

Chapter 4

Internal Organization and Implementation

This section heavily relies on other subdisciplines of Practical Computer Science:

- System Structures, down to the physical level
- Operating Systems Aspects: Caching
- Algorithms (mainly: for joins) and Data Structures (tree indexes, hashing)

156

PHYSICAL DATA ORGANIZATION

- the **conceptual schema** defines which data is described and its semantics.
- the **logical schema** defines the actual relation names with their attributes (and datatypes), keys, and integrity constraints.
- the **physical schema** defines the **physical database** where the data is actually stored.
⇒ efficiency
- system: the data is actually stored in **files**: data that semantically belongs together (a relation, a part of a relation (**hashing**), some relations (**cluster**)).
- additionally, there are files that contain auxiliary information (**indexes**).
- data is accessed **pagewise** or **blockwise** (typically, 4KB – 8KB).
- each page contains some **records** (tuples). Records consist of **fields** that are of an elementary type, e.g., bit, integer, real, string, or pointer.

157

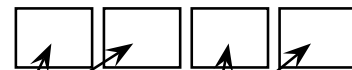
DB SERVER ARCHITECTURE: SECONDARY STORAGE AND CACHING

runtime server system: accessed by user queries/updates

- parser: translates into algebra, determines the required relations + indexes
- file manager: determines the file/page where the requested data is stored
- buffer/cache manager: provides relevant data in the cache
- query/update processing: uses only the cache

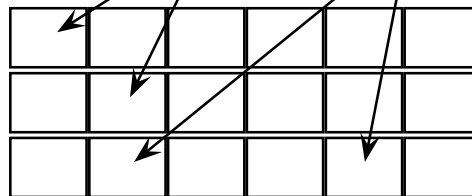
Cache (main memory): pagewise organized

- Accessed pages are fetched into the cache
- pages are also changed in the cache
- and written to the database later ...



Secondary Storage (Harddisk): pagewise organized

- data pages with tuples
- index pages with tree indexes (see later)
- database log etc. (see later)



158

DATABASE ACCESS MECHANISM

Records must be loaded from (and written to) the secondary memory for processing:

- the **file manager** determines the page where the record is stored.
- the **buffer/cache manager** is responsible to provide the page in the buffer (**buffer management**):
 - maintains a **pool** of pages (organized as frames).
for every page, it is stored if the page has been changed, how often/frequently it has been used, and if it is currently used by transactions
 - if the required page is not in the cache, some stored page is replaced (if it has been changed, it must be written to the secondary memory)
- complex **prefetching** strategies, based on knowledge about transactions.
[see lecture on Operating Systems]
- for now, it is sufficient to note that we have to deal with pagewise access.

159

Storage of Files, Pages, and Records

- Inside a file, every tuple/record has a **tuple identifier** of the form (p, n) where p is the page number and n is its index inside the page.
Each page then contains a **directory** that assigns a physical address to each n .
- memory management for deleted records
- different strategies for fixed-length and variable-length records

Simplified storage of a page of the Country table:

•	•	•	•	...																	
1	I	5	I	t	a	I	y	int: 301230	bigint: 57460274				4	R							
o	m	e	5	L	a	z	i	o	2	C	H	11	S	w	i	t	z	e	r	l	a
n	d	int: 41290			bigint: 7207060				4	B	e	r	n	2	B	E					
1	B	7	B	e	l	g	i	u	m	int: 30510			bigint: 10170241								
8	B	r	u	s	s	e	l	s	7	B	r	a	b	a	n	t	...				
⋮																					

... so far to the physical facts ...

160

4.1 Efficient Data Access

- efficiency depends on the detailed organization and additional algorithms and data structures
- support **generic** operations:
 - **Scan**: all pages that contain records are read.
 - **Equality Search**: all records that satisfy some equality predicate are read.
`SELECT * FROM City WHERE Country = 'D';`
 - **Range Search**: all records that satisfy some comparison predicate are read.
`SELECT * FROM City WHERE Population > 100.000;`
 - **Modify, Delete**: analogously
 - **Insert**: analogously: search for an appropriate place where to put the record.
- linear search (scan) ??
- **Need for efficient searching (equality and/or range)**

161

INDEXING

Indexes (for a file) are auxiliary structures that support special (non-linear) **access paths**

- Based on **search keys**
- not necessarily the relational “keys”, but *any* combination of attributes
- a relation may have several search keys
- an *index* is a set of data entries on some pages together with efficient access mechanisms for locating an entry according to its search key value.
- different types of indexes, depending on the operations to be supported:
 - equality search
 - range search (ordered values)
 - search on small domains
- in general, joins by key/foreign-key references are supported by indexes.

162

TREE INDEXES

This topic brings data structures and databases (= applications of data structures) together.

- introductory lecture “Computer Science I”: store numbers in trees.
- databases: tree *index* over the values of a column of a relation
 - search tree based on the values (numbers, strings)
 - the tuples themselves are not stored in the tree
 - entries (or leaf entries only) hold the values *and* point to the respective tuples in the database
- special trees with higher degrees:
 - each node (of the size of a storage page) has multiple entries and multiple children.

163

ASIDE, APPLICATION AND REVIEW: BINARY SEARCH TREES

- Binary Search Trees are a typical topic in “Computer Science I”: store numbers.
- Often used for *implementation* of other concepts, e.g., sets (cf. also topic “Abstract Datatypes” in some CS I lectures)
 - set: add(x), contains(x)?, list-all()
how to implement efficiently?
Java: classes TreeSet, HashSet

164

Example: BSB-based TreeSet in SQL Query Answering

[Exercise/Demonstrate on whiteboard]

- (1) `SELECT country FROM City WHERE population > 1000000;` contains duplicates.
- (2) `SELECT distinct country FROM City WHERE population > 1000000;`
 - initialize an empty TreeSet rs ,
 - evaluate (1),
 - during computation, for each result r :
 - if $r \notin rs$, then add it, and output r ;
 - requires $n \cdot \log n$ steps.
- sketch Java class: `treenode(left: node, right: node, value: String)`
- sketch representation of BSB in storage page/array (cf. CS I)
[node at position $n \rightarrow$ left child at position $2n$, right child at position $2n + 1$]
- (3) `SELECT country, count(*) FROM City WHERE population > 1000000
GROUP BY country;`
 - same strategy as above, additional data (count) stored in the tree,
 - if $r \notin rs$, then add $(r, count : 1)$ it, and output r ,
 - if $r \in rs$, then increment count of r .

165

B- AND B*-TREES

A **B-tree** (R. Bayer & E. McCreight, 1970) of order (m, ℓ) , $m \geq 3$, $\ell \geq 1$, is a search tree [see lecture on Algorithms and Data Structures]:

- the root is either a leaf or it has at least 2 children
- every inner node has at least $\lceil m/2 \rceil$ and at most m children
- all leaves are on the same level (balanced tree), and hold at most ℓ entries
- inner nodes have the form $(p_0, k_1, p_1, k_2, p_2, \dots, k_n, p_n)$ where $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ and
 - k_i are search key values, ordered by $k_i < k_j$ for $i < j$
 - p_i points to the $i + 1$ th child
 - all search key values in the left (i.e., p_{i-1}) subtree are less than the value of k_i (and all values in the right subtree are greater or equal)

B-trees are used for “simply” organizing items of an ordered set (e.g. for sorting) as an extension of binary search trees.

166

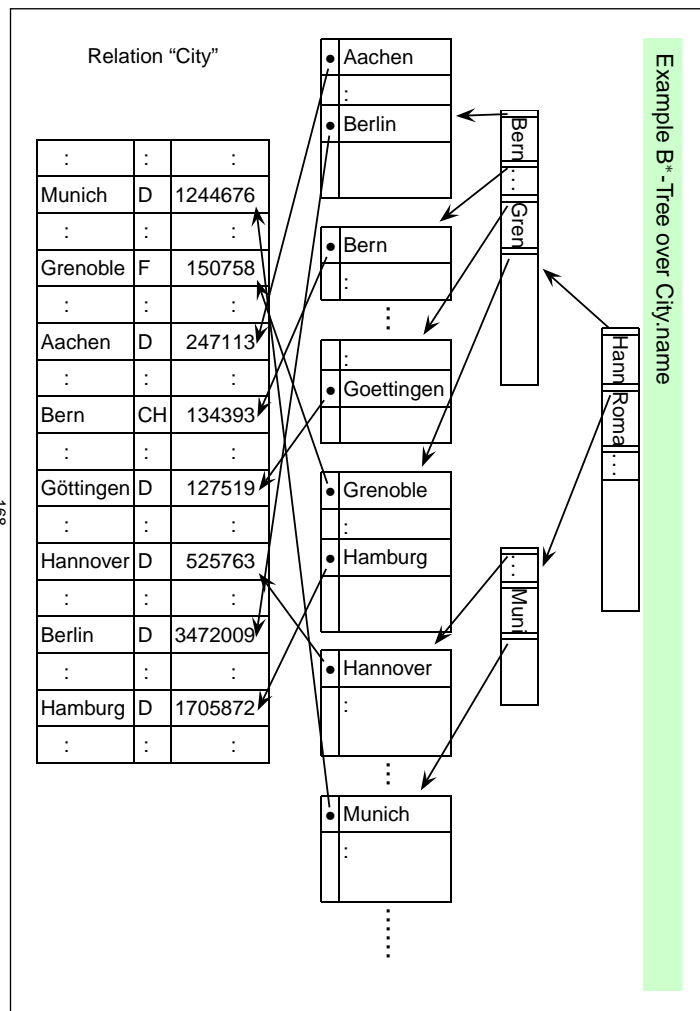
B*-TREES

(sometimes also called B⁺-Trees)

A **B*-tree** of order (m, ℓ) , $m \geq 3$, $\ell \geq 1$, is closely related, except:

- they are intended to associate *data* with search key values:
- the inner nodes do not hold additional data, but are still intended to guide the search. (organized internally e.g. as binary search trees)
- The leaves are of the form $([k_1, s_1], [k_2, s_2], \dots, [k_g, s_g])$ where $g \leq \ell$, k_i is a search key value, and s_i is a data record or (in databases) a pointer to the corresponding record.

167



Properties

- Let N the number of entries. Then, for the height h of the tree,
 - $h \leq \lceil \log_{m/2}(2N/\ell) \rceil$ ($\ell/2$ entries per leaf, inner nodes half filled) and
 - $h \geq \lceil \log_m(N/\ell) \rceil$ (ℓ entries/leaf, inner nodes completely filled)
- equality search needs h steps
inside each of the inner nodes, search is also in $O(n)$
- if the leaves are connected by pointers, ordered sequential access (range search) is also supported
- insertions and modifications may be expensive (tree reorganization)

Use of B*-Trees as Access Paths in Databases

- databases: cities stored in data files, index trees hold *pointers* to city records in their tree entries.
- separation between index files/pages and data files/pages.
- multiple search trees for each relation possible.

Example

Example 4.1

Consider the MONDIAL database with 3000 cities, with an index over the name. Assume the following sizes:

- every leaf (tuple) page contains 10 cities,
- every inner node contains 20 pointers

Then

- every inner node on the lowest level covers 200 cities
- every inner node on the second lowest level covers 4000 cities
- minimal: only one level of inner nodes
- maximal: two levels of inner nodes (nodes about only 2/3 filled)
- access every city with `WHERE Name = "..."` in 3 or 4 steps
- index on population, e.g., for `WHERE Population > 1.000.000 ORDER BY Population`
- realistic numbers: block size 4K: lowest level (keys+pointers to DB): 100 cities; inner nodes: 100 references. □

170

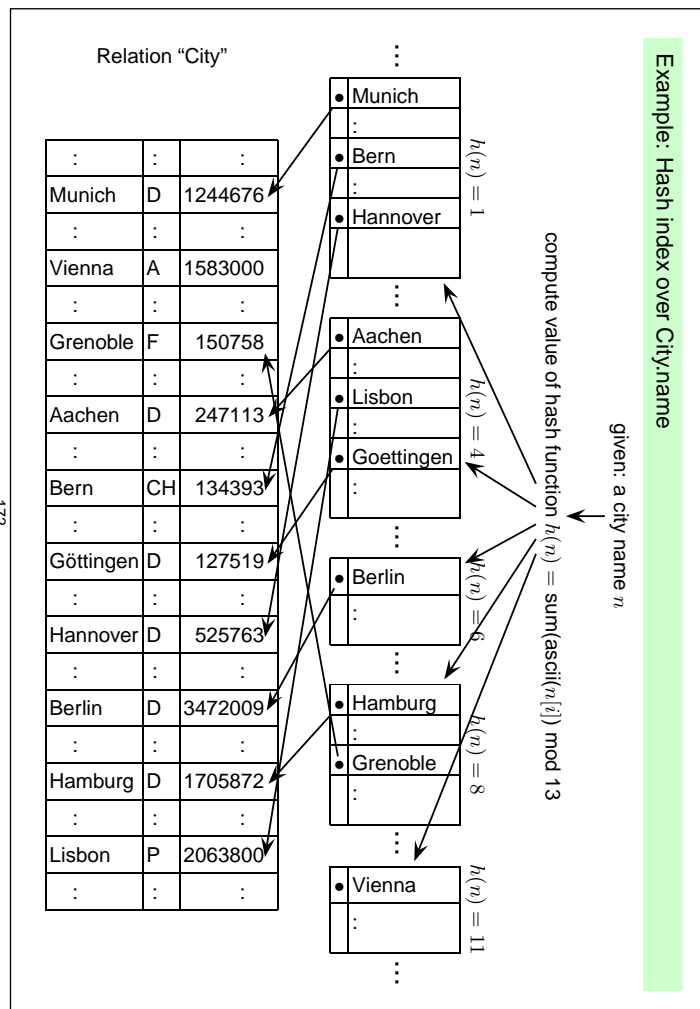
HASH-INDEX (DICTIONARY)

Hash index over the value of one or more columns ("hash key" – is not necessarily a key of the relation):

The values are distributed over k **tiles**.

- A **hash function** h is a function that maps each value to a tile number.
operation: `lookup(value)`
- each tile holds pairs (key,pointer) to all tuples whose hash-key value is mapped to this tile;
- each tile consists of one or more pages.
- common technique: convert value to an integer i . $i \bmod k$ gives the tile number.

171



Hash-Index

Example 4.2

Multi-attribute hash keys:

Consider a hash index on *City*.(name, province, country). h computes the sum of the ASCII numbers of the letters and takes the remainder modulo 111. □

Properties:

- equality search and insert in constant time (+ time for searching in the tile)
- does not support range queries or ordered output

Comments:

- maintenance of overflow pages: see Info III
- lookup inside each tile can be organized by a B-tree (kept on a single page)

TREE AND HASH INDEXES: OBSERVATIONS

- structure of tree leaf nodes and hash tiles is the same:
 - pairs: (search key value, reference to a tuple in storage page)
- tree leaf nodes and hash tiles are actually *projections* of a table R on the search key attribute/attributes
 - $\pi[\text{name}](\text{City})$ can be answered by just using the above tree/hash indexes.

⇒ indexes not only be as access paths, but also to support frequently used projections.

DUPLICATES IN TREE OR HASH INDEXES

What if $\pi[\text{search-key-attrs}](\text{relation})$ contains duplicates, e.g., index on City.country?

- straightforward: multiple subsequent (value, ref) entries for the same value in the leaf/tile pages,
- saves some space: pairs (value, set-of-references).
- any index that is not over a superset of a candidate key potentially contains duplicates.

174

BIT LISTS

If the number of possible values of a search key value is small wrt. the number of tuples, **bit lists** are useful as indexes

Let a an attribute of the records stored in a file f , where k values of a exist. Then, a bit list index to f consists of k bit vectors $B(v_i)$, $i \leq 1 \leq k$.

- $B(v_i)(j) = 1$ if the j th record in f has the value v_i for the attribute a .

Properties:

- access all tuples with a given value:
 - without bit list: linear search over all pages
 - with bit list: access bit list, and access those pages where a “1”-tuple is located.
- modifications: no problem
- deletions: depends (if gaps are allowed on the pages)

175

Example 4.3

Consider the relation $isMember(organization, country, type)$ where $type$ has only the values “member”, “applicant”, and “observer”:

<i>isMember</i>		
<i>Org.</i>	<i>Country</i>	<i>Type</i>
EU	D	member
UN	D	member
UN	CH	observer
UN	R	member
EU	PL	applicant
EU	CZ	applicant
:	:	:

Bit list for $type=member$: 1 1 0 1 0 0 ...

Bit list for $type=observer$: 0 0 1 0 0 0 ...

Bit list for $type=applicant$: 0 0 0 0 1 1 ...

Bit list for $org=EU$: 1 0 0 0 1 1 ...

Bit list for $org=UN$: 0 1 1 1 0 0 ...

- Search for members of the UN:
without bit list: linear search over all pages
with bit list: access bit list, and access those pages where a 1-tuple is located.
- 2nd bit list on organization column: “members of UN” by logical “and” of bit lists. □

ORDERED STORAGE

Tuples of a relation can be stored grouped/ordered wrt. one search key

- Example: table City grouped by country (or even by (country, province))

Relation “City”	Vienna	A	1583000
	Innsbruck	A	118000
	·	·	·
	Bern	CH	134393
	·	·	·
	Berlin	D	3472009
	Hamburg	D	1705872
	Munich	D	1244676
	Hannover	D	525763
	Göttingen	D	127519
	Aachen	D	247113
	·	·	·
	Paris	F	2152423
	Grenoble	F	150758
·	·	·	
Lisbon	P	2063800	
·	·	·	

- things that are frequently needed together can be fetched within the same page,
- Index on City.country needs only a reference to the first tuple in each country (note that the index is still useful to have access in $O(\log n)$ or $O(1)$).
- insertions and modifications more costly (strategies to allow and keep free space between blocks)

CLUSTERING

- keep data that semantically belongs together on the same pages
- obviously done for relations, by ordered storage even inside a relation
- data of several relations $R_1(\bar{X}_1), \dots, R_n(\bar{X}_n)$ can also be grouped to **clusters**:
 - choose a set $\bar{Y} \subseteq \bar{X}_1 \cap \dots \cap \bar{X}_n$ as **cluster key**.
 - combine relations by their \bar{Y} -values.
 - provides obvious advantages when evaluating joins over the cluster key.

178

Example 4.4

Consider the relation schemata

organization(name, abbrev, established, ...)

and

isMember(organization, country, type).

The foreign key

isMember.organization → *organization.abbreviation*
 is used as cluster key.

Organization		
Abbrev		
EU	Name	established
	European Union	07.02.1992
	Country	Type
	D	member
	A	member
	:	:
UN	Name	established
	United Nations	26.06.1945
	Country	Type
	D	member
	CH	observer
	:	:

179

4.1.1 Algorithms and Data Structures: Basic Techniques

Iteration: all tuples in the input relations are processed iteratively. If possible, instead of the tuples themselves, an index can be used.

Indexes: selections and joins can be based on processing of an index for determining the tuples that satisfy the selection condition (a join condition is also a selection condition).

Indexes on single attributes: obvious.

Indexes on multiple attributes:

- a **hash index** to a conjunction of **equality** predicates of the form **field = value** can be used if every field of the search key occurs exactly in one predicate together with a constant,
- a **tree index** to a conjunction of **comparison** predicates of the form **field θ value** can be used if a prefix of the search key exists such that each field of the prefix occurs in exactly one predicate together with a constant.

Example: if (Country,Province,CityName) is the search key of an index, it can also be used as an index for the key's prefix (Country,Province).

180

4.1.2 Implementation of Algebra Operations

SELECTION

Selection: $\sigma[R.field \theta value]R$.

- no index, unordered tuples: linear scan of the file
- no index, tuples ordered wrt. *field*: find the tuple(s) that satisfy "*field θ value*" (binary search) and process these tuples.
- tree index: find the tuple(s) that satisfy "*field θ value*" using the tree index and process these tuples.
- hash: suitable only if θ is equality.

181

Selection: boolean combination of predicates

Boolean combinations of predicates can be evaluated by set operations on references (from the tree leaves and hash tiles):

Consider $\pi[\text{name}](\sigma[\text{country}=\text{"D"} \wedge \text{population} > 500.000](\text{City}))$

- hash index on City.country,
- tree index on City.population (supports range search for > 500.000)

Leaf entries and entries on hash tiles are of the form

(search-key-value, reference-to-tuple)

(i) hash lookup on City.country="D" results in a set of references,

(ii) index search on City.population > 500.000 results in a set of references.

- compute intersection:
- put (i) into a TreeSet or HashSet s (fits in main memory)
- for each reference in (ii): if it is contained in s , access actual tuple and output its name.

182

PROJECTION

Projection: $\pi[\text{field}_1, \text{field}_2, \dots, \text{field}_m]R$.

Main problem: remove duplicates (relational algebra does not allow duplicates, SQL does).

- if an index over $\text{field}_1, \text{field}_2, \dots, \text{field}_m$ or a superset of it exists: scan only the index leaf nodes and apply a projection to them.
- by sorting:
 - scan the file, create a new file with projected tuples.
 - sort the new file (over all fields, $n \log n$).
 - scan the result, remove duplicates.
- by hash: scan the file, put the projected tuples into a hash.
 - duplicates all end up in the same tile (still in files).
 - remove duplicates separately for each tile (iterating the hashing process with different functions until tile fits in main memory)
 - collect the tiles in the output file.

⇒ hashing not only as an index for search *structures*, but also as an *algorithm* for partitioning.

183

JOIN

Consider an Equijoin: $R \bowtie_{R.A=S.B} S$.

Nested-loop-join

```
foreach tuple  $r \in R$  do
  foreach tuple  $s \in S$  do
    if  $r_A = s_B$  then add  $[r, s]$  to result
```

Very inefficient:

- **assumption: only one page of each relation fits into main memory**
- m tuples of R per page, n tuples of S per page:
 $|R|/m + |R| \cdot |S|/n$ filesystem accesses (for each tuple of R , loop over all pages of S).
- $|R| \cdot |S|$ comparisons.

obvious: if possible, process the smaller relation in inner loop to keep it in main memory ($|R|/m + |S|/n$ filesystem accesses).

184

Block-nested-loop-Join

Optimization by page-oriented strategy:

Divide S into blocks that fit in main memory and process each of the blocks separately.

- **assumption: only one page of each relation fits into main memory**
- m tuples of R per page, n tuples of S per page:
 - take first page (R_1) of R and first page (S_1) of S ;
 - combine *each* tuple of R_1 with each tuple with S_1 .
 - continue with R_1 and second page (S_2) of S ... up to R_1 with S_{last}
 - continue with R_2 and S_1 etc.
- $|R|/m + |R|/m \cdot |S|/n$ filesystem accesses (for each page of R , loop over all *pages* of S).
- still $|R| \cdot |S|$ comparisons.
- **same for $k > 1$ available cache pages: read $k - 1$ pages from R for each block.**

185

Straightforward optimizations for Block-Nested-Loop-Join

For each block, some optimizations can be applied:
(that motivate also the subsequent algorithms)

- idea: do not consider cartesian product + selection as the base for join, but look up matching tuples in S .

Applied to the Block-Nested-Loop-Join base algorithm, these do not reduce the number of page accesses, but the number of comparisons:

- generate a temporary index on $S.B$ over the loaded fraction of S ;
- in case of duplicates of $R.A$: generate a temporary index also over $R.A$. on the single loaded page to process tuples groupwise;
- [do not discuss yet: generate temporary ordered indexes both on $R.A$ and $S.B$, process them merge-style]

next step: consider these optimizations on the global level of the algorithm

186

Index-Nested-Loop-Join

If for one of the input relations there is an index over the join attribute, choose it as the inner relation.

- given index on S 's join attribute
- loop over R , for each value access matching tuple(s) in S .
- $|R|/m + |Result|$ filesystem accesses,
- up to $|R|/m + |R| \cdot |S|$ filesystem accesses,
- maybe less efficient than block-nested-loop!

Parallelization

- Divide R into partitions and process each of them on a separate processor (share the index on S).

187

Sort-merge-join

Algorithm type: "Scan line"

First: simple case that illustrates the principle:

Assumption: relations are sorted wrt. $R.A$ and $S.B$

- search for matches as follows:
 - proceed through the ordered R and S stepwise, always doing a step in the index where the "smaller" value is.
- if a match is found:
 - generate a result tuple.
 - check for tuples which have the same values of the join attributes (must immediately follow this match in both relations).
- $|R|/m + |S|/n$ filesystem accesses.

General Case: relations are not sorted

... next slide

188

Sort-merge-join (cont'd)

- Sort relations first:
 - sort R according to $R.A$:
 - copy first k pages in cache, sort them in place (Quicksort), store them,
 - do this for pages $2k...3k - 1$ etc.
 - requires $2 \cdot |R|/m$ page accesses (read and write)
 - mergesort the sorted packets (linear; traverse all packets in parallel and write out in different place)
 - requires again $2 \cdot |R|/m$ page accesses (read and write)
 - do the same for S

Sort+merge: $4 \cdot |R|/m + 4 \cdot |S|/n + |R|/m + |S|/n$ page accesses

- [note: the step "mergesort the sorted packets" can be omitted by running merge join directly on all of them; then $2 \cdot |R|/m + 2 \cdot |S|/n + |R|/m + |S|/n$.]

189

Sort-merge-join (cont'd)

- Use only sorted indexes:
 - compute sorted indexes of both relations on the join attributes
(fit into main memory, requires $|R|/m + |S|/n$)
(if tree indexes on join attributes are available, simply use them)
 - search for matches as above:
 - * proceed through the ordered indexes R and S stepwise, always doing a step in the index where the “smaller” value is.
 - if a match is found:
 - * access actual tuples and generate a result tuple.
 - * check for tuples which have the same values of the join attributes (must immediately follow this match in both relations).
 - at most $|R|/m + |S|/n + |Result|$ filesystem accesses
(when processing duplicates with same $R.A = S.B$ value, less than $+ |Result|$)
 - with sorted indexes: costs depend on number of results (“selectivity of the join”).
- ⇒ decide to sort relations or to use indexes based on selectivity/heuristics.

190

Sort-merge-join: basic Algorithm

```
if R not sorted on attribute A, sort it;
if S not sorted on attribute B, sort it;
Tr := first tuple in R;
Ts := first tuple in S;
Gs := first tuple in S;
while Tr ≠ eof and Ts ≠ eof do {
    while Tr ≠ eof and Tr.A < Ts.B do Tr = next tuple in R after Tr;
    while Ts ≠ eof and Tr.A > Ts.B do Ts = next tuple in S after Ts;
    // now, Tr.A = Ts.B: match found
    while Tr ≠ eof and Tr.A = Ts.B do {
        Gs := Ts;
        while Gs ≠ eof and Gs.B = Tr.A do {
            add [Tr, Gs] to result;
            Gs = next tuple in S after Gs;}
        Tr = next tuple in R after Tr;
    }
    Ts := Gs;
}
```

191

Hash-join

Algorithm type: “Divide and Conquer”

Partitioning (building) phase:

- Partition the smaller relation R by a hash function h_1 applied to $R.A$.
- Partition the larger relation S by the same hash function applied to $S.B$.
(into different hash tables)

Matching (probing) phase:

- potential(!) matches have been mapped to “face-to-face” partitions.
- thus, consider each pair of corresponding partitions:
- if partition of R does not fit into main memory, proceed recursively with another h .
- otherwise – i.e., R -partition fits in main memory:
 - if corresponding S -partition also fits into main memory, compute the join.
 - otherwise proceed recursively with another (finer) h_2 inside main memory and process S -partition pagewise.

192

EXERCISE

Exercise 4.1

Consider the join of two relations R and S wrt. the join condition $R.A = S.B$. R uses M pages with p_R tuples each, and S uses N pages with p_S tuples each. Let $M = 1000$, $p_R = 100$, $N = 500$, $p_S = 80$; the cache can keep 100 pages. Compute the number of required I/O-operations for computing the join (without writing the result relation) for:

- simple nested-loop-join,
- pagewise nested-loop-join,
- index-nested-loop-join (Index on S_B),
- sort-merge-join (sorted relation/sorted index),
- hash-join.

□

193

SEMIJOINS

- Natural Semijoins or Condition Semijoins as subqueries:

$R \bowtie [\text{condition}(R_{i_1}, \dots, R_{i_1}, S_{i_1}, \dots, S_{i_\ell})]S$

It is often sufficient to use an index over $S_{i_1} \dots S_{i_\ell}$ instead of the whole relation.

- Semijoins as preparation for joins: $R \bowtie S = (R \bowtie S) \bowtie S$ (and symmetrically $= R \bowtie (R \bowtie S)$)

Is that simpler? – Sometimes Yes!

- cf. Join $R \bowtie S$ with if an existing index over the join attribute(s) in S : iterate over R and access S via index. No access, if not found.

This actually *is* $(R \bowtie S) \bowtie S$.

Consider $R \bowtie S$ to be a hash join:

- First hash S , build an index over the join attribute(s), keep the index in main memory.
- iterate over R : for every tuple μ , if $\mu \in (R \bowtie S)$ (can be checked against the main memory index), then hash it.

194

GROUPING AND AGGREGATION

General Structure:

SELECT A_1, \dots, A_n	list of attributes
FROM R_1, \dots, R_m	list of relations
WHERE F	condition(s)
GROUP BY B_1, \dots, B_k	list of grouping attributes
HAVING G	condition on groups
ORDER BY H	sort order

- SUM, AVG, COUNT: linear scan necessary (need to consider all tuples)
- MIN, MAX: index can be used
- support for grouping (SQL: GROUP BY) using an index or a hash table

UNION, DIFFERENCE, INTERSECTION \Leftrightarrow CARTESIAN PRODUCT

- intersection and cartesian product are special cases of join
- union and difference: analogous to sort-merge-join or hash-join

195

SITUATION-DEPENDENT OPTIMIZATION

Determining the optimal **execution plan** depends on **cost models**, **heuristics**, and the actual physical schema and the actual database contents:

- index structures
- statistics on data (i.e., to some extent state dependent)
 - cardinality of relations
 - distribution of values

196

Notions

The **selectivity** sel of operations can be used for estimating the size of the result relation:

- Selection with condition p :

$$sel_p = \frac{|\sigma[p](R)|}{|R|}$$

Proportion of tuples that satisfy the selection condition.

- for p an equality test $R.A = c$ where A is a key of R , $sel_p = 1/|R|$.
- if $R.A$ distributes evenly on i different values, $sel_p = 1/i$.

- Join of R and S :

$$sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

is the proportion of result tuples wrt. the cartesian product.

for $R \bowtie_{R.A=S.B} S$ with A a key attribute, $|R \bowtie_{R.A=S.B} S| \leq |S|$, thus $sel_{RS} \leq 1/|R|$.

- optimal order of applying joins
- optimal order of applying selections

197

APPLICATION-LEVEL ALGORITHMIC OPTIMIZATION

And **never** forget about using efficient algorithms for querying the database!

- analyze the problem from the **algorithmic** point of view
- before hacking

EXAMPLES

Use the MONDIAL database for the following examples.

Example 4.5

Compute all pairs of european countries that are adjacent to the same set of seas. □

Example 4.6

Compute all political organizations that have at least one member country on every continent (this operation is called relational division). □

198

4.1.3 Exercises and Examples

Solution to Exercise 4.1

R : 1000 pages, each of them with 100 tuples

S : 500 pages, each of them with 80 tuples

Cache: 100 pages

a) simple nested loop join

– outer loop: R

Load each page of R . For all tuples, iterate over S 's pages.

$$1000 \cdot (1 \text{ (load)} + 100 \text{ (tuples per page)} \cdot 500 \text{ (pages of } S)) = 50,001,000$$

– outer loop: S

Load each page of S . For all tuples, iterate over R 's pages.

$$500 \cdot (1 \text{ (load)} + 80 \text{ (tuples per page)} \cdot 1000 \text{ (pages of } R)) = 40,000,500$$

199

Solution to Exercise 4.1 (cont'd)

R: 1000 pages, each of them with 100 tuples

S: 500 pages, each of them with 80 tuples

Cache: 100 pages

b) pagewise nested loop:

- outer loop: *R*. Load each page of *R*. Combine all tuples of that page with all tuples from each page of *S*.

$$1000 \cdot (1 + 500 \text{ (pages of } S)) = 501,000$$

- outer loop: *S*. $500 \cdot (1 + 1000 \text{ (pages of } R)) = 505,000$

b2) maximum-pages nested loop: load as many (first 99) pages of *R* as possible in the cache and join with one page of *S* after the other. Then continue with 2nd 99 pages.

- 11 times, i.e., $10 \cdot 99$ *R* pages + $1 \cdot 10$ *R* pages through all *S* pages.

$$\text{Overall: } 1000 + 11 \cdot 500 = 6500.$$

- symmetric with $(5 \cdot 99 + 1 \cdot 5)$ pages of *S*:

$$\text{Overall: } 500 + 6 \cdot 1000 = 6500.$$

- can be algorithmically optimized by on-the-fly indexes during the main-memory join

200

Solution to Exercise 4.1 (cont'd)

R: 1000 pages, each of them with 100 tuples

S: 500 pages, each of them with 80 tuples; Cache: 100 pages

c) Index-nested-loop (index on *S.B*): Assumptions:

- every *R.A* matches – average – 4 *S*-tuples.
- index over *S.B* already exists (if not: creating an index over *S.B* requires 500 accesses) and can be kept in main memory

Iterate over *R*, for each tuple search for the value of *R.A* in the index over *S.B* and access the tuples

$$1000 \cdot (1 \text{ (load)} + 100 \cdot 4 \text{ (access tuples)}) = 401,000.$$

Note: the number of page accesses depends on the number of results since for every actual result there is one page access.

$$\text{In case that } S \text{ is ordered wrt. } S.B: 1000 \cdot (1 \text{ (load)} + 100 \cdot 1 \text{ (access tuples)}) = 101,000.$$

201

Solution to Exercise 4.1 (cont'd)

R: 1000 pages, each of them with 100 tuples

S: 500 pages, each of them with 80 tuples; Cache: 100 pages

d) sort-merge:

different settings

- if already ordered: linear scan:

$$1000 + 500 = 1500$$

- sort relations first:

$$2 \cdot 1000 + 2 \cdot 500 + 1000 + 500 = 4500$$

- using an index (assume: every *R.A* matches – average – 4 *S*-tuples):

$$1000 + 500 + 4 \cdot 1000 \cdot 100 = 401,500$$

Solution to Exercise 4.1 (cont'd)

R: 1000 pages, each of them with 100 tuples

S: 500 pages, each of them with 80 tuples

Cache: 100 pages

e) Hash-Join:

- Hash *R*:

100 pages fit in memory - read one page of *R* after the other and distribute over 99 partitions (whenever a page of a partition is full, move it to the disk).

In the average, each partition them contains about 10.1 (\rightarrow 11) pages; last page of each partition is not completely filled.

maximum 1000 (read) + 1089 (write) = 2099.

- Same for *S*. Get 99 partitions of average 5.1 (\rightarrow 6) pages (maximum 500 + 594 = 1094 accesses).

- now we have 99 corresponding pairs of partitions (one from *R*, one from *S*).

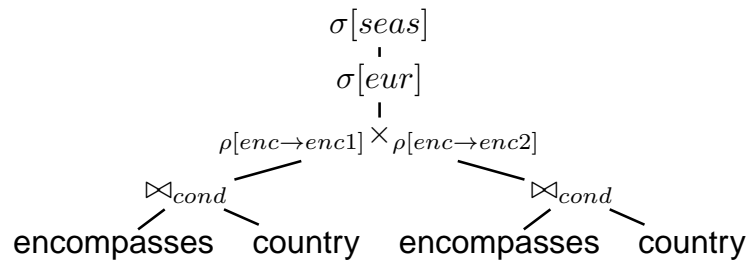
Join each pair: read 11 + 6 pages per partition and process them.

(about $99 \cdot (11 + 6) = 1683$ accesses).

Overall: 4876 accesses.

Solution to Exercise 4.5

First, compute all pairs of european countries. Note that we are interested in their names, thus we need also the *Country* relation.



- $cond = \text{"enc.country=country.code"}$.
- $eur = \text{"enc1.continent='europe' and enc2.continent='europe'"}$.
- $seas$ expresses that both countries border the same set of seas. It is a correlated subquery.
- add suitable projections.
- move $\sigma[eur]$ downwards both sides directly to *encompasses*
- obviously, both subtrees of \times are identical.

204

Solution to Exercise 4.5 (cont'd)

- $\sigma[seas(C_1, C_2)]$ is a correlated subquery that takes two country codes as input:

$$\sigma[seas(C_1, C_2)] = seas(C_1) - seas(C_2) = \emptyset \wedge seas(C_2) - seas(C_1) = \emptyset$$

$$= (seas(C_1) - seas(C_2)) \cup (seas(C_2) - seas(C_1)) = \emptyset$$
- $seas(C) = \pi[sea](\sigma[country = C](geo_sea))$
- for each country, $seas(C)$ is computed only once and then reused.

Resulting SQL skeleton (using subqueries in the FROM clause):

```
SELECT ...
FROM (SELECT european countries) as C1,
     (SELECT european countries) as C2
WHERE  $\sigma[seas(C_1, C_2)]$ 
```

205

Solution to Exercise 4.6

- $\pi[abbrev](\sigma[all_continents(org)])(organization)$ where $\sigma[all_continents(org)]$ is true for

$\{org \mid \forall cont : cont \text{ is a continent} \rightarrow org \text{ has a member on } cont\}$

- convert \forall into $\neg exists$:

$\{org \mid \neg \exists cont : cont \text{ is a continent and } org \text{ has no member on } cont\}$

- Thus, $\sigma[all_continents(org)]$ checks if

$\pi[name](continent) - \pi[enc.continent](\sigma[organization = org]isMember) \bowtie encompasses$

is empty.

Resulting SQL skeleton (uses a correlated subquery):

```
SELECT ...
FROM organization
WHERE NOT EXISTS
  ((SELECT continents)
   MINUS
   (SELECT continents where org has a member))
```