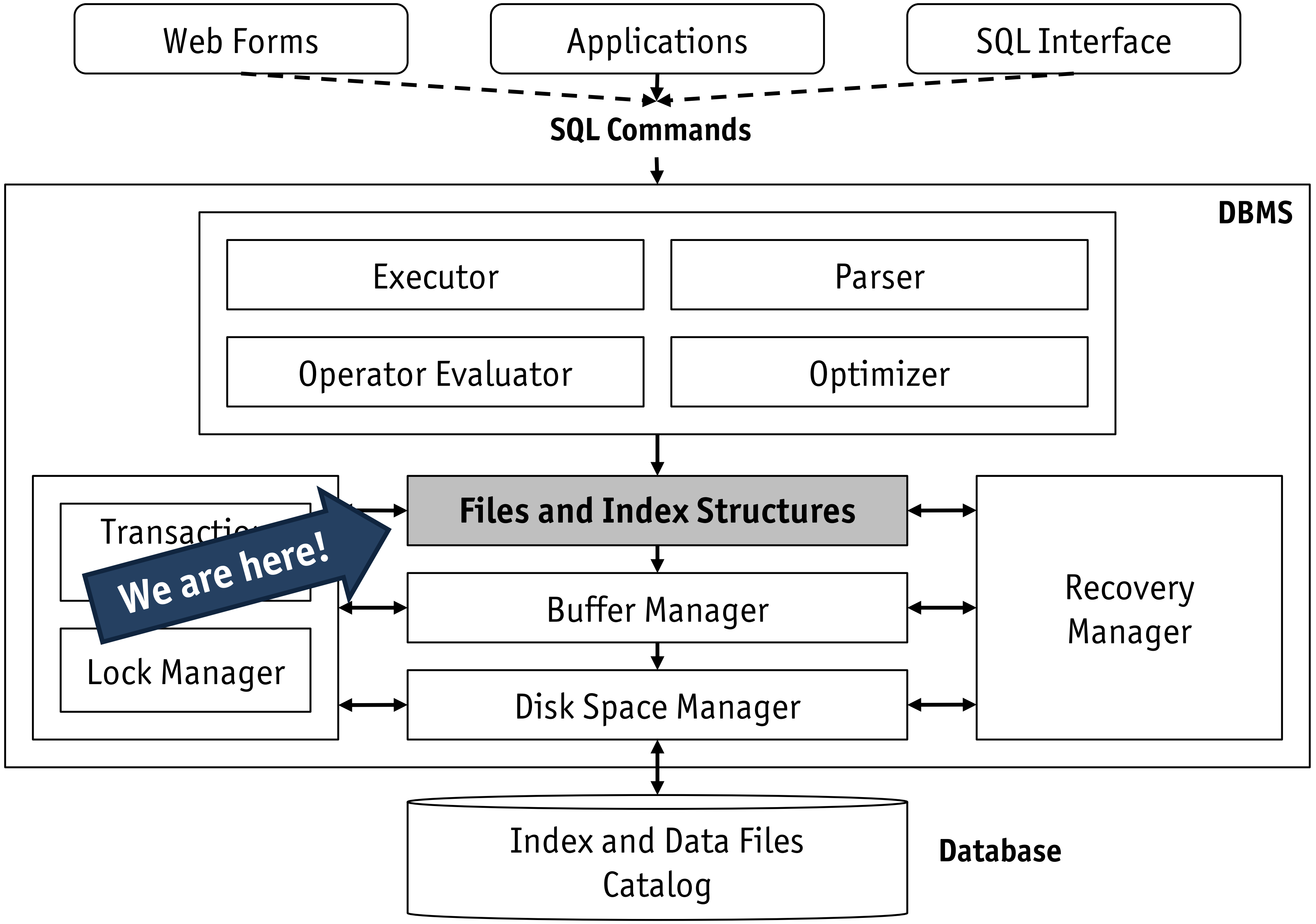


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Recall Heap Files

- Heap files provide **just enough** structure to maintain a collection of records (of a table)
- The heap file **supports sequential** (`openScan(·)`) over the collection

SQL query leading to a sequential scan

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

- **No other operations** get specific support from heap files

Systematic File Organization

SQL queries calling for systematic file organization

```
SELECT A, B
FROM   R
WHERE  C > 45
```

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

- For the above queries, it would definitely be helpful if the SQL query processor could rely on a particular file organization of the records in the file for table **`R`**

Exercise

Which organization of records in the file for table **`R`** could speed up the evaluation of **both** queries above?

Module Overview

- Three different **file organizations**
  1. files containing **randomly ordered** records (heap files)
  2. files **sorted** on one or more record fields
  3. files **hashed** on one or more record fields
- Comparison of file organizations
  - simple **cost model**
  - application of cost model to **file operations**
- Introduction to **index** concept
  - clustered vs. unclustered indexes
  - dense vs. sparse indexes

Comparison of File Organizations

- Competition of three file organizations in five disciplines
  1. **scan:** read all records in a give file
  2. **search with equality test**
  3. **search with range selection** (upper or lower bound may be unspecified)
  4. **insert** a given record in the file, respecting the file's organization
  5. **delete** a record (identified by its *rid*), maintain the file's organization

SQL queries calling for equality test and range selection support

```
SELECT *                          SELECT *
FROM   R                          FROM   R
WHERE  C = 45                     WHERE  A > 0 AND A < 100
```

Simple Cost Model

- A **cost model** is used to analyze the execution time of a given database operations
  - **block I/O** operations are typically a major cost factor
  - **CPU time** to account for searching inside a page, comparing a record field to selection constant, etc.
- To **estimate** the execution time of the five database operation, we introduce a **coarse** cost model
  - omits cost of network access
  - does not consider cache effects
  - neglects burst I/O
  - …
- Cost models play an important role in **query optimization**

Simple Cost Model

Simple cost model parameters

Parameter	**Description**
b	number of pages in the file
r	number of records on a page
D	time to read/write a **d**isk page
C	**C**PU time needed to process a record (e.g., compare a field value)
H	CPU time take to apply a function to a record (e.g., a comparison or **h**ash function)

- Some typical values
  - $D \approx 15$ ms
  - $C \approx H \approx 0.1$ μs

Back to the Future

A simple hash function

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`**, **`'dog'`**$\rangle)$	$\to$	2	$(42 = 101010_2)$
$h(\langle$**`14`**, **`true`**, **`'cat'`**$\rangle)$	$\to$	6	$(14 = 1110_2)$
$h(\langle$**`26`**, **`false`**, **`'mouse'`**$\rangle)$	$\to$	2	$(26 = 11010_2)$

- The hash function determines the page number only, record placement inside a page is not prescribed
- If a page *p* is filled to capacity, a chain of overflow pages is maintained to store additional records with $h(\langle\ldots\rangle) = p$
- To avoid immediate overflowing when a new record is inserted, pages are typically filled to 80% only when a heap file is initially (re)organized into a hash file

Cost of 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.**

$$Scan_{heap} = b \cdot (D + r \cdot C)$$

2. **Sorted file**

   The sort order **does not help** here. However, the scan retrieves the record in sorted order (which can be a big plus for subsequent operators).

$$Scan_{sort} = b \cdot (D + r \cdot C)$$

3. **Hashed file**

   The hash function **does not help**. We simply scan from the beginning (skipping over the spare free space typically found in hashed files).

$$Scan_{hash} = \underbrace{(100/80)}_{= 1.25} \cdot b \cdot (D + r \cdot C)$$

Hashed File

Scanning a hashed file

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

Cost of Search with Equality Test

1. Heap file

The equality test is Ⓐ on a *primary key* or Ⓑ *not* on a *primary key*.

Ⓐ $Search_{heap} = {}^{1}/_{2} \cdot b \cdot (D + r \cdot C)$

Ⓑ $Search_{heap} = b \cdot (D + r \cdot C)$

2. Sorted file (sorted on `A`)

We assume the equality test is on `A`. The sort order enables the use of **binary search.**

$$Search_{sort} = \log_2 b \cdot D + \log_2 r \cdot C$$

If more than one record qualifies, all other matches are stored **right after** the first hit.

✎ **Nevertheless, no DBMS will implement binary search for value lookup**

Why?

Cost of Search with Equality Test

3. Hashed file (hashed on **`A`**)

We assume the equality test is on **`A`**. **Hashed files support equality searching best.** The hash function directly leads to the page containing the hit (overflows chains are ignored here).

The equality test is Ⓐ on a *primary key* or Ⓑ *not* on a *primary key*

Ⓐ $Search_{hash} = H + D + {}^{1}/_{2} \cdot r \cdot C$

Ⓑ $Search_{hash} = H + D + r \cdot C$

Note that there is **no dependence** on the file size *b* here. All qualifying records live on the same page or, if present, in its overflow pages.

Cost of Search with Range Selection

1. **Heap file**

   Qualifying records can appear anywhere in the file.

   $$Range_{heap} = b \cdot (D + r \cdot C)$$

2. **Sorted file** (sorted on **`A`**)

   Use equality search with **`A`** = *lower*, then sequentially scan the file until a record with **`A`** > *upper* is found.

   $$Range_{sort} = \log_2 b \cdot D + \log_2 r \cdot C + \lceil {}^{n}/_{r} \rceil \cdot D + n \cdot C$$

   where *n* denotes the number of hits in the range.

3. **Hashed file** (hashed on **`A`**)

   Hashing offers no help as has functions are designed to scatter records all of the hashed file, e.g., *h*(⟨**`7`**, …⟩) = 7, *h*(⟨**`8`**, …⟩) = 0.

   $$Range_{hash} = 1.25 \cdot b \cdot (D + r \cdot C)$$

Cost of Insert

1. Heap file

The record can be added to some arbitrary page, e.g., the last page. This involves reading and writing the page.

$$Insert_{heap} = 2 \cdot D + C$$

2. Sorted file

On average, the new record will belong in the middle of the file. Then, all subsequent records in the latter half of the file must be shifted.

$$Insert_{sort} = \underbrace{\log_2 b \cdot D + \log_2 r \cdot C}_{\textit{search}} + \underbrace{{}^{1}/_{2} \cdot b \cdot (2 \cdot D + r \cdot C)}_{\textit{shift latter half}}$$

3. Hashed file

Search for the record, then read and write the page determined by the hash function (assume spare 20% space is sufficient to hold new record)

$$Insert_{hash} = \underbrace{H + D}_{\textit{search}} + C + D$$

Cost of Delete

1. Heap file

Assume that the file is not compacted after the record is found and removed, i.e., the file uses free space management.

$$Delete_{heap} = \underbrace{D}_{\textit{search by record id}} + C + D$$

2. Sorted file

Access the record's page and then (on average) shift the latter half of the file to compact it.

$$Delete_{sort} = D + \underbrace{{}^{1}/_{2} \cdot b \cdot (2 \cdot D + r \cdot C)}_{\textit{shift latter half}}$$

3. Hashed file

Page access by rid is faster than the hash function: same cost as heap file.

$$Delete_{hash} = D + C + D$$

Performance Comparison

- Performance of **range selections** for files of increasing size (D = 15 ms, C = 0.1 µs, r = 100, n = 10)

Performance graph

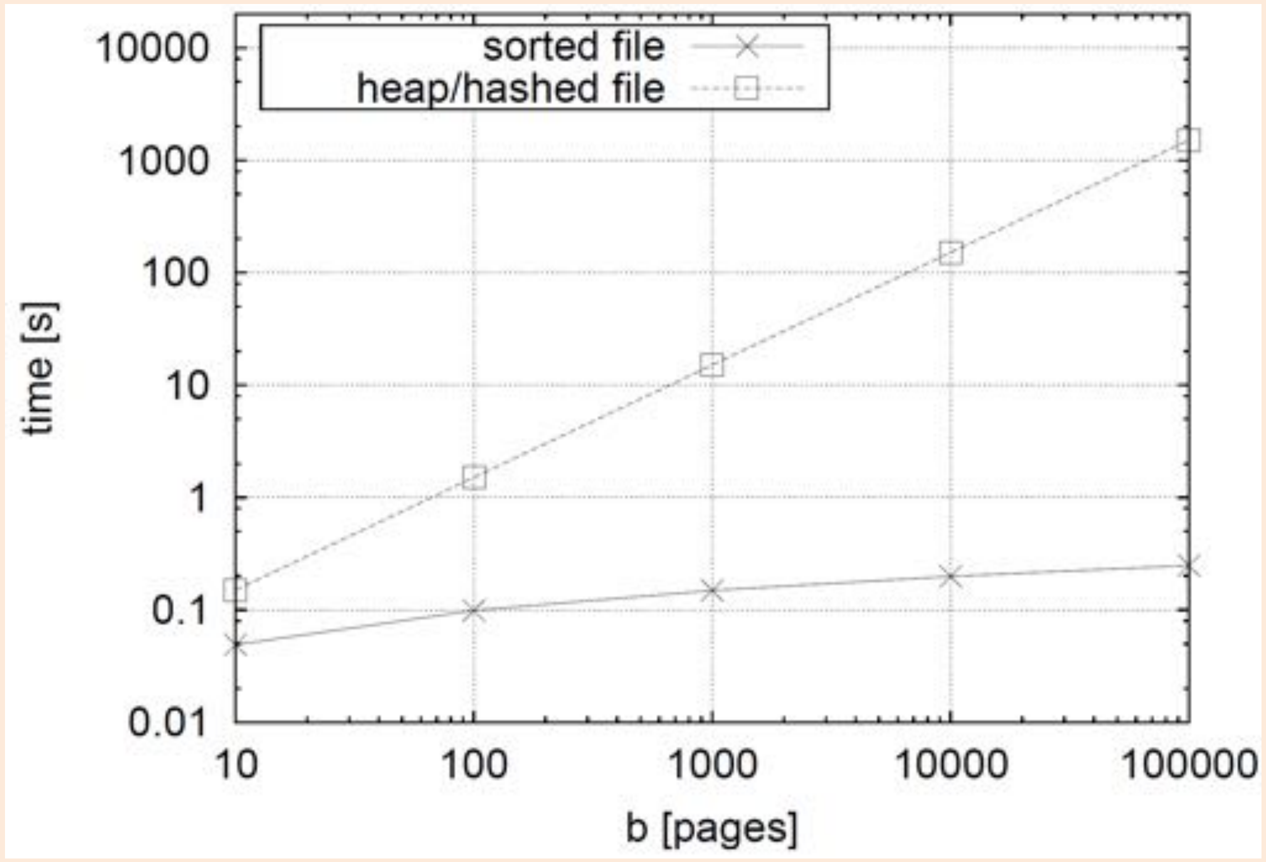


Figure Credit: Marc H. Scholl, University of Konstanz, Germany

Performance Comparison

- Performance of **deletions** for files of increasing size ($D = 15$ ms, $C = 0.1$ µs, $r = 100$, $n = 1$)

Performance graph

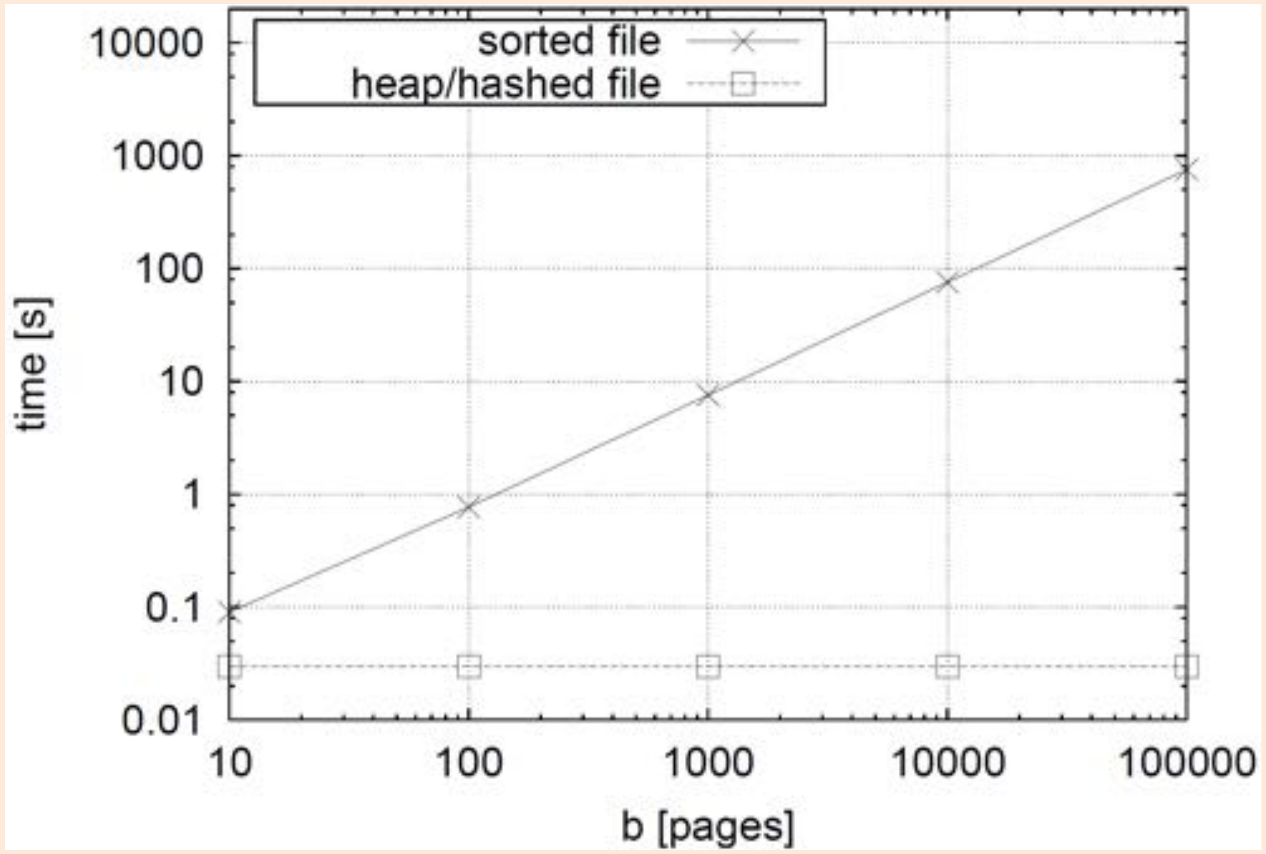


Figure Credit: Marc H. Scholl, University of Konstanz, Germany

And the Winner Is...

- There is **no single file organization** that responds equally fast to all five operations
- This is a **dilemma** because more advanced file organizations can make a real difference in speed (see previous slides)

- There exist **index structures** which offer all advantages of a sorted file *and* support insertions/deletions efficiently (at the cost of a modest space overhead): **B+ trees**
- Before discussing B+ trees in detail, the following introduces the **index concept** in general

Indexes

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

Example

```
SELECT A, B, C
FROM   R
WHERE  A > 0 AND A < 100
```

If the file for table **R** is sorted on **C**, it **cannot** be used to evaluate *Q* more efficiently. A solution is to add an index *that supports range queries* on **A**.

Indexes

- A DBMS uses indexes like **guides**, where each guide is specialized to accelerate searches on a specific attribute (or a combination of attributes) of the records in its associated file

Usage of index on attribute A

$$\begin{tikzcd} k \arrow[r, "①"'] & \boxed{\text{index}} \arrow[r, "②"'] & k^* \arrow[r, "③"'] & \langle \ldots, \mathbf{A} = k, \ldots \rangle \end{tikzcd}$$

1. Query 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*.

The Small Print: Only "exact match" indexes will return records that contain the value *k* for field **A**. In the more general case of "similarity" indexes, the records are not guaranteed to contain the value *k*, they are only candidates for having this value.

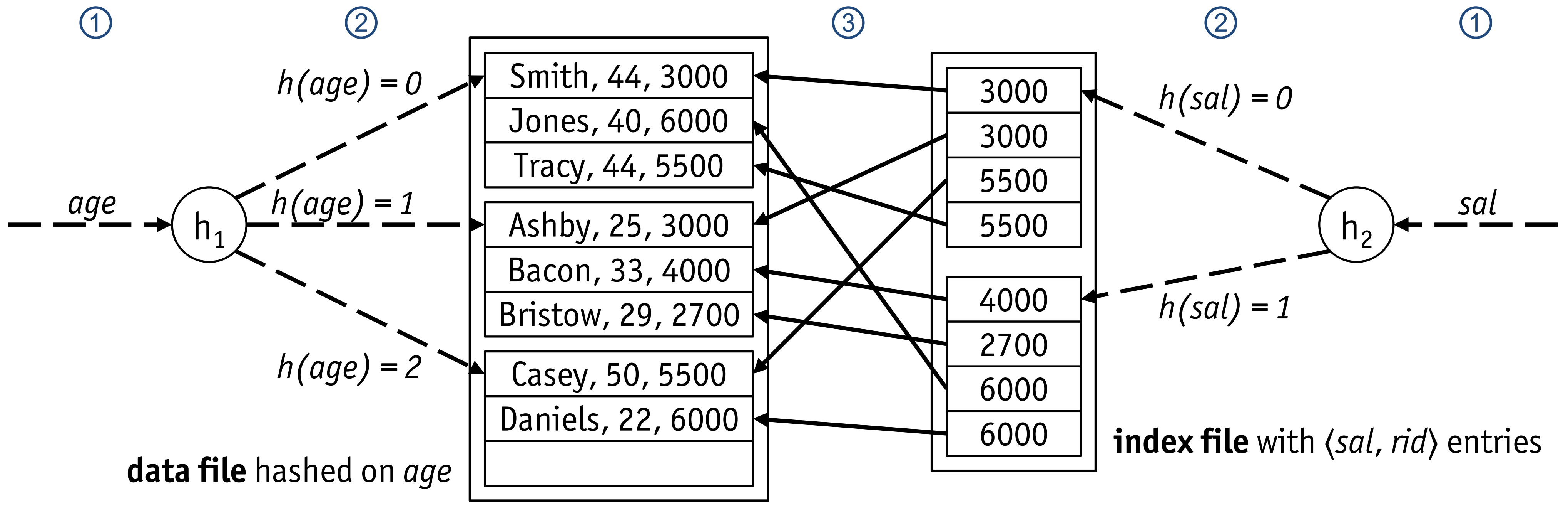
Index Entries

Index Entry Design

Variant	Index entry $k*$
❶	$\langle k, \quad \langle \ldots, \mathtt{A} = k, \ldots \rangle\rangle$
❷	$\langle k, \quad rid \rangle$
❸	$\langle \mathrm{k}, \quad [rid_1, rid_2, \ldots] \rangle$

- Remarks
  - With variant ❶, there is no need to store the data records in addition to 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 ❶ to avoid redundant storage of records
  - Variants ❷ and ❸ use *rid*(s) to point into the actual data file
  - Variant ❸ leads to fewer index entries if multiple records match a search key *k*, but index entries are of variable length

Index Example



- **Data file** contains ⟨*name*, *age*, *sal*⟩ records and is hashed on age, using hash function h_1 (index entry variant ❶)
- **Index file** contains ⟨*sal*, *rid*⟩ index entries (variant ❷), pointing to data file (hash function h_2)
- This file organization plus index **efficiently** supports equality searches on both key *age* **and** key *sal*

Clustered vs. Unclustered Indexes

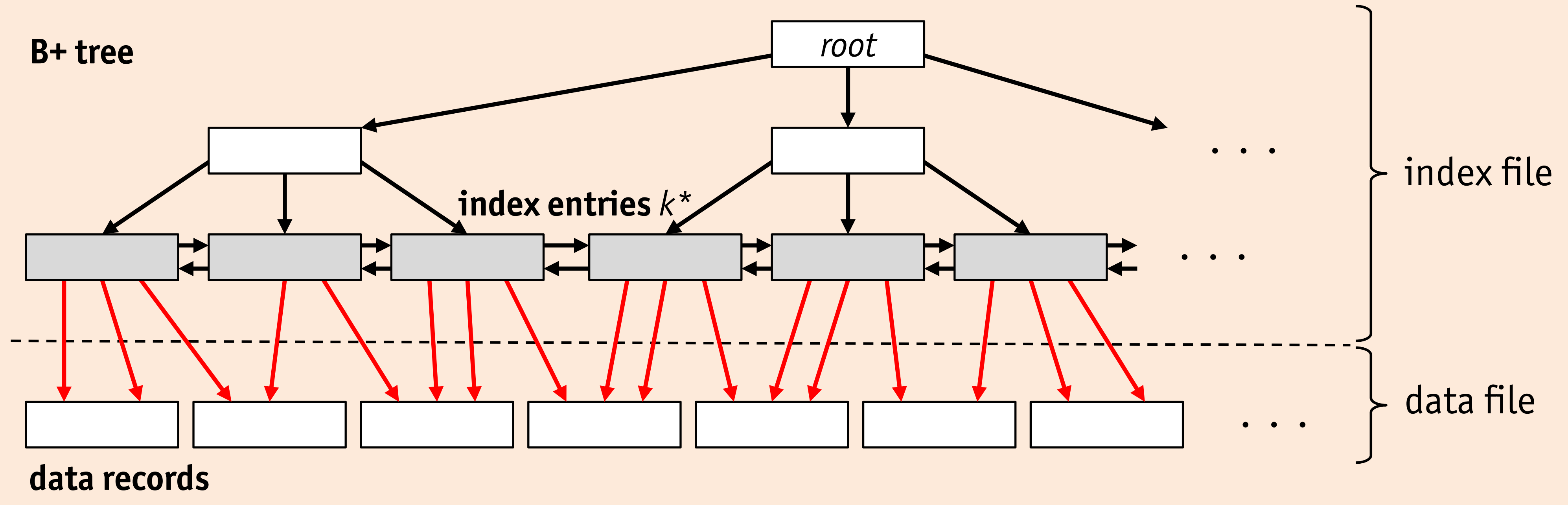
- Suppose, **range selections** such as *lower* ≤ **`A`** ≤ *upper* need to be supported on records of a data file
- If an index on field **`A`** is maintained, range selection queries could be evaluated using the following algorithm
  1. **query the index** *once* for a record with field **`A`** = *lower*
  2. **sequentially scan the data file** from there until we encounter a record with field **`A`** > *upper*

Name the assumption!

Which important assumption does the above algorithm make in order for this **switch from index to data file** to work efficiently?

Clustered vs. Unclustered Indexes

Index over data file with matching sort order



- Remark
  - in a B+ tree, for example, the index entries $k*$ stored in the leaves are sorted on the key k

Clustered vs. Unclustered Indexes

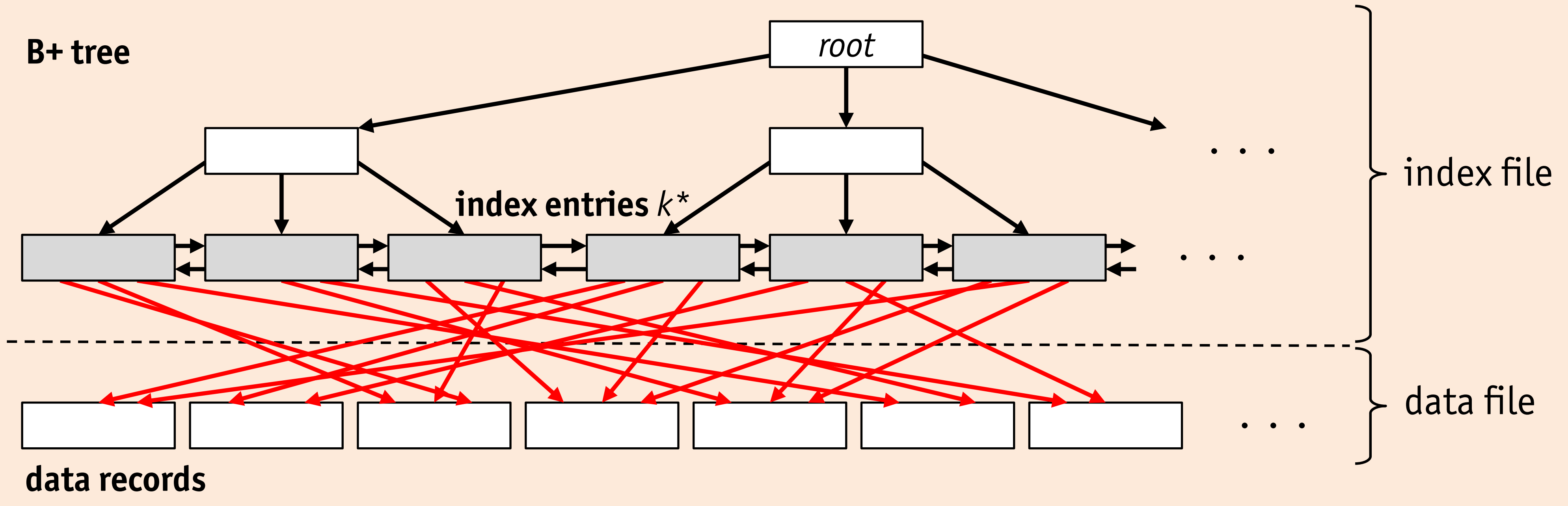
Definition: Clustered Index

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**
  - proximity of index entries **does not** imply proximity of data records
  - as before, the index can be queried for a record with **`A`** = *lower*
  - however, to continue the scan it is necessary to **revisit the index entries**, which point to **data pages scattered** all over the data file
- Remarks
  - an index that uses entries $k*$ of variant ❶, is clustered by definition
  - a data file can have at most one clustered index (but any number of unclustered indexes)

Clustered vs. Unclustered Indexes

Unclustered index



Clustered vs. Unclustered Indexes

Variant ❶ in Oracle 8i

```
CREATE TABLE … ( … PRIMARY KEY ( … )) ORGANIZATION INDEX;
```

Clustered indexes in DB2

Create a clustered index **`IXR`** on table **`R`**, index key is attribute **`A`**

```
CREATE INDEX IXR ON R(A ASC) CLUSTER;
```

From the DB2 V9.5 manual

> *"[* **`CLUSTER`** *] specifies that the index is* ***the*** *clustering index of the table. The* ***cluster factor*** *of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to* ***insert new rows physically close to the rows for which the key values of this index are in the same range****. Only one clustering index may exist for a table so* **`CLUSTER`** *may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8)."*

Slide Credit: Torsten Grust, University of Tübingen, Germany

Clustered vs. Unclustered Indexes

Cluster a table based on an existing index in PostgreSQL

Reorganize the rows of table **`R`** so that their physical order matches the *existing* index **`IXR`**

```
CLUSTER R USING IXR;
```

- If **`IXR`** indexes attribute **`A`** of **`R`**, rows will be sorted in ascending **`A`** order
- Range queries will touch less pages, which additionally, will be physically adjacent
- **Note:** Generally, future insertions will compromise the perfect **`A`** order
  - may issue **`CLUSTER R`** again to re-cluster
  - in **`CREATE TABLE`**, use **`WITH (fillfactor =`** f **`)`**, $f \in$ **`10…100`**, to reserve space for subsequent insertions

- The SQL-92 and SQL-99 standard do not include any statement for the specification (creation, dropping) of index structures
  - SQL **does not even require** SQL systems to provide indexes at all!
  - almost all SQL implementations support one or more kinds of indexes

Slide Credit: Torsten Grust, University of Tübingen, Germany

Dense vs. Sparse Indexes

- Another advantage of a **clustered index** is the fact that it can be designed to be **space efficient**

Definition: Sparse Index

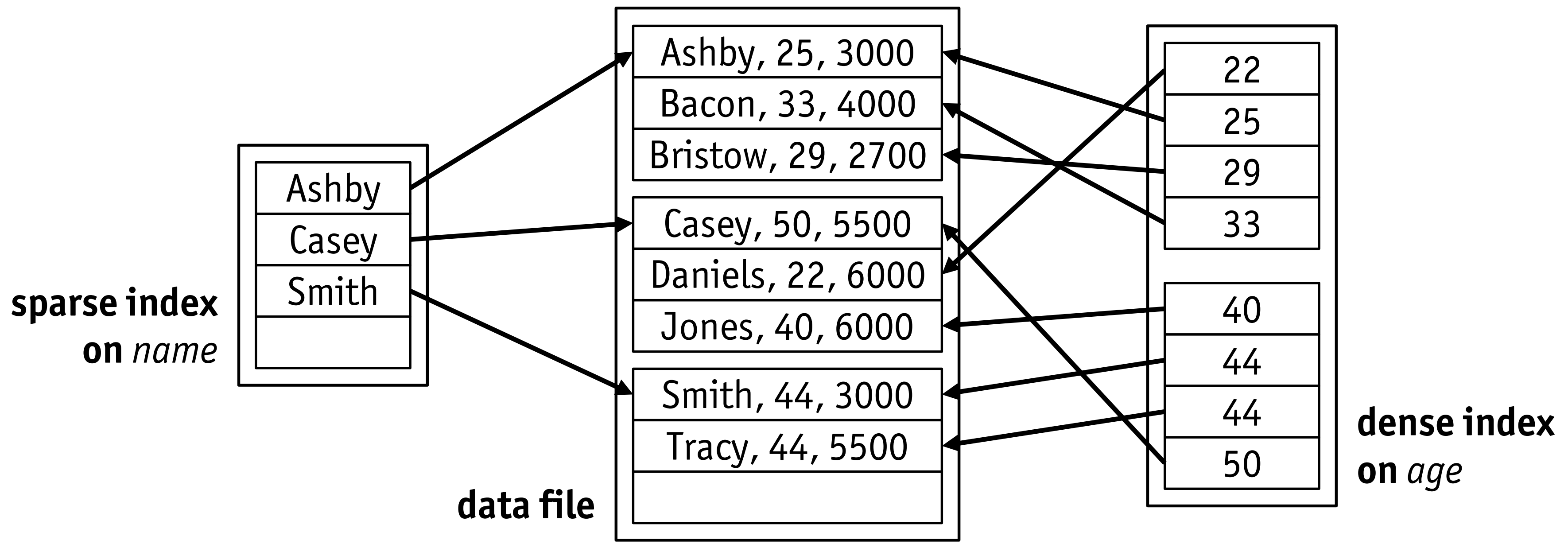
To keep the size of the index small, maintain **one index entry** $k*$ **per data file page** (not one index entry per data record). The key k is the **smallest key** on that page.

Indexes of this kind are called **sparse**. Otherwise indexes are referred to as **dense**.

Search a record with field `A` = k in a sparse `A`-index

1. Locate the largest index entry $k'*$ such that $k'* \leq k'$
2. Then access the page pointed to by $k'*$
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`**, matching records are guaranteed to be found in proximity

Dense vs. Sparse Index Example



- Again, the data file contains ⟨*name*, *age*, *sal*⟩ records
- Two indexes are maintained for the data file
  - **clustered sparse index** on field *name*
  - **unclustered dense index** on field *age*
- Both indexes use entry variant ❷ to point into the data file

Dense vs. Sparse Indexes

- Final remarks
  - sparse indexes need 2-3 orders of magnitude **less space** than dense indexes
  - it is **not possible** to 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 the following SQL queries?

-
  ```
  SELECT MAX(age)
  FROM   employees
  ```

-
  ```
  SELECT MAX(name)
  FROM   employees
  ```

Primary vs. Secondary Indexes

Terminology

In the literature, there is often a distinction between **primary** (mostly used for indexes on the primary key) and **secondary** (mostly used for indexes on other attributes) indexes.

This terminology, however, is not very uniform and some text books may use those terms for different properties.

For example, some text books use **primary** to denote variant ❶ of indexes, whereas **secondary** is used to characterize the other two variants ❷ and ❸.

Multi-Attribute Indexes

- Each of the indexing techniques sketched so far can be applied to a **combination of attribute values** in a straightforward way
  - concatenate indexed attributes to form an index key, e.g., ⟨`lastname`, `firstname`⟩ → `searchkey`
  - define index on `searchkey`
  - index will support lookup based on both attribute values, e.g., … `WHERE lastname='Doe' AND firstname='John'` …
  - possibly, it will also support lookup based on a “prefix” of values, e.g., … `WHERE lastname='Doe'` …
- So-called **multi-dimensional indexes** provide support for **symmetric** lookups for all subsets of the indexed attributes
- Numerous such indexes have been proposed, in particular for geographical and geometric applications

Universität
Konstanz

Database System Architecture and Implementation

TO BE CONTINUED...