# Chapter 4
# Efficiency and Optimization

## PHYSICAL DATA ORGANIZATION

- the **conceptual schema** defines which data is described and its semantics.
- the **physical schema** defines the **physical database** where the data is actually stored.
  $\Rightarrow$ 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.



## DB SERVER ARCHITECTURE: SECONDARY STORAGE AND CACHING

runtime server system: accessed by user queries/updates

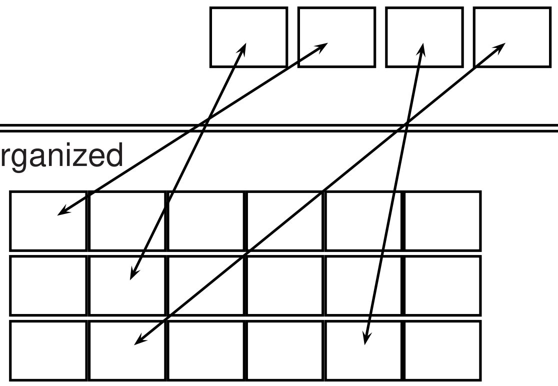
- parser: translates into algebra, determines the required relations + indexes
- file manager: determines the file/page where the requested data is stored
- buffer/cache manager: provides relevant data in the cache
- query/update processing: uses only the cache

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)



## 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.



### 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

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

## 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)



## 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.

## 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.



## B- AND B$^*$-TREES

A **B-tree** (R. Bayer & E.McCreight, 1970) of order $(m, \ell)$, $m \geq 3$, $\ell \geq 1$, 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 $\ell$ entries
- inner nodes have the form $(p_0, k_1, p_1, k_2, p_2, \ldots, k_n, p_n)$ 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.

# B*-Trees

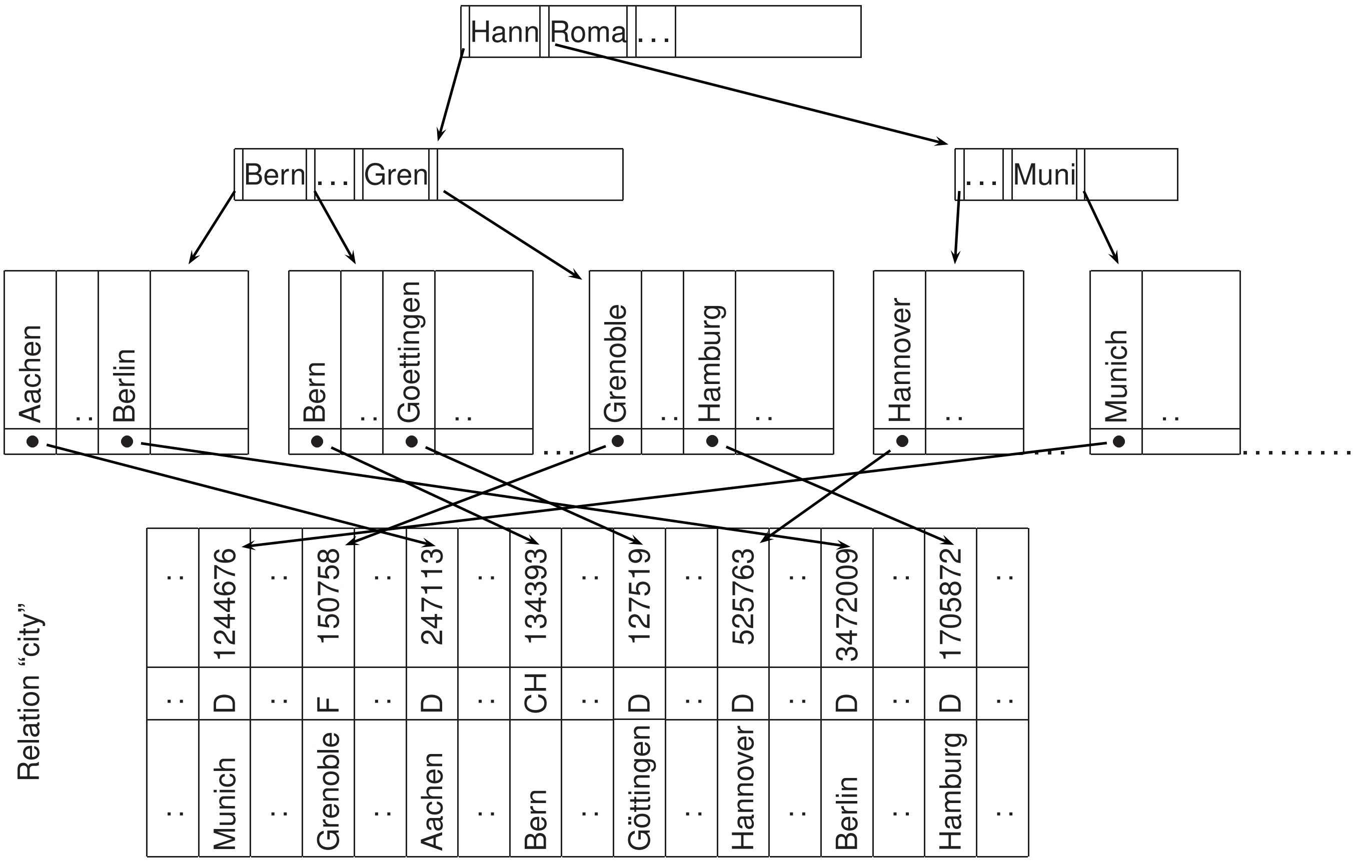
(sometimes also called B$^+$-Trees)

A **B*-tree** of order $(m, \ell)$, $m \geq 3$, $\ell \geq 1$, 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.
- The leaves are of the form $([k_1, s_1], [k_2, s_2], \ldots, [k_g, s_g])$ 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.



## Example B*-Tree over city.name



## 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
  $h \geq \lceil log_m(N/\ell) \rceil$ ($\ell$ entries/leaf, inner nodes completely filled)
- equality search needs $h$ steps
  inside each of the inner nodes, binary search is applied
- if the leaves are connected by pointers, ordered sequential access (range search) is also supported
- insertions and modifications may be expensive (tree reorganization)

## Use as Access Paths in Databases

- previous page: example tree holds actual city data in its leaves.
- 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.* □

## HASH-INDEX (DICTIONARY)

Hash index over the value of a column:
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 pointers to all tuples whose column value is mapped to this tile; each tile consists of one or more pages.
- common technique: convert value to an integer $i$. $i$ mod $k$ gives the tile number.

**Example 4.2**
*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



## 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)

**Example 4.3**
*Consider the relation is_member(organization, country, type) where type has only the values "member", "applicant", and "observer":*

| *is_member* | | |
|---|---|---|
| ***Org.*** | ***Country*** | ***Type*** |
| *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 ...

- *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: "members of UN" by logical "and" of bit lists.* □



## CLUSTERING

Data from the secondary storage is fetched by **pages**.
Pages are cached in the main memory.

- keep data that semantically belongs together on the same pages
- obviously done for relations
- data of several relations $R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n)$ can also be grouped to **clusters**:
  - choose a set $\bar{Y} \subseteq \bar{X}_1 \cap \ldots \cap \bar{X}_n$ as **cluster key**.
  - combine relations by their $\bar{Y}$-values.

**Example 4.4**

*Consider the relation schemata*

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

*and*

*is_member(organization, country, type).*

*The foreign key*

*is_member.organization* $\rightarrow$ *organization.abbrev*

*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* |
| | *:* | *:* |



## 4.2 Efficient Query Evaluation

Queries are formulated *declaratively* (e.g., SQL or algebra trees), actually built over a small set of basic operations (cf. the definition of the relational algebra).

**Semantical optimization:** consider integrity constraints in the database.

Example: $population > 0$, thus, a query that asks for negative values can be answered without explicit computation.

- not always obvious
- general case: first-order theorem proving.
- special cases: [see lecture on Database Theory]

**Logical/algebraic optimization:** search for an equivalent algebra expression that performs better:

- size of intermediate results,
- implementation of operators as algorithms,
- presence of indexes and order.

## 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).

- 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** $\theta$ **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 $(A_1, A_2, A_3)$ is the search key, $(A_1, A_2)$ is a prefix.



### 4.2.1 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 "$field\ \theta\ value$" (binary search) and process the these tuples.
- tree index: find the tuple(s) that satisfy "$field\ \theta\ value$" using the tree index and process these tuples.
- hash: suitable only if $\theta$ is equality.

\+ strategies for compound selection conditions.

## Projection

Projection: $\pi[field_1, field_2, ..., field_m]R$.

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

- 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 table.
  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.



## Join

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

**Nested-loop-join**

$\texttt{foreach}$ tuple $r \in R$ $\texttt{do}$

$\qquad \texttt{foreach}$ tuple $s \in S$ $\texttt{do}$

$\qquad\qquad \texttt{if}$ $r_A = s_B$ $\texttt{then}$ add $[r, s]$ to result

Very inefficient:

- assume: 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 ($|R|/m + |S|/m$ filesystem accesses).

## 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.

- assume: 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.

Further optimization:

when a page of $S$ is loaded, generate a temporary index over the join attribute(s) that is used in the comparison (then, only matching pairs are considered).



## 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 $|R| \cdot |S|$ filesystem accesses,
- maybe less efficient than block-nested-loop!

## Parallelization

Divide one of the relations into partitions and process each of them on a separate processor.

## Sort-merge-join

Algorithm type: "Scan line"

- sort both relations according to the join attributes
  (if tree indexes on join attributes are available, simply use them)
- search for matches as follows:
  - proceed through the ordered relations/indexes R and S stepwise, always doing a step in the relation/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).
- $|Result|$ filesystem accesses.
- much less, if relations are *stored* in this order.



**Sort-merge-join: 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` $Tr \neq eof$ `and` $Ts \neq eof$ `do` {
    `while` $Tr \neq eof$ `and` $Tr.A < Ts.B$ `do` $Tr$ = next tuple in $R$ after $Tr$;
    `while` $Ts \neq eof$ `and` $Tr.A > Ts.B$ `do` $Ts =$ next tuple in $S$ after $Ts$;
    // now, $Tr.A = Ts.B$: match found
    `while` $Tr \neq eof$ `and` $Tr.A = Ts.B$ `do` {
        $Gs := Ts$;
        `while` $Gs \neq eof$ `and` $Gs.B = Tr.A$ `do` {
            add $[Tr, Gs]$ to result;
            $Gs =$ next tuple in $S$ after $Gs$;}
        $Tr =$ next tuple in $R$ after $Tr$;
        }
    $Ts := Gs$;
}

## 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.



## EXERCISE

**Exercise 4.1**
*Consider the join of two relations $R$ and $S$ wrt. the join condition $R_i = S_j$. $R$ 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:*

*(a) simple nested-loop-join,*

*(b) pagewise nested-loop-join,*

*(c) index-nested-loop-join (Index on $S_j$),*

*(d) sort-merge-join,*

*(e) hash-join.* □

## Grouping and Aggregation

**General Structure:**

| | |
|---|---|
| `SELECT` $A_1, \ldots, A_n$ | list of attributes |
| `FROM` $R_1, \ldots, R_m$ | list of relations |
| `WHERE` $F$ | condition(s) |
| `GROUP BY` $B_1, \ldots, B_k$ | 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 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



## 4.2.2 Algebraic Optimization

The operator tree of an algebra expression provides a base for several optimization strategies:

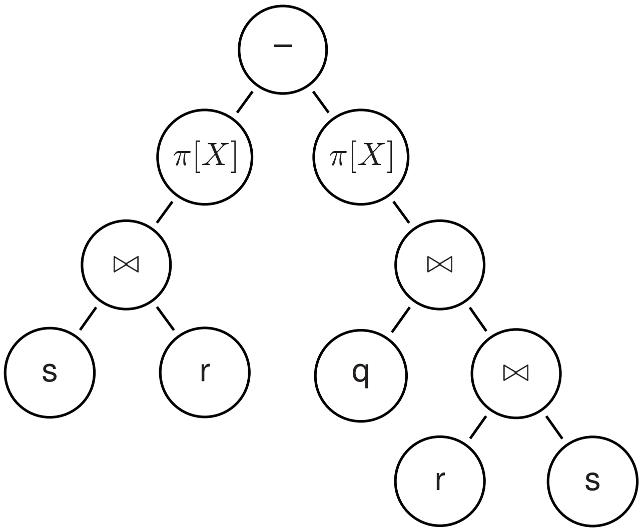
- reusing intermediate results
- equivalent restructuring of the operator tree
- "shortcuts" by melting several operators into one
  (e.g., join + equality predicate $\to$ equijoin)
- combination with actual situation: indexes, properties of data

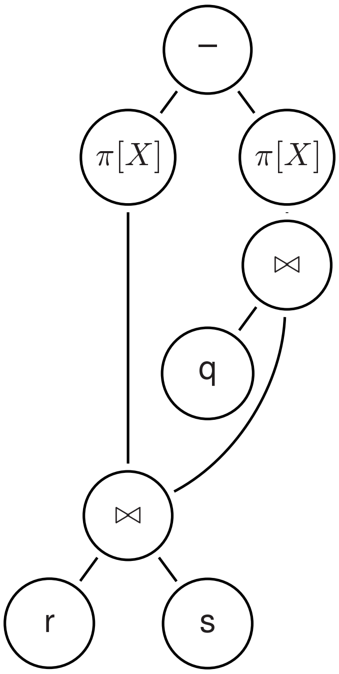
Real-life databases implement this functionality.

- SQL: **declarative** specification of a query
- internal: algebra tree + optimizations

## Reusing intermediate results

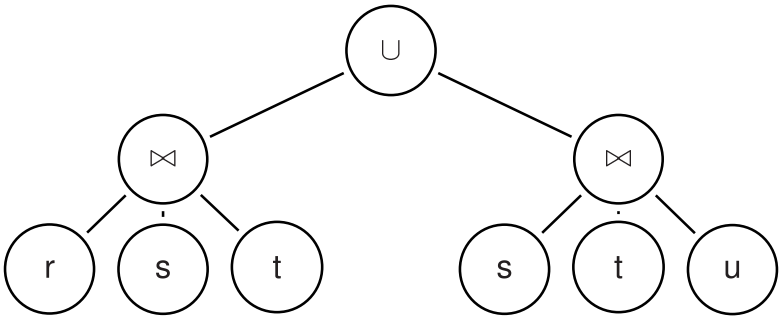
- Multiply occurring subtrees can be reused
  (directed acyclic graph (DAG) instead of algebra tree)

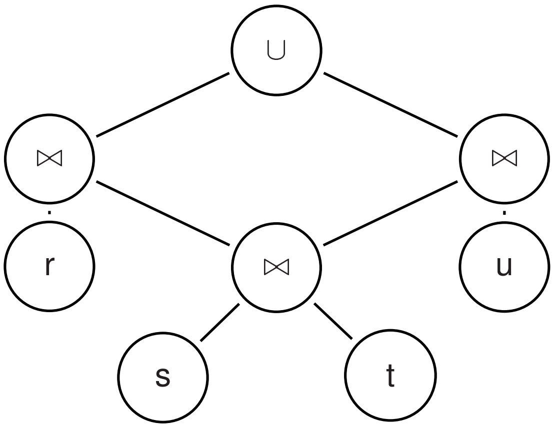





### Reusing intermediate results



## Optimization by Tree Restructuring

- Equivalent transformation of the operator tree that represents an expression
- Based on the equivalences shown on Slide 101.
- minimize the size of intermediate results
  (reject tuples/columns as early as possible during the computation)
- selections reduce the number of tuples
- projections reduce the size of tuples
- apply both as early as possible (i.e., before joins)
- different application order of joins



### Push Selections Down

Assume $r, s \in \text{Rel}(\bar{X})$, $\bar{Y} \subseteq \bar{X}$.

$$\sigma[cond](\pi[\bar{Y}](r)) \equiv \pi[\bar{Y}](\sigma[cond](r))$$

(condition: $cond$ does not use attributes from $\bar{X} - \bar{Y}$, otherwise left term is undefined)

$$\sigma_{pop>1E6}(\pi[name, pop](country)) \equiv \pi[name, pop](\sigma_{pop>1E6}(country))$$

$$\sigma[cond](r \cup s) \equiv \sigma[cond](r) \cup \sigma[cond](s)$$

$$\sigma_{pop>1E6}(\pi[name, pop](country) \cup \pi[name, pop](city))$$
$$\equiv \sigma_{pop>1E6}(\pi[name, pop](country)) \cup \sigma_{pop>1E6}(\pi[name, pop](city))$$

$$\sigma[cond](\rho[N](r)) \equiv \rho[N](\sigma[cond'](r))$$

(where $cond'$ is obtained from $cond$ by renaming according to $N$)

$$\sigma[cond](r \cap s) \equiv \sigma[cond](r) \cap \sigma[cond](s)$$

$$\sigma[cond](r - s) \equiv \sigma[cond](r) - \sigma[cond](s)$$

$\pi$ : see comment above. Optimization uses only left-to-right.

## Push Selections Down (Cont'd)

Assume $r \in \mathrm{Rel}(\bar{X})$, $s \in \mathrm{Rel}(\bar{Y})$. Consider $\sigma[cond](r \bowtie s)$.

Let $cond = cond_{\bar{X}} \wedge cond_{\bar{Y}} \wedge cond_{\overline{XY}}$ such that

- $cond_{\bar{X}}$ is concerned only with attributes in $\bar{X}$
- $cond_{\bar{Y}}$ is concerned only with attributes in $\bar{Y}$
- $cond_{\overline{XY}}$ is concerned both with attributes in $\bar{X}$ and in $\bar{Y}$.

Then,

$$\sigma[cond](r \bowtie s) \equiv \sigma[cond_{\overline{XY}}](\sigma[cond_{\bar{X}}](r) \bowtie \sigma[cond_{\bar{Y}}](s))$$

**Example 4.5**

*Names of all countries that have been founded earlier than 1970, their capital has more than 1.000.000 inhabitants, and more than half of the inhabitants live in the capital.* □



**Example 4.5 (Continued)**

*(Solution)*

$$\begin{aligned}
&\pi[Name](\sigma[establ < \text{“01 01 1970”} \wedge city.pop > 1.000.000 \wedge country.pop < 2 \cdot city.pop] \\
&\qquad (country \bowtie_{country.(capital,prov,code)=city(name,prov,country)} city) \\
&\equiv \pi[Name](\sigma[country.pop < 2 \cdot city.pop] \\
&\qquad\qquad (\sigma[establ < \text{“01 01 1970”}](country) \\
&\qquad\qquad\qquad \bowtie_{country.(capital,prov,code)=city(name,prov,country)} \\
&\qquad\qquad\qquad\qquad \sigma[city.pop > 1.000.000](city)))
\end{aligned}$$

□

- Nevertheless, if $cond$ is e.g. a complex mathematical calculation, it can be cheaper first to reduce the number of tuples by $\cap$, $-$, or $\bowtie$

$\Rightarrow$ data-dependent strategies (see later)

## Push Projections Down

Assume $r, s \in \text{Rel}(\bar{X})$, $\bar{Y} \subseteq \bar{X}$.

Let $cond = cond_{\bar{X}} \wedge cond_{\bar{Y}}$ such that

- $cond_{\bar{Y}}$ is concerned only with attributes in $\bar{Y}$
- $cond_{\bar{X}}$ is the remaining part of $cond$ that is also concerned with attributes $\bar{X} \setminus Y$.

$$\begin{array}{lcl}
\pi[\bar{Y}](\sigma[cond](r)) & \equiv & \sigma[cond_{\bar{Y}}](\pi[\bar{Y}](\sigma[cond_{\bar{X}}](r))) \\
\pi[\bar{Y}](\rho[N](r)) & \equiv & \rho[N](\pi[\bar{Y}'](r)) \\
& & \text{(where } \bar{Y}' \text{ is obtained from } \bar{Y} \text{ by renaming according to } N\text{)} \\
\pi[\bar{Y}](r \cup s) & \equiv & \pi[\bar{Y}](r) \cup \pi[\bar{Y}](s)
\end{array}$$

- Note that this does *not* hold for "$\cap$" and "$-$"!
- advantages of pushing "$\sigma$" vs. "$\pi$" are data-dependent
  Default: push $\sigma$ lower.

Assume $r \in \text{Rel}(\bar{X})$, $s \in \text{Rel}(\bar{Y})$.

$$\pi[\bar{Z}](r \bowtie s) \equiv \pi[Z](\pi[\bar{X} \cap \overline{ZY}](r) \bowtie \pi[\bar{Y} \cap \overline{ZX}](s))$$

- complex interactions between reusing subexpressions and pushing selection/projection



## Application Order of Joins

Minimize intermediate results:

```
SELECT organization.name, country.code
FROM organization, country, is_member
WHERE organization.abbrev = is_member.organization
  AND country.code = is_member.country
```

Exploit selectivity of join:

- $\underbrace{\underbrace{(\text{org} \times \text{country})}_{200 \cdot 200 = 40000} \bowtie \text{is\_member}}_{7000}$
- $\underbrace{\underbrace{(\text{org} \bowtie \text{is\_member})}_{200,7000 \leadsto 7000} \bowtie \text{country}}_{7000}$

If indexes on country.code and organization.abbrev are available:

- loop over is_member
- extend each tuple with matching country and organization by using the indexes.

Example/Exercise

Consider the equivalent (to the previous example) query:

```
SELECT organization.name, country.code
FROM organization, country,
WHERE EXISTS
  (SELECT *
   FROM is_member
   WHERE organization.abbrev = is_member.organization
     AND country.code = is_member.country)
```

- suggests the non-optimal evaluation!
- transform the above query into algebra
- ... yields the same "broad" join as before ...
- ... and leads to the optimized join ordering.



## Operator Evaluation by Pipelining

- above, each algebra operator has been considered separately
- if a query consists of several operators, the materialization of intermediate results should be avoided
- **Pipelining** denotes the immediate propagation of tuples to subsequent operators

**Example 4.6**

- $\sigma[A = 5 \wedge B > 6]R$:

  *Assume an index that supports the condition $A = 5$.*
  - *without pipelining: compute $\sigma[A = 5]R$ using the index, obtain $R'$. Then, compute $\sigma[B > 6]R'$.*
  - *with pipelining: compute $\sigma[A = 5]R$ using the index, and check **on-the fly** each qualifying tuple against $\sigma[B > 6]$.*

□

- **Unary** (i.e., selection and projection) operations can always be pipelined with the next lower binary operation (e.g., join)

**Example 4.6 (Continued)**

- $\sigma[cond](R \bowtie S)$:
  - *without pipelining: compute* $R \bowtie S$*, obtain* $RS$*, then compute* $\sigma[cond](RS)$*.*
  - *with pipelining: during computing* $(R \bowtie S)$*, each tuple is immediately checked whether it satisfies* $cond$*.*
- $(R \bowtie S) \bowtie T$:
  - *without pipelining: compute* $R \bowtie S$*, obtain* $RS$*, then compute* $RS \bowtie T$*.*
  - *with pipelining: during computing* $(R \bowtie S)$*, each tuple is immediately propagated to one of the described join algorithms for computing* $RS \bowtie T$*.* □

Most database systems use iterator-based implementation of algebra operators with pipelining.

- no materialization of intermediate results
- no indexes on them, no sorting, no hashing
- index-based joins only if a base relation is joined
  [see lecture on Database Theory]



## 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

## 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



# 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 4.7**
*Compute all pairs of european countries that are adjacent to the same set of seas.* □

**Example 4.8**
*Compute all political organizations that have at least one member country on every continent (this operation is called* relational division*).* □

### 4.2.3 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.
     $1000 \cdot (1$ (load) $+ 100$ (tuples per page) $\cdot 500$ (pages of $S$)$) = 50,001,000$
   - outer loop: $S$
     Load each page of $S$. For all tuples, iterate over $R$'s pages.
     $500 \cdot (1$ (load) $+ 80$ (tuples per page) $\cdot 1000$ (pages of $R$)$) = 40,000,500$

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 $S$.
     $1000 \cdot (1 + 500$ (pages of $S$)$) = 501,000$
   - outer loop: $S$. $500 \cdot (1 + 1000$ (pages of $R$)$) = 505,000$



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 $S_j$): Assumptions:
   - every $S_j$ occurs – average – $k = 4$ times.
   - index can be kept in main memory

   Iterate over $R$, for each tuple search for the $S_j$ key and access the tuples
   $1000 \cdot (1$ (load) $+ 100 \cdot 4$ (access tuples)$) = 401,000$.
   In case that $S$ is ordered wrt. $S_j$: $1000 \cdot (1$ (load) $+ 100 \cdot 1$ (access tuples)$) = 101,000$.

d) sort-merge: sort $R$ according to $R.i$:
   sort each 100 pages in place (Quicksort), store them:
   $10$ (times) $\cdot 100$ (pages) $\cdot 2$ (read and write) $= 2000$
   mergesort the sorted pages (linear) – read and write: $2000$
   sort $S$ according to $S_j$ analogously: $2000$
   merge-join of sorted relations: linear scan: $1000 + 500 = 1500$
   Overall: $7500$

## 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:

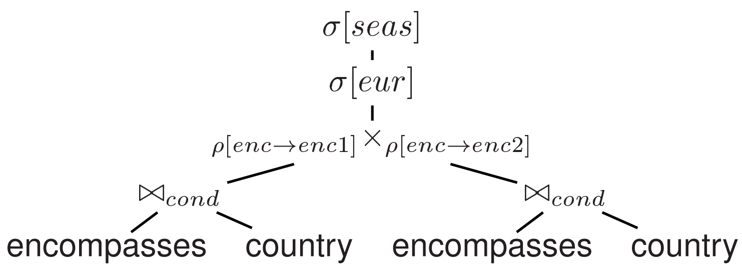
- Hash $R$:
  100 pages fit in memory - read 34 pages and distribute over 60 partitions. Do this 30 times.
  (about $30 \cdot (34$ (read) $+ 60$ (write) $) < 3000$ accesses).
  In the average, each partition them contains about 17 pages.
- Same for $S$. Get 60 partitions of average 9 pages. (about 1500 accesses).
- now we have 60 corresponding pairs of partitions (one from $R$, one from $S$).
  Join each pair: read $17 + 9$ (overestimate) pages per partition and process them.
  (about $60 \cdot (17 + 9) = 1560$ accesses).

Overall: 6060 accesses.



## Solution to Exercise 4.7

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




- $cond =$ "enc.country=country.code".
- $eur =$ "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.

## Solution to Exercise 4.7 (cont'd)

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

$$\begin{aligned}\sigma[seas(C_1, C_2)] &= seas(C_1) - seas(C_2) = \emptyset \wedge seas(C_2) - seas(C_1) = \emptyset \\ &= (seas(C_1) - seas(C_2)) \cup (seas(C_2) - seas(C_1)) = \emptyset\end{aligned}$$

- $seas(C) = \pi[sea](\sigma[country = C](geo\_sea))$
- 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 σ[seas(C1, C2)]
```



## Solution to Exercise 4.8

- $\pi[abbrev](\sigma[all\_continents(org)])(organization)$ 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

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

  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))
```