File Organisation and Indexing

Werner Nutt

Introduction to Databases

Free University of Bozen-Bolzano

Data Storage Principles

- Database relations are implemented as **files** of **records**.

- This is still an *abstraction:* the real storage medium are **disks**, which consist of **pages** (ca. 0.5–5 kbytes)

- Pages are *read* from disk and *written* to disk: high cost operations!

- Mapping: each record has a **record identity (rid)**, which identifies the *page* where it is stored and its *offset* on that page

- The DBMS reads (and writes) *entire pages* and stores a number of them in a **buffer pool**

- The **buffer manager** decides which pages to load into the buffer
  (Replacement policy: e.g., "least recently used" or "clock")

Alternative File Organisations

Alternatives are *ideal for some situation,* and *not so good in others:*

Heap Files: *No order on records.* Suitable when typical access is a file scan retrieving all records.

Sorted Files: *Sorted by a specific record field* (key). Best if records must be retrieved in some order, or only a "range" of records is needed.

Hashed Files:
- File is a collection of **buckets.**

    (Bucket = *primary* page plus zero or more *overflow* pages.)
- **Hashing function** h: computes $h(r) =$ bucket into which record r belongs (looks at only some of the fields of r, the *search fields.*)

    Good for equality selections.

Cost Model

We ignore CPU costs, for simplicity:

- P: number of **data pages**

- D: (average) time to read or write **disk page**

Simplifications:

- Measuring number of page I/O's ignores

  gains of pre-fetching blocks of pages

  $\rightsquigarrow$ even I/O cost is only approximated

- Average case analysis   *(based on several simplistic assumptions)*

Good enough to show overall trends!

Assumptions in Our Analysis

- *Single record* insert and delete

- **Heap Files**:
  - Equality selection on key; exactly one match
  - Insert always at end of file

- **Sorted Files:**
  - Selections on sort field(s)
  - Files compacted after deletions

- **Hashed Files:**
  - No overflow buckets, 80% page occupancy

Cost of Operations

	Heap File	Sorted File	Hashed File
Scan all recs			
Equality Search			
Range Search			
Insert			
Delete			

Indexes

An **index** on a file speeds up selections on the **search key fields**
for the index

- *Any subset of the fields* of a relation can be the *search* key for an
  index on the relation

- *Search key* is not the same as *key*
    (minimal set of fields that uniquely identify a record in a relation)

An index contains a collection of **data entries**, and supports efficient
retrieval of all data entries K^* with a given key value K.

Data Entries in Indexes

Three alternatives:

- Data record with key value K

- $\langle K, r \rangle$, where r is rid of a record with search key value K

- $\langle K, [r_1, \ldots, r_n] \rangle$, where $[r_1, \ldots, r_n]$ is a list or rid's of records with search key value K

Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value K

- – Examples: B+-trees, hash-based structures

- – Index may contain auxiliary information that directs searches

Data Entries in Indexes: Alternative 1

> *"Data record with key value K"*

- Means: **index structure** is a **file organisation** for data records

  (like heap files or sorted files)

- For a given collection of data, **at most one index** can use this.

  Otherwise, duplication of data records

  $\rightsquigarrow$ redundant storage

  $\rightsquigarrow$ potential inconsistency

- If: large data records

  $\implies$ high number of pages containing data entries

  $\implies$ large size of auxiliary information

Data Entries in Indexes: Alternatives 2 and 3

$$\text{``}\langle K, r \rangle\text{'' or ``}\langle K, [r_1, \ldots, r_n] \rangle\text{'', with rids } r, r_1, \ldots, r_n$$

- Typically, **data entries** are much **smaller than data records**
  $\implies$ better than Alt1 (with large data records),
     especially if search keys are small

     Portion of index structure to direct search
     is much smaller than with Alt1.

- If more than one index is required on a given file, **at most one** index can use **Alt1**; the **rest** must use Alt2 or Alt3

- Alt3 is **more compact** than Alt2, but leads to **variable sized data entries** even if search keys are of fixed length
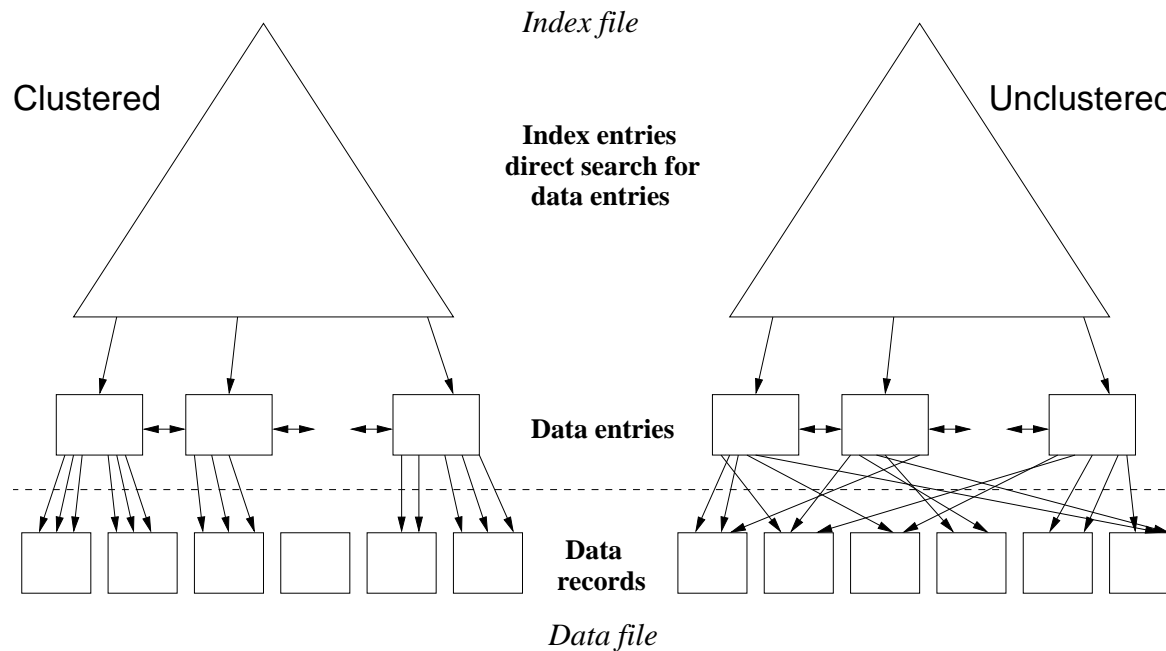

Index Classification

Primary vs. **Secondary:** Primary, if search key contains a primary key

- **Unique** index: search key contains a candidate key

Clustered vs. **Unclustered:** Clustered, if order of data records is the same as, or "close to", order of data entries

- Alternative 1 implies clustered, but not vice-versa
- A file can be clustered on at most one search key
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not

Clustered vs. Unclustered

Index file

Clustered

Unclustered

**Index entries
direct search for
data entries**

Data entries

**Data
records**

Data file

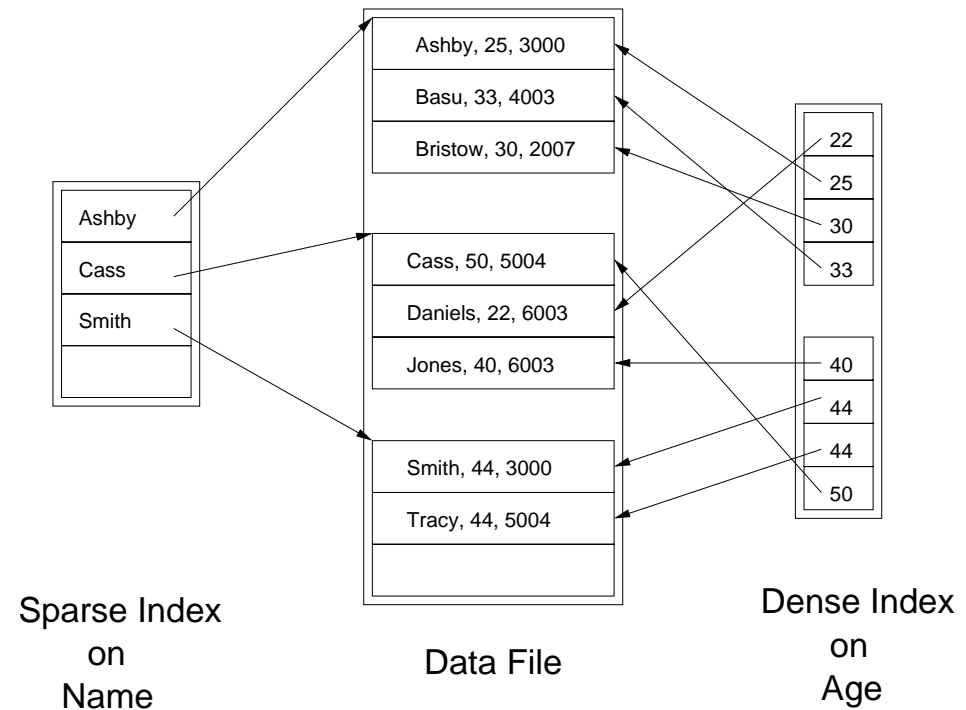If Alt2 is used for data entries, and data records are stored in a heap file:

- To build clustered index, first sort the heap file

  (with some free space on each page for future inserts)

- Overflow pages may be needed for inserts

  $\rightsquigarrow$ order of data recs is "close to", but not identical to the sort order
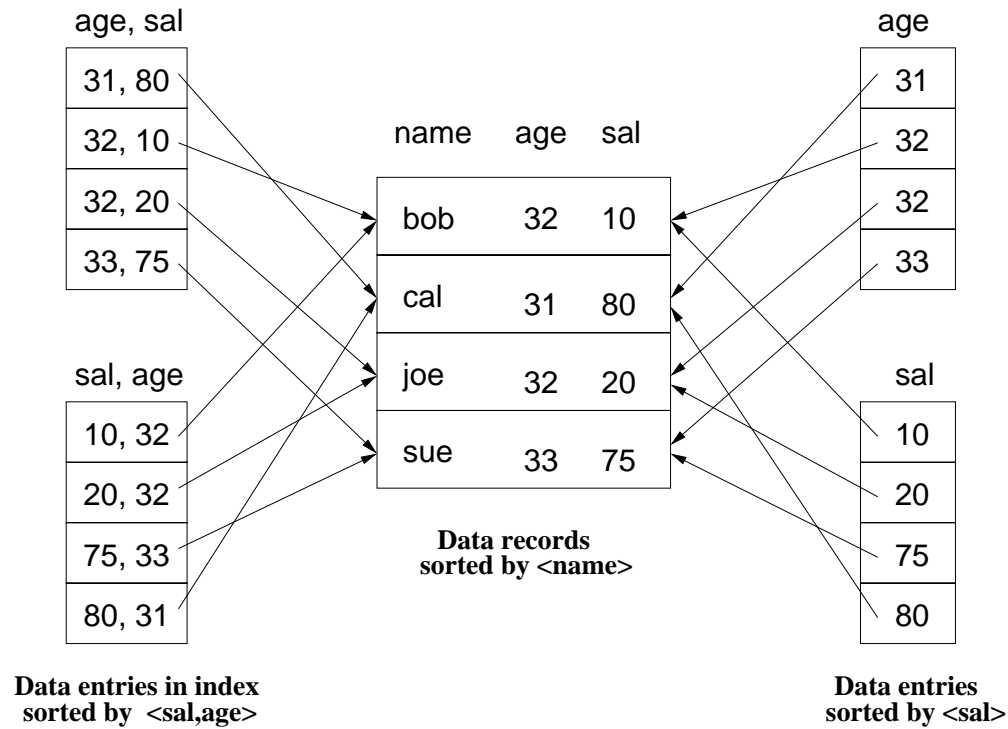

Dense vs. Sparse

Dense, if there is at least one data entry per search key value

- Alternative 1 always leads to dense index

- Every sparse index is clustered

- Sparse indexes are smaller; however, some optimisations are based on dense indexes

Sparse Index
on
Name

Data File

Dense Index
on
Age

Composite Key Indexes

age, sal

| 31, 80 |
| 32, 10 |
| 32, 20 |
| 33, 75 |

age

| 31 |
| 32 |
| 32 |
| 33 |

name	age	sal
bob	32	10
cal	31	80
joe	32	20
sue	33	75

**Data records
sorted by <name>**

sal, age

| 10, 32 |
| 20, 32 |
| 75, 33 |
| 80, 31 |

sal

| 10 |
| 20 |
| 75 |
| 80 |

**Data entries in index
sorted by <sal,age>**

**Data entries
sorted by <sal>**

Summary

- Many alternative file organisations exist, each appropriate in some situation.

- If selection queries are frequent, sorting the file or building an *index* is important.

  - *Hash-based* indexes only good for *equality search*

  - *Sorted files* and *tree-based* indexes best for *range search* (and also equality search)

    (files rarely sorted in practice; B+-tree index is better)

- Index is a collection of data entries plus a way to quickly find entries with given key values

Summary (Cntd.)

- **Data entries** can be (1) actual data records, (2) $\langle \text{key}, \text{rid} \rangle$-pairs, (3) $\langle \text{key}, \text{rid-list} \rangle$-pairs

  Choice orthogonal to **indexing technique**
  used to locate data entries with a a given key value.

- There may be **several** indexes on a given file of data records, each with a **different search key.**

- Indexes can be **classified** as

  - *clustered* vs. *unclustered*

  - *primary* vs. *secondary*

  - *dense* vs. *sparse.*

  Differences have important consequences for *utility/performance*

References

These slides are based on Chapter 8 of the book *Database Management Systems* by R. Ramakrishnan and J. Gehrke, and on slides by the authors published at

`www.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html`