Database Management Systems II

Robert Gottstein

gottstein@dvs.tu-darmstadt.de

Exercise 4.1

- File Organization Techniques
- File Indexes and Access Paths

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

a) Alternative file organizations:

- **Heap files**
- **Sorted files**
- **Hashed files**

b) File indexing techniques and access paths.
Index classification.

c) Advantages/disadvantages of:

- **ISAM Indexes**
- **B+Trees**
- **Hash Indexes**

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

a). Alternative File Organizations

First: Definition of File organization

…is a way of *physically arranging* the records of a file when the file is stored on disk.

Answers the quesion:

*„**How** are the records arranged on disk?“*

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

1.Heap File (Unordered File)

- File records ordered randomly
- Literally a „heap“ of records

Evaluation:

- Good space utilization / storage efficiency
- Fast file scan (retrieve all records), fast inserts and deletes
- Slow searches

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

2.Sorted File (Ordered File)

- File records sorted on a *'set of fields'* → *search key*

Evaluation:

- Good space utilization / storage efficiency
- Slow inserts and deletes
- Quite fast for searches (binary-search can be used)
- **Fastest for range selections and ordered access**
  - **E.g. retrieval in search key order**

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

3.Hashed File

- File hashed on a *'set of fields'* → *search key*
- File is a collection of **buckets**
- **Each bucket** = primary page + *n* overflow pages
- Hashing function **h()** (defined on search key)

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

Evaluation (Hashed File):

- Worse space utilization ⇒more pages used to store file
- Fast inserts and deletes
- **Fastest for equality selections (point queries)**
- **No support for range selections** (range queries) and file scan a little slower

Summary

Heap file:

Good if *full file scans* dominate

Sorted file:

Good if *range queries and ordered accesses* dominate

Hashed file:

Good if *point queries* dominate

TECHNISCHE UNIVERSITÄT DARMSTADT

b). File Indexing Techniques and Access Paths

Definition Index

An auxiliary structure designed *to speed up operations* that are not efficiently supported by the basic file organization.

Index - Properties

- Each index is built on a set of fields called **search key**
- The index speeds up selections on the **search key fields**
- **Search key** <> **key**
  (minimal set of fields that uniquely identify a record)
- An index contains a collection of **data entries**
- Each data entry contains a ***search key value*** and some additional information allowing us to quickly ***locate*** all **data records** with the respective search key value
- The index supports efficient retrieval of all data entries with a given search key value '**k**'

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

3 alternatives for what to store in data entries:

Alternative 1:

{k, *data record* s with s. key value k**}**

Alternative 2:

{k, ***RID of data*** record with s. key value k**}**

Alternative 3:

{k, ***list of RIDs*** of records with s. key value k**}**

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

Typical indexing techniques:

1. ***Tree-structured indexing*** -(e.g. B+trees, ISAM)
2. ***Hash indexing***
3. **[Encoded]** ***Bit-map Indexing***

Alternative 1 for data entries allows an index to be used as a primary file organization (e.g. index-organized tables in Oracle)

Exercise 4.1 File Organization Techniques File Indexes and Access Paths

Definition: Access Paths

The different ways to retrieve records of a file (tuples of a relation) are called **access paths**.

For example to locate records one can use

- a file scan
- the primary file organization
- an auxiliary index structure

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

Index Classification:

1. Primary vs. Secondary Indices:

- **Primary Indices**: Clustered Indices are sometimes called Primary Indices. A Primary Index is is not necessarily an index on the primary key of a relation:
  - „Professor“ relation could be sorted on Department Attribute (which is not even a key) and a Clustered Index (Primary Index) could be built, while the Id attribute of that relation (which is also ist primary key) will be unclustered, because rows won‘t be sorted on that attribute.
- **Secondary Indices:**
  - An index which is unclustered (the records are not sorted on the indexed attribute)
- A relation can have at most one Primary Index

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Clustered vs. Unclustered Indexes:

Definition Clustered Index:

File is organized so that the ordering of data records is the same as or close to the ordering of index data entries

- Logically related records are stored in ***physical proximity***
- ***A file can have at most one clustered index (=Primary Index)***
- Indexes with Alternative 1 **{**data record**}** for data entries are clustered by definition. Indexes using alternative 2 or 3 {RID, List of RIDs} are only clustered when data records are sorted in the order of the index search key.
- **Clustering is very important for performance**

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

3. Dense vs. Sparse Indexes:

Dense Index:

- If there is at least one data entry for each search key value

Sparse Index:

- Otherwise.

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

Major indexing techniques:

1. ***Tree-structured Indexing -***(e.g. ISAM, B+trees)
2. ***Hash Indexing*** (static vs. dynamic)
3. **[Encoded] *Bit-map Indexing***

Tree-Structured Indexing

– support both ***equality*** and ***range*** selections

– support ordered access efficiently (e.g. sorted file scan)

– two major types: ***ISAM:*** static structure, ***B+Tree:*** dynamic structure

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

1c). ISAM, B+Trees and Hash Indexes

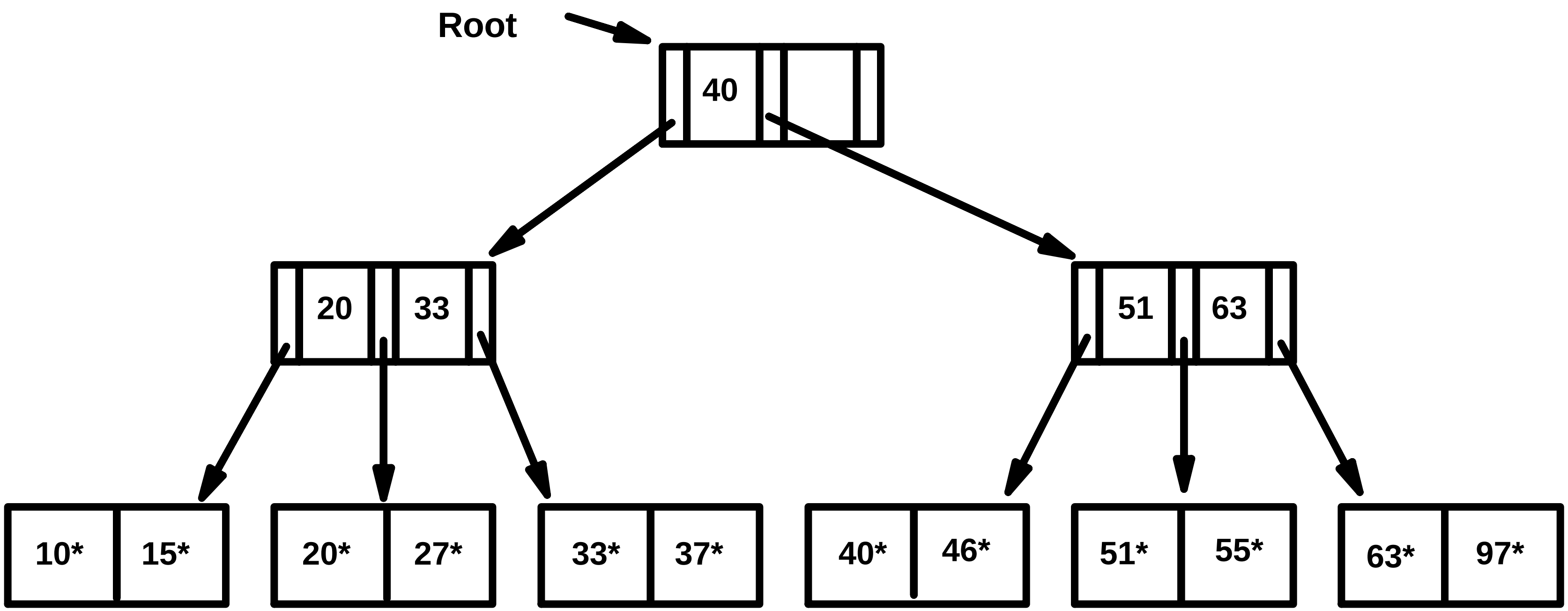
ISAM (Indexed Sequential Access Method):

- ***static index structure***
  - *only* leaf pages modified, overflow pages needed
- if files dynamic, overflow chains grow ⇒***poor performance***
- leaf pages usually kept contiguous ⇒***sequential disk accesses*** for large range queries or sorted file scans
- low locking overhead and high concurrency: since non-leaf pages are static and are never locked.
- *Bottom-line:* Suitable only for static files.

(See http://www.cs.umd.edu/class/spring2005/cmsc424-0201/notes/ISAM.ppt)

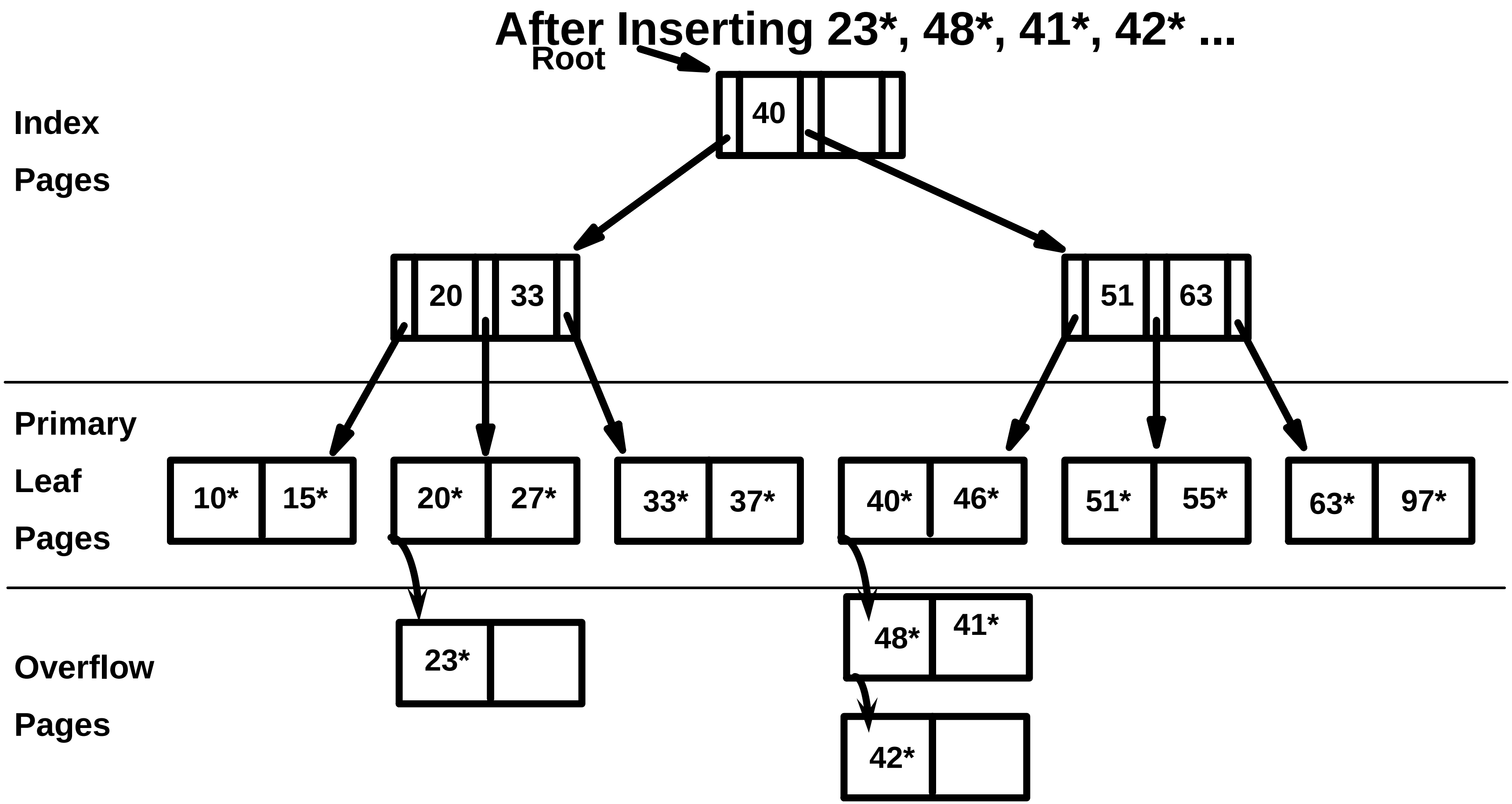
Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

- Example: ISAM with Fixed Size of two



(See http://www.cs.umd.edu/class/spring2005/cmsc424-0201/notes/ISAM.ppt)

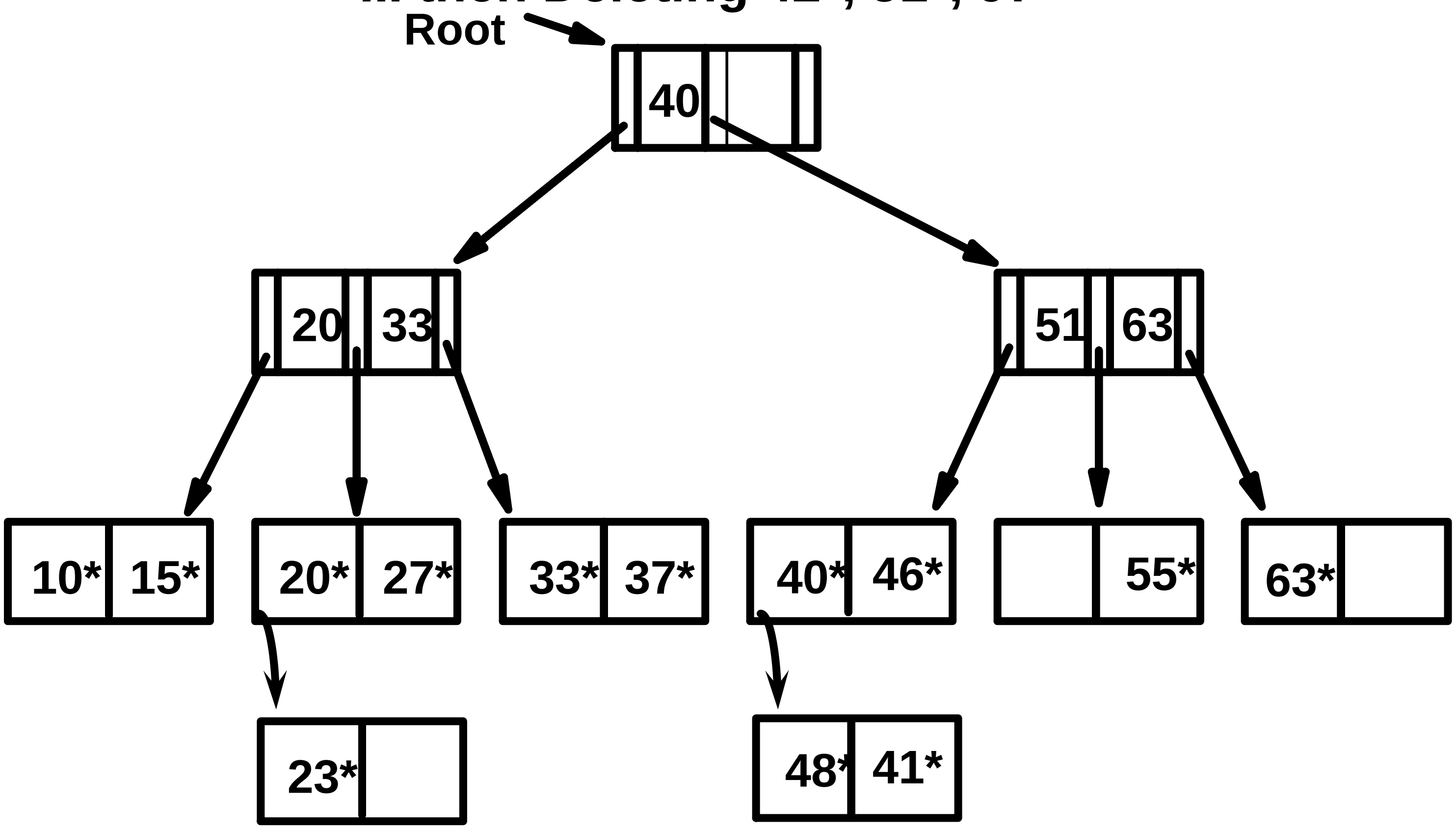
Exercise 4.1
File Organization Techniques
File Indexes and Access Paths



(See http://www.cs.umd.edu/class/spring2005/cmsc424-0201/notes/ISAM.ppt)

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

... then Deleting 42*, 51*, 97*

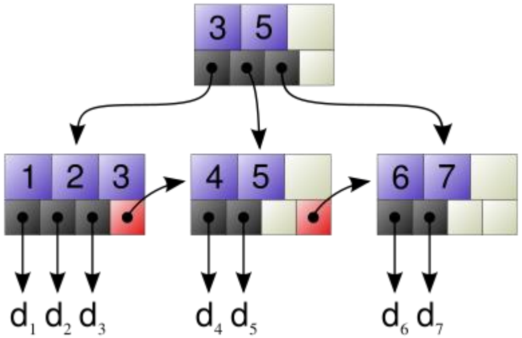


(See http://www.cs.umd.edu/class/spring2005/cmsc424-0201/notes/ISAM.ppt)

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

B+Trees



- dynamic index structure
- supports inserts and deletes efficiently
- index reorganized dynamically (balanced)
- high **fan-out** means depth rarely > 4 levels
- **usually cost is 3 (max 4) I/Os** to get respective data entries
- lower concurrency and higher locking overhead than with ISAM (non-leaf pages)
- leaf pages eventually get out of sequence as file grows ⇒random disk accesses for large range queries or sorted file scans

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

TECHNISCHE UNIVERSITÄT DARMSTADT

Note (B+ Trees):

Ideal for ***dynamic files requiring sorted access***. B+Trees are the most popular index structures in commercial database systems. For files that are ***updated frequently*** and ***require sorted access***, using a B+Tree index as a primary file organization is almost always superior to using a simple "sorted file" organization.

Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

Hash-based Indexing (static vs. dynamic):

Idea:

A hash function maps search key values into a range of **bucket** numbers. Bucket number determines page where respective data entry is stored.

- **Fastest method for equality selections** - Usually 1 I/O to locate data entry
- **Do not support range queries**
- B+trees support range queries and are almost as good for equality selections - usually preferred in commercial systems
- Hashing principle useful in implementing relational operators

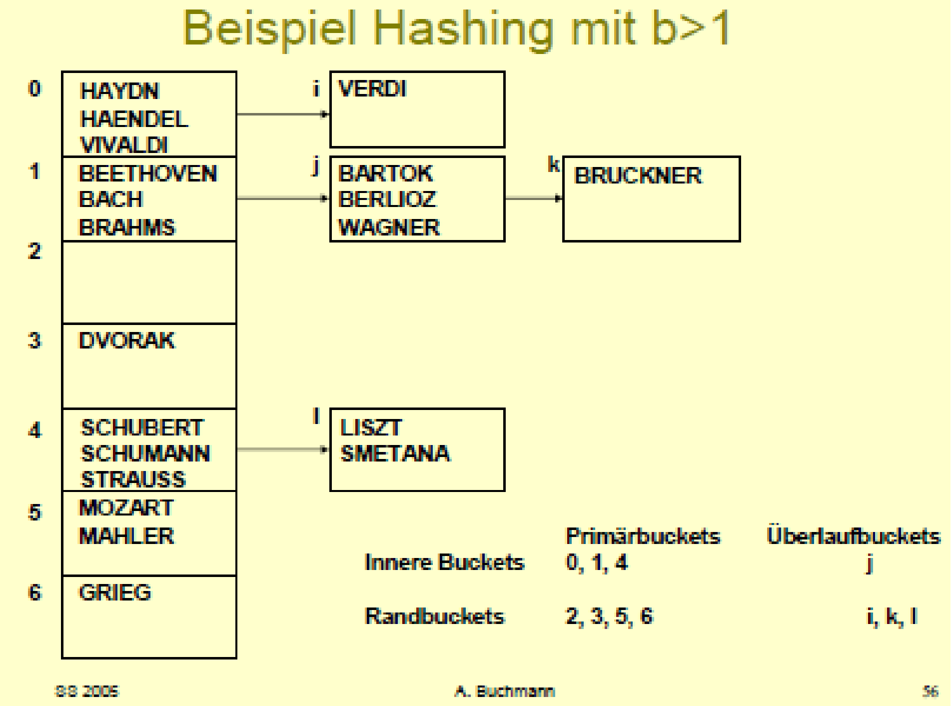
Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

Static Hashing:

- # buckets fixed, overflow pages used for expansion
- Poor performance for dynamic files

- As known from GDI2 →

GDI2 Script SS2005 A. Buchmann



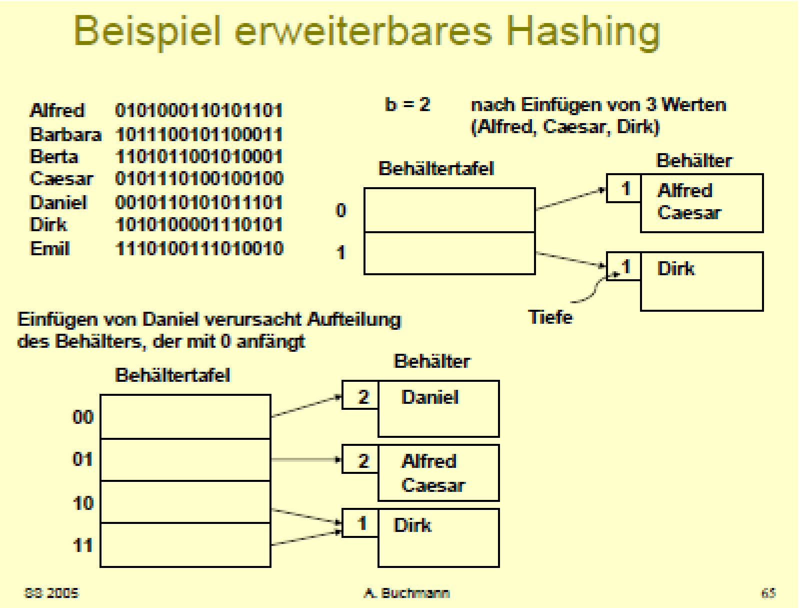
Exercise 4.1
File Organization Techniques
File Indexes and Access Paths

Dynamic Hashing:

- # buckets grows/shrinks dynamically, no overflow pages used
- better performance for dynamic files
- two most popular implementation variants: **extendible hashing** (with cost usually 2 I/Os), **linear hashing** (with cost usually 1 I/O)

- As known from GDI2 →

GDI2 Script SS2005 A. Buchmann



Last time….

- File Organization(s)
  - Heap/Sorted/Hash

- Access Paths
  - Scanning & Searching
  - Indexing (Hash, Trees...)
  - ISAM

This Session….

- Case Study B-Tree in Informix
- (External) Merge Sort, Sorting
- Joins
- Cost Functions

Exercise 4.2

- **Case Study "B-Tree Index in Informix"**

Exercise 4.2
Case Study "B-Tree Index in Informix"

TECHNISCHE UNIVERSITÄT DARMSTADT

Physical layout of a B+Tree index will be examined.
The *page size is 2048 Bytes*.

Each index page contains the following data:

Page Header	24 Bytes
Slot Table	4 bytes per slot entry
Non-leaf pages (Index Entries)	(s + 4) bytes/entry (SKEY: s; POINTER: 4)
Leaf pages (Data Entries)	(s + 5) bytes/ entry (SKEY: s; RID: 4; Delete Flag 1)
Page-Ending Timestamp	4 Bytes
Char(8)	8 Bytes

Exercise 4.2
Case Study "B-Tree Index in Informix"

Note:

s = Size of Search Key (SKEY) in bytes

Suppose that an index on a CHAR(8) primary key attribute is to be created.

s := 8 bytes

Exercise 4.2
Case Study "B-Tree Index in Informix"

TECHNISCHE UNIVERSITÄT DARMSTADT

a) How many **data entries at most can fit in a leaf-page** of the index?

b) How many **index entries at most can fit in a branch page** (non-leaf page) of the index?

c) What **depth** would the index have for 100.000 data entries (how many levels)?

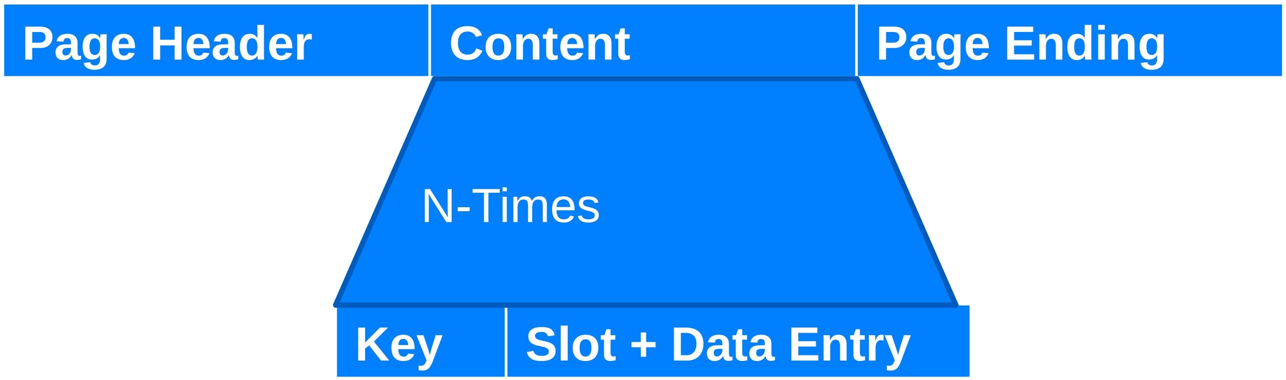
1. For a FILLFACTOR of 100%
2. For a FILLFACTOR of 75%

Note:

bfr := blocking factor - number of records that fit in a page.

Exercise 4.2
Case Study "B-Tree Index in Informix"

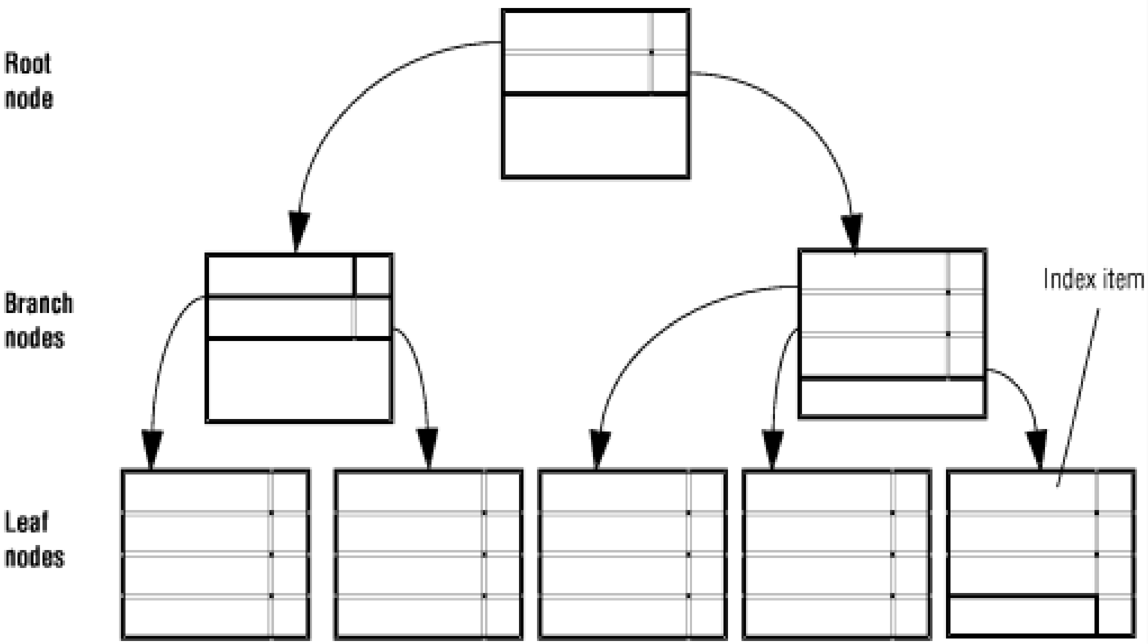
„How does a simple leaf-page look like?“

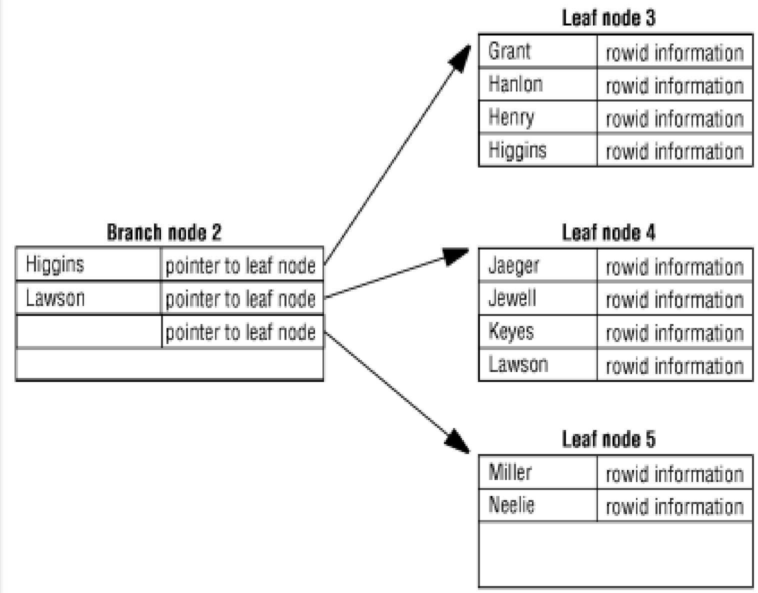


Remember: This is a ‚simplified‘ presentation

Exercise 4.2
Case Study "B-Tree Index in Informix"

„How does Informix B-Tree look like?“





*http://publib.boulder.**ibm.com**/infocenter/idshelp/v10/index.jsp?topic=/com.ibm.adref.doc/adref235.htm*

Exercise 4.2
Case Study "B-Tree Index in Informix"

a). How many data entries **at most** can fit in **a leaf-page** of the index?

bfr leaf:

$$2048 \geq \overbrace{24}^{\text{Page header}} + \overbrace{4}^{\text{Page Ending TS}} + N\overbrace{\left(s + 5 + 4\right)}^{\text{Data Entry +Slot}}$$

$$s = 8 \Rightarrow 2020 \geq N \cdot 17 \Leftrightarrow N \leq \frac{2020}{17} = 118.823$$

$$\Rightarrow N \leq 118 \;\; \textit{data entries per page}$$

Exercise 4.2
Case Study "B-Tree Index in Informix"

b). How many index entries **at most** can fit in a **branch page** (non-leaf page) of the index?

bfr branch:

$$2048 \geq \overset{\text{Page header}}{24} + \overset{\text{Page Ending TS}}{4} + M(s + \overset{\text{Data Entry +Slot}}{4} + 4) + \overset{\text{Extrapointer}}{4}$$

$$s = 8 \Rightarrow 2016 \geq M \cdot 16 \Leftrightarrow M \leq \frac{2016}{16} = 126$$

$$\Rightarrow M \leq 126 \text{ data entries per page}$$

Exercise 4.2
Case Study "B-Tree Index in Informix"

TECHNISCHE UNIVERSITÄT DARMSTADT

c) What **depth** would the index have for ***100.000 data entries*** (how many levels)?

1. For a FILLFACTOR of 100%
2. For a FILLFACTOR of 75%

Exercise 4.2
Case Study "B-Tree Index in Informix"

TECHNISCHE UNIVERSITÄT DARMSTADT

Index depth for 100.000 data entries

1.FILLFACTOR = 100%

$$no\ leaf\ \ nodes = \frac{no\ data\ entries}{bfr\ leaf} = \frac{100000}{118} \approx 848$$

$$\log_{126}(848) + 1 = \frac{\ln(848)}{\ln(126)} + 1 = 1.39 + 1 = 3$$

126 Data Entries
per branch-page

Exercise 4.2
Case Study "B-Tree Index in Informix"

TECHNISCHE UNIVERSITÄT DARMSTADT

2.FILLFACTOR = 75%

$$brf\,leaf\ :\ 0.75\ bfr\ =\ 118 \cdot 0.75 = 89$$

$$brf\,branch\ :\ 0.75\ bfr = 126 \cdot 0.75 = 95$$

$$no\ leaf\ nodes\ =\ \frac{no\ data\ entries}{bfr\ leaf} = \frac{100000}{89} = 1123.59 \approx 1124$$

$$\log_{95}(1124) + 1 = \frac{\ln(1124)}{\ln(95)} + 1 = 1.54 + 1 \approx 3$$

Exercise 4.2
Case Study "B-Tree Index in Informix"

Index FILL-FACTOR:

- Typical: 67%
- Does not affect the tree-depth so much;
  - usually no more than 3 or 4 levels
- *Important for concurrency* (when index grows)
- Storage efficiency (space utilization)

Exercise 4.3

- **Internal and External Sorting Algorithms and Cost Functions**

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

a) Discuss the **general cost function for internal sorting** algorithms.

b) Describe the **external merge-sort algorithm** and derive a cost function for the # of page I/Os.

c) Discuss how the use of techniques such as blocked I/O and double buffering can improve the performance of the external merge-sort algorithm.

d) Discuss how B+Tree Indexes can be used for sorting. Consider both cases when the index is clustered and when the index is unclustered.

External Sorting - recommended literature: Ramakrishnan, Gehrke Database Management Systems Chapter 13

External Sort Merge: Chapter Sorting in „Database System Concepts“ by A. Silberschatz

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Why is it usefull to sort?

- Receive query answers in order
- bulk loading a tree index
- eliminate duplicate entries
- usage of a fast join algorithm

Why use External Sort?

- To be able to efficiently sort large amounts of data that exeed the available bufferspace „Data is too large to fit into ram“
- Minimize the disk access

Sort Merge Join:

- Cost function **sorted**: *C = b**r** + b**s** + (b**rs**)*
- Cost function **unsorted**: *C= b**r** + b**s** + b**r** log b**r** + b**s** log b**s** + (b**rs**)*

Where is external sort used?

- IBM DB2, Microsoft SQL Server, Oracle 8, Sybase ASE use External Sort

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

b) Sorting of relations that do not fit in memory = external Sorting

Stage 1:
Create number of **Sorted Runs**; each run is a part of the relation but contains only **sorted records**.
(Read Part of Relation; Sort it in Memory; Write sorted „run“ out)
Each part contains M or less blocks of the relation given a total of N runs.

Stage 2:
N Runs are merged.

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

Database System Concepts. A. Silberschatz page 548

G	24
A	19
D	31
C	33
B	14
E	16
R	16
G	21
M	3
P	2
D	7
A	14

A	19
D	31
G	24

B	14
C	33
E	16

D	21
M	3
R	16

A	14
D	7
P	2

A	19
B	14
C	33
D	31
E	16
G	24

A	14
D	7
D	21
M	3
P	2
R	16

A	14
A	19
B	14
C	33
D	7
D	21
D	31
E	16
G	24
M	3
P	2
R	16

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

c) Blocked Access and Double Buffering

Blocked access

reading multiple pages with a single I/O request

reduces disk positioning delays and can therefore lead to significant performance improvements.

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Blocked Access in context of **external-merge-sort algorithm**

- double the size of the input and output buffers during the merging phase
- will reduce the number of runs merged in a single pass (the fan-in), but will allow us to access disk in larger units.
- decrease the "per-page I/O cost" at the cost of increasing the number of passes.
  - (your absolute buffer size is still the same) – reading larger granules of sorted runs means you have less space for the merge, therefore more passes are necessary

→ This is a tradeoff which must be considered when implementing the external-sorting algorithm.

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

Blocked Access in context of **block-nested-loop-join algorithm**

- instead of allocating just one page for scanning the inner relation we can do better by **splitting the buffer space evenly between the two relations**
- results in more passes over the inner relation, but **cuts down on disk positioning costs**.

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Double Buffering

- *Idea:* To keep the CPU busy while an I/O request is being carried out -*to overlap CPU and I/O processing*.

- Double the size of each input buffer. When half of the buffer is filled, start processing data while continuing to read data in the second half. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delays for all but the first block transfer.

Exercise 4.3
Internal and External Sorting Algorithms and Cost Functions

- Note that although double buffering *can considerably reduce the **response time** for a given query*, it may not have a significant effect on **throughput**, because the CPU can be kept busy by working on other queries while waiting for an I/O request to complete. On the other hand, we shouldn't neglect the advantages that double buffering bring in enabling sequential accesses.

EQUI-JOIN Implementations and Cost Functions

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

a) ***Describe the algorithms*** for the following alternative implementations of EQUI-JOIN and ***derive the approximate cost functions*** (for I/O costs):
   - nested-loops-join (block-oriented)
   - index-join with Primary Index
   - index-join with Secondary Index
   - sort-merge-join
   - hash-join

b) Show that for a block-nested-loop EQUI-JOIN of relations R and S where |R|<<|S|, it is more cost-effective to use R as the ***outer relation*** (resp. S as the inner).
   Assume that, if the buffer size is *B pages*, *(B-2) pages are used for the relation in the outer loop* and 1 page is used for the relation in the inner loop. The remaining 1 page is used as an output buffer for the resulting relation.

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

M = # pages of R
N = # pages of S
B = # pages in buffer
|R| = # tuples of R
|S| = # tuples of S

$$R \bowtie_{X=Y} S$$

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Nested-loops-join (not block oriented)

foreach tuple r ∈ **R do**

    foreach tuple s ∈ **S do**

        if ri == sj then add ⟨r,s⟩ to result

flashyDB DVS

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Nested-loops-join (block oriented)

Use:
(B -2) buffer pages to scan R
1 page to scan S
1 page to write result - output buffer

foreach ***BLOCKS*** **of R of size (B-2) pages:**
scan S sequentially and probe for matching tuples
write tuples of result using the output buffer page

Smaller relation should be used in second foreach (S) (see b) later)

Special case: |R|<(B-2)

Exercise 4.4
EQUI-JOIN Implementations and C… …ns

Nested-loops-join (block oriente…

Every page of M one time (outer relation)

N pages of S are scanned sequentially for each BLOCK of R

$$Cost: M + \left(\left\lceil \frac{M}{B-2} \right\rceil \cdot N \right) + result$$

Every page of N one time per iteration (inner relation)

$$result = \left\lceil \frac{js \cdot |R| |S|}{bfr_{RES}} \right\rceil$$

=#iterations $\left\lceil \frac{M}{B-2} \right\rceil$

IO for result

Note: **js := join selectivity** := Ratio of tuples in result to tuples in cross product

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Index-Join (Index-Nested-Loop

x+1:
x block accesses to locate index entry
1 block access to read tuples
(!simplified!)

Assume that ***S*** has a ***Primary Index o***

foreach **BLOCK 'b'** of R:
foreach **tuple 't'** in BLOCK '**b**'
use the index to probe for matching tuples

$$Cost: M + |R|(x+1) + result$$

x: number of ***page accesses to locate index*** data entry (leaf page for tree index, bucket for hash index)
Typically **x** is:
2-4 for a ***B-tree index***
1-2 for a ***hash index***

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Index-Join (Index-Nested-Loops-Join) with Secondary Idx

Assume that ***S*** has a ***Secondary Index on the join attributes***

For all BLOCK **'b'** of R:

  For all tuple **'t'** in BLOCK '**b**'

    use the index to probe for matching tuples

simplified

$$Cost: M + |R|(x + y) + result$$

x: number of ***page accesses to locate index*** data entry

y: number of ***accesses to retrieve matching tuples*** of S.

Depends on:

- number of matching tuples
- whether index is clustered

$$clustered\ index: y = \left\lceil \frac{\#matching\ tuples}{bfr_s} \right\rceil$$

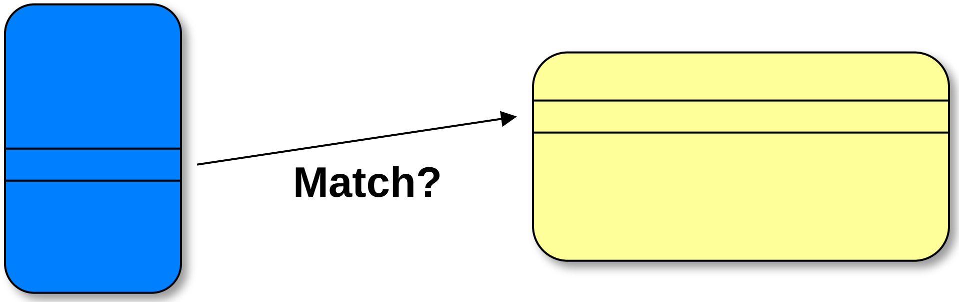
$$unclusted\ index: y \leq \#matching\ tuples$$

flashyDB DVS

TECHNISCHE UNIVERSITÄT DARMSTADT

Sort-Merge Join

- If relations are sorted on the join attribute a very efficient join is possible (especially for key attributes):
- Descend R and S and compare tuples
  - since tuples are in sorted order, *if no match is found*, we can advance and ***need not consider previous values***



Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Sort-Merge Join

Assume: ***no partition of S is scanned multiple times*** (or the necessary pages are found in the buffer after the first pass).

- This is always the case for key-foreign key joins.

Method:

- ***Sorting Phase (Partitioning Phase):***
  Sort R and S using an external sort algorithm (e.g. ***external merge sort***)
- ***Merging Phase (Probing Phase):***
  Scan R and S looking for matching tuples

$$Cost : O(M \log(M)) + O(N \log(N)) + M + N + result$$

Sorting

Merging

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Refinement of Sort-Merge Join

Refinement:

Combine merging phase of external sort with merging phase of the join.

Runs of R merged in the same time as runs of S merged - resulting R and S streams probed for matching tuples on the fly.

Requirement: B up to $\sqrt{L}$

$\mathrm{L} = \text{Size of larger relation}$

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

Assume one record per page:

Fill factor =0.5

R =4 Pages ={7,2,6,4}

S =4 Pages ={2,7,4,6}

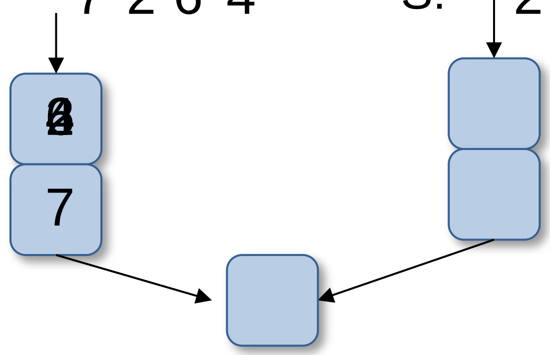
B = 5 Pages

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

R: 7 2 6 4

S: 2 7 4 6

I O Costs



(M+N)
(read)

Result R1: 2 6 7

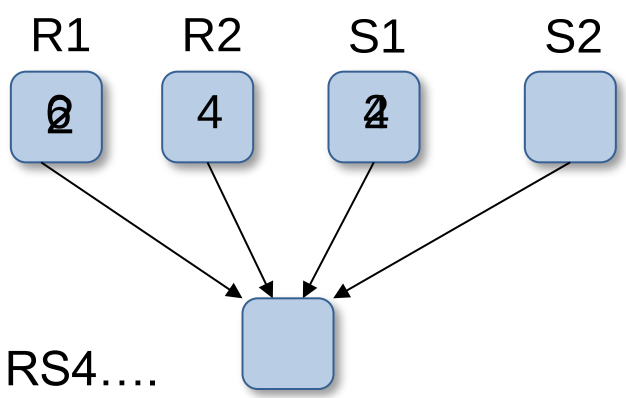
Result S1: 2 4 6 7

(M+N)
(write)

Result R2: 4

Result S2:

(M+N)
(read)



Result: RS2 RS4….

result
(write)

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

➔ $$Cost : 3(M + N) + result$$

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Hash-Join

Idea: Use hashing to partition instead of sorting

Method:

- ***Partitioning Phase:*** Apply hash function h1() on the join attributes to partition R and S each into *k* partitions (***k*<=B-1**, need 1 input page and 1 output page per partition).
- ***Probing Phase:***
  For all partitions R*i* of R:
    Load R*i* into an in-memory hash-table using h2()   h1()
    Scan corresponding partition Si of S, probing for matching tuples

Memory Requirements Hash Join

***f* = fudge factor** (accounts for increase in memory requirement of hashed vs. non-hashed partition)

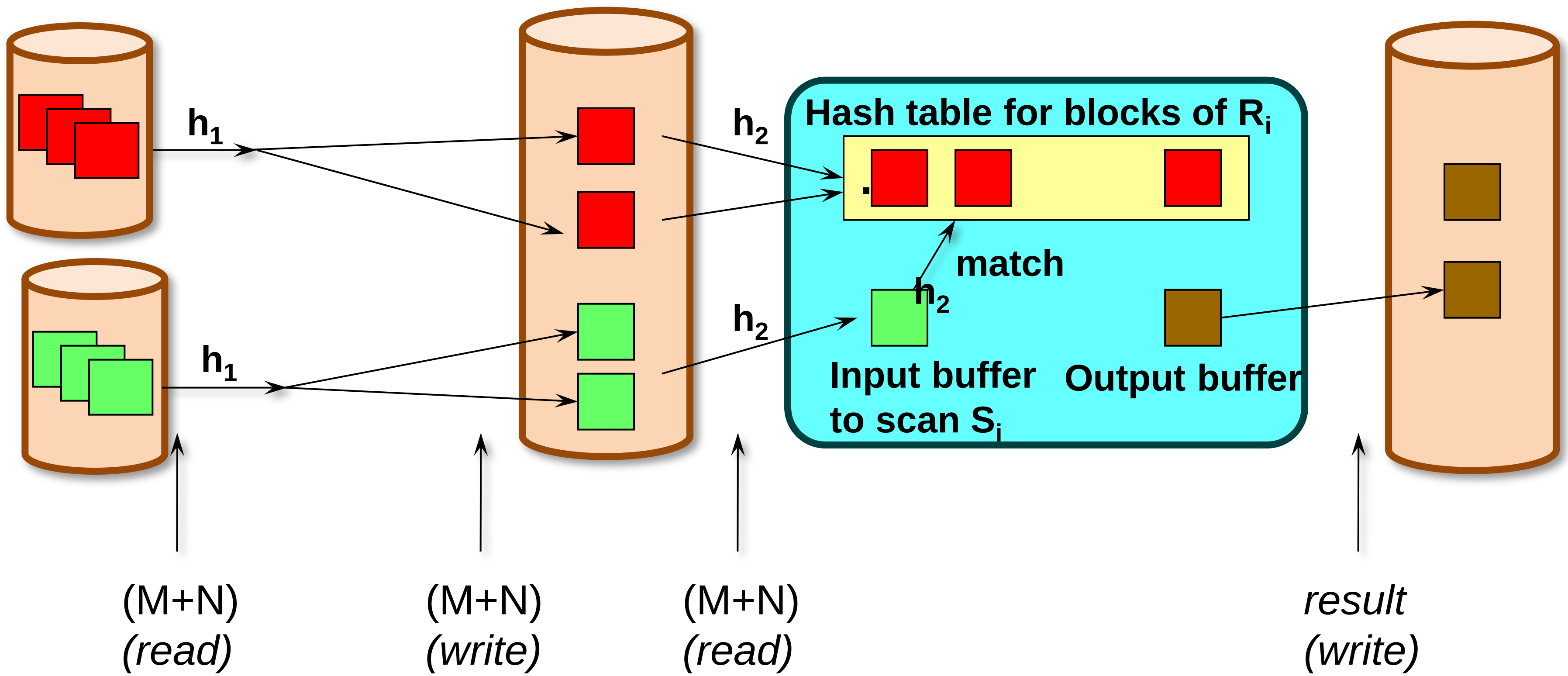
In Probing Phase: $Need: B \geq \frac{f \cdot M}{B-1} + 2 \Leftrightarrow B \geq \sqrt{f \cdot M}$

Buffers for input / output

If a ***partition overflows*** use another hash function to partition it further into smaller partitions (*apply technique recursively* until partitions small enough).

$\frac{M}{B-1}$ =Size of Partition of R

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions



➔ $$Cost: 3(M+N) + result$$

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

Hybrid-Hash-Join

f = fudge factor (accounts for increase in memory requirement of hashed vs. non-hashed partition)
k=number of partitions

Idea: Combines the partitioning phase with the joining of the first partitions of R and S. This saves writing/reading first partitions of R and S.

$$Need: B-((k-1)+2) \geq f\,\frac{M}{k} \Rightarrow B \geq ((k-1)+2)+f\,\frac{M}{k}$$

1 output per partition (without 1st)

Buffers for input / output

Size of 1st partition

TECHNISCHE UNIVERSITÄT DARMSTADT

$f.M$ = In-mem. hash table of M

Hybrid-Hash-Join – special case

If all partitions fit in memory ⇔ $B \geq f.M + 2$
(possible to build in-memory hash table of R and scan S probing for matching tuples)

Min. Cost = (M+N)+Res

flashyDB DVS

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

b) Show that for a **block-nested-loop EQUI-JOIN** of relations R and S where |R|<<|S|, it is more cost-effective to use R as the ***outer relation*** (resp. S as the inner).

Assume that, if the buffer size is *B pages*, *(B-2) pages are used for the relation in the outer loop* and 1 page is used for the relation in the inner loop. The remaining 1 page is used as an output buffer for the resulting relation.

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

Nested-loops-join (block oriented) (*reminder*)

N pages of S are scanned sequentially for each BLOCK of R

$$Cost: M + \left(\left\lceil \frac{M}{B-2} \right\rceil \cdot N\right) + result$$

$$result = \left\lceil \frac{js \cdot |R||S|}{bfr_{RES}} \right\rceil$$

Every page of M one time (outer relation)

Every page of N one time per iteration (inner relation)

$\left\lceil \frac{M}{B-2} \right\rceil$ =#iterations

Note: **js := join selectivity** := Ratio of tuples in result to tuples in cross product

Exercise 4.4
EQUI-JOIN Implementations and Cost Functions

i) R outer, S inner

$$C1 = M + \left(\left\lceil \frac{M}{B-2} \right\rceil \cdot N\right) + result$$

|R|<<|S|

ii) S outer, R inner

$$C2 = N + \left(\left\lceil \frac{N}{B-2} \right\rceil \cdot M\right) + result$$

$$C1 - C2 = M - N + \left(\left\lceil \frac{M}{B-2} \right\rceil \cdot N - \left\lceil \frac{N}{B-2} \right\rceil \cdot M\right)$$

$$\left\lceil \frac{M}{B-2} \right\rceil \cdot N - \left\lceil \frac{N}{B-2} \right\rceil \cdot M \approx 0 \; for \; M, N >> B-2$$

$$\Rightarrow C1 - C2 \approx M - N$$

$$|R| << |S| \Rightarrow M << N \Rightarrow C1 - C2 < 0 \Rightarrow C1 < C2$$

Exercise 4.5

Case Study
"Estimating the I/O costs of EQUI-JOIN"

Exercise 4.5
Case Study "Estimating the I/O costs of EQUI-JOIN"

TECHNISCHE UNIVERSITÄT DARMSTADT

Given is the following relational schema (PK attributes are underlined):

STUDENT(<u>MNR</u>, NAME, VORNAME, FB, SEM)

FACHBEREICH(<u>FBNR</u>, BEZ, ANSCHRIFT, DEKAN)

The ***record size for STUDENT is 128 Bytes, for FACHBEREICH 256*** Bytes, the ***file block size (and database page size) is 4096 Bytes***. The relation ***STUDENT*** has a ***clustered index on FB*** (Fachbereichsnummer) with **depth x=3**. There are ***17980*** students in total and ***31*** Fachbereiche. Every student is registered in exactly one Fachbereich. We assume that students are uniformly distributed among Fachbereiche, that is, every Fachbereich has the same number of registered students.

Calculate the average expected costs for an Equi-Join:

STUDENT $_{*FB=FBNR}$ FACHBEREICH

utilizing the available access paths on the STUDENT relation. The resulting relation should contain all attributes of STUDENT and FACHBEREICH.

Exercise 4.5
Case Study "Estimating the I/O costs of EQUI-JOIN"

TECHNISCHE UNIVERSITÄT DARMSTADT

In order to do this proceed as follows:

- ***Derive the general formula for estimation of the I/O costs*** (in block accesses).
- Calculate ***the number of the blocks needed for storage of the FACHBEREICH relation***, as well as ***the blocking factor of STUDENT*** and the ***blocking factor of the resulting relation.***
- Derive an ***estimation for the Join-Selectivity***.
- Calculate the expected value of the ***selection cardinality*** STUDENT.

Exercise 4.5
Case Study "Estimating the I/O costs of EQUI-JOIN"

TECHNISCHE UNIVERSITÄT DARMSTADT

Every page of M one time (outer relation)

Every page of N one time per iteration (inner relation)

$\left\lceil \frac{M}{B-2} \right\rceil$ =#iterations

$$Cost: M + \left(\left\lceil \frac{M}{B-2} \right\rceil \cdot N\right) + result$$

$$result = \left\lceil \frac{js \cdot |R| \, |S|}{bfr_{RES}} \right\rceil$$

Note: **js := join selectivity** := Ratio of tuples in result to tuples in cross product

Exercise 4.5
Case Study "Estimating the I/O costs of EQUI-JOIN"

Blocks of FB:

$$b_{FB} = \left\lceil \frac{256 \cdot 31}{4096} \right\rceil = 2$$

$$S_S = \frac{17980}{31} = 580$$

Selection cardinality of S on FB

(since students uniformly distributed among FBs)

$$Cost = b_{FB} + (|FB| \cdot (x + \left\lceil \frac{S_S}{bfr_S} \right\rceil)) + js \cdot \frac{|F||S|}{bfr_{RES}}$$

x=3

$$bfr_{RES} = \left\lfloor \frac{4096}{128+256} \right\rfloor = \left\lfloor 10\frac{2}{3} \right\rfloor = 10$$

$$bfr_S = \left\lfloor \frac{4096}{128} \right\rfloor = 32$$

$$js = \frac{|S \text{ join } F|}{|S||F|} = \frac{|S|}{|S||F|} = \frac{1}{|F|} = \frac{1}{31}$$

(FBNR is PK!)

Exercise 4.5
Case Study "Estimating the I/O costs of EQUI-JOIN"

$$C = 2 + (31 \cdot (3 + \left\lceil \frac{580}{32} \right\rceil)) + \frac{1}{31} \cdot \frac{31 \cdot 17980}{10}$$

$$C = 684 + 1798 = 2482 \text{ page accesses}$$

Exercise 4.6

TECHNISCHE UNIVERSITÄT DARMSTADT

Selection Costs

(Database Management Systems 15.2)

Exercise 4.6
Selection Costs

TECHNISCHE UNIVERSITÄT DARMSTADT

Part I

Employees(<u>eid: integer</u>, ename: string, sal: integer, title: string, age: integer)

Suppose that the following indexes exist: a hash index on ***eid***, a B+ tree index on ***sal***, a hash index on ***age***, and a clustered B+ tree index on <***age sal***>
Each Employees record is 100 bytes long, and you can assume that each index data entry is 20 bytes long. The Employees relation contains 10 000 pages (20 relations per page).
Consider each of the following selection conditions and, assuming that the reduc- tion factor (RF) for each term that matches an index is 0.1, compute the cost of the most selective access path for retrieving all Employees tuples that satisfy the Condition:

(a) sal > 100
(b) age = 25
(c) age > 20
(d) eid = 1, 000
(e) sal > 200 ∧ age > 30
(f) sal > 200 ∧ age = 20
(g) sal > 200 ∧ title =‚C F O‘

Exercise 4.6
Selection Costs

TECHNISCHE UNIVERSITÄT DARMSTADT

(a) sal > 100

No clustered index on sal:

A filescan would probably be the best

B+ Tree lookup

Using the unclustered index :

2+10,000 pages * (20bytes / 100bytes) * 0.1 +
10,000 pages * 20 tuples per page * 0.1 = ***20 202***

Record lookup

Filescan:

10,000 Pages

Exercise 4.6
Selection Costs

(b) age = 25

Possiblities:

Clustered B+ Tree Index <***age sal***> vs. Hash Index on **age**

B+ Tree lookup

Clustered B+ tree index:

2 + 10,000 * 0.4 * 0.1 + 10,000 pages * 0.1 = ***1402***

Record lookup

Hash Index:

10 000 * 0.1 * 20 = ***20 000***

No. Tuppels per Page

Exercise 4.6
Selection Costs

(c) age > 20

Clustered B+ tree index

2 + 10,000 * 0.2 * 0.1 + 10,000 pages * 0.1 = ***1202***

Again B+ tree is the best.

Exercise 4.6
Selection Costs

(d) eid = 1, 000

eid is a candiate key

➔ One can assume that only one record will be found.

Total cost is roughly 1.2 (lookup) + 1 (record access) ➔Total costs are 2 or 3.

Exercise 4.6
Selection Costs

(e) sal > 200 $\wedge$ age > 30

If the age > 30 clause is examined first:

Similar to the age > 20 case.

Total Costs: 1202

Exercise 4.6
Selection Costs

TECHNISCHE UNIVERSITÄT DARMSTADT

(f) sal > 200 ∧ age = 20

Again Clustered B+ index on **< age, sal >** since only 10 % of all relations fulfill sal > 200.

Assuming *a linear distribution* of values for sal for age, one can assume a cost of:

2 + 10,000 pages * 0.1 * 0.1 + 10,000 * 0.4 * 0.1 * 0.1 = 142.

selectivity for sal

selectivity for age

Exercise 4.6
Selection Costs

(g) sal > 200 ∧ title = ‘C F O’

Filescan is the best available method to use.

Total cost: 10,000

Exercise 4.6
Selection Costs

TECHNISCHE UNIVERSITÄT DARMSTADT

Part II

Suppose that, for each of the preceding selection conditions, you want to retrieve the ***average salary*** of qualifying tuples.

For each selection condition, describe the least expensive evaluation method and state its cost.

Exercise 4.6
Selection Costs

(a) sal > 100

The result is only the average salary.

An index-only scan can be performed

Unclusterd B+ tree on *sal* for a total cost:

2 + 10,000 * 0.1 * 0.2 = 202

Exercise 4.6
Selection Costs

(b) age = 25

To avoid a relational lookup:

Use the clustered index on < age, sal >

Total cost:

2 + 10000 * 0.1 * 0.4 = 402.

Exercise 4.6
Selection Costs

(c) age > 20

Similar to the age = 25 case

Total Cost: 402

Exercise 4.6
Selection Costs

(d) eid = 1000

Candidate key

Therefore: using the hash index again is the best option:

Total Cost: 1.2 + 1 = 2.2

hash lookup

relation retrieval

Exercise 4.6
Selection Costs

(e) sal > 200 ∧ age > 30

Using the clustered B+ tree again

Total cost: 402

Exercise 4.6
Selection Costs

(f) sal > 200 ∧ age = 20

Similarly to the sal > 200 ∧ age = 20 case in the previous problem:

Clustered B+ index for an index only scan.

Total Cost: 2 + 10000 * 0.1 * 0.1 * 0.4 = 42

B+ lookup

selectivity for sal

selectivity for age

smaller tuple sizes
index-only scan

Exercise 4.6
Selection Costs

(g) sal > 200 ∧ title = ‘C F O’

In this case, an index-only scan may not be used, and individual relations must be retrieved from the data pages.

The cheapest method available is a simple filescan. ***Total cost:*** 10,000 I/Os.

Exercise 4.7

Query Optimizer

(Database Management Systems 15.8)

Exercise 4.7
Query Optimizer

TECHNISCHE UNIVERSITÄT DARMSTADT

Consider the following relational schema and SQL query:

Suppliers(*sid:* integer, *sname*: char(20), *city*: char(20))

Supply(*sid:* integer, *pid:* integer)

Parts(*pid:* integer, *pname*: char(20), *price*: real)

SELECT S.sname, P.pname

FROM Suppliers S, Parts P, Supply Y

WHERE

S.sid = Y.sid **AND**

Y.pid = P.pid **AND**

S.city = ‘Madison’ **AND**

P.price <= 1 000

Exercise 4.7
Query Optimizer

a) What information about these relations does the query optimizer need to select a good query execution plan for the given query?

The query optimizer will need information such as ***what indexes exist*** (and what type) on:

S.sid, Y.sid, Y.pid, P.pid, S.city, P.price

It will also ***need statistics*** about the database such as low/high index values and distribution between fields.

Exercise 4.7
Query Optimizer

b) What indexes might be of help in processing this query? Explain briefly.

A sorted, clustered index on P.price would be useful for range retrieval.

A B+ Tree index on S.sid, Y.sid, Y.pid, P.pid could be used in an index-only sort-merge.

Exercise 4.7
Query Optimizer

c) How does adding DISTINCT to the SELECT clause affect the plans produced?

To support the DISTINCT selection, we must **sort** the results (unless they already are in sorted order) and scan for multiple occurences.

Exercise 4.7
Query Optimizer

d) How does adding GROUP BY sname to the query affect the plans produced?

The GROUP BY sname clause requires us:
- to sort the results of the earlier steps on sname, and
- to compute some aggregate (e.g., SUM) for each group (i.e., set of tuples with the same sname value).