Credits

Based on

- German lecture by Gunter Saake (translated by Sandro Schulze)
- According german book:
Overview

1. Functionality and principles of database systems
2. Architecture of database systems
3. Management of storage device(s)
4. File organisation and data structures
5. Data structures for specific applications
6. Basic algorithms for database operations
7. Query optimization
8. Further aspects and outlook
Required Knowledge

Databases I:
- Basic principles of database systems
- Tables, attributes, keys
- Relational algebra and SQL

*Short repetition at the beginning of the lecture!*
Literature

- H"arder, T.; Rahm, E.: *Datenbanksysteme — Konzepte und Techniken der Implementierung*. Springer-Verlag, 2001 (in German)
1. Requirements and Principles of DBS

- Repetition of basic terms (of databases)
- Overview over discussed components
Basic Terms: Components

External Level
- Queries
- Updates

Conceptual Level
- Optimizer
- Computation

Internal Level
- Disc Access

Data Dictionary

DB Operations
API
Forms

View Definition
Data Organisation
Data Definition
Classification of Components

User Components

Programming Components

Transformation Components

Data Dictionary

Definition Components
Nine Rules of Codd

1. Integration
2. Operations
3. Catalog (Data Dictionary)
4. (Multi)User Views
5. Consistency
6. (Data) Privacy
7. Transactions
8. Synchronization
9. Data Backup
Database Models and Data Definition

Important models in commercial systems

- **Hierarchical model**: Data exist in a tree-like shape with hierarchical structured data sets,

- **Network model**: Support of networks of data sets, referenced by pointers,

- **Relational data model**: Data exist in tabular form,

- **Object(-oriented) model**: object-oriented modelling of data by objects, organized in classes and referenced by pointers,

- **Semi-structured model**: Management of self-describing data without schemata in graph structures (XML).
## Relational Databases

<table>
<thead>
<tr>
<th>Borrowing</th>
<th>InventoryNo</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4711</td>
<td>Brown</td>
</tr>
<tr>
<td></td>
<td>1201</td>
<td>Smith</td>
</tr>
<tr>
<td></td>
<td>0007</td>
<td>Miller</td>
</tr>
<tr>
<td></td>
<td>4712</td>
<td>Brown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Book</th>
<th>InventoryNo</th>
<th>Title</th>
<th>ISBN</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0007</td>
<td>Dr. No</td>
<td>3-125</td>
<td>James Bond</td>
</tr>
<tr>
<td></td>
<td>1201</td>
<td>Object Databases</td>
<td>3-111</td>
<td>Heuer</td>
</tr>
<tr>
<td></td>
<td>4711</td>
<td>Databases</td>
<td>3-765</td>
<td>Vossen</td>
</tr>
<tr>
<td></td>
<td>4712</td>
<td>Databases</td>
<td>3-891</td>
<td>Ullman</td>
</tr>
<tr>
<td></td>
<td>4717</td>
<td>Pascal</td>
<td>3-999</td>
<td>Wirth</td>
</tr>
</tbody>
</table>
create table Book ( 
    ISBN   char(10),
    Title   varchar(200),
    PublisherName   varchar(30),
    primary key (ISBN),
    foreign key (PublisherName)
    references Publishers (PublisherName)
)
Queries

Basic principles
- Relational algebra as well as
- Tuple or range calculus.
Relational Algebra

\[ \sigma_{\text{Name}=\text{`Brown'}}(r(\text{Borrowing})) \]

\[ \pi_{\text{Title}}(r(\text{Book})) \]

\[ \pi_{\text{InventoryNo},\text{Title}}(r(\text{Book})) \bowtie \sigma_{\text{Name}=\text{`Brown'}}(r(\text{Borrowing})) \]
Alteration Component

This component of a database system enables,
- to insert tuples,
- to delete tuples and
- to update/change tuples.
Languages and Views: SQL

```sql
select  Book.InventoryNo, Title, Name
from    Book, Borrowing
where   Name = 'Brown' and
        Book.InventoryNo = Borrowing.InventoryNo

update Employees
set     Salary = Salary + 1000
where   Salary < 5000

insert into Book values
(4867,'Wissensbanken', '3-876','Karajan')

insert into Customer
( select LName, LAdr, 0 from Provider )
```
Views in SQL

```sql
create view Browns as
select Book.InventoryNo, Title, Name
from Book, Borrowing
where Name = 'Brown' and
    Book.InventoryNo = Borrowing.InventoryNo
```
Overview of Components discussed in this lecture

- **Optimizer**
- **Data file organisation and access paths**
- **Organisation of the secondary storage**

Not in this lecture:
- **Transaction management (concurrency control)**
- **Recovery component**
Optimizer

- Equivalence of algebra terms
  1. $\sigma_{A=\text{const}} \left( \text{REL1} \bowtie \text{REL2} \right)$ und A aus REL1
  2. $\sigma_{A=\text{const}} \left( \text{REL1} \right) \bowtie \text{REL2}$

- common strategy: Selections as early as possible $\Rightarrow$ decreases the amount of tuples in relations

- Example: REL1 100 tuples, REL2 50 tuples
  - internal: tuples are stored sequential
    1. $5000 \left( \bowtie \right) + 5000 \left( \sigma \right) = 10000$ operations
    2. $100 \left( \sigma \right) + 10 \cdot 50 \left( \bowtie \right) = 600$ operations
      if 10 tuples in REL1 fulfill the condition $A = \text{const}$
Joins

- **Merge-Join:** *Join by merging* of $R_1$ and $R_2$
  - particularly efficient, if one (or both) relation(s) are ordered by the join attributes, i.e., for join attributes $X$ has to apply: $X := R_1 \cap R_2$
  - $r_1$ and $r_2$ will be ordered by $X$
  - Merging of $r_1$ and $r_2$, i.e., both relations are traversed sequentially and matched pairs are added to the result

- **Nested-Loops-Join:** twofold loop over $R_1$ and $R_2$
  - availability of access path for $X$ (for one relation) $\Rightarrow$ replacing inner loop by access via access path
Complexity of Operations

- **Selection**
  - Hash-based data structures: $O(1)$
  - Sequential traversal: $O(n)$
  - Generally (tree-based access paths): $O(\log n)$

- **Join**
  - Ordered tables exist: $O(n + m)$ (Merge Join)
  - Else: up to $O(n \times m)$ (Nested Loop Join)

- **Projection**
  - Existing access path or projection on key: $O(n)$
  - Duplicate elimination (by sorting): $O(n \log n)$
Types of Optimization

- **Logical optimization:**
  - Uses only algebraic attributes of operations
  - *NO* information about storage structures and access paths
  - Usage of heuristic rules instead of exact optimization
  - Examples:
    - Removal of redundant operations
    - Move/Shift of operations, so that Selections are executed as early as possible

\[\Downarrow\] **Algebraic optimization**
Types of Optimization II

- **Internal optimization:**
  - Usage of information about existing storage structures
  - Choice of implementation strategies of single operations (Merge Join vs. Nested-Loops-Join)
  - Examples:
    - Order of Joins by means of size and support of the relations by access paths
    - Order of Selections by means of selectivity of attributes and the existence of access paths
Algebraic Optimization

- *Removal of redundant operations* \((r \Join r = r)\)

\[
\begin{align*}
r(\text{BookAway}) &= \\
&= r(\text{Book}) \Join \pi_{\text{ISBN},\text{Date}}( \\
&\quad \cdots \sigma_{\text{Date} < '31-12-1990'}(r(\text{Borrowing})))
\end{align*}
\]

- Query to View:

\[
\pi_{\text{Title}}(r(\text{Book}) \Join r(\text{BookAway}))
\]

- Insertion of view definition:

\[
\pi_{\text{Title}}(r(\text{Book}) \Join r(\text{Book}) \Join \pi_{\cdots}(\ldots))
\]
Algebraic Optimization II

- **Shift of Selections**

\[ \sigma_{\text{Author}=\text{'Vossen'}} (r(\text{Book}) \bowtie \pi_{\text{ISBN,Date}}(\ldots)) \]

Join on downsized intermediate results:

\[ (\sigma_{\text{Author}=\text{'Vossen'}} (r(\text{Book}))) \bowtie \pi_{\text{ISBN,Date}}(\ldots) \]

Commutate Selection and Join
Algebraic Optimization III

- Order of Joins

\[(r(Publisher) \Join r(Borrowing)) \Join r(Book)\]

Drawback: first Join degenerates into cartesian product, since there are no shared attributes

\[r(Publisher) \Join (r(Borrowing) \Join r(Book))\]

\[\Join \text{ associative and commutative}\]
# Data Organisation and Access Paths

<table>
<thead>
<tr>
<th>Conceptual Level</th>
<th>Internal Level</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations</td>
<td>Files</td>
<td></td>
</tr>
<tr>
<td>Tuples</td>
<td>Records</td>
<td>Blocks</td>
</tr>
<tr>
<td>(Attribute) Values</td>
<td>Fields</td>
<td></td>
</tr>
</tbody>
</table>
Access Paths

- Primary versus Secondary index
- Sequential files, B-Trees, hashing
- One-dimensional versus multidimensional
- Specific applications
2. Architecture of Database Systems

- Problems covered in this chapter
- Layered architecture of a relational DBMS
- Hardware and operating system
- Buffer management
- Storage system
- (Internal) access system
- Data system
Problems covered (in this chapter)
Five-Layer Architecture (for DBMS)

- Based on the idea of Senko (1973)
- Further developments of Harder (1987)
- Realized in the development of the IBM prototype System R
- Detailed description of transformation components
  - Step-wise transformation of queries/changes up to the point of access on storage devices
  - Definition of interfaces between components
Five-Layer Architecture: Interfaces I

- Set-oriented interface \((SOI)\)
  - declarative DML on tables, views and tuples

- Record-oriented interface \((ROI)\)
  - Records, logical files, logical access paths \((Indices)\)
  - navigating access on internal representation of relations

- Internal record interface \((IRI)\)
  - Records, access paths
  - Manipulation of records and access paths
5-Layered-Architecture: Interfaces II

- System buffer interface (*SBI*)
  - Pages, page addresses
  - Release and allocation
- File interface (*DS*)
  - Fetch page, write page
- Device interface (*DI*)
  - Tracks, cylinder
  - Movement of disk heads
5-Layered-Architecture: Functionality

Set-Oriented Interface (SOI)

Record-Oriented Interface (ROI)

Internal Record Interface (IRI)

System Buffer Interface (SBI)

File Interface (FI)

Device Interface (DI)

Data System

Access System

Storage System

Buffer Management

Operating System

Translation, Access Path Selection, Access Control, Integrity Control

Data Dictionary, Currency Pointer, Sorting, Concurrency Control

Record Manager, Access Path Management, Lock Management, Log/Recovery

System Buffer Management with Page Replace Strategy

External Storage Management
5-Layered-Architecture: Objects

Set-Oriented Interface (SOI) ——— Data System ——— Record-Oriented Interface (ROI)

Internal Record Interface (IRI) ——— Access System ——— Storage System

System Buffer Interface (SBI) ——— Buffer Management ——— Operating System

File Interface (FI) ———

Device Interface (DI) ———

Relations
Views
External records, Scans, Index structures
Internal records, Trees, Hash tables
Segments
Pages
Files
Blocks
Cylinder
Trace

SQL : select ... from ...
QBE, QUEL, ...
FIND NEXT record
STORE record
Store internal record s
INSERT in B–Tree
Provide Page j
Unlock Page j
Read block k
Write block k
Driver
Explanations (I)

- Set-oriented interface **SOI**:  
  - declarative data manipulation language (DML) on tables and views (e.g., SQL)
- is converted to the record-oriented interface **ROI** via the file system:
  - navigating access on internal representation of relations
  - manipulated objects: typed data sets and internal relations as well as logical access paths (Indices)
  - Functions of data system: Translation and optimization of SQL queries
Explanations (II)

- is converted to the internal record interface **IRI** via the access system:
  - Uniform management of internal tuples (no typing)
  - Storage structures of access paths (concrete operations on B*-"Trees und hash tables) implemented
  - Multi user mode with transactions is realized
Explanations (III)

- Data structures and operations (of IRI) are converted via storage system to internal pages of a virtual, linear address space
  - Manipulation of address space by operations of the system buffer interface SBI
  - Typical objects: internal pages, page addresses
  - Typical operations: Release and allocation of pages, page replace strategy, lock management, maintenance of log book (e.g., writing)

- Mapping of internal pages to blocks of file interface FI via buffer management
  - Realization of FI operations on device interface carried out by OS
Hardware and Operating System

- OS layer: foundation for database related layer
  - Required: Driver (programs) for data access of storage media; caching mechanisms
- Processors and computer architecture: industrial standards (though: database computer available)
- Storage media: specific requirements, result in storage hierarchy
Classical Storage hierarchy

Primary, secondary, and tertiary storage

- Cache
- Main Memory
- Disk Memory
- Optical Disk
- Magnetic Tapes

Primary Storage
Secondary Storage
Tertiary Storage
## Characteristics of Storage Media

<table>
<thead>
<tr>
<th></th>
<th>Primary</th>
<th>Secondary</th>
<th>Tertiary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>fast</td>
<td>slow</td>
<td>very slow</td>
</tr>
<tr>
<td>Price</td>
<td>expensive</td>
<td>low cost</td>
<td>cheap</td>
</tr>
<tr>
<td>Stability</td>
<td>volatile</td>
<td>non-volatile</td>
<td>non-volatile</td>
</tr>
<tr>
<td>Size</td>
<td>small</td>
<td>big</td>
<td>very big</td>
</tr>
<tr>
<td>Granularity</td>
<td>fine</td>
<td>coarse</td>
<td>coarse</td>
</tr>
</tbody>
</table>
Primary Storage

- Cache and main memory
- very fast, fine-grained access on data: byte-by-byte addressable
- 32-Bit addressing: only $2^{32}$ bytes directly addressable $\rightarrow$ size of primary storage very restricted
- High purchase costs per byte
- volatile and non-reliable storage medium
Secondary Storage

- Online storage
  - Usually disk memory, non-volatile and reliable
  - much bigger, several gigabyte memory space per medium
  - Purchase costs are considerable less
  - Data not directly processable
  - Granularity for access more coarse-grained: Blocks, usually 512 bytes (or a multiple thereof)
  - Access vacancy: Factor $10^5$ more slowly access

- Required: intelligent buffer management, good query optimization

**However:** *minimized by new technologies, e.g., Flash*
Tertiary Storage

- For long-term data backup (archiving) or short-term logging (journal) of database and database changes
- Secondary storage not useful (too small/expensive)
- Several hundreds of gigabyte or even terabyte of data: aka offline storage, archive storage
- Usual: optical disks, magnetic tapes
- "Offline storage" mostly removable medium
- Drawbacks: Access vacancy extremely high: Access on sequential medium, get magtape, insert magtape (even with automatization: magtape robots, Jukeboxes)
Provided Services

- Driver (programs) → read(fetch)/write of blocks
- Assignment of blocks to pages
- Enhancement of block information with check sum → detection of write/read error
- Caching mechanisms, to keep and manage already read data in main memory
- Operations of file system of OS (usual: Only one file used by database systems)
Buffer Management

- Management of required blocks (of secondary storage) in main memory
- Memory space only for limited amount of pages in main memory: \textit{Buffer}
- Function of buffer management: Replacement of no longer required pages in buffer (page replace strategies)
- Difference: buffer managed by database system $\leftrightarrow$ Cache on operating system level
Buffer

- Size depends on main memory (usually up to 12 GB)
- Nevertheless only a minor fraction of database (less than 1%)
- All read/write operations on pages in the buffer
- Hence, buffer as a bottleneck
- Main memory (available) huge, database quite small ⇒ whole database (e.g., during system start) in the buffer: *main memory databases*
Buffer Management: Functions

- Allocation of memory space for pages
- Looking for/replacement of pages in the buffer
- Optimization of (work)load sharing between parallel transactions
Page Access (Procedure) (I)

Page request of higher layer (storage system) (*logical page reference*)

- Requested page in the buffer: provided to the storage system
- Requested page not in the buffer (*page fault*): *physical page reference* from buffer management to operating system layer; in general page replace in case of full buffer; if the page to be going to replaced has been changed → write to secondary storage

Effort per I/O operations: 2500 instructions in CPU; 15 to 30 ms for access on secondary storage
Page Access (Procedure) (II)

Storage System

logical page reference

Main Memory

replace

Buffer Management

physical page reference

load

Operating System

Secondary Storage

page reference
logical
physical page reference
replace
Secondary Storage
load
Main Memory
System
Operating
Management
Buffer
Storage System
## Storage System

Buffer: Pages (byte container) ↔ Storage system: internal data sets/records ↔ Access system: logical data records, internal tuples

<table>
<thead>
<tr>
<th>(Data) Structure</th>
<th>System Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple</td>
<td>Data system</td>
</tr>
<tr>
<td>Internal tuple or logical data record</td>
<td>Access system</td>
</tr>
<tr>
<td>internal data record</td>
<td>Storage system</td>
</tr>
</tbody>
</table>

- Objects in storage system (represented) as internal records
- Auxiliary data, e.g., index entries, are represented as internal records as well
Adressing of Records

- Problem: Update/Change of database ⇒ Efficient update of addresses
- Example: Internal records addressed with offset $x$, relatively to begin of page (internal record starts at byte $x$) ⇒ changes on this page have impact on used tuple address
- Improvement: TID concept (tuple identifier)
  - Address: Page number and offset in a list of tuple pointer at the begin/head of the page
  - Entry in pointer field ⇒ offset of record
  - Position of internal record within the page changed ⇒ only local entry in pointer field has to be changed; all addresses, used "‘outside’", remain stable
Types of Records

- **Unspanned records** at most on one page; record too large for currently processed page ⇒ request the storage management for new page

- **Spanned records** can span several pages; record too large ⇒ Beginning of record on current page, overflow stored on new page

- **Records with fixed length**: for a certain type of tuples a definite number of bytes (for *string* all attribute values are stored with same number of bytes)

- **Records with variable length**: only the number of bytes really needed is stored (number of bytes per record is variable)
Access System

- Abstraction from internal representation of records to pages
- *Logical data records, internal tuples*
- Internal tuples can be
  - Elements of a data representation of the conceptual relation *or*
  - Elements of an access path on the conceptual relation
- Internal tuples consist of fields (corresponding to the attributes of conceptual tuples)
- Typical operations of the access system: *Scans* (internal cursor on files or access paths)
Index Files

- Access path on a file is a file as well: *Index file*
- Index consists of attribute values of conceptual relation (realizing a fast access on relation) as well as a list of tuple addresses
- Assigned addresses point to tuples, which contain the indexed attribute value
- Index on primary key \(\Rightarrow\) List of tuple addresses with singleton entries: *Primary index*
- Index arbitrary set of attributes: *Secondary key* (although attribute values must **not** possess any key characteristics)
- Index on secondary key: *Secondary index*
Data Operations

- Insertion of a data record \textit{(insert)}
- Removal of a data \textit{(remove oder delete)}
- Modification of a data record \textit{(modify)}
- Locking for and find a record \textit{(lookup or fetch)}
Data Operations: Types of lookup

- Given the attribute value for a certain field (column) ⇒ sought-after internal tuples possessing this value: *single-match query*.
- Given a combination of values for a certain combination of fields ⇒ sought-after all tuples, possessing these values:
  - Values for all values of (index) file: *exact-match query* (*single match query* is a special case)
  - Values only for some fields of (index) file: *partial-match query*
- Given a range of values for one or more attributes ⇒ sought-after all internal tuples, possessing attribute values within this range: *range query*
Access on Data Records

- Data Records in a file depending on value of primary key
  - ordered or
  - hashed (scattered)

stored ⇒ fast access over primary key

- Fast access over other attributes (secondary key) realized over index files by default
File Organisation and Access Paths (I)

- **Primary key (PK) / Secondary key (SK):**
  - Access on PK (only one (distinct) tuple address per attribute value)
  - Access on SK (several tuple addresses per attribute value possible)
  - Usual: Primary key determines file organisation, secondary key determines access paths (index files)

- **one-dimensional / multidimensional:**
  - Support of access for a fixed combination of fields (*exact-match*)
  - Support of access for a variable combination of fields (*partial-match*)
File Organisation and Access Paths II

- **static / dynamic:**
  - Static file organisation or access path \(\Rightarrow\) only useful for a fixed number of data records (to be managed)
  - Dynamic file organisation or access path \(\Rightarrow\) independent of number of data records (automatic, efficient adaptation to growing/shrinking data sets)
Examples

- **B-Tree**
  - Dynamic, one-dimensional access paths
  - in most database systems definable over several attributes of a file
  - However: only one *exact-match* possible (on this combination of fields)

- **Hashing (classical hash method)**
  - Static and one-dimensional kind of file organisation
  - Increasing amount of tuples $\Rightarrow$ increasing number of collisions expected
Data System (I)

- **Optimization**: Set-oriented query (SQL) has to be optimized by the system
  - Transformation of query expression into more efficient, processable expression (*Query Rewriting*, *conceptual* or *logical optimization*)
  - Selection of most efficient query expression by cost estimation (*cost-based optimization*)
  - *Internal optimization*: Selection of useful access paths and computation algorithms for query processing for every relational-algebraic operator
Data System (II)

- *Selection of access paths*: determines internal structures to be exploited for query processing
- *Computation*: Selection of algorithm influences response time of a query
- Data system has to select the computation algorithms in cooperation with selection of access paths
3. Storage Management

- Storage media
- Storage arrays: RAID
- Backup media: Tertiary storage
- Structure of (physical) storage
- Pages, records and addressing
- Buffer management in detail
- Cryptographic methods
Storage Media

Several purposes:
- Provide data for processing
- Long-term data storage (and at the same time having them available anyway)
- Archiving data in the long term and cheap by accepting longer access time

In this section:
- Storage hierarchy
- Hard-disk/magnetic disk
- Capacity, costs, speed
Storage Hierarchy

1. Extreme fast processor with register
2. Very fast cache memory
3. Fast main memory
4. Slow secondary storage with randomized access
5. Very slow nearline tertiary storage with automatically provided storage media
6. Extreme slow offline tertiary storage with manually provided storage media

Tertiary storage: CD-R (Compact Disk Recordable), CD-RW (Compact Disk ReWritable), DVD (Digital Versatile Disks), magnetic tapes, e.g., DLT (Digital Linear Tape)
Cache Hierarchy (I)

Characteristics of storage hierarchy

- Level \( x \) (e.g., level 3, main memory) has a considerably faster access time than level \( x + 1 \) (e.g., level 4, secondary storage)
- At the same time, much higher costs per memory/storage space
- Thus, much lower capacity
- Durability of data increase corresponding to the levels
Cache Hierarchy (II)

- Reducing *access vacancy* (differences between access times on data) \(\Rightarrow\) cache storage buffers the data of level \(x + 1\) on level \(x\):
  - *Cache* (equivalent to main memory cache) \(\Rightarrow\) faster semiconductor technology for the supply of data to the processor (level 2 in storage hierarchy)
  - *Disk storage cache* in main memory: *Buffer*
  - Cache in the context of WWW (accessing data over HTTP): Part of disk storage, which buffers parts of data, provided by the internet
Access Vacancy

- Magnetic disk 70% more storage density per year
- Magnetic disk only 7% faster per year
- CPU power increased by 70% per year
- Access vacancy between main memory and magnetic disk is $10^5$

Measurements:
- $ns$ for nanoseconds (i.e., $10^{-9}$ seconds), $ms$ for milliseconds (i.e., $10^{-3}$ seconds)
- KB (KiloByte = $10^3$ bytes), MB (MegaByte = $10^6$ bytes), GB (GigaByte = $10^9$ bytes) und TB (TeraByte = $10^{12}$ bytes)
## Numerical Access Vacancy (2005)

<table>
<thead>
<tr>
<th>Type of storage</th>
<th>typical access time</th>
<th>typical capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>6 ns</td>
<td>512 KB up to 32 MB</td>
</tr>
<tr>
<td>Main memory</td>
<td>60 ns</td>
<td>32 MB up to 1 GB</td>
</tr>
<tr>
<td>— Access Vacancy $10^5$ —</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>12 ms</td>
<td>1 GB up to 10 GB</td>
</tr>
<tr>
<td>Disk farm or array</td>
<td>12 ms</td>
<td>in the range of TB</td>
</tr>
</tbody>
</table>
Access Locality

- Caching principle does not work, if always new data (from higher levels) is needed
- In most use cases: *Locality* of (data) access
- This means, mostly the access takes place on data provided by the cache of the respective level (usually more than 90%)
- Hence: Buffer management of database system is an important concept
Hard-Disk/Magnetic Disk
Recording Component

- Disk pack with 5 to 10 disks
- Up to 10,000 revolutions per minute (rpm)
- For every disk surface a read/write head (i.e. between 10 and 20)
- Disk surface: concentrical circles (Tracks)
- Stacked tracks of all disk surfaces: Cylinder
- Tracks consist of Sectors (512 bytes, 1 KB)
- Sector contains data for autocorrection of errors — parity bits or Error Correcting Codes (ECC)
- Smallest access unit on disk: Block (multiple of sector size)
Positioning Component

- Addressing of blocks realized by cylinder/track/sector number
- Access time (from request to the transfer of block content):
  - *Seek time* (time for moving access arm)
  - *Latency time* (time for rotating to the respective sector)
  - *Data-transfer time* (time for data transfer)
### Typical Characteristics of Hard-Disks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average seek time</td>
<td>8 ms</td>
<td>16 ms</td>
<td>30 ms</td>
</tr>
<tr>
<td>Revolution time</td>
<td>6 ms</td>
<td>16.7 ms</td>
<td>16.7 ms</td>
</tr>
<tr>
<td>Track capacity</td>
<td>100 KB</td>
<td>47 KB</td>
<td>13 KB</td>
</tr>
<tr>
<td>Disk surface(s)</td>
<td>20</td>
<td>15</td>
<td>19</td>
</tr>
<tr>
<td>Cylinder</td>
<td>5000</td>
<td>2655</td>
<td>411</td>
</tr>
<tr>
<td>Capacity</td>
<td>10 GB</td>
<td>1.89 GB</td>
<td>93.7 MB</td>
</tr>
</tbody>
</table>
Controller and further Developments

(Disk) Controller

- Transfer rate: 40 MB per second
- IDE- or SCSI-Controller

Further developments

- *Disk farms* ⇒ loose coupling of few but big disks; distribution of data on disk realized by operating system or database system
- *Disk arrays*, esp. RAID systems (*Redundant Array of Inexpensive Disks*): High amount of cheap standard disks (8 bis 128) organized by an intelligent controller ⇒ Increase of fault tolerance (reliability) or increase of parallelism of access (increase in efficiency)
# Storage Capacity and Costs I

<table>
<thead>
<tr>
<th>Size</th>
<th>Information or Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>= 1.000 (precise: 1024 Byte)</td>
</tr>
<tr>
<td>0.5 KB</td>
<td>Book page as text</td>
</tr>
<tr>
<td>30 KB</td>
<td>scanned, packed book page</td>
</tr>
<tr>
<td>1 MB</td>
<td>= 1.000.000 (...)</td>
</tr>
<tr>
<td>5 MB</td>
<td>The bible as text</td>
</tr>
<tr>
<td>20 MB</td>
<td>scanned book</td>
</tr>
<tr>
<td>500 MB</td>
<td>CD-ROM; Oxford English Dictionary</td>
</tr>
<tr>
<td>1 GB</td>
<td>= 1.000.000.000 (...)</td>
</tr>
<tr>
<td>4.7 GB</td>
<td>Digital Versatile Disk (DVD)</td>
</tr>
<tr>
<td>10 GB</td>
<td>packed movie</td>
</tr>
<tr>
<td>100 GB</td>
<td>one floor of a library</td>
</tr>
<tr>
<td>200 GB</td>
<td>Capacity of a video tape</td>
</tr>
</tbody>
</table>
### Storage Capacity and Costs II

<table>
<thead>
<tr>
<th>Size</th>
<th>Information or Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 TB</td>
<td>= 1.000.000.000.000.000 (…)</td>
</tr>
<tr>
<td>1 TB</td>
<td>Library with 1M books</td>
</tr>
<tr>
<td>1 TB</td>
<td>huge (external) hard-disk</td>
</tr>
<tr>
<td>11 TB</td>
<td>big(gest) Data Warehouse (Wal-Mart)</td>
</tr>
<tr>
<td>20 TB</td>
<td>huge storage/disk array</td>
</tr>
<tr>
<td>20 TB</td>
<td>Library of Congress books stored as text</td>
</tr>
<tr>
<td>1 PB</td>
<td>= 1.000.000.000.000.000.000 (…)</td>
</tr>
<tr>
<td>1 PB</td>
<td>scanned books of a national library</td>
</tr>
<tr>
<td>15 PB</td>
<td>worldwide hard-disk production in 1996</td>
</tr>
<tr>
<td>200 PB</td>
<td>worldwide magnetic tape production in 1996</td>
</tr>
</tbody>
</table>
## Storage Medium and Costs (ca. 1999)

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Storage Medium</th>
<th>Preis (Euro/MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Memory</td>
<td>64-MB-SDRAMs</td>
<td>0.5000 Euro</td>
</tr>
<tr>
<td>Secondary Storage</td>
<td>10-GB-IDE-HD</td>
<td>0.0275 Euro</td>
</tr>
<tr>
<td>Tertiary Storage</td>
<td>opt. disk 5 GB</td>
<td>0.0160 Euro</td>
</tr>
<tr>
<td></td>
<td>650-MB-CD-RW</td>
<td>0.0150 Euro</td>
</tr>
<tr>
<td></td>
<td>650-MB-CD-R</td>
<td>0.0015 Euro</td>
</tr>
<tr>
<td></td>
<td>70-GB-DLT-Band</td>
<td>0.0010 Euro</td>
</tr>
</tbody>
</table>
Current Developments

- Classical storage hierarchy in change

- Flash storage
  - Well-known from digi cams, MP3 player, PDA, etc.
  - Block by block deletion before every writing
    - Sequential reading identical with disk
    - Random reading considerably faster
    - Writing is more slowly
    - Durability is limited to 100,000 until 1,000,000 writing processes
  - 256 GB Chips for 40 Dollar predicted for 2012
    - Even the still more expensive than disk (approx. by factor 10)

- RAM storage \(\Rightarrow\) Even faster, random access
  - Main memory databases
Storage Arrays: RAID

- Connection of cheap standard magnetic tapes to one logical device supervised by a certain controller
- Distribution of data over the several physical disks is realized by the controller
- Two opposed objectives:
  - Increasing the fault tolerance (system stability, reliability) by redundancy
  - Increase in efficiency through parallelism of access
Increasing of Fault Tolerance

- Usage of additional disks for storage of duplicates (mirrors) of the original data ⇒ Failures: switch over to mirror disk
- Particular RAID-Levels (1, 0+1) permit such a mirroring
- Alternative: Control information, e.g., parity bits, are stored on a separate disk instead of the same sector as the original data
- RAID levels 2 until 6 restore defective data using parity bits or Error Correcting Codes (ECC)
- Parity bit can detect and correct a media error (if the defective disk is known)
Increase of Efficiency (I)

- Database distributed over several disks, that can be accessed parallel ⇒ Access time decreases nearly linear to the number of disks available
- Distribution
  - Bit by bit (i.e., in the case of 8 available disks, one byte can be distributed)
  - Byte by byte (bytewise)
  - Block by block (blockwise)
Increase of Efficiency (II)

- Higher RAID levels (from level 3) combine error correction and bitwise blockwise distribution of data

- Differences (to lower levels):
  - *faster access* on certain data
  - *increased throughput* for many transactions (queued for parallel processing) by load balancing the overall system
RAID Levels 0, 1 und 0+1

Blocks

RAID 0

RAID 1

RAID 0 + 1
RAID Level 0

- Data blocks are distributed on physical disks (available for the RAID system) by rotation principle (*Striping*).
- Here: blockwise Striping.
- Advantages:
  - Parallel reading of data of sequenced blocks.
  - Load balancing if many parallel read TAs are queued (needed blocks exist probably on different disks).
- Disadvantages:
  - No speedup for random access on one specific block.
  - No redundancy or control information.
RAID Level 1

- Blocks (to be stored) are mirrored/duplicated on several disks
- Disadvantages:
  - Solely increase of efficiency: load balancing for reading of parallel transactions
- Advantages
  - In case of errors simply switch to one of the mirror disks
  - Subsequently, the defective disk can be replaced and filled with correct data
RAID Level 0+1

- Combination of level 0 and 1 to obtain the advantages of both
- Foundation for higher RAID-Levels: Combination of Striping for increase of efficiency and actions for error correction are retained
- However, doubling the memory space requirements is avoided
- Instead of mirror disks, control information, e.g., parity bits or Error Correcting Codes (ECC) are used
RAID Levels 2 und 3

RAID 2
A[1]  
B[1]  
C[1]  
D[1]  
A[2]  
B[2]  
C[2]  
D[2]  
...  
A ECC[1]  
B ECC[1]  
C ECC[1]  
D ECC[1]  
A ECC[2]  
B ECC[2]  
C ECC[2]  
D ECC[2]

RAID 3
A[1]  
B[1]  
C[1]  
D[1]  
A[2]  
B[2]  
C[2]  
D[2]  
...  
A ECC  
B ECC  
C ECC  
D ECC
RAID Level 2

- Bitwise Striping
- Parity bits or extended ECC on additional disks
- Reading in one data requires 8 parallel read operations
- Access not more efficient, but the overall throughput is octuplicated (i.e., eight times higher)
RAID Level 3

- Principle: Disk detects error $\Rightarrow$ byte can be reconstructed with the help of the 7 remaining bits and one parity bit
- Parity bits use only one single, dedicated disk
- Control information are reduced to the basics, and thus, memory space is reduced further
RAID Levels 4 to 6

RAID 4

RAID 5

RAID 6
RAID Level 4

- Blockwise Striping
- One dedicated disks for parity bits
- Small data sets are readable more efficient, since only one physical disk is affected
RAID Level 5

- No additional, dedicated disk for storage of parity bits
- Instead, parity bits are distributed over existing disks
- Hence, bottleneck of level 3 and 4 is removed: So far, every write operation on parity disk as well $\implies$ every write operation waits for dedicated disk
- Improved load balancing by usage of different disks
RAID Level 6

- Based on RAID level 5
- More control information on data disk
- More than one media error can be corrected
- Only barely implemented
# Overview RAID Levels

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0+1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Striping blockwise</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Striping bitwise</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity dedicated disk</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity distributed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Detecting several errors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Comparison of particular RAID Level (I)

Application profile
- Number of read operations
- Number or write operations
- Requirements regarding system stability
Comparison of particular RAID Level (II)

Characteristics

- Fastest error correction: Level 1 (well-suited for log files)
- Solely increase of efficiency: Level 0 (well-suited for video server)
- Level 3 and 5 improvements of level 2 and 4
- Level 3 and 5 for working with huge amounts of data
- Level 3 increases overall throughput
- Level 5 improves random access
- Level 6 would be improvement of level 5, but rarely realized
- Archive media are not (!) replaced by RAID systems
Backup Media: Tertiary Storage

Requirements

- Less used parts of database, with probably huge size (Text, Multimedia) should be stored "‘cheaper'" than with standard mag tapes
- Currently used databases have to be back upped additionally (archiving)

Tertiary storage: Medium replaceable

- offline: Media have to be changed manually
- nearline: Media are changed automatically (jukeboxes, mag tape robots)
Optical Disks

- CD-ROM, CD-R, CD-RW; DVD, DVD-R, DVD-RW, DVD+RW, ...
- Differences: executable operations, storage capacity
- Storage capacity CD: approx. 700 MB
- Storage capacity DVD: 4.7 GB to 17 GB data
- Optical disks and devices quite cheap
- Although random access is possible, the access is very slow (about 250 ms)
- Durability is comparatively high (approx. 30 years)
- Popular as storage for data required rarely but continuous
Tapes

- Very cheap medium with low cost devices
- Access yet more slowly than optical disks
- Sequential access: Transfer rate between 1 and 10 MB per second (acceptable)
- Random access: in the range of some minutes (on average)
- Very high capacity in the range of GBs (DLT (Digital Linear Tape) contains up to 70 GB packed data)
- Popular as archive storage
Jukeboxes and Robots

- Shut down / Rewind,
- Remove media from device,
- Mounting the medium,
- Grab new medium,
- Loading of media and boot up

Cost over 20 seconds with optical disks and more than 2 minutes with tapes

- Typical capacities: Tape robots with more than 10,000 cartridges (range of petabytes possible)
Long-term Backup/Archival

Aspects of durability:

- Physical stability of the medium ensures the *integrity* of data
- Availability of devices and drivers ensure the *readability* of data
- Available meta data ensure the *interpretability* of data
- Availability of programs, processing or working on the data ensure the *reusability* of data
Physical Stability: Integrity

- 10 years for magnetic tapes, 30 years for optical disks
- In comparison to classical archival media, e.g., paper, very low
- However, integrity of data is even longer guaranteed than other aspects
Readability

- Problem: Which systems, used today, can read tapes, floppy disks and punchcards mostly used in the 60s and 70s?
- New devices/systems would have to maintain numerous parameters like recording formats, type of parity information etc. for several decades
- Furthermore, procedure (of recording) not standardized or standardization is enhanced continuously
Interpretability

- Different codes for representation of characters (EBCDIC, ASCII, 16-Bit-Unicode)

- Document formats: Which (current) SW can read formatted text documents, created by word processing SW in the 70s or 80s?

- Solution: Storage in open standards like page description languages (Postscript) or markup languages (HTML, XML)
Management of Physical Storage

Abstraction of specific storage and backup media
Model: Sequence of Blocks

Set–Oriented Interface (SOI)
Record–Oriented Interface (ROI)
Internal Record Interface (IRI)
System Buffer Interface (SBI)
File Interface (FI)
Device Interface (DI)

Data System
Access System
Storage System
Buffer Management
Operating System
Operating System Files (I)

- Every relation or access path in one OS file respectively
- One or more OS files, relations and access paths are managed by the DBS itself within these files
- DBS itself manages the hard-disk and uses blocks in their original form (*raw device*)
Operating System Files (II)

Why not always using OS file management?

- Platform (OS) independence
- In 32-Bit operating systems: file size at most 4 GB
- OS files only on one medium (storable)
- OS-specific buffer management (i.e., blocks of secondary storage are used in main memory) does not meet the requirements of DBS
Blocks and Pages

- Assignment of physical blocks to *pages*
- Usually with constant multiplier: 1, 2, 4 or 8 blocks (of one track) on one page
- Here (for simplicity): "one block — one page"
- Addressing of higher levels of DBS via page numbers
Services

- Allocation or deallocation of memory space
- Fetch or write of page content
- Allocation in a way, that logical consecutive data areas (e.g., a relation) are preferably stored in consecutive blocks on disk
- After numerous update operations: Reorganisation methods
- (Free) storage management: Doubly-linked list of pages
### Mapping of Data Structures

- Mapping of conceptual level to internal data structures
- Supported by *meta data* (in the Data Dictionary, e.g., the internal schema)

<table>
<thead>
<tr>
<th>Conc. level</th>
<th>Internal level</th>
<th>File system/Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations →</td>
<td>Log. files →</td>
<td>Phys. files</td>
</tr>
<tr>
<td>Tuple →</td>
<td>Records →</td>
<td>Pages/Blocks</td>
</tr>
<tr>
<td>Attribute values →</td>
<td>Fields →</td>
<td>Bytes</td>
</tr>
</tbody>
</table>
Alternatives of Mapping

- Example 1: Every relation in one logical file, these files altogether in a single physical file
- Example 2: Cluster storage – several relations in one logical file
Common Form of Storage (I)

Relation

<table>
<thead>
<tr>
<th>PANr</th>
<th>Surname</th>
<th>Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Heuer</td>
<td>DBR</td>
</tr>
<tr>
<td>5588</td>
<td>Saake</td>
<td>MD</td>
</tr>
<tr>
<td>6834</td>
<td>Korn</td>
<td>MD</td>
</tr>
</tbody>
</table>

several logical files

Index File

Main File

Index File

Heuer

Korn

Saake

4711 Heuer ... DBR

5588 Saake ... MD

6834 Korn ... MD
Common Form of storage (II)

several logical files

Main File

Index File

one physical file

Index File
Adaptation of Records to Blocks

- Records (maybe of variable length) have to be adapted to blocks, consisting of a constant number of bytes: *Blocking*
- Blocking depends on length of data fields/records (variable or fix)
  - Records with variable length: Higher effort for read/write operations, record length has to be redetermined
  - Records with fixed length: Higher storage effort (overhead)
### Blocking Techniques

- **Unspanned Records**: every record in at most one block
- **Spanned Records**: record probably in several blocks

Standard: Unspanned records (only in the case of BLOBs or CLOBs spanned records are used)
Pages, Records and Addressing

- Structure of pages (in storage management): Double-linked list (*Free-List*)
- Free pages in (free) storage management

![Diagram showing storage management structure](image)
Page

- **Header**
  - Information about predecessor and successor page
  - Probably page number itself
  - Information about type of record
  - Free space (available)

- **Records**

- Unassigned bytes
Page: Addressing of Records

- Addressable units
  - Cylinder
  - Tracks
  - Sectors
  - Blocks or pages
  - Records in blocks or pages
  - Data fields/areas in records (e.g., a certain attribute)

- Example: Address of record realized by page number and offset (relative address from top of the page in bytes)
  
  (115, 142)
Page Access as Bottleneck

- Measurement for the speed of DB operations: Number of page accesses on secondary storage (caused by access vacancy)
- Rule of thumb: Time of access $\leftrightarrow$ Quality of access path $\leftrightarrow$ Number of required pages accesses
- Main memory operations not arbitrarily negligible

next slides: **Types of Records**
Pinned records

- *Pinned records* are bounded to their position.
- Example: References (directly on record) by pointer from other, hardly detectable page(s); e.g., moving of record (115, 142) to page 136.
- Risk: Pointer referencing to nothing (*dangling pointers*)
Unpinned records

- **Unpinned records**: References by "logical pointer"
- E.g.: Matriculation number for referencing to a student record
- This logical pointer can be mapped to current address at a central place (e.g., index file for student relation)
- Disadvantage: Moving of record requires loading both directly affected pages (source and target page of moving operation) as well as loading the index page
- This is resolved by TID concept
Records with fixed length

SQL: Data types with fixed and variable length
- char(n) string with fixed length n
- varchar(n) string with variable length with maximum length n

Structure of records, if all data fields with fixed length:
1. *Administration block* with
   - Type of record (if different types on one page allowed)
   - Deletion bit

2. *Free space* for adjusting the offset

3. *User data* of record
Records with variable length

- Required in administration block: Record length $l$, so that the length of the user data area $d$ is known

<table>
<thead>
<tr>
<th>$l$</th>
<th>$d$</th>
</tr>
</thead>
</table>

Length (or administration block)  (useful) data
Records with variable length (II)

a) I n aI A1 ... aln An

number attributes

attribute lengths

attribute values

b) I n aI A1 ... apn A1 ... An

number attributes

attribute pointer

attribute values
Storage of Records with variable Length

- **Strategy a)**: Every data field with variable length $A_i$ starts with a *length pointer* $al_i$, specifying the length of the following data field.

- **Strategy b)**: After length pointer $l$ and number of attributes at the beginning of the record, a pointer field $ap_1, \ldots, ap_n$ for all data fields with variable length is established.

Advantage of strategy b): easier navigation within one record (even for records in pages ⇒ TID)
Application of variable Records

"‘Repetition Groups’": List of values with same data type

- Strings of variable length, e.g., varchar\((n)\) are repetition groups with char as basis data type $\Rightarrow$ mathematical, this is the Kleene closure (char)*

- Attribute values with a set or list nature, which should be stored denormalized within the record (e.g., storage as nested relation or cluster storage); for a list of integer values this would be (integer)*

- Address field for index file, which points to several records for one attribute value (secondary index), i.e., (pointer)*
Huge, unstructured Records

- RDBS data types for very large, unstructured Information:
  - *Binary Large Objects (BLOBs)*: Byte sequences like images, audio or video sequences
  - *Character Large Objects (CLOBs)*: Sequences of ASCII characters (unstructured ASCII text)
- Long fields generally exceed the limit of one page, hence only non-BLOB fields are stored on original page
BLOB Storage: Solution 1

- Pointer as attribute value: Pointer points to the beginning of page or block list, which contains the BLOB

  Advantage for insert, delete or modify operation

  Disadvantage for random access within the BLOB
BLOB Storage: Solution 2

- BLOB directory as attribute value:
  - Size of BLOB
  - Further administration information
  - Several pointers, referencing to single pages

- Advantage: fast access on partitions of the BLOB

- Disadvantage: Fixed, limited (maximum) size of the BLOB (e.g., Gigabyte BLOB; 8-byte addressing, page size 1 KB ⇒ 8 MB for one BLOB directory)

- More efficient: B-Tree for storage of BLOBs (v.i.)
### BLOB Storage: Solution 2 (II)

#### BLOB Directory

<table>
<thead>
<tr>
<th>BLOB size</th>
<th>Administration info.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to block 1</td>
<td></td>
</tr>
<tr>
<td>Pointer to block 2</td>
<td></td>
</tr>
<tr>
<td>Pointer to block 3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Pointer to Block k</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing pointers to blocks and administration info.
Addressing: TID Concept

- *Tuple-Identifier* (TID) record address consisting of page number and offset
- Offset is used to reference within a single page ⇒ an offset value of \( i \) points to the \( i \)-th entry in a list of *tuple pointer* at the beginning of the page
- Each tuple pointer contains offset value
- Movements within a page: All references from "‘outside”” remain unchanged
- Movements to another page: Old record is replaced by new TID pointer
- This two-staged reference not desireable for efficiency reasons: Reorganisation in periodical intervals
TID Concept: One-Staged Reference

- TID
- Page p
- i–th tuple
- tuple pointer

Gunter Saake
Database Implementation Techniques
Last Changes: 21.4.2015
TID Concept: Two-Staged Reference

![Diagram showing TID concept](image_url)
Buffer Management in Detail

- **SOI**: Set-Oriented Interface
- **ROI**: Record-Oriented Interface
- **IRI**: Internal Record Interface
- **SBI**: System Buffer Interface
- **FI**: File Interface
- **DI**: Device Interface
Functions of Buffer Management

- **Buffer**: Designated part of the main memory
- Divided into *buffer frames*, each frame can contain one page from the hard-disk
- Functions:
  - Buffer management has to search required page in the buffer \(\Rightarrow\) efficient *search techniques*
  - Parallel DB transactions: clever *storage allocation* within the buffer
  - Buffer full: adequate *page replace strategies*
  - *Differences between OS buffer and DB buffer*
  - Specific application of buffer management: *shadow storage concept*
Searching a Page

- **Direct Search:** Linear search without additional means
- **Indirect Search:**
  - *Ordered or unordered table:* all pages of buffer are observed
  - *Linked list:* enables a faster, ordered insertion
  - *Hash table:* with adequate hash function an advantageous effort for searching and modifying is possible
Storage Allocation in the Buffer

for several transactions to be executed in parallel

- **Local strategies**: Disjunctive parts of the buffer are made available for every transaction (size can be determined statically before execution of TA or dynamically at program runtime)

- **Global strategies**: Access behaviour of all TAs determines storage allocation (pages, jointly referenced by several TAs can be regarded better this way)

- **Strategies related to page type**: Partition of the buffer: Buffer frames for data pages, access path pages, data dictionary pages, ...
Page Replace Strategies

- Storage system requests page $E_2$, which is not in the buffer
- All buffer frames are reserved
- Before loading $E_2$, a buffer frame has to be cleared
- The page has to be selected by the strategies described below
- If page has been modified within the buffer, *rewrite* the page to the hard-disk
- If page has only be read within the buffer, it can be overwritten (*replaced*)
Page Replace schematic

Main Memory

Storage System

Buffer

Secondary Storage

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>...</th>
<th>An-1</th>
<th>An</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>B2</td>
<td>...</td>
<td>Bn-1</td>
<td>Bn</td>
</tr>
<tr>
<td>C1</td>
<td>C2</td>
<td>...</td>
<td>Cn-1</td>
<td>Cn</td>
</tr>
<tr>
<td>D1</td>
<td>D2</td>
<td>...</td>
<td>Dn-1</td>
<td>Dn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E1</th>
<th>E2</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Page Replace: Techniques (I)

- **Demand-paging technique**: Exactly one page in the buffer is replaced by requested page

- **Prepaging technique**: Besides the requested page further pages, potentially required in future, are loaded into the buffer (e.g., reasonable for BLOBs)

- **Optimal strategy**: Which page has the maximum distance to the next usage? (not feasible, future reference behaviour not predictable)

→ Feasible techniques exhibit no knowledge over future reference behaviour

- **Fortune strategy**: Same reuse probability is assigned to every page
Page Replace: Techniques (II)

- Suitable, realizable techniques should use previous reference behaviour on pages, for estimating the expected value for reuse
  - Better than fortune strategy
  - Approximation to optimal strategy
Characteristics of established Strategies

- **Age** of page in the buffer:
  - Age of page after "storage" (the global strategy (G))
  - Age of page after last time of reference (strategy of youngest behaviour (Y))
  - Age of page is not considered (−)

- **Number** of references to page in the buffer:
  - Number of all references to a page (the global strategy (G))
  - Only number of last references to the page is considered (strategy of youngest behaviour (Y))
  - Number of references is not considered (−)
# Classification of established Strategies

<table>
<thead>
<tr>
<th>Technique</th>
<th>Principle</th>
<th>Age</th>
<th>Numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>oldest page replaced</td>
<td>G</td>
<td>–</td>
</tr>
<tr>
<td>LFU (least frequently used)</td>
<td>replace page with least frequency</td>
<td>–</td>
<td>G</td>
</tr>
<tr>
<td>LRU (least recently used)</td>
<td>replace page, not referenced for longest time (System R)</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DGCLOCK (dyn. generalized clock)</td>
<td>logging the replace frequency of important pages</td>
<td>G</td>
<td>YG</td>
</tr>
<tr>
<td>LRD (least reference density)</td>
<td>replace page with least reference density</td>
<td>YG</td>
<td>G</td>
</tr>
</tbody>
</table>
Lack of Appropriateness of OS Buffer (I)

Natural Join of relations \( A \) and \( B \) (corresponding sequence of pages: \( A_i \) bzw. \( B_j \)) \( \sim \) Implementation: *Nested-Loop*

Buffer

\[
\begin{array}{cccccc}
A1 & B1 & B2 & B3 & B4 & B5 \\
\end{array}
\]

Relation A

\[
\begin{array}{cccccc}
A1 \\
A2 \\
A3 \\
\ldots \\
\end{array}
\]

Relation B

\[
\begin{array}{cccc}
B1 \\
B2 \\
B3 \\
B4 \\
B5 \\
B6 \\
\end{array}
\]

A1 will be replaced by B1, B2, B3, B4, B5, and B6. Must be loaded.
Lack of Appropriateness of OS Buffer (II)

- FIFO: $A_1$ replaced, since oldest page in buffer
- LRU: $A_1$ replaced, since page has only been used during the first step for reading the compared tuple

**Problem**

- For next step, $A_1$ is required again
- Further "build up": for loading $A_1, B_1$ has to be removed (but needed in the next step) usw.

**Page Replace Strategies of DBS**

- *Fixing of pages*
- *Release of pages*
- *Rewriting* a page (e.g., at the end of a TA!)
Shadow Storage Concept (I)

- Modification of buffer concept
- If page has to be (re)written to the disk during a TA: not to original page, but to a new page
- Two parallel auxiliary structures (instead of one)
- Auxiliary structures: virtual page tables
Shadow Storage Concept (II)
Shadow Storage Concept (III)

- In case of TA abort, only access on original version $V_0$ necessary for recovery of the old state (before TA started) of the pages.
- Toggle between both versions of virtual page tables using a "'switch'"
- TA committed successfully $\Rightarrow V_1$ becomes original version and $V_0$ references to new shadow pages.
- Disadvantage: Formerly coherent page areas of one relation are "'scattered'" over secondary storage (after change operations on DB)
Cryptographic Methods

- Prohibit unauthorized access on DB
  - Within DBS: Right management using grant
  - On OS level: file encryption against "Dump"
  - Network: file encryption and, if data are transferred, secure channels (SSL)

- Common methods
  - Data Encryption Standard
  - Public-Key techniques: RSA, PGP
4. File Organisation - Access Structures

- Classification of storage techniques
- Statical techniques (Heap, index-sequential, indexed non-sequential)
- Tree techniques (B-Trees and different variants)
- Hash techniques
- Bitmap Indices
- Multidimensional storage techniques
- Clustering
- Physical data definition and realization in SQL systems
- Data catalogue / Data Dictionary
Classification into 5-Layer-Architecture

- Storage system requests pages via the system buffer interface (SBI)
- Interprets pages as *internal records*
- Internal realization of logical records with the help of pointer, specific index entries and further auxiliary structures
- Access system abstracts from concrete realization
Classification (II)

- **SOI** (Set-Oriented Interface)
- **ROI** (Record-Oriented Interface)
- **IRI** (Internal Record Interface)
- **SBI** (System Buffer Interface)
- **FI** (File Interface)
- **DI** (Device Interface)
Classification of Storage Techniques

Criteria for *access structures* or *access techniques*:

- Organizes internal relation (*file organisation form*) or only additional access opportunity on existing internal relation (*access path*)
- Kind of assigning given attribute values to record address
- Types of queries, which are supported (efficient) by file organisation forms and access paths
Primary Key vs. Secondary Key

Distinctive feature: Types of supported attribute(s)

- **Primary key**: Important characteristics
  - Devoid of duplicates
  - Identifying set of attributes
  - Join operations often used on primary key

- **Secondary key** arbitrary (other) set of attributes
  - Mostly no real key characteristics
  - Usual no identifying feature
  - Support of particular queries (e.g., Selection on secondary key)
Further distinctive Features

- Usually: Primar key supported by *primary index* or *file organisation form*; can be *clustered index*
- Usually: Secondary key supported by *secondary index* or *access path*, index is not *clustered*
- *Sparsely populated index* suitable only for primary key
- *Densely populated index* has to be used for secondary key
Primary Index vs. Secondary Index

- **Primary index**: Access path on internal relation, which can exploit the file organisation form of internal relation

- Internal relation $\Rightarrow$ Primary index can make use of sorting $\Rightarrow$ *Clustered index* or *sparsely populated index* possible

- Primary index usually on primary key, but on secondary key possible as well

- **Secondary index**: Every other access path on internal relation, which can’t exploit the file organisation form of the internal relation

- Secondary index: *non-clustered index* or *densely populated index*
Primary Index vs. Secondary Index (II)

- One primary index, several secondary indices per internal relation possible
- Some RDBS: Secondary indices as access path $\Rightarrow$ no index makes use of the type of storage of the internal relation
File Organisation vs. Access Path (I)

- **File organisation form**: Type/Kind of storage of internal relation
  - Unsorted storage of internal tuples: *Heap organisation*
  - Sorted storage of internal tuples: *Sequential organisation*
  - Scattered storage of internal tuples: *Hash organisation*
  - Storage of internal tuples in multidimensional space: *multidimensional file organisation forms*

- Usual: Sorting or hash functions on primary key

- Sorted storage plus additional primary index on sorting attributes: *index-sequential organisation form*
File Organisation vs. Access Path (II)

*Access path*: Access structure beyond basic file organisation form, e.g., index file

- Entries: \((K, K \uparrow)\)
- \(K\) is value of primary or secondary key
- \(K \uparrow\) is record or reference to record
- \(K\): *Search key*, more precisely: *Access attribute* and *access attribute value*
File Organisation vs. Access Path (III)

Entry $K \uparrow$:

- $K \uparrow$ is record by itself: Access path is degenerated to file organisation form
- $K \uparrow$ is address of internal tuple: Primary key; secondary key possible with $(K, K \uparrow_1), \ldots, (K, K \uparrow_n)$ with the same access attribute value $K$
- $K \uparrow$ is list of tuple addresses: Secondary key; variable length of index entries is harmful

Tuple addresses: TIDs, page addresses only, . . .

Index file can be stored by itself in file organisation form and hence, access paths can be added to it
Sparsely vs. Densely Populated Index (I)

- **Sparsely populated index**: Not for every access attribute value $K$ an entry in index file
  - Internal relation ordered by access attribute: one entry per page in the index ⇒ Index points with $(K_1, K_1↑)$ to *page leader*, next index entry $(K_2, K_2↑)$
  - Record with access attribute value $K? \text{ with } K_1 \leq K? < K_2 \text{ can be found on the page of } K_1↑$

- **Index-sequential file**: Sorted file with sparsely populated index as primary index
Sparingly vs. Densely Populated Index (II)

- *Densely populated index*: For every record of internal relation one entry in index file
- Primary index can be densely populated index, if file organisation form is heap file, but even in the case of sorting (*clustered index*)
Clustered vs. Non-Clustered Index

- *Clustered index*: Sorted according to the internal relation
- Ex.: Internal relation `students` sorted by matriculation number
  ⇒ Usually, index file clustered on attribute `matriculation number`
Clustered vs. Non-Clustered Index

- **Non-clustered index**: is organized differently to internal relation
- Ex.: Secondary index on *field of study*, file ordered by matriculation number
- Primary index can be sparsely populated and clustered
- Every sparsely populated index is a clustered index as well, but not vice versa
- Secondary index can only be a densely populated, non-clustered index (aka: inverted file)
Key access vs. Key Transformation

- **Key access**: Assignment of primary and secondary key values to addresses in auxiliary structure, e.g., index file
  - Ex.: Index-sequential organisation, B-Tree, KdB-tree, …

- **Key transformation**: Tuple address is computed based on primary and secondary key values (algorithm is stored instead of index entries)
  - Ex.: Hash techniques
Non-Composite vs. Composite Index

- **Non-composite index**: Access path on one single access attribute
- **Composite index**: Access path on several attributes
- Ex.: Attributes Name and ZIP support
  - either two non-composite indices
  - or one composite index on both attributes
- Advantage of composite index: For *exact-match* only one index access (and thus, less page accesses)
- Composite index: Kind of realization determines, whether *partial-match* is supported efficient besides *exact-match* (one- or multidimensional)
Onedimensional vs. Multidimensional Access Structure (I)

- Non-composite index always *onedimensional access structure*: Access attribute values define linear order in onedimensional space.

- Composite index can be onedimensional or *multidimensional access structure*
  - *Onedimensional case*: Combinations of different access attribute values are concatenated and considered as one single access attribute value (i.e., linear order in onedimensional space again)
Onedimensional vs. Multidimensional Access Structure (II)

- Ex. onedimensional case: \((\text{Name}, \text{ZIP})\) supports no *partial-match* on ZIP

<table>
<thead>
<tr>
<th>Name</th>
<th>ZIP-code</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andreas</td>
<td>18209</td>
<td>•</td>
</tr>
<tr>
<td>Christa</td>
<td>69121</td>
<td>•</td>
</tr>
<tr>
<td>Gunter</td>
<td>39106</td>
<td>•</td>
</tr>
</tbody>
</table>

- *Multidimensional case*: Set of access attribute values spans a multidimensional space \(\Rightarrow\) In the case of *partial-match*, horizontal or vertical straight line (in the space) determines number of hits

- Ex.: Queries on \(\text{ZIP-code}\) possible as well
Statistical vs. Dynamical Structure (I)

- **Statistical access structure**: Provides only for a fixed number of (managed) records an optimal solution
  - Ex. 1: Address transformation for identity card number $p$ of persons with $p \mod 5$.
  - 5 pages, page size 1 KB, average record length 200 Bytes, equipartition of ID card numbers.
  - Ideal for 25 persons, for 10,000 persons not sufficient anymore.
Statical vs. Dynamical Structure (II)

- **Statical access structure:**
  - Ex. 2: Phone books of telecommunication company *Jingle-and-Bell*
    - Three-staged index structure: Region index, in phone book an index on location, within a location users are sorted by surname
    - *Jingle-and-Bell* has 30 customers overall ⇒ only one page necessary (no three-staged index structure)
    - *Jingle-and-Bell* becomes global corporation with 3 billions customers: Three-staged index structure not sufficient, at least a fourth stage (country indices) necessary
Statical vs. Dynamical Structure (III)

- *Dynamical access structures* ideal, independent of number of records
  - Dynamic address transformation techniques modify dynamically the range of transformation
  - Dynamic index techniques modify dynamically the number of index stages ⇒ usual in DBS
Examples for Classifications

- Example for dynamical and onedimensional technique: *B-Tree* (most popular access path in relational database systems)
- Usually created with `create index`
- Only *exact-match*, no *partial-match*
- Example for statical and onedimensional technique: classical *Hash technique*
Requirements to Storage Techniques

- Dynamical behaviour
- Efficiency for single access (key search for primary index)
- Efficiency for multi-access (key search for secondary access)
- Usability for sequential traverse (sorting, clustered index)
- Clustering
- Query types: exact-match, partial-match, range queries
Statical Techniques

- Heap, index-sequential, indexed non-sequential
- Usual the basic storage technique in RDBS
- *Direct organisation forms*: No auxiliary structure, no address computation (Heap, sequential)
- Statical index techniques for primary and secondary index
Heap Organisation

- Records are stored completely unsorted
- Physical order of records corresponds to chronological order of receiving records

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>First Name</th>
<th>Date</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>8832</td>
<td>Tamara</td>
<td></td>
<td></td>
<td>11.11.73</td>
</tr>
<tr>
<td>5588</td>
<td>Gunter</td>
<td>Saake</td>
<td></td>
<td>5.10.60</td>
</tr>
<tr>
<td>4711</td>
<td>Andreas</td>
<td>Heuer</td>
<td></td>
<td>31.10.58</td>
</tr>
<tr>
<td>9999</td>
<td>Christa</td>
<td>Preisendanz</td>
<td></td>
<td>10.5.69</td>
</tr>
<tr>
<td>6834</td>
<td>Michael</td>
<td>Korn</td>
<td></td>
<td>24.9.74</td>
</tr>
<tr>
<td>7754</td>
<td>Andreas</td>
<td>Möller</td>
<td></td>
<td>25.2.76</td>
</tr>
<tr>
<td>9912</td>
<td>Antje</td>
<td>Hellhof</td>
<td></td>
<td>4.4.70</td>
</tr>
</tbody>
</table>

*frei*
Heap: Operations

- **insert**: Access on last page of the file. Free space sufficiently ⇒ append record. Otherwise fetch next free page.
- **delete**: lookup, afterwards set deletion bit to 0
- **lookup**: Sequential traverse of whole file, hence maximum effort (Heap file often used together with secondary index or only for very small relations)
- Complexity: Recording of data $O(1)$, search $O(n)$
Sequential Storage

- Sorted storage of records

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Andreas</td>
<td>Heuer</td>
<td>......</td>
<td>31.10.58</td>
</tr>
<tr>
<td>5588</td>
<td>Gunter</td>
<td>Saake</td>
<td>......</td>
<td>5.10.60</td>
</tr>
<tr>
<td>6834</td>
<td>Michael</td>
<td>Korn</td>
<td>......</td>
<td>24.9.74</td>
</tr>
<tr>
<td>7754</td>
<td>Andreas</td>
<td>Möller</td>
<td>......</td>
<td>25.2.76</td>
</tr>
<tr>
<td>8832</td>
<td>Tamara</td>
<td>Jagellowsk</td>
<td>......</td>
<td>11.11.73</td>
</tr>
<tr>
<td>9912</td>
<td>Antje</td>
<td>Hellhof</td>
<td>......</td>
<td>4.4.70</td>
</tr>
<tr>
<td>9999</td>
<td>Christa</td>
<td>Preisendanz</td>
<td>......</td>
<td>10.5.69</td>
</tr>
</tbody>
</table>

frei
Sequential File: Operations

- **insert**: Search page, insert record (sorted) ⇒ During creating or sequential filling of file every page should only be filled up to a certain degree (approx. 66%)

- **delete**: Effort is identical, but lookup becomes faster

The following file organisation forms exhibit:
- Faster lookup
- More required space (caused by auxiliary structures, e.g., index files)
- More time required for insert and delete

- Classical index form: Index-sequential file organisation
Index-Sequential File Organisation (I)

- Combination of sequential main file and index file: *index-sequential file organisation form*
- Index file can be clustered, sparsely populated index
- Conceivable as an, at least, two-staged tree
  - Leaf level is *main file* (records)
  - any other level is *index file*
## Index-Sequential File Organisation (II)

**Page 18**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name1</th>
<th>Name2</th>
<th>Date</th>
<th>Name3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Andreas</td>
<td>Heuer</td>
<td>31.10.58</td>
<td></td>
</tr>
<tr>
<td>5588</td>
<td>Gunter</td>
<td>Saake</td>
<td>5.10.60</td>
<td></td>
</tr>
<tr>
<td>6834</td>
<td>Michael</td>
<td>Korn</td>
<td>24.9.74</td>
<td></td>
</tr>
<tr>
<td>7754</td>
<td>Andreas</td>
<td>Möller</td>
<td>25.2.76</td>
<td></td>
</tr>
</tbody>
</table>

**Page 45**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Name1</th>
<th>Date</th>
<th>Name2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8832</td>
<td>Tamara</td>
<td>Jagellowsk</td>
<td>11.11.73</td>
<td></td>
</tr>
<tr>
<td>9912</td>
<td>Antje</td>
<td>Hellhof</td>
<td>4.4.70</td>
<td></td>
</tr>
<tr>
<td>9999</td>
<td>Christa</td>
<td>Preisendanz</td>
<td>10.5.69</td>
<td></td>
</tr>
</tbody>
</table>

frei
Structure of Index File

- Records in index file:
  (primary key value, page number)
  For every page of the main file exactly one index record in index file
- Problem: ""Root"" of the tree on more than one page in the case of one-level index (and rapidly growing data sets)
Structure of Index File (II)
Multi-Level Index

- Optional: Index file is managed index-sequential as well
- Ideally: Index of highest level fits into one page

Index file, 2nd stage

<table>
<thead>
<tr>
<th>2413</th>
<th>107</th>
</tr>
</thead>
<tbody>
<tr>
<td>10017</td>
<td>122</td>
</tr>
</tbody>
</table>

Index file, 1st stage

<table>
<thead>
<tr>
<th>2413</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>18</td>
</tr>
<tr>
<td>8832</td>
<td>45</td>
</tr>
</tbody>
</table>

Page 107

<table>
<thead>
<tr>
<th>10017</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>11732</td>
<td>3</td>
</tr>
</tbody>
</table>
**lookup** for Index-Sequential Files

**lookup** operation looks for corresponding record for access attribute value \( w \)

Index file is traversed sequential, while looking for \((v_1, p)\) within the index \( v_1 \leq w \):

- If \((v_1, p)\) is last record in index file, the record, corresponding to \( w \) can only be stored on this last page (if it exists)
- Next record \((v_2, p')\) of index with \( v_2 > w \), hence, the record, corresponding to \( w \), must be stored on page \( p \)

\((v_1, p)\) *overlaps* access attribute value \( w \)
**insert** for Index-Sequential Files

- **insert**: Firstly, find page with **lookup**
- If enough space, record is stored (sorted) in respective page; adjust index, if new record is the first record of the page
- No space, then fetch new page from free storage management; record of "crowded" page are equally distributed on old and new page; create index entry for new page
- Alternative: New record can be stored on overflow page (corresponding to the page found by **lookup**)
delete for Index-Sequential Files

- **delete**: Firstly, find page with lookup
- Delete record on page (set deletion bit to 0)
- Record is first one on page: adjustment of index
- If page is empty after deletion: adjustment of index, page back to free storage management
Problems of Index-Sequential Files

- *Heavily growing files*: Number of linear linked index pages grows; automatic adaptation of number of levels not envisioned
- *Heavily shrinking files*: Only a slowly decrease of index and main file pages
- *Unbalanced pages* in main file (unnecessarily high demand for storage space, too long access times)
Indexed Non-Sequential Access Path

- Used to support secondary keys
- Several access paths (of this form) per file possible
- Single- or multi-level: Higher index levels are organized index-sequential again
Structure of Index File

- Secondary index, densely populated and non-clustered index
- For every record in main file, a record \((w, p)\) exists in index file
- \(w\) secondary key value, \(p\) assigned page
  - Either several records are contained in index file for a certain \(w\)
  - or a list of addresses is given in the main file for a certain \(w\)
Structure of Index File (II)

Access path "name"

Andreas 18
Andreas 18
Antje 45
Christa 45
Gunter 18

Main file

Page 18

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Name</th>
<th>. . . . .</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Andreas</td>
<td>Heuer</td>
<td>. . . . .</td>
<td>31.10.58</td>
</tr>
<tr>
<td>5588</td>
<td>Gunter</td>
<td>Saake</td>
<td>. . . . .</td>
<td>5.10.60</td>
</tr>
<tr>
<td>6834</td>
<td>Michael</td>
<td>Korn</td>
<td>. . . . .</td>
<td>24.9.74</td>
</tr>
<tr>
<td>7754</td>
<td>Andreas</td>
<td>Möller</td>
<td>. . . . .</td>
<td>25.2.76</td>
</tr>
</tbody>
</table>

Page 45

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Name</th>
<th>. . . . .</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>8832</td>
<td>Tamara</td>
<td>Jagellowsk</td>
<td>. . . . .</td>
<td>11.11.73</td>
</tr>
<tr>
<td>9912</td>
<td>Antje</td>
<td>Hellhof</td>
<td>. . . . .</td>
<td>4.4.70</td>
</tr>
<tr>
<td>9999</td>
<td>Christa</td>
<td>Preisendanz</td>
<td>. . . . .</td>
<td>10.5.69</td>
</tr>
</tbody>
</table>
Structure of Index File (III)

Access path "city"

- BS: 45
- DBR: 18, 45
- HD: 45
- HRO: 45
- MD: 18

Main file

- Page 18
  4711 | Andreas | Heuer | DBR … 31.10.58
  5588 | Gunter  | Saake | MD … 5.10.60
  6834 | Michael | Korn  | MD … 24.9.74
  7754 | Andreas  | Möller| DBR … 25.2.76

- Page 45
  8832 | Tamara  | Jagellowsk| BS … 11.11.73
  9912 | Antje   | Hellhof | HRO … 4.4.70
  9999 | Christa | Preisendanz | HD … 10.5.69
  10015 | Denny  | Liebe | DBR … 5.8.77

Last Changes: 21.4.2015
Operations

- **lookup**: $w$ can occur several times, overlapping technique not needed
- **insert**: Adaptation of index files
- **delete**: Remove index entry
Tree Techniques

- Number of levels can be changed dynamically
- Most important tree techniques: \textit{B-Trees} and its variants
- Variants of B-Trees are even more omnipresent in modern database systems than SQL
- SQL only common in relational and object-relational database technology; B-Trees are used as basic technique anywhere
B-Trees

- Starting point: Balanced search tree
- *Balanced*: All paths from root to leaves of the tree are equidistant
- Main memory implementation structure: Binary search trees, e.g., AVL-Tree by Adelson-Velskii and Landis
- Database area: Nodes of search tree are tailored according to the page structure of DBS
- Several access attribute values on one page
- *Multiway-Trees*
Principle of the B-Tree

- *B-Tree* by Bayer (B acronym for balanced, broad, bushy, Bayer, **NOT**: Binary)

- Dynamic, balanced Index tree, where every index entry points to one page in the main file

Multiway-Tree totally balanced, if

1. all paths (from root to leaves) are equidistant
2. every node has the same amount of index entries

Total balancing too expensive, hence B-Tree criterion:

*Every page, except for the root page, contains between m and 2m data*
Characteristics of B-Tree (I)

\( n \) records in main file \( \Rightarrow \log_m(n) \) page accesses needed from root to leaf

- By the use of the balancing criterion a characteristic close to total balancing can be achieved (1. criterion is fulfilled completely, 2. criterion approximately)
- Criterion guarantees 50% utilization of memory space
- Simple, fast algorithms for search, insert and delete records (complexity about \( O(\log_m(n)) \))
Characteristics of B-Tree (II)

- B-Tree suitable as primary and secondary index
- Records directly in index pages ⇒ File organisation form
- Reference from index pages to records in main pages ⇒ Secondary index
Definition B-Tree

*Order* of B-Tree is minimal number of entries on index pages, except for the root page

Ex.: B-Tree of order 8 contains on every inner index page between 8 and 16 entries

Def.: An index tree is a B-Tree of order $m$, if the following requirements are fulfilled:

1. Every page contains at most $2m$ elements.
2. Every page, except for the root page, contains at least $m$ elements.
3. Every page is either a leaf page without successor or has $i + 1$ successors, if $i$ is the number of elements.
4. All leaf pages are on the same level.
Insert into a B-Tree: Example

1. Insert 1: $\{1\}$

2. Insert 5: $\{1\} \rightarrow \{1, 5\} \rightarrow \{1, 2, 5\}$

3. Insert 2: $\{2\}$

4. Insert 6: $\{2\} \rightarrow \{2, 6\} \rightarrow \{1, 2, 5\} \rightarrow \{1, 2, 5, 6\} \rightarrow \{1, 2, 5, 6, 7\}$

5. Insert 4: $\{4\}$

6. Insert 7: $\{7\}$

7. Insert 3: $\{3\}$

8. Insert 8: $\{8\}$

Split operations are indicated at each step.
Search in B-Trees

- **lookup** as in statical index techniques

Starting at the root page, the entry in B-Tree is detected, which overlaps the access attribute value \( w \) ⇒ Following the pointer, load page of next level

- **Search:** 38, 20, 6
Insert into B-Trees

Insert of value $w$

- Search of respective leaf page with **lookup**
- Relevant page found with $n < 2m$ elements, insert $w$ (sorted)
- Relevant page found with $n = 2m$ elements, create new page,
  - First $m$ values on original page
  - Last $m$ values on new page
  - The "splitting" element is displaced upwards to the respective index page
- Eventually, this process is done recursively up to the root
Delete in B-Trees

Less than $m$ Elements on page: Underflow
Deletion of value $w$: Example: 24; 28, 38, 35

- Search corresponding page with lookup
- $w$ is stored on leaf page $\Rightarrow$ Delete value, managing (possible existing) underflow
- $w$ not stored on leaf page $\Rightarrow$ Delete value, replace by lexicographically next smaller element of a leaf page, managing (possible existing) underflow on leaf page
Delete in B-Trees (II)

Dealing with underflows

- Balancing with neighboring page (if this page has \( n \) elements with \( n > m \))

- or merge two pages to one page (neighboring page with \( n = m \) elements), furthermore the "middle" element of upper index page is added, maybe underflow on this upper index page has to be handled
Complexity of Operations

- Effort for insert, lookup and delete in B-Tree always $O(log_m(n))$ operations
- Corresponds to the "'height'" of the tree
- Concrete: Page size 4 KB, access attribute value 32 bytes, 8-byte pointer: between 50 and 100 index entries per page; order of tree is 50
- 1,000,000 records: $\log_{50}(1,000,000) = 4$ page accesses in worst case
- Usually root page of every B-Tree is in the buffer: 3 page accesses
Insert and Delete in B-Tree

Insert of element 22; deletion of 22
Variants

- $\mathbf{B^+}$-Trees: Main file integrated as undermost (leaf) level of tree
- $\mathbf{B^*}$-Trees: Splitting pages is avoided by ""Shuffle"
- Prefix-B-Trees: Strings as access attribute values, only prefix is indexed
**B⁺-Tree**

- Most commonly used variant of B-Tree (in practice): more efficient change operations, decrease of tree height
- Records of main file are integrated on leaf pages of tree
- Inner nodes containing only access attribute value and pointer to successive page on next level
B-Tree and $B^+$-Tree by Comparison

The diagram illustrates the structure of B-Tree and $B^+$-Tree index files and main files. The key comparison points between the two types of trees are highlighted.

The B-Tree uses a simple search mechanism, whereas the $B^+$-Tree supports range queries more efficiently.
Order; Operations

- **Order** for $B^+$-Tree: $(x, y)$, $x$ minimum assignment of index pages, $y$ minimum assignment of record pages
- **delete** more efficient in comparison to B-Tree ("‘borrow’" an element of leaf page is not applicable)
- Access attribute values of inner nodes even remain
- Usually used as primary index
- $B^+$-Tree is dynamical, multi-level, index-sequential file
B*- and B#-Tree

- Drawback of B-Tree: Frequent splitting of pages and low exploitation of memory space (about 50 %)

- B*-Tree, B#-Tree:
  - Redistribution of records to potentially not fully loaded neighboring pages instead of splitting pages in the case of underflow
  - If not applicable: Two pages are partitioned to three pages (enables average memory space usage of 66% compared to 50%)
B⁺-Tree for BLOBs

- Instead of access attribute values in B⁺-Tree: Positions or offsets within the BLOB are indexed

- BLOB-B⁺-Tree: Position-B⁺-Tree

- Suitable for other large storage objects as well (as usual in object-oriented DBs)
Digital and Prefix Trees

- B-Trees: Problems with strings to be indexed
- Solution: Digital and prefix trees
- Digital trees index (fixed) the characters of the underlying alphabet
  can become unbalanced
- Examples: Tries, Patricia-Trees
- Prefix trees index prefix of the strings
Tries

- Originate from ""(Text) Retrieval"", but it is pronounced like try

```
  H
  J
  K
  L
  A
  O
  E
  T
  U
```

- Node E:
  - Child H: Hellhof
  - Child L: Heuer
  - Child U: Jagellowsk

- Node U:
  - Child E: Korn
  - Child T: Lansen
  - Child O: Loeser

- Node T:
  - Child L: Lottermann
  - Child U: Ludwig

Gunter Saake

Database Implementation Techniques

Last Changes: 21.4.2015
Nodes of a Trie

Problems: long, shared partial words, non-existing characters and character combinations, possibly empty nodes, extremely unbalanced trees
Patricia-Trees

- Tries: Problems with inventory IDs, path names, URLs (long, shared partial strings)
- Solution: *Practical Algorithm To Retrieve Information Coded In Alphanumeric* (Patricia)
- Principle: Skip partial strings
- Problem: Searched term "‘engine lock’" not in DB ⇒ Not detected until comparison with node "‘database language’"
Patricia-Tree and Trie by Comparison

Skipped partial strings additionally stored: Prefix trees
Prefix Trees

Dat 3

org.
a

base 4

odel

m

language

s

ystem

e

Gunter Saake

Database Implementation Techniques

Last Changes: 21.4.2015
Hash Techniques

- Key transformation and overflow handling
- DB technique: Transformation range complies with page address space
- Dynamic: Dynamic hash functions or re-hashing
Basic Principles

- Basic hash function: $h(k) = k \mod m$
- $m$ preferably prime number
- Overflow handling
  - Overflow pages as linked list
  - Linear sounding
  - Quadratic sounding
  - Double hashing
Hash Techniques for Databases

![Hash Table Diagram]

The diagram illustrates a hash table with a hash function `h` that maps keys to indices in the table. The keys include names such as Arnie, Gunter, Jerry, Regis, Andreas, Zuse, Tamara, Gunther, Hastur, and more. The hash table is structured to show how different keys are hashed to different indices. The keys are distributed across the table, and the diagram highlights how the hash function `h` is applied to different keys, resulting in their respective positions in the table.
Operations and Time Complexity

- **lookup, modify, insert, delete**
- **lookup** needs at most $1 + \#B(h(w))$ page accesses
- $\#B(h(w))$ number of pages (including overflow pages) of bucket for hash value $h(w)$
- Lower bound 2 (Access on hash directory plus access on first page)
Static Hashing: Problems

- Lack of dynamic
- Increase of (storage) range requires completely new hashing
- Choice of hash function is essential; Ex.: Hash index of 100 buckets, students to hash on 6-digit MATRNR (is assigned consecutive)
  - First two positions as hash value: Records are stored on few pages (nearly sequential)
  - Last two positions as hash value: Records are distributed equally on all pages
- Sorted output of a relation is inefficient
Dynamic Hashing

- Static hash techniques
  - Re-hashing if used capacity increases

- Dynamic hash techniques
  - Hash techniques with dynamical resizeable transformation range
    - Usual doubling of range...
  - Often in combination with additional index for addressing of blocks

- Example: Linear hashing
Linear Hashing

Sequence of hash functions, which are characterized as follows:

- \( h_i : \text{dom}(\text{primary key}) \rightarrow \{0, \ldots, 2^i \times N\} \) is a sequence of hash functions with \( i \in \{0, 1, 2, \ldots\} \) and \( N \) as initial size of the hash directory.

- Value of \( i \) is also referred to as *Level* of the hash function. 
*dom*(primary key) is abbreviated for the remainder as *dom*(Prim).
Linear Hashing (II)

For these hash functions apply the following conditions:

- $h_{i+1}(w) = h_i(w)$ for approx. the half of all $w \in \text{dom(Prim)}$
- $h_{i+1}(w) = h_i(w) + 2^i \times N$ for the other half

For instance, conditions are fulfilled, if $h_i(w)$ is defined as $w \mod (2^i \times N)$

- Representation by bit strings, adding one bit results in doubled transformation range
Principle of Linear Hashing

For one $w$ at most two responsible hash functions, whose level differs only by 1, decision for one function (of both) by *split pointer*

- $sp$ split pointer (determines which page is splitted next)
- $lv$ level (determines which hash functions are used)

Based on split pointer and level, the total number $Num$ of reserved pages can be computed:

$$Num = 2^{lv} + sp$$

Both values are initialized with 0 at the beginning.
# Principle of Linear Hashing (II)

<table>
<thead>
<tr>
<th>Page No.</th>
<th>h(w)</th>
<th>Pages</th>
<th>Overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00011011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10101010 10110101</td>
<td>10011001</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>01010101 01011010</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>11111111 11111110</td>
<td>11111101</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00100000</td>
<td></td>
</tr>
</tbody>
</table>
Lookup

\[ s := h_w(w); \]
\[ \text{if } s < sp \]
\[ \quad \text{then } s := h_{w+1}(w); \]

- Firstly, hash value is determined with "'smaller'" hash function
- If hash value less than value of split pointer ⇒ "'bigger'" hash function is used
**Split of a Page**

1. Records of a page (bucket), where $sp$ points to, are redistributed using $h_{lv+1}$ (approx. half of the records are moved to the page (bucket) with hash number $2^{lv} \times N + sp$)

2. The split pointer is increased: $sp := sp + 1$

3. If this process is finished for one level, it starts with page 0 again; level is increased by 1:

   ```
   if $sp = 2^{lv} \times N$ then
   begin
      $lv := lv + 1$
      $sp := 0$
   end;
   ```
### Example after Split

<table>
<thead>
<tr>
<th>Page No.</th>
<th>$h(w)$</th>
<th>Pages</th>
<th>Overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00011011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>10011001</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>01010101 01011010</td>
<td>11111101</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>11111111 11111110</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00100000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>10101010 10110101</td>
<td></td>
</tr>
</tbody>
</table>
Problems of Linear Hashing

<table>
<thead>
<tr>
<th>Page No.</th>
<th>( h(w) )</th>
<th>Pages</th>
<th>Overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00011011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>10011001</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>01010101 01011010</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00100000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>10101010 10110101</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>11111111 11111110</td>
<td></td>
</tr>
</tbody>
</table>

\( lv = 3 \)
\( sp = 0 \)
Improvements

- Extendable hashing
- Spiral hashing
- Combination of both techniques
Extendable Hashing

- Problem: Split takes place at fixed position, but not necessarily where overflow of pages occurs
- Idea: binary Trie for access on index pages
- Leaves of different depth
  - Index pages with depth value
  - Split realized in the case of overflow
- But: Storage not as Trie, but as an array
  - Corresponds to a complete Trie with maximum depth
    - “shared” pages as leaves
  - Array of size $2^d$ for maximum depth $d$
    - Only one storage access required!
  - Overflow: Index size possibly has to be doubled!
### Initial situation:
- Insert of 00111111 would lead to overflow, if the maximum depth is reached

<table>
<thead>
<tr>
<th>Depth</th>
<th>Index</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = 2</td>
<td>00</td>
<td>00011011 00100000</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>01010101 01011010</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10101010 10110101</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11111111 11111110</td>
</tr>
</tbody>
</table>
Extendable Hashing III

- Doubling of index size

<table>
<thead>
<tr>
<th>Depth</th>
<th>Index</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000</td>
<td>00011011</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>00100000</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>01010101</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>01011010</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>10101010</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>10110101</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>11111111</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td>11111110</td>
</tr>
</tbody>
</table>
Extendable Hashing IV

- Now possible: Split of page

<table>
<thead>
<tr>
<th>Depth</th>
<th>Index</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000</td>
<td>00011011</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>01010101 01011010</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>10101010 10110101</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>11111111 11111110</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111</td>
<td>00100000 00111111</td>
</tr>
</tbody>
</table>
Spiral Hashing I

- Problem: Cyclical increasing likelihood of splitting
- Solution: Different density of hash values
  - Interpretation of bit strings as binary representation of decimal places/digits of a number between 0.0 and 1.0
  - Function from \([0.0, 1.0] \rightarrow [0.0, 1.0]\) so that density of equally distributed values close to 1.0 is twice the number as close to 0.0
Spiral Hashing II

- Redistribution using exponential function
- Function $exp(n)$
  
  $$exp(n) = 2^n - 1$$

  fulfills the conditions
  - Particularly it is valid that $2^0 - 1 = 0$ and $2^1 - 1 = 1$
- Hash function $exhash$
  
  $$exhash(k) = exp(h(k)) = 2^{h(k)} - 1$$
Spiral Hashing III

Impact of used hash function in interval 0.0 to 1.0

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2^n - 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0717735</td>
</tr>
<tr>
<td>0.2</td>
<td>0.1486984</td>
</tr>
<tr>
<td>0.3</td>
<td>0.2311444</td>
</tr>
<tr>
<td>0.4</td>
<td>0.3195079</td>
</tr>
<tr>
<td>0.5</td>
<td>0.4142136</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5157166</td>
</tr>
<tr>
<td>0.7</td>
<td>0.6245048</td>
</tr>
<tr>
<td>0.8</td>
<td>0.7411011</td>
</tr>
<tr>
<td>0.9</td>
<td>0.866066</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Spiral Hashing IV

- Spiral expansion
  - Starting point: 4 pages of depth 2
Spiral Hashing V

- Spiral expansion
  - Split of page with highest density
  - Result: 5 pages, 3 of depth 2 and 2 of depth 3
Bitmap Indices

- **Idea:** *Bit array* for encoding the assignment of tuple to attribute value

- **Comparison with tree-based index structures:**
  - Avoidance of degenerated B-Trees
  - More insensitive against higher number of attributes
  - Easier support of queries, where only some (of the indexed) attributes are confined
  - In return, usually higher effort for updates
    - For example, in Data Warehouses this is quite unproblematic because of the predominant read access
Bitmap Index: Realization

- **Principle:** Replacement of TIDs (rowid) by a bit list for a key value in $B^+$-Tree
- **Structure of nodes:**

<table>
<thead>
<tr>
<th>F</th>
<th>010010...01</th>
<th>O</th>
<th>0111010...00</th>
<th>P</th>
<th>100000...10</th>
</tr>
</thead>
</table>
- **Advantage:** Less memory requirements
  - Example: 150,000 tuples, 3 different key values, 4 byte for TID
    - $B^+$-Tree: 600 KB
    - Bitmap: $3 \cdot 18750$ Byte = 56KB
- **Disadvantage:** Effort for update
Bitmap Index: Realization /2

- Definition in Oracle
  
  ```sql
  CREATE BITMAP INDEX order_status_idx ON order(status);
  ```

- Storage in compressed form
Standard Bitmap Index

- Every attribute is stored separately
- For each value of an attribute a bitmap vector is created:
  - For each tuple one bit exists, which is set to 1, if the indexed attribute in the tuple contains the reference value of this bitmap vector
  - The number of resulting bitmap vectors per dimension corresponds to the number of different values, which exist for the attribute
### Standard Bitmap Index /2

- **Example: Attribute gender**
  - 2 values (m/f)
  - 2 bitmap vectors

<table>
<thead>
<tr>
<th>PersId</th>
<th>Name</th>
<th>gender</th>
<th>Bitmap-f</th>
<th>Bitmap-m</th>
</tr>
</thead>
<tbody>
<tr>
<td>007</td>
<td>James Bond</td>
<td>M</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>008</td>
<td>Amelie Lux</td>
<td>F</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>Harald Schmidt</td>
<td>M</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>Heike Drechsler</td>
<td>F</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Standard Bitmap Index /3

- Selection of tuples by concatenation of respective bitmap vectors
- Example: Bitmap index on attributes gender and month of birth
  - (i.e., 2 bitmap vectors $B_f$ and $B_m$ for gender 12 bitmap vectors $B_1$, ..., $B_{12}$ for the month, if all months occur)
- Query: "all women, born in march"
  - Computation: $B_f \land B_3$ (concatenated bitwise conjunctive)
  - Result: all tuples, where at the position in the bitmap vector (in the result) a 1 occurs
Multicomponent Bitmap Index

- With bitmap standard index ⇒ attributes with many values result in a great many of bitmap vectors
- \(< n, m >\)-Multicomponent bitmap indices enable that \(n \cdot m\) possible values can be indexed by \(n + m\) bitmap vectors
- Every value \(x (0 \leq x \leq n \cdot m - 1)\) can be represented by two values \(y\) and \(z\):

\[
x = n \cdot y + z \text{ with } 0 \leq y \leq m - 1 \text{ and } 0 \leq z \leq n - 1
\]

- As a result, at most \(m\) bitmap vectors for \(y\) and \(n\) bitmap vectors for \(z\) are needed
- Memory requirements reduced from \(n \cdot m\) to \(n + m\)
- In exchange, for a exact match query two bitmap vectors have to be read
Multicomponent Bitmap Index (II)

- Example: Two-component bitmap index
- for $M = 0..11$ about $x = 4 \cdot y + z$
- y-values: B-2-1, B-1-1, B-0-1
- z-values: B-3-0, B-2-0, B-1-0, B-0-0

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>B-2-1</td>
<td>B-1-1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: Postal Code

- Encoding of postal code
- Values from 00000 to 99999
- Direct realization: 100,000 columns
- Two-component bitmap index (first 2 digits + 3 last digits): 1,100 columns
- Five components: 50 columns
  - Suited for range queries "ZIP 39***"
- Binary coded (up to $2^{17}$): 34 columns
  - only for exact match queries!
- Note: encoding with base 3 results in only 33 columns...
Multidimensional Storage Techniques

- So far: onedimensional (no partial-match queries, only linear order)
- Now: multidimensional (even partial-match queries, positioning in multidimensional data space)
- $k$ dimensions $= k$ attributes can be supported equal
- This section
  - multidimensional B-Tree
  - multidimensional hash technique
  - Grid files
- Further multidimensional techniques for multimedia and geo(graphical) data in next chapter
Multidimensional Tree Techniques

*KdB-Tree* is a $B^+$-Tree, where the index pages are realized as binary trees with access attributes, access attribute values and pointer

Variants of $k$-dimensional index trees:

- **kd-Tree** by Bentley and Friedman: Multidimensional basic structure (binary tree) developed for main memory algorithms
- **KDB-Tree** by Robinson: Combination of kd-Tree and B-Tree ($k$-dimensional index tree with higher branching factor)
- **KdB-Tree** by Kuchen: Improvement of Robinsons variant, covered in this lecture
Multidimensional Tree Techniques (II)

- KdB-Tree can support primary key and several secondary keys in parallel
- Usage as file organisation form $\Rightarrow$ additional secondary indices are unnecessary
Definition KdB-Tree

Idea: on each index page a partial tree is represented, which branches after several consecutive attributes

- **KdB-Tree of Type** \((b, t)\) consists of
  - *Inner nodes* (*range pages*) which contain a *kd-Tree* with at most \(b\) internal nodes
  - *Leaves* (*record pages*) which can contain up to \(t\) tuples of the stored relation

- **Range pages**: *kd-Tree* contained with *slice elements* and two pointer
  - Slice element contains *access attribute* and *access attribute value*; left pointer: smaller access attribute values; right pointer: bigger access attribute values
Example

- **Name**: Gunter
- **PANr**: 7754
- **SName**: Möller
- **PANr**: 8832
- Records:
  - **Record**: 5834, Korn, Michael
  - **Record**: 4711, Heuer, Andreas
KdB-Tree: Structure

- Range pages
  - Number of slice and address elements of the page
  - Pointer to root of contained kd-Tree
  - *Slice and address elements.*

- Slice element
  - Access attribute
  - Access attribute value
  - Two pointer to child node of the kd-Tree of this page (can be slice as well as address elements)

- Address elements: Address of a successor of the range page in KdB-Tree (range or record page)
KdB-Tree: Operations

- Complexity lookup, insert and delete for exact-match $O(\log n)$
- For partial-match better than $O(n)$
- If $t$ of $k$ attributes are specified in the query: Access complexity of $O(n^{1-t/k})$
KdB-Tree: Slice attributes

- Order of slice attributes
  - Determined cyclic
  - or selectivities are included: Access attribute with high selectivity should be used earlier and more frequently as slice element

- slice attribute value: Because of information on distribution of attribute values a suitable ""center"" of a attribute value range to be sliced should be determined
KdB-Tree: Brickwall

2d-Tree \((A_1, A_2)\)

- \((6, -)\)
- \((-6)\)
- \((4, -)\)
- \((-4)\)
- \((3, -)\)
- \(S_7\)
- \(S_3\)
- \(S_4\)
- \(S_6\)
- \(S_7\)
- \(S_1\)
- \(S_2\)

Diagram:

```
   S_7
   /    \
S_3    S_1\n   |    |  \    \
S_4    S_6  S_2 \\
   |    |      |    \
S_5    |      |    S_7
```

```
A_2
7
6
5
4
3
2
1

1 2 3 4 5 6 7 A_1
```

Gunter Saake
Database Implementation Techniques

Last Changes: 21.4.2015
Multidimensional Hashing

- Idea: Bit interleaving
- The bits of the address of different access attribute values are computed alternately
- Example: Two dimensions

<table>
<thead>
<tr>
<th></th>
<th>0<em>0</em></th>
<th>0<em>1</em></th>
<th>1<em>0</em></th>
<th>1<em>1</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0<em>0</em></td>
<td>0000</td>
<td>0001</td>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>0<em>1</em></td>
<td>0010</td>
<td>0011</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>1<em>0</em></td>
<td>1000</td>
<td>1001</td>
<td>1100</td>
<td>1101</td>
</tr>
<tr>
<td>1<em>1</em></td>
<td>1010</td>
<td>1011</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>
MDH by Kuchen

Idee

- MDH is based on linear hashing
- Hash values are bit sequences, where the beginning sequence is used as current hash value respectively
- One bit string per participating attribute is computed respectively
- Beginning sequences are now processed cyclical by the principle of bit interleaving
- Hash value is composed in turns from the bits of the single values
MDH formal (I)

- Multidimensional value $x$

\[
x := (x_1, \ldots, x_k) \in D = D_1 \times \cdots \times D_k
\]

- Sequence of hash functions, indexed with $i$, is constructed

- $i$-th hash function $h_i(x)$ is composed by a composition function $\bar{h}_i$ from the respective $i$-th beginning sequences of the local hash values $h_{ij}(x_j)$: $h_i(x) = \bar{h}_i(h_{i1}(x_1), \ldots, h_{ik}(x_k))$

- Local hash functions $h_{ij}$ result into bit vector of length $z_{ij}$:

\[
h_{ij} : D_j \rightarrow \{0, \ldots, z_{ij}\}, \ j \in \{1, \ldots, k\}
\]
MDH formal (II)

- \( z_{ij} \) should be equal, so that the dimensions are considered equally.
- Composition function \( \tilde{h}_i \) composes local bit vectors to bit vector of length \( i \):

\[
\tilde{h}_i : \{0, \ldots, z_{i_1}\} \times \cdots \times \{0, \ldots, z_{i_k}\} \rightarrow \{0, \ldots, 2^{i+1} - 1\}
\]
MDH formal (III)

- Balanced length of $z_{ij}$ is determined by following definition, which increases the lengths cyclical for each enhancement step (for one position):

$$z_{ij} = \begin{cases} 
2^{\lfloor \frac{i}{k} \rfloor} + 1 - 1 & \text{if } j - 1 \leq (i \mod k) \\
2^{\lfloor \frac{i}{k} \rfloor} - 1 & \text{if } j - 1 > (i \mod k)
\end{cases}$$

- Composition function:

$$\tilde{h}_i(x) = \sum_{r=0}^{i} \left( \frac{(x_{(r \mod k)+1} \mod 2^{\lfloor \frac{r}{k} \rfloor} + 1) - (x_{(r \mod k)+1} \mod 2^{\lfloor \frac{r}{k} \rfloor})}{2^{\lfloor \frac{r}{k} \rfloor}} \right) 2^r$$
MDH Illustration

Clarifies composition of hash function $h_i$ for three dimensions and the value $i = 7$

Highlighted parts of bit string correspond to the values $h_{71}(x_1)$, $h_{72}(x_2)$ und $h_{73}(x_3)$

During the step to $i = 8$ a further bit of $x_2$ (precise: of $h_{82}(x_2)$) would be used
MDH Complexity

- Exact-Match queries: $O(1)$
- Partial-Match queries, with $t$ of $k$ attributes determined, complexity $O(n^{1 - \frac{t}{k}})$
- Results from the number of pages, if certain bits are "unknown"
- Special cases: $O(1)$ for $t = k$, $O(n)$ for $t = 0$
Grid Files

- Most common and attractive multidimensional file organisation form (regarding the underlying technique)
- Separate category: Elements are combinations of key transformation (like hash techniques) and index files (like tree techniques)
- Multidimensional space is partitioned equally (in contrast to Brickwall)
Grid File: Objectives

Hinrichs and Nievergelt

- Principle of two disk accesses: Every record should be achievable in two accesses, regarding an *exact-match* query
- Decomposition of data space in cuboids: *n*-dimensional cuboids represent the search regions of the grid file
- Principle of adjacency preservation: Similar objects should be stored on the same page
- Symmetric treatment of all space dimensions: *partial-match* queries possible
- Dynamical adaptation of grid structure in the case of delete and insert operations
Principle of Two Disk Accesses

For exact-match

1. sought-after $k$-tuple is mapped to the intervals of the *scales*; index values are computed as combination of determined intervals; scales in main memory ⇒ no disk access

2. Access on *Grid-Directory* via computed index values; here, addresses of record pages are stored; first *disk access*.

3. Record access: second *disk access*. 
Structure of a Grid File (I)

Diagram (Intervals)

Grid–Directory (Grid cells)

Regions

Pages
Structure of a Grid File (II)

- **Grid**: $k$ onedimensional fields (scales), every scale represents an attribute
- **Scales** consist of partition of the corresponding range of values in intervals
- **Grid directory** consist of grid cells, which partition the data space in cuboids
- **Grid cells** represent grid region; to each region exactly one record page is assigned
- **Grid region**: $k$-dimensional, convex structure (regions are pairwise disjunctive)
Operations

At the beginning: Cell = Region = one record page

- **Page overflow**: Page is split. If the grid region, corresponding to the page, consists of only one grid cell, an interval on the scale must be partitioned into two intervals. If a region consists of several cells, these cells have to be separated in several (single) regions.

- **Page underflow**: Two regions are merged to one region, if the result yields to a new, convex region.
Example

- **Initial grid file**

- Insert records: (45, D), (2, A), (87, S), (75, M), (55, K), (3, Z), (15, D), (25, K), (48, F)

- Every page of the grid file can contain up to three records
Buddy System

- Described technique: *Buddy system* (twin system)
- Cells, resulting from the same step can be merged to regions; no other aggregation of cells is allowed within the buddy system
- Inflexible deletion: Only merging of regions is allowed, which have been evolved as twins
- Example: Delete (15,D): Merge pages 1 and 4; delete (87,S), page 2 is understaffed, but can not be merged with any other page
Cluster Creation

- Jointly storage of records on pages
- Important special cases:
  - *Cluster on key attributes.* Support of range queries and grouping: Records are stored coherent in sorting order on pages ⇒ *index-organized tables* or clustered, densely populated primary indices
  - *Cluster on foreign key attributes.* Groups of records, sharing one attribute value, are clustered on pages (support of join queries)
- Component relations instead of join attributes in OODBs
Index-Organized Tables

- Tuples are directly contained in index
- However, in case of frequent splitting the TID concept is meaningless
- Further secondary index can not be created due to missing TID
- E.g., unique not possible
Cluster for Join Queries

Join attribute: Cluster key

<table>
<thead>
<tr>
<th>OrderNo</th>
<th>Order date</th>
<th>Customer</th>
<th>Delivery date</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>15.04.98</td>
<td>Orion Enterprises</td>
<td>01.01.2001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Position</th>
<th>Part</th>
<th>Amount</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aluminiumtorso</td>
<td>2</td>
<td>3145,67</td>
</tr>
<tr>
<td>2</td>
<td>Antenna</td>
<td>2</td>
<td>32,50</td>
</tr>
<tr>
<td>3</td>
<td>Overkill</td>
<td>1</td>
<td>1313,45</td>
</tr>
<tr>
<td>4</td>
<td>Rivets</td>
<td>1000</td>
<td>−.50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OrderNo</th>
<th>Order date</th>
<th>Customer</th>
<th>Order date</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>05.10.98</td>
<td>Kirk Enterpr.</td>
<td>31.12.1999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Position</th>
<th>Part</th>
<th>Amount</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Beamer</td>
<td>1</td>
<td>13145,67</td>
</tr>
<tr>
<td>2</td>
<td>Energiekristall</td>
<td>2</td>
<td>32,99</td>
</tr>
<tr>
<td>3</td>
<td>Phaser</td>
<td>5</td>
<td>1313,45</td>
</tr>
<tr>
<td>4</td>
<td>Rivets</td>
<td>2000</td>
<td>−.50</td>
</tr>
</tbody>
</table>
Definition of Cluster

```sql
create cluster OrderCluster
    (OrderNo number(3))
pctused 80 pctfree 5;

create table T_Order
    (OrderNo number(3) primary key, ...)
cluster OrderCluster (OrderNo);

create table T_Orderposition
    (Position number(3),
    OrderNo number(3) references T_Order,
    ...
    constraint OrderPosKey
        primary key (Position, OrderNo)
    )
cluster OrderCluster (OrderNo);
```
Organisation of Cluster

- **Indexed cluster** use an sorted index (e.g. $B^+$-Tree) on the cluster key for access on the cluster.

- **Hash cluster** determine the suitable cluster with a hash function.

Indices for cluster correspond to normal indices for the cluster key.

Cluster identifiers or direct storage addresses are used instead of TIDs (for hash techniques).
Indexed Cluster

create index OrderClusterIndex
on cluster OrderCluster
Hash Cluster

```sql
create cluster OrderCluster (  
    OrderNo number(5,0))  
pctused 80  
pctfree 5  
size 2k  
hash is OrderNo  
hashkeys 100000;
```
## Physical Data Definition in SQL

<table>
<thead>
<tr>
<th></th>
<th>Ingres</th>
<th>Oracle</th>
<th>DB2</th>
<th>Informix</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential, Heap</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>index-seq. (sparsely)</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>indexed non-seq. (densely)</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>multilevel</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B-Tree</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B⁺-Tree (densely)</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>KdB-Tree</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Hash</td>
<td>+</td>
<td>(+)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MDH</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cluster</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>
Ingres

```
create [ unique ] index indexname 
    on relname ( 
    attrname [ asc | desc ],
    ... ) [ with-clause ];
```

Information of physical storage in `with`=clause

```
structure = cbtree | btree | cisam | isam 
    | chash | hash,
key = ( attrname, ... ),
minpages = n, maxpages = n, fillfactor = n,
leaffill = n, noleaffill = n,
location (location, ...)
```
**Ingres: modify**

```
modify  rel-or-index-name  
to  storage-structure  
[  on  attr-name  [asc|desc]  
{},  attr-name  [asc|desc]  } ]  
with-clause
```

- Furthermore compressed versions for all organisation forms
- Especially useful for strings to be indexed: (CHeap, CHash, CBtree, ...)
- Addressing of records takes place using the TID concept
Oracle

Database

Physical Database structures

Logical Database structures

Tablespace J
Tablespace K
Tablespace I

Disk B
Disk A
Block

Data File 1
Data File 2
Data File 3
Data File n

Table R
Index R1
Index R2
Table S
Index S1

Daten Seg. r
Index Seg. i1
Index Seg. i2
Daten Seg. s
Index Seg. s1

Basis
Extent
Extent 1
Extent 2

Block 1
Block 2
Block 3

Record 032
Record 517
Record 389
Record 782
Oracle: Blocks

DB block

Head

Free area

Data area

Block information

Table directory

Record directory

fixed = 24 Byte

fixed = 4 Byte for non-clustered tables

variable – grows with amount of tuples
Oracle: Records

Head | Column 1 | Column 2 | Column 3
--- | --- | --- | ---
Number of columns | Cluster key | Chain address | Column length | Data
optional | optional | optional |
Oracle: Data Organisation

- Standard index is structured as $B^+$-Tree
- Index-organized tables store tuples directly in the leaves of $B^+$-Tree
- Clustering of several relations possible; cluster indices can be organized as $B^+$-Tree or hash index
- Bitmap indices store bit matrices for enumeration attributes (see Data Warehouses)
- Reverse indices interpret bytes of primary key in reverse order ⇒ take off of storage in sorting order
Informix

- **dbspace**: (several Chunks – raw file) und **blobspace**
- Chunks - Extents - Page (Timestamp, TID field)
- Pages within an extent (guaranteed clustered)
  - *Bitmap Page*: Directory of all pages in extent
  - *Data Page*: Real data from tables
  - *Remainder Page*: Spanned records with overflow pages
  - *Index Page*: Index data
  - *Blob Page*: BLOBs with buffer and log mechanisms (in original record reference to start page of list of BLOB pages)
  - *Free Page*: Free pages in this extent

- **tablespace**: Several Extents
Informix: Data Definition

- **Index structure:** B⁺-Tree

```sql
create [ unique | distinct ] [ cluster ]
  index indexname on relname ( attrname [ asc | desc ], attrname [ asc | desc ], ...)
  [ using indexart,]
  [ fillfactor = percent ];
```

- **cluster** option: clustered index (only for new creation this characteristic is ensured)
- **using:** ""B-tree"" for B⁺-Tree, ""R-tree"
- **fillfactor:** initial 90%
- Methods can be indexed instead of attributes
DB2

Secondary index: $B^+\text{-Tree}$

- Clustered and densely populated \textit{clustered}
- Non-clustered and densely populated \textit{non-clustered}
DB2 (II)

tablespace

- Set of container
- Container: OS directory, OS file, secondary storage medium
- tablespace: For several tables (physical clustering)
- Several tablespaces per table: (index files, BLOB files, . . . )
- Granularity for allocation: Extents
- OS-managed (SMS, System Managed Space)
- DB2-managed (DMS, Database Managed Space)
DB2: bufferpool

tablescape in main memory is per Default a bufferpool

- Changes of size of bufferpools
- Creation of several bufferpools for one tablescape
- prefetchsize: Number of pages to be loaded in case of page fault
DB2: \textit{reorg} 

- Records of a table are stored in interrelated parts of the storage again.
- Redetermine sorting of a file.
- Eliminate overflow pages after increase of records (two-level TID concept).
5. Specific Access Structures

Access structures for specific applications

- Object-oriented databases
- Geometric access structures
- Access techniques for multimedia and text data
Object-Oriented Databases

Structure of an OODBMS

Application

Query Manager

Object Manager

Object Buffer

Storage Manager

Buffer Management

Page Buffer

Operating System

OODBMS
Object Identity

- **Variants**
  - *Representation of object identity by surrogates and their implementation through indirect references*: (GemStone, ORION/ITASCA, Postgres)
  - *Representation and implementation of object identity through direct references*: (ONTOS, ObjectStore)

- First technique is (logical) better, but more slowly (because of indirections)
Object Identity (II)

- Surrogates:
  - Always *unique*, even after deletion of the corresponding object
  - In *distributed environment* unique as well (encoding of computer ID in surrogate)
  - *Abstract class*, in which the object is instantiated, can be made explicitly within the surrogate

- Length: 32 or 64 Bit
Classes

(without complex attribute values or component objects)

- **Binary storage**
  Object is stored together with one attribute respectively as binary relation

- **Object structure with integrated schema**
  Schema information integrated in storage structure of each object (e.g., in ORION/ITASCA)

- **Object structure with external schema**
Classes (II)

- **Object structure with integrated schema**

<table>
<thead>
<tr>
<th>Surrogate</th>
<th>Bytes</th>
<th>Num_Attribute</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>011001010</td>
<td>38</td>
<td>3</td>
<td>workplace</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Werte–Offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>salary</td>
<td>supervisor</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

- **Object structure with external schema**

<table>
<thead>
<tr>
<th>Surrogate</th>
<th>Bytes</th>
<th>Attribute Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>011001010</td>
<td>10</td>
<td>comp. science</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3050</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10011011</td>
</tr>
</tbody>
</table>
Complex Attributes

- **Principle**
  - Object bigger than page ⇒ several pages in form of a B-Tree (e.g., EXODUS)
  - Set of objects of one class are clustered

- **Complex attributes**
  - *Divided storage* (normalized like in RDBS)
    Iris, OSCAR
  - Entire object structure is stored into one *Cluster*
    DASDBS

- Private *component objects*: Cluster possible

- Shared component object: Reference to (original) component object
Cluster

- Cluster definition at time of
  - *Class definition (in the schema)*: \( \text{O}_2 \)
  - *Object instantiation (per Object)*: ObjectStore, ObServer, ONTOS and GemStone

- Cluster structures used for
  - all objects of a class
  - certain parts of classes, e.g., a partition of classes, based on certain attribute values
  - all instances of a class, which belong to a specified part of the class hierarchy
  - composed object (object with component objects)
  - complex attribute values
Class Hierarchies

- Object exactly in one class:
  - State of the object is stored in this class (*Home Class Model*) (done in OODBPLs and some new developments, e.g., ORION)
Class Hierarchies (II)

- Object in multiple classes
  - Object in smallest class (regarding the class hierarchy), together with inherited attribute values (*Leaf Overlap Model*) (in ORDBMS like Illustra)
  - Object in every class, together with local attribute values (*Split Instance Model*) (OpenODB)
  - Object in every class, together with attribute values defined there as well as inherited attribute values (*Repeat Class Model*) (deep extension, direct storage, e.g., UniSQL)
  - All objects in one file, non-applicable attributes are set to **null** (*Universal Class Model*)
  - All Objects in a ternary file with surrogate, attribute and attribute value (*Value Triple Model*)
Access Paths for Classes

- Basic file organisation form already determined by storage structure of class
- Can be supported additionally by hash functions or B-Trees
- Support of access on objects in class and component hierarchies
- RDBMS: Access path supports only one relation
- OODBMS: Support of set of classes by access path
Access Paths for Class Hierarchies

- Index for hierarchy of classes on attribute of one (super)class $C$; reference to
  - all occurrences of suitable objects in class hierarchy with root $C$, if objects are stored using the split instance method
  - occurrence of object in class hierarchy in all other cases (i.e., methods), whereas the object can be stored in one of the subclasses of $C$ as well

- *Class hierarchy index* (ORION/ITASCA, O$_2$)

- Example: Persons, employees and students: Index on Name of student
Component Hierarchies

- Path expressions have to be supported; attribute values given for a (even indirect) component class
  
  \[ \text{Book.Publisher.PublisherLocation} \]
  
  (Access on book over location of publisher)

  \[ \text{Book.Publisher.Lector.Name} \]
  
  (Access on book over name of lector of the publisher)
## Component Hierarchies (II)

![Diagram of component hierarchies]

<table>
<thead>
<tr>
<th>Book</th>
<th>Publ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>α₁</td>
<td>β₁</td>
</tr>
<tr>
<td>α₂</td>
<td>β₁</td>
</tr>
<tr>
<td>α₃</td>
<td>β₂</td>
</tr>
<tr>
<td>α₄</td>
<td>β₂</td>
</tr>
<tr>
<td>α₅</td>
<td>β₂</td>
</tr>
<tr>
<td>α₆</td>
<td>β₃</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Publ.</th>
<th>Lector</th>
</tr>
</thead>
<tbody>
<tr>
<td>β₁</td>
<td>γ₁</td>
</tr>
<tr>
<td></td>
<td>γ₂</td>
</tr>
<tr>
<td></td>
<td>γ₃</td>
</tr>
<tr>
<td>β₂</td>
<td>γ₂</td>
</tr>
<tr>
<td></td>
<td>γ₃</td>
</tr>
<tr>
<td></td>
<td>γ₄</td>
</tr>
<tr>
<td>β₃</td>
<td>γ₄</td>
</tr>
<tr>
<td></td>
<td>γ₅</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lector</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ₁</td>
<td>Maria</td>
</tr>
<tr>
<td>γ₂</td>
<td>Peter</td>
</tr>
<tr>
<td>γ₃</td>
<td>Martin</td>
</tr>
<tr>
<td>γ₄</td>
<td>Sandra</td>
</tr>
<tr>
<td>γ₅</td>
<td>Evelyn</td>
</tr>
</tbody>
</table>
equality and identity index

- **Support of attributes, whose type is a standard data type (e.g., String), analog to classical access paths in relational systems**
  GemStone: *equality index* for sub-paths
  Publisher.PublisherLocation or Lector.Name

- **Support of component objects (i.e., object-value attributes)**
  GemStone: *identity index* for sub-paths Book.Publisher or Publisher.Lector
Types of Realization: Overview

- Path index
- Multi index
- Join index
- Nested index
- Access support relation

(Type 1 and 2 in ORION and GemStone)
Illustration: Index Graph

Legend:
- component relation
- index support
**Multi Index**

- *Multi index*: Binary index files, pointing from \( n \)-th component of path expression to \( n - 1 \)-th component

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Lector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Martin</td>
<td>{\gamma_3}</td>
</tr>
<tr>
<td>Eve</td>
<td>{\gamma_5}</td>
</tr>
<tr>
<td>Sandra</td>
<td>{\gamma_4}</td>
</tr>
<tr>
<td>Peter</td>
<td>{\gamma_2}</td>
</tr>
<tr>
<td>Maria</td>
<td>{\gamma_1}</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Lector</th>
<th>Publ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\gamma_1</td>
<td>{\beta_1}</td>
</tr>
<tr>
<td>\gamma_2</td>
<td>{\beta_1, \beta_2}</td>
</tr>
<tr>
<td>\gamma_3</td>
<td>{\beta_1, \beta_2}</td>
</tr>
<tr>
<td>\gamma_4</td>
<td>{\beta_2, \beta_3}</td>
</tr>
<tr>
<td>\gamma_5</td>
<td>{\beta_3}</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Publ.</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>\beta_1</td>
<td>{\alpha_1, \alpha_2}</td>
</tr>
<tr>
<td>\beta_2</td>
<td>{\alpha_3, \alpha_4, \alpha_5}</td>
</tr>
<tr>
<td>\beta_3</td>
<td>{\alpha_6}</td>
</tr>
</tbody>
</table>
```
Join Index

- **Join index**: symmetric multi index

Books → Publisher → Lector → Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Lector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>(\gamma_3)</td>
</tr>
<tr>
<td>Eve</td>
<td>(\gamma_5)</td>
</tr>
<tr>
<td>Sandra</td>
<td>(\gamma_4)</td>
</tr>
<tr>
<td>Martin</td>
<td>(\gamma_2)</td>
</tr>
<tr>
<td>Maria</td>
<td>(\gamma_1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lector</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\gamma_1)</td>
<td>Maria</td>
</tr>
<tr>
<td>(\gamma_2)</td>
<td>Martin</td>
</tr>
<tr>
<td>(\gamma_3)</td>
<td>Peter</td>
</tr>
<tr>
<td>(\gamma_4)</td>
<td>Sandra</td>
</tr>
<tr>
<td>(\gamma_5)</td>
<td>Eve</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lector</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\gamma_1)</td>
<td>{(\beta_1)}</td>
</tr>
<tr>
<td>(\gamma_2)</td>
<td>{(\beta_1, \beta_2)}</td>
</tr>
<tr>
<td>(\gamma_3)</td>
<td>{(\beta_1, \beta_2)}</td>
</tr>
<tr>
<td>(\gamma_4)</td>
<td>{(\beta_2, \beta_3)}</td>
</tr>
<tr>
<td>(\gamma_5)</td>
<td>{(\beta_3)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Lector</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta_1)</td>
<td>{(\gamma_1, \gamma_2, \gamma_3)}</td>
</tr>
<tr>
<td>(\beta_2)</td>
<td>{(\gamma_2, \gamma_3, \gamma_4)}</td>
</tr>
<tr>
<td>(\beta_3)</td>
<td>{(\gamma_4, \gamma_5)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta_1)</td>
<td>{(\alpha_1, \alpha_2)}</td>
</tr>
<tr>
<td>(\beta_2)</td>
<td>{(\alpha_3, \alpha_4, \alpha_5)}</td>
</tr>
<tr>
<td>(\beta_3)</td>
<td>{(\alpha_6)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Book</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha_1)</td>
<td>{(\beta_1)}</td>
</tr>
<tr>
<td>(\alpha_2)</td>
<td>{(\beta_1)}</td>
</tr>
<tr>
<td>(\alpha_3)</td>
<td>{(\beta_2)}</td>
</tr>
<tr>
<td>(\alpha_4)</td>
<td>{(\beta_2)}</td>
</tr>
<tr>
<td>(\alpha_5)</td>
<td>{(\beta_2)}</td>
</tr>
<tr>
<td>(\alpha_6)</td>
<td>{(\beta_3)}</td>
</tr>
</tbody>
</table>
Nested Index

- *Nested index*: One single index file for $n$-th and first component of path expression

![Diagram](attachment:diagram.png)

<table>
<thead>
<tr>
<th>Name</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>Martin</td>
<td>${\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5}$</td>
</tr>
<tr>
<td>Eve</td>
<td>${\alpha_6}$</td>
</tr>
<tr>
<td>Sandra</td>
<td>${\alpha_3, \alpha_4, \alpha_5, \alpha_6}$</td>
</tr>
<tr>
<td>Peter</td>
<td>${\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5}$</td>
</tr>
<tr>
<td>Maria</td>
<td>${\alpha_1, \alpha_2}$</td>
</tr>
</tbody>
</table>
Path Index

- **Path index**: Generalized nested index

```
Name | Lector  | Name | Publisher | Name | Book
-----|---------|------|-----------|------|------
Martin| {γ3}    | Martin| {β1, β2}  | Martin| {α1, α2, α3, α4, α5}
Eve   | {γ5}    | Eve  | {β3}      |       | {α6} |
Sandra| {γ4}    | Sandra| {β2, β3}  |       | {α3, α4, α5, α6} |
Peter  | {γ2}    | Peter| {β1, β2}  |       | {α1, α2, α3, α4, α5} |
Maria | {γ1}    | Maria| {β1}      |       | {α1, α2} |
```

Gunter Saake

Database Implementation Techniques

Last Changes: 21.4.2015
Access Support Relation (ASR)

- Generalization of all previous access paths, e.g., generalized (compact) path index

<table>
<thead>
<tr>
<th>Name</th>
<th>Lector</th>
<th>Publisher</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>Martin</td>
<td>γ₃</td>
<td>β₁, β₂</td>
<td>α₁, α₂, α₃, α₄, α₅</td>
</tr>
<tr>
<td>Eve</td>
<td>γ₅</td>
<td>β₃</td>
<td>α₆</td>
</tr>
<tr>
<td>Sandra</td>
<td>γ₄</td>
<td>β₂, β₃</td>
<td>α₃, α₄, α₅, α₆</td>
</tr>
<tr>
<td>Peter</td>
<td>γ₂</td>
<td>β₁, β₂</td>
<td>α₁, α₂, α₃, α₄, α₅</td>
</tr>
<tr>
<td>Maria</td>
<td>γ₁</td>
<td>β₁</td>
<td>α₁, α₂</td>
</tr>
</tbody>
</table>
Access Paths for Methods

- Results of method execution are stored in index
  - Non-parametrized method: one method result per object in index
  - Parametrized method: method result is stored per object and per possible parameter value in index; not efficient, hence, only certain ranges from value set in index or only parameter in index, already used for a query (adaptive, learning index)

- Materialization of method results: *Function-Materialization technique*
Object Buffer

- So far:
  - Application data are loaded from page buffer to main memory parts, available for application programs.
  - This "costs" one transformation from internal representation to the representation desired by the application program.

- In some systems, objects are loaded directly from disk to application storage: \( O_2 \), Objectivity, ONTOS, probably with certain address transformations (ObjectStore, see "Pointer Swizzling").

- Other OODBSs like GemStone, ORION/ITASCA, DASDBS and OSCAR have second buffer: *Object buffer*
Task: An object, which is present in main memory and accessed by an application program, should be found very fast with object buffer and logical object identities:

1. Object $\alpha$ in main memory possesses a component object $\beta$, which cannot be found in the object buffer

2. Object $\beta$ is sought-after in page buffer
   - Search through of a Resident Object Table (ROT)
   - Assumption: $\beta$ is not in page buffer
Pointer Swizzling (II)

- with object buffer and logical object identities (cont.):
  3. (Re)load $\beta$ from secondary storage
     - Search in *Persistent Object Table (POT)* to determine the secondary storage address for the given object identity
  4. After loading the object into the page or object buffer respectively:
     For every access (on object) in application program, the corresponding main memory address has to be searched using the logical object identity

$\leadsto$ to cumbersome and indirect
Pointer Swizzling (III)

- In the case that direct references for the implementation of the object identity are used
  - Direction is omitted
  - Nevertheless: Direct secondary storage address not useful in main memory $\Rightarrow$ has to be transformed for every access.

- If object buffer is omitted:
  - No transformation from page buffer needed
  - However, objects contained in the page buffer have to be searched using the current main memory address as well
Pointer Swizzling (IV)

- Hence: Transformation of indirect or direct (secondary storage) references to main memory addresses ➔ Pointer Swizzling

- Variants:
  - Original or copy:
    - Pointer transformation to original page (in page buffer) or to the copy (in object buffer)
  - Immediately or delayed:
    - Pointer of all objects are transformed while they are loaded or delayed with the first access on the object in main memory
Pointer Swizzling (V)

- **Variants (cont.):**
  - Direct or indirect:
    - Transformation to direct main memory address or "only" to a descriptor (indirect pointer), which contains main memory address

- **Systems**
  - ORION: immediate, indirect Swizzling
  - $O_2$: no Pointer Swizzling
  - Exodus: delayed strategy
  - ObjectStore: immediate, direct Swizzling (VMMA)
Geometric Access Structures

- Huge amount of geometric objects ($\geq 10^6$)
- Characteristics of geometric objects (geo objects)
  - Geometry (e.g., frequency polygon)
  - For support of queries: Additionally, a $d$-dimensional, describing rectangle (*bounding box*) can be used
  - Non-geometric attributes

- Application scenarios: Geo information systems (land register data, maps), CAX applications (e.g., VLSI design), ...

- Access is primarily realized over geo data: Search window (screen capture of data), access on adjacent objects
Typical Operations

- **Exact search**
  - Input: Exact geometric search data
  - Result: One, single object (maximum)

- **Range query for given, n-dimensional window**
  - Search window: $d$-dimensional rectangle (corresponds to a multidimensional interval)
  - Result: All geo objects, which subtend the search window
  - Result quantity depends on parameter (of search window)
Typical Operations (II)

- Insertion of geo objects
  - *Desireable without global reorganisation!*

- Deletion of geo objects
  - *Desireable without global reorganisation!*
Adjacency-preservative Search Trees

- Partition of geometric space in *regions*
- Adjacent objects should be assigned to *same* region, if possible
- If not applicable, distribute these objects on *adjacent* regions
- Tree structure is created by refinement of regions to adjacent partial regions
- Storage of objects in leaf regions
Adjacency-preservative Search Trees (II)

- Degrees of freedom
  - Shape of regions
  - Disjunctive (total) partitioning or overlapping of regions
  - Unique or multiple assignment of objects to regions
  - Storage and access using original or derived geometry for objects
  - Degree of tree & organisation form
Multi-Level Processing of geom. Queries

Query

Geo-Index

Candidates

Query Result

Filtering

load geo object

Query to geo object

no hit

hit

Refinement

Gunter Saake

Database Implementation Techniques

Last Changes: 21.4.2015
Geometric Tree Structure: BSP-Tree
## Realization Variants

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Tree Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BSP-Tree</td>
</tr>
<tr>
<td>Region shape</td>
<td>convex</td>
</tr>
<tr>
<td>Part. regions</td>
<td>complete</td>
</tr>
<tr>
<td>Overlapping</td>
<td>no</td>
</tr>
<tr>
<td>Balanced</td>
<td>no</td>
</tr>
</tbody>
</table>
R-Trees (I)

- **R-Tree**: Generalization of B-Tree principle to several dimensions
  - Root of tree is a rectangle, which encompasses all geo objects
  - Geo objects (itself) are represented by their enclosing rectangles
  - Partition in regions takes place using non-disjunctive rectangles
  - Each geo object is assigned unique to one leaf
R-Trees (II)

- Region partition through rectangles in the R-Tree
R-Trees (III)

Tree structure for R-Tree

A

B

C

D

E

F

G

H

I

J

K

L

M

N
Problems of R-Trees

- A certain rectangle can be overlapped by many regions, but it is stored in exactly one region (only)
- Even point queries can result into a lookup of many rectangle regions
- Inefficient for exact match (which is also needed for insert and delete operations!)
- Problems with insertion
  - Insert requires often an expansion of regions (propagated upwards)
Expansion in case of Insert

old

new

rectangle to be inserted
R$^+$-Trees (I)

- R$^+$-Tree: Partition in *disjunctive* regions
- Hence, each stored point of a geometric object is assigned a unique leaf
- On each tree level, at most one rectangle is assigned to a point as well $\rightarrow$ distinct path from root to leaf (storing an point)
- ‘Clipping’ of geo objects necessary!
R^+-Trees (II)
R\(^+\)-Trees (III)
Problems of $R^+\text{-Trees}$

- Objects have to be stored in several rectangle regions (*clipping*) — increased storage & modification effort
- Insert of objects possibly requires modification of several rectangle regions

![Diagram of R+ Trees](image-url)
Problems of $R^+$-Trees (II)

- Insert can lead to *unavoidable* partition of regions in certain situations

- Region modifications have consequences in both directions, to the leaves as well as to the root

- Upper bound for entries in leaf nodes can not be guaranteed anymore
Cell-Trees
Point Data Structures

- Storage of rectangle using point data structures (e.g., grid file)
  - Transformation of spacious objects (multidimensional rectangles) to point data
  - Transformation maps $d$-dimensional rectangles to points in $2d$-dimensional space $\mathcal{R}^{2d}$
  - $d$-dimensional rectangle (notation):

$$r = [l_1, r_1] \times \cdots \times [l_d, r_d]$$
Point Data Structures (II)

- Corner transformation

\[ p_r = (l_1, r_1, \ldots, l_d, r_d) \in \mathbb{R}^{2d} \]

per interval as coordinates: upper bound, lower bound

- Center transformation

\[ p_r = \left( \frac{l_1 + r_1}{2}, \frac{r_1 - l_1}{2}, \ldots, \frac{l_d + r_d}{2}, \frac{r_d - l_d}{2} \right) \in \mathbb{R}^{2d} \]

per interval as coordinates: center point, half width
Corner Transformation
Center Transformation

Example

interval center
width
interval
half
y
x

Gunter Saake
Database Implementation Techniques
Last Changes: 21.4.2015
5–348
Search Window

![Search Window Diagram]
Grid File Degeneration
Grid File Extensions: LSD-Trees

Possible partition of one layer/plane for a LSD-Tree
Example for LSD-Trees
Access Structures for Multimedia DB

- Media objects $\rightarrow$ huge *binary objects*

<table>
<thead>
<tr>
<th>Format</th>
<th>Megabyte</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG</td>
<td>75</td>
<td>uncompressed</td>
</tr>
<tr>
<td>MPEG-1</td>
<td>12.5</td>
<td>compressed, loss of quality</td>
</tr>
<tr>
<td>MPEG-2</td>
<td>17</td>
<td>compressed, high quality</td>
</tr>
</tbody>
</table>

1 minute of combined Audio/Video recording

- Indexing on derived attributes (so-called *Features*)
  $\rightarrow$ Indexing on highdimensional *feature vectors*

- *Continuous data types*
Continuous Data Types

- Data types for storage of *continuous media*
  - Data have to be loaded *fast enough* (in real-time) into main memory for playing.
  - Since bandwidth for (data) transfer from secondary storage to main memory is not constant, data are usually loaded *ahead into main memory cache* (*prefetching strategy*)
  - Caution: Avoiding *cache overflow*
  - Especially for continuous media: *Storage on tertiary storage medium* is common → two-level caching strategy necessary
Highdimensional Indices

- Feature vectors for characterizing media objects
- Feature vector: Point in highdimensional space
- Database as set of points of the $d$-dimensional space:

\[ DB = \{ P_0, \ldots, P_{n-1} \} \]

- Every point of this database is a $d$-dimensional vector:

\[ P_i \in DB \subseteq \mathbb{R}^d \]
Typical Operations on Feature Vectors

- **Range queries** compute all adjacent points in database for a given vector:

\[
\text{range}(DB, Q, r, M) = \{ P \in DB \mid \delta_M(P, Q) \leq r \}
\]

with:

- **DB**: Database as set of feature vectors (search space)
- **r**: Distance determining the range for searching
- **Q**: Search vector
- **M**: Metric (e.g., euclidean distance)
Typical Operations on Feature Vectors (II)

- **Point query** defines the exact search and thus, corresponds to range query with distance 0:

\[
\text{point}(DB, Q, M) = \{ P \in DB \mid \delta_M(P, Q) = 0 \}
\]

- **Nearest neighbor query** determines for a given (search) vector the nearest vector in the database:

\[
\text{nearest-neighbor}(DB, Q, M) = \\
\{ P \in DB \mid \forall (P' \in DB) : \delta_M(P, Q) \leq \delta_M(P', Q) \}
\]
Distance Function $\delta$

1. $\delta(p, p) = 0$
2. $\delta(p_1, p_2) = \delta(p_2, p_1)$
3. $\delta(p_1, p_2) + \delta(p_2, p_3) \geq \delta(p_1, p_3)$ (triangle inequality)
Metrics

- **Euclidean distance**

  \[ \delta_{Euclid}(P, Q) = \sqrt{\sum_{i=1}^{d} (Q_i - P_i)^2} \]

  \(Q_i\) is value of \(Q\) for the \(i\)-th dimension

- **Manhattan distance**

  \[ \delta_{Manhattan}(P, Q) = \sum_{i=1}^{d} |Q_i - P_i| \]

- **Maximum distance**

  \[ \delta_{Max}(P, Q) = \max\{|Q_i - P_i|\} \]
Comparison of Metrics

Euclid

Manhattan

Max
Nearest Neighbor Search in R-Trees
MinDist and MinMaxDist

- **MinDist**: *Min-distance* is minimal distance between query point and MBR

- **MinMaxDist**: for every hyperplane \((n - 1)\) dimensions) the point, most far away from query point is determined (most disadvantageous case)
  
  ▶ Min-Max point is then the minimal distant point of all of these max-points and its distance to query point is min-max distance

\[
\text{MinDist} \leq \text{MinMaxDist}
\]
MinDist and MinMaxDist (II)

R1

R2

MinDist

MinMaxDist

Gunter Saake

Database Implementation Techniques Databases II

Last Changes: 21.4.2015 5–363
Nearest Neighbor

- Currently assumed minimal distance $\text{Dist}$
  - $\text{Dist} < \text{MinDist} \rightarrow$ Child nodes must not be sought, since they can not contain any point, which can be considered as hit
  - $\text{MinDist} \leq \text{Dist} \leq \text{MinMaxDist} \rightarrow$ Child nodes have to be sought, since a candidate could be contained
  - $\text{Dist} > \text{MinMaxDist} \rightarrow$ the considered MBR or a child node contain a candidate, which is closer than the point currently assumed to be the nearest one
  - Detection of closer candidate $\rightarrow$ Update of $\text{Dist}$
X-Tree as an Example

- Performance problems of R-Tree structures with high number of dimensions:
  - Usually, large overlapping results from splitting of inner nodes, caused by selecting wrong dimensions (for the split)
  - For large overlapping, linear search is faster than block-oriented access on tree node
X-Tree as an Example (II)

- X-Tree: Modifications of R-Trees
  - Constant factor \texttt{max\_overlap} determines a maximum degree of overlapping
  - For MBR, a \textit{split history} containing dimensions for previous splits is maintained
  - \textit{Super nodes} have the size of several normal, inner tree nodes and hence, can contain many entries
X-Tree as an Example (III)

- Types of nodes:
  - Simple inner nodes: same structure as in R-Tree, but additionally split histories are stored
  - Inner *super nodes* span several data blocks and hence, reference to a higher number of entries (data)
  - Leaf nodes correspond to the leaves of the R-Tree
X-Tree graphical
X-Tree - number of Dimensions

d=4

d=8

d=32
Storage of Images
Quadtrees
Quadtrees (II)
Video Data

Video segments:

- Object E
- Object D
- Object C
- Object B
- Object A
Video Data: Segment-Tree
Information Retrieval

- *Information Retrieval (IR):*
  - Technique for storage and retrieval of full text documents

- Examples:

  - `[‘Databases’] in Text`
  - `[‘Databases’ and ‘Multimedia’] in Text`
  - `[‘Object’ 1 word prior to ‘Orientation’] in Text`
  - `[‘Object’ in same sentence as ‘Orientation’] in Text`
  - `[‘Object’ inside 2 sections with ‘Orientation’] in Text`
Quality Measures for IR

\[
\text{Recall} = \frac{\text{number of located, relevant docs}}{\text{total number of relevant docs}}
\]

\[
\text{Precision} = \frac{\text{number of located, relevant docs}}{\text{total number of located docs}}
\]

\[
\text{Fallout} = \frac{\text{number of located, irrelevant docs}}{\text{total number of irrelevant docs}}
\]
Concept Indexing

- **Linguistic analysis:**
  - *morphologic analysis* generates a word stem by elimination of pre-/suffixes and plural endings → *Stemming*
  - Deriving new word forms from already known word forms → especially for German!

- **Synonyms:** binary relation between equivalent words

- **Term hierarchy** in (subject-specific) *thesaurus*:
  Inheritance of attributes → Queries to more general/specific terms possible
Inverted Lists

- Indexed words (strings) form a lexicographical sorted list.
- Particular entry consists of a *word* and a list of document identifier which references to documents where the word occurs.
- Additionally, further information for the word-document combination can be stored:
  - Position of (first occurrence of) word in text
  - Frequency of word in text
Inverted Lists

- Hetze → (d01, 1)
- Heu → (d01, 2)
- Heuer → (d13, 2)
- Heureka → (d13, 2)
- Heurose → (d01, 1)
- Heurost → (d01, 2)
- (d42, 2) → (d73, 1)
6. Basic Algorithms for DB Operations

- Database parameter
- Complexity of basic algorithms
- Unary operations (Scan, Selection, Projection)
- Binary operations: Set operations
- Computation of Joins
Classification

- **SOI** (Set-Oriented Interface)
- **ROI** (Record-Oriented Interface)
- **IRI** (Internal Record Interface)
- **SBI** (System Buffer Interface)
- **FI** (File Interface)
- **DI** (Device Interface)

Diagram:

```
  Data System
   ↑      ↓
   SOI    Set-Oriented Interface

  Access System
   ↑      ↓
   ROI    Record-Oriented Interface

  Storage System
   ↑      ↓
   IRI    Internal Record Interface

  Buffer Management
   ↑      ↓
   SBI    System Buffer Interface

  Operating System
   ↑      ↓
   FI     File Interface

  Operating System
   ↑      ↓
   DI     Device Interface
```
Database Parameters

- Consideration of complexity ($O(n^2)$)
- Cost estimation/evaluation (concrete)
- Database parameters as basis
- Stored in the data dictionary of the DBS
Database Parameters (II)

- $n_r$: Number of tuples in relation $r$
- $b_r$: Number of blocks (pages), containing tuples from $r$
- $s_r$: (average) size of tuples from $r$
- $f_r$: Blocking factor (tuples from $r$ per block)

$$f_r = \frac{bs}{s_r},$$

with $bs$ block size

- tuples of one relation are packed tightly in blocks:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$
Database Parameters (III)

- $V(A, r)$: Number of different values for attribute $A$ in relation $r$:
  \[ V(A, r) = |\pi_A(r)| \]
- $A$ is primary key: $V(A, r) = n_r$
- $SC(A, r)$: *selection cardinality*; average number of result tuples for $\sigma_{A=x}(r)$ with $x \in \pi_A(r)$
- Key attribute $A$: $SC(A, r) = 1$
- General:
  \[ SC(A, r) = \frac{n_r}{V(A, r)} \]

Further information: Branching factor of B-Tree indices, height of tree, number of leaf nodes
Complexity of Basic Algorithms

Assumptions

- Indices realized as $B^+$-Trees
- Dominant cost factor: Block access (on secondary storage)
- Access on secondary storage for intermediate relations as well
- Intermediate relation at first for each (basic) operation
- Intermediate relation (hopefully) to a large extent in the buffer
- Some operations (set operations) on set of addresses (TID lists)
Main Memory Algorithms

Important for throughput of the overall system, because they are frequently used

- **Comparison of tuples**
  (For duplicate detection, specify sorting order, ...) realized iterative by comparison of single attributes, firstly attributes with high selectivity

- **TID access**
  TID within main memory: Usual approach for resolving indirect addresses
Access on Records

- **Relations**: internal identifier \( \text{RelID} \)
- **Indices**: internal identifier \( \text{IndexID} \)
  - **Primary index**, e.g., \( I(\text{Persons(PANr)}) \)
    for \( A = a \) at most one tuple per access is provided
  - **Secondary index**, e.g., \( I(\text{Borrowing(PANr)}) \)
    Ex.: \( \text{PANr} = 4711 \) provides in general several tuples

Index access: Result usually as TID list(s)
Access on Records (II)

- **fetch-tuple**: direct access on tuple using TID value tuple is loaded to *tuple buffer*
  
  \[\text{fetch-tuple}(\text{RelID}, \text{TID}) \rightarrow \text{tuple buffer}\]

- **fetch-TID**: Determine TID for (primary key) attribute value
  
  \[\text{fetch-TID}(\text{IndexID}, \text{attribute value}) \rightarrow \text{TID}\]

- Further operation on relations and indices: *Scans*
Example in SQL

```
select *
from CUSTOMER
where CName = 'Brown'
```

- Query for equality on a key
- `put`: used here for displaying the result

```
currentTID := fetch-TID(CUSTOMER-CName-Index, 'Brown');
currentBuffer := fetch-tuple(CUSTOMER-RelationID, currentTID);
put(currentBuffer);
```
External Sorting Algorithms

External sorting by merging; Complexity: $O(n \log n)$ operations for ordering (during merge process)
Unary Operations

Scan traverses tuples of a relation

- *Full table scan* traverses all tuples of a relation in arbitrary order
  Costs: $b_r$

- *Index Scan* uses index to select tuples in sorting order
  Costs: Number of tuples plus height of index

Comparison

- Full table scan performs better by exploiting the blocking
- Index scan performs better in the case that only few data are needed (but worse for reading many tuples)
Operations on Scans

- Open full table scan
  \[ \text{open-rel-scan}(\text{RelationID}) \rightarrow \text{ScanID} \]
  returns \text{ScanID} that can be used for identification for the following operation

- Initialize index scan
  \[ \text{open-index-scan}(\text{IndexID}, \text{Min}, \text{Max}) \rightarrow \text{ScanID} \]
  returns \text{ScanID}; \text{Min} and \text{Max} determines range of range query

- \text{next-TID} provides next TID; scan cursor is increased

- \text{end-of-scan} returns \text{true}, if no TID is left for the scan

- \text{close-scan}
Example: Scan

```sql
select * 
from Persons 
where surname between 'Heuer' and 'Jagellowsk'
```
Example: Full table scan

currentScanID := open-rel-scan(People-RelationID);
currentTID := next-TID(currentScanID);
while not end-of-scan(currentScanID) do
begin
    currentBuffer :=
        fetch-tuple(People-RelationID, currentTID);
    if currentBuffer.Surname >= 'Heuer'
        and currentBuffer.Surname <= 'Jagellowsk'
    then put (currentBuffer);
    endif;
    currentTID := next-TID(currentScanID);
end;
close (currentScanID);
Example: Index Scan

currentScanID :=
   \texttt{open-index-scan}(\texttt{Persons-Surname-IndexID},
   \texttt{\textquoteleft Heuer\textquoteleft}, \texttt{\textquoteleft Jagellowsk\textquoteleft});
currentTID := \texttt{next-TID}(\texttt{currentScanID});
while not \texttt{end-of-scan}(\texttt{currentScanID}) do
begin
   currentBuffer :=
      \texttt{fetch-tuple}(\texttt{Persons-RelationID}, \texttt{currentTID});
   \texttt{put}(\texttt{currentBuffer});
   currentTID := \texttt{next-TID}(\texttt{currentScanID});
end;
\texttt{close}(\texttt{currentScanID});
Selection

- *exact search, range selections*, complex, composed selection criteria
- Composite predicate $\varphi$ composed from atomic predicates (exact search, range query) with **and**, **or**, **not**

Tuplewise approach

- Given $\sigma_\varphi(r)$
- Relation scan: For all $t \in r$ compute $\varphi(t)$
- Complexity $O(n_r)$, more precisely $b_r$
Selection: Conjunctive Normal Form

- Apply access path to complex predicates \( \Rightarrow \) analyze and transform (suitable) \( \varphi \)

- For instance, transform \( \varphi \) to conjunctive normal form CNF; consists of *conjunctives*

- Select conjunctive (heuristically), which can be good analyzed by an index (e.g., \( A = c \) in the case that index on \( A \) exists)

- Analyze selected conjunctive; for resulting TID list, all other parts of CNF are analyzed tuplewise

- Alternative: Analyze several suitable conjunctives and intersect the resulting TID lists
Selection: Filter Methods

- For filter method all conditions are set to true, which are not supported by an access method.
- Resulting predicate: $\varphi'$.
- $r' = \sigma_{\varphi'}(r)$ can now be analyzed using indices.
- $\sigma_{\varphi}(r')$ on the (hopefully much smaller) intermediate result $r'$ can now be analyzed using the tuplewise approach.
- Filter methods are only suitable, if $\varphi'$ actually reduces the data amount (attention with disjunctions).
Projection

- Relational algebra: with duplicate elimination
- SQL: no duplicate elimination, if not required with `distinct` (modified scan)
- With duplicate elimination:
  - Sorted output of an index is helpful for duplicate elimination
  - Projection on indexed attributes without access on stored tuples
Projection (II)

Projection $\pi_X(r)$:
1. $r$ have to be sorted by $X$
2. $t \in r$ are added to the result, in the case that $t(X) \neq \text{previous}(t(X))$

Time complexity: $O(n_r \log n_r)$

If $r$ is already sorted by $X$: $O(n_r)$

Key $K \subseteq X$: $O(n_r)$
Scan Semantics

- Scan-based (positional) change operations: Definition of scan semantics \(\leadsto\) Effectiveness on subsequent scan operations
- Example: Deletion of current record
- States: before first record, point to a record, between two records, behind last record, in empty set
- Furthermore: Transition rules for states
Scan Semantics (II)

Halloween problem (System R):

- SQL statement:

```sql
UPDATE employee e
SET salary = salary * 1.05
```

- Record-oriented analysis using index scan on \texttt{I}_{\text{employee}}(\texttt{salary})
and immediate index update

- Without specific precautions: infinite number of salary increases
Binary Operations: Set Operations

Binary operations mostly realized on the basis of tuplewise comparison of particular sets of tuples

- Nested-Loops technique or loop iteration
  - The inner relation \( r \) is completely traversed for every tuple of an outer relation \( s \)
  - Complexity: \( O(n_s \times n_r) \)

- Merge technique or merge method
  - \( r \) and \( s \) (sorted) are traversed stepwise in the given tuple order
  - Complexity: \( O(n_s + n_r) \)
  - If sorting has to be carried out: Sort-Merge technique
  - Complexity: \( n_r \log n_r \) and/or \( n_s \log n_s \)
Set Operations (II)

- **Hash methods**
  - Smaller relation in hash table
  - Tuples of second relation are assigned to comparison fellow using a hash function
  - Ideally complexity $O(n_s + n_r)$
Classes of Binary Operations

r

A
B
C
S
## Classes of Binary Operations (II)

<table>
<thead>
<tr>
<th>Result extensions</th>
<th>Compliance for all attributes</th>
<th>Compliance for some attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>Difference $r - s$</td>
<td>Anti-Semi Join</td>
</tr>
<tr>
<td>$B$</td>
<td>Intersection $r \cap s$</td>
<td>Join, Semi Join</td>
</tr>
<tr>
<td>$C$</td>
<td>Difference $s - r$</td>
<td>Anti-Semi Join</td>
</tr>
<tr>
<td>$A \cup B$</td>
<td></td>
<td>Left Outer Join</td>
</tr>
<tr>
<td>$A \cup C$</td>
<td>Symmetric Difference $(r - s) \cup (s - r)$</td>
<td>Anti Join</td>
</tr>
<tr>
<td>$B \cup C$</td>
<td></td>
<td>Right Outer Join</td>
</tr>
<tr>
<td>$A \cup B \cup C$</td>
<td>Union $r \cup s$</td>
<td>Full Outer Join</td>
</tr>
</tbody>
</table>
Union with Duplicate Elimination

Union by insertion

- Variant of Nested-Loops techniques
- Create copy of one relation \( r_2 \) with name \( r'_2 \), subsequently insert tuples \( t_1 \in r_1 \) in \( r'_2 \) (time complexity depends on organisation form of copy)

Specific techniques for the union

- Concatenate \( r \) and \( s \)
- Projection on all attributes of the concatenated relation

Time complexity: \( O((n_r + n_s) \times \log(n_r + n_s)) \) (like projection)
Union (II)

Union through merge techniques (merge-union)

1. Sorting of \( r \) and \( s \) (if unsorted)

2. Merging of \( r \) and \( s \)
   - \( t_r \in r \) smaller than \( t_s \in s \): add \( t_r \) to the result, read next \( t_r \in r \)
   - \( t_r \in r \) bigger than \( t_s \in s \): add \( t_s \) to the result, read next \( t_s \in s \)
   - \( t_s = t_r \): add \( t_r \) to the result, read next \( t_r \in r \) and \( t_s \in s \) respectively

- Time complexity: \( O(n_r \times \log n_r + n_s \times \log n_s) \) with sorting, \( O(n_r + n_s) \) without sorting
Computation of Joins

Variants

- Nested-Loops Join
- Block-Nested-Loops Join
- Merge Join
- Hash Join
- ...
Nested-Loops Join

Double loop iterates over all \( t_1 \in r \) and all \( t_2 \in s \) for an operation \( r \Join s \):

\[
\begin{align*}
\text{for each } & t_r \in r \text{ do} \\
& \text{begin} \\
& \quad \text{for each } t_s \in s \text{ do} \\
& \quad \quad \text{begin} \\
& \quad \quad \quad \text{if } \varphi(t_r, t_s) \text{ then put}(t_r \cdot t_s) \text{ endif} \\
& \quad \quad \text{end} \\
& \quad \text{end} \\
& \text{end}
\end{align*}
\]

\( r \Join \varphi s : \)
Algorithm

Nested-Loops Join with Scan

\[ \text{R1ScanID} := \text{open-rel-scan}(R1ID); \]
\[ \text{R1TID} := \text{next-TID}(\text{R1ScanID}); \]
\[ \text{while not end-of-scan}(\text{R1ScanID}) \text{ do} \]
\[ \begin{align*}
    & \text{R1Buffer} := \text{fetch-tuple}(R1ID, R1TID); \\
    & \text{R2ScanID} := \text{open-rel-scan}(R2ID); \\
    & \text{R2TID} := \text{next-TID}(\text{R2ScanID}); \\
    & \text{while not end-of-scan}(\text{R2ScanID}) \text{ do} \]
\[ \begin{align*}
    & \ldots/* \text{ Scan on inner relation */} \\
    & \text{end;} \\
    & \text{close} (\text{R2ScanID}); \\
    & \text{R1TID} := \text{next-TID}(\text{R1ScanID}); \\
\end{align*} \]
\[ \text{end;} \]
\[ \text{close} (\text{R1ScanID}); \]
Nested-Loops Join with Scan (II)

/* Scan on inner relation */
R2Buffer := fetch-tuple(R2ID,R2TID);
if R1Buffer.X = R2Buffer.Y
then insert into RES
     R2.Buffer.B1, ..., R1.Buffer.Bm);
endif;
R2TID := next-TID(R2ScanID);

Improvement: Nested-Loops Join connects all \( t_1 \in r \) with the result
\[ \sigma_{X=t_1(X)}(s) \] (good for index on \( X \) in \( r_2 \))
Block-Nested-Loops Join

Iteration takes place on blocks instead of tuples

```
for each Block \( B_r \) of \( r \) do
begin
  for each Block \( B_s \) of \( s \) do
  begin
    for each Tuple \( t_r \in B_r \) do
    begin
      for each Tuple \( t_s \in B_s \) do
      begin
        if \( \varphi(t_r, t_s) \) then put \( t_r \cdot t_s \) endif
      end
    end
  end
end
```

Costs: \( b_r \cdot b_s \)
Merge Techniques

\[ X := R \cap S; \] if unsorted, first of all sorting of \( r \) and \( s \) according to \( X \)

1. \( t_r(X) < t_s(X) \), read next \( t_r \in r \)
2. \( t_r(X) > t_s(X) \), read next \( t_s \in s \)
3. \( t_r(X) = t_s(X) \), connect \( t_r \) with \( t_s \) and all successors of \( t_s \), which are identical with \( t_s \) according to \( X \)
4. For the first \( t'_s \in s \) with \( t'_s(X) \neq t_s(X) \) starting with original \( t_s \) the same procedure is repeated with the successors of \( t'_r \) von \( t_r \), as long as \( t_r(X) = t'_r(X) \)
Merge Techniques: Complexity

- All tuples with same $X$-value: $O(n_r \times n_s)$
- $X$ is key of $R$ or $S$: $O(n_r \log n_r + n_s \log n_s)$
- In the case of (pre)sorted relations even: $O(n_r + n_s)$
Merge Join with Scan

- Join attributes of both relations have key characteristic
- $\text{min}(X)$ and $\text{max}(X)$: minimal and maximal stored value for $X$ respectively
Merge Join with Scan (II)

R1ScanID := open-index-scan(R1XIndexID, min(X), max(X));
R1TID := next-TID(R1ScanID);
R1Buffer := fetch-tuple(R1ID,R1TID);
R2ScanID := open-index-scan(R2YIndexID, min(Y), max(Y));
R2TID := next-TID(R2ScanID);
R2Buffer := fetch-tuple(R2ID,R2TID);
while not end-of-scan(R1ScanID) and not end-of-scan(R2ScanID) do
  begin
    .../* merge */
  end;
close (R1ScanID);
close (R2ScanID);
Algorithm

Merge Join with Scan (III)

/* merge */

if R1Buffer.X < R2Buffer.Y
    then R1TID := next-TID(R1ScanID);
        R1Buffer := fetch-tuple(R1ID,R1TID);
    else if R1Buffer.X > R2Buffer.y
        then R2TID := next-TID(R2ScanID);
            R2Buffer := fetch-tuple(R2ID,R2TID);
        else insert into RES
            R2.Buffer.B1, ..., R1.Buffer.Bm);
            R1TID := next-TID(R1ScanID);
            R1Buffer := fetch-tuple(R1ID,R1TID);
            R2TID := next-TID(R2ScanID);
            R2Buffer := fetch-tuple(R2ID,R2TID);
        endif
    endif
endif
Join by Hashing

Idea:

- Exploiting available main memory for minimizing the accesses on secondary storage
- Locating join fellows by hashing
- Querys such as \( r \bowtie_{r.A=s.B} s \)
Classical Hashing

- Preparation: Smaller relation becomes $r$
- Procedure
  1. Tuples of $r$ are read into main memory using scan and inserted into hash table $H$ using hash function $h(r.A)$
  2. If $H$ is full (or $r$ is read completely):
     Scan on $S$ and lookup for join fellow with $h(s.B)$
  3. If scan on $r$ is not finished:
     $H$ is restructured and a scan is executed on $S$ again
- Complexity: $O(b_r + p \times b_s)$ with $p$ as number of scans on $s$
Partitioning using Hash Function

- Tuples from $r$ and $s$ over $X$ are "hashed" in common file with $k$ blocks (buckets)
- Tuples in same bucket are connected using a join algorithm
Partitioning using Hash Function (II)
Algorithm

Partitioning using Hash Function (III)

for each $t_r$ in $r$ do
  begin
    $i := h(t_r(X))$;
    $H^r_i := H^r_i \cup t_r(X)$;
  end;
for each $t_s$ in $s$ do
  begin
    $i := h(t_s(X))$;
    $H^s_i := H^s_i \cup t_s(X)$;
  end;
for each $k$ in $0...\max$ do
  $H^r_k \bowtie H^s_k$;
Comparison of Techniques

Nested−Loops Join

Merge Join

Hash Join
Aggregation & Grouping

Queries:

```
select A, count(*)
from T
group by A
```

Algebra operator: \( \gamma_{\text{count}(\ast), A(r(t))} \)

Implementation variants:
- Nested Loops
- Sorting
- Hashing
7. Query Optimization

- Basic principles, motivating examples
- Phases of query processing
- Translation of SQL into relational algebra
- Logical optimization (algebraic, tableau)
- Internal optimization
- Cost-based selection
Classification

Set-Oriented Interface (SOI)

Record-Oriented Interface (ROI)

Internal Record Interface (IRI)

System Buffer Interface (SBI)

File Interface (FI)

Device Interface (DI)

Operating System

Buffer Management

Storage System

Access System

Data System
Basic Principles

Basic languages
- SQL
- Relational calculi
  - here: Relational algebra

Objective(s) of optimization
- Query processing as fast as possible ⇒
- as few as possible page accesses during query processing ⇒
- For all operations, only a minimum number of pages (tuples) should be considered
Partial Objectives of Optimization

1. Selections as early as possible
2. Encompass basic operations and realize them without intermediate storage
3. Remove redundant operations, idempotencies or empty intermediate relations
4. Merging of identical (partial) expressions: Reuse of intermediate results
Example

CUSTOMER { CName, Cadr, Kto }
ORDER { CName, Product, Amount }

select CUSTOMER.CName, Kto
from CUSTOMER, ORDER
where CUSTOMER.CName = ORDER.CName
    and Product = 'Coffee'

- Relation CUSTOMER: 100 tuples; per page: 5 tuples
- Relation ORDER: 10,000 tuples; per page: 10 tuples
- 50 entries in ORDER table concern Coffee
- Considering tuples (CName, Kto): 50 per page
- 3 lines of Customer × ORDER fits into one page
- Buffer of size 1 for each relation, no spanned records
Direct Computation

1. $R_1 := \text{CUSTOMER} \times \text{ORDER}$
   
   Page accesses:
   
   ▶ $r : (100/5 \times 10.000/10) = 20.000$
   
   ▶ $w : (100 \times 10.000)/3 \approx 333.000$ (ca.)

2. $R_2 := \sigma_{\text{SEL}}(R_1)$
   
   ▶ $r : 333.000$ (ca.)
   
   ▶ $w : 50/3 \approx 17$ (ca.)

3. $RES := \pi_{\text{PROJ}}(R_2)$
   
   ▶ $r : 17$
   
   ▶ $w : 1$

Overall approx. 687.000 page accesses and approx. 333.000 pages for intermediate storage
Optimized Computation

1. \( R_1 := \sigma_{\text{Product} = 'Coffee'}(\text{ORDER}) \)
   - \( r : \frac{10.000}{10} = 1.000 \)
   - \( w : \frac{50}{10} = 5 \)

2. \( R_2 := \text{CUSTOMER} \bowtie_{\text{CName} = \text{CName}} R_1 \)
   - \( r : \frac{100}{5} \times 5 = 100 \)
   - \( w : \frac{50}{3} = 17 \)

3. \( RES := \pi_{\text{PROJ}}(R_2) \)
   - \( r : 17 \)
   - \( w : 1 \)

approx. 1.140 page accesses (improved by factor 500)
Computation with Usage of Index

Indices \( I(\text{ORDER(Product)}) \) and \( I(\text{CUSTOMER(CName)}) \)

1. \( R_1 := \sigma_{\text{Product} = 'Coffee'}(\text{ORDER}) \) on \( I(\text{ORDER(Product)}) \)
   - \( r \): minimal 5, maximal 50; \( w \): \( 50/10 = 5 \)

2. \( R_2 := \) sorting of \( R_1 \) by \( \text{CName} \)
   - \( r + w \): \( 5 \times \log 5 = 15 \) (ca.)

3. \( R_3 := \text{CUSTOMER} \bowtie_{\text{CName}=\text{CName}} R_2 \)
   - \( r \): \( 100/5 + 5 = 25 \); \( w \): \( 50/3 = 17 \)

4. \( RES := \pi_{\text{PROJ}}(R_3) \)
   - \( r \): 17; \( w \): 1

maximal approx. 130 and minimal approx. 85 page accesses
## Comparison of Variants

<table>
<thead>
<tr>
<th>Variant</th>
<th>Read/Write accesses</th>
<th>Accesses</th>
<th>Pages for intermediate results</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct computation</td>
<td>ca. 687.000</td>
<td></td>
<td>ca. 333.000</td>
</tr>
<tr>
<td>optimized computation</td>
<td>ca. 1.140</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>computation with index</td>
<td>min. 85</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>with Pipelining</td>
<td>max. 130</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>51 to 96</td>
<td></td>
<td>5 (plus sorting)</td>
</tr>
</tbody>
</table>
Phases of Query Processing

1. *Translation and view expansion*
   - Simplification of arithmetic expressions in query plan
   - Resolve sub queries
   - Usage of view definition

2. *Logical (aka algebraic) optimization*
   - Convert query plan independent of concrete organisation form; e.g., move Selection into other operations
Phases of Query Processing II

3 Internal optimization
   ▶ Considering concrete storage techniques (Indices, Cluster)
   ▶ Selection of algorithms
   ▶ Several alternative internal (execution) plans

4 Cost-based selection
   ▶ Statistic information (size of tables, selectivity of attributes) are used for the selection of a concrete internal plan

5 Code generation
   ▶ Compilation of access plan in executable code
Procedure and Languages

SQL

Translation

View Resolving

Optimization

Code Generation

Execution

Algebra

Access Plan

Code

logical optimization

physical optimization

cost-based selection
Phases of Optimization

Optimization

- logical optimization
- physical optimization
- cost-based selection

- algebra term
- several execution plans

execution plan
Translation into Relational Algebra

\[
\pi_{A_1, \ldots, A_m}(\sigma_F(r(R_1) \times r(R_2) \times r(R_3) \times \cdots \times r(R_n)))
\]

Realization in relational algebra:

\[
\text{select } A_1, \ldots, A_m \text{ from } R_1, R_2, \ldots, R_n \text{ where } F
\]

Improve query tree (following slide) according to:

- Detection of Joins instead of cartesian product
- Resolving sub queries (not exists-queries to difference)
- SQL constructs without counterpart in relational algebra: group by, order by, arithmetic, multiset semantic
Realization of SFW Block

\[
\sigma_{F} \left( \pi_{A_{0}, \ldots, A_{m}} \times \times r(R_{u}) \times r(R_{3}) \times r(R_{1}) \times r(R_{2}) \right) \]

Gunter Saake
Normalization

- Simplification of following optimization steps through uniform (canonical) query format
- Especially for selection and join conditions
  - Conjunctive normal form vs. disjunctive normal form
  - Conjunctive normal form (CNF) for simple predicates $p_{ij}$:
    \[(p_{11} \lor p_{12} \lor \cdots \lor p_{1n}) \land \cdots \land (p_{m1} \lor p_{m2} \lor \cdots \lor p_{mn})\]
  - Disjunctive normal form (DNF):
    \[(p_{11} \land p_{12} \land \cdots \land p_{1n}) \lor \cdots \lor (p_{m1} \land p_{m2} \land \cdots \land p_{mn})\]
  - Transformation to CNF/DNF by application of equivalence relations for logical operations
Normalization /2

Equivalence relations

- \( p_1 \land p_2 \iff p_2 \land p_1 \) and \( p_1 \lor p_2 \iff p_2 \lor p_1 \)
- \( p_1 \land (p_2 \land p_3) \iff (p_1 \land p_2) \land p_3 \) and \( p_1 \lor (p_2 \lor p_3) \iff (p_1 \lor p_2) \lor p_3 \)
- \( p_1 \land (p_2 \lor p_3) \iff (p_1 \land p_2) \lor (p_1 \land p_3) \) and
  \( p_1 \lor (p_2 \land p_3) \iff (p_1 \lor p_2) \land (p_1 \lor p_3) \)
- \( \neg(p_1 \land p_2) \iff \neg p_1 \lor \neg p_2 \) and \( \neg(p_1 \lor p_2) \iff \neg p_1 \land \neg p_2 \)
- \( \neg(\neg p_1) \iff p_1 \)
Normalization: Example

Query:

```sql
select * from Project P, Assignment A
where P.PNr = A.PNr and
    Budget > 100.000 and
    (Location = 'MD' or Location = 'B')
```

Selection condition in CNF:

\[
P.PNr = A.PNr \land Budget > 100.000 \land (Loc = 'MD' \lor Loc = 'B')
\]

Selection condition in DNF:

\[
(P.PNr = A.PNr \land Budget > 100.000 \land Location = 'MD') \lor
(P.PNr = A.PNr \land Budget > 100.000 \land Location = 'B')
\]
Logical Optimization

- Heuristic methods
  - For instance, algebraic optimization
  - For relational algebra + grouping, ...

- Exact methods
  - Tableau optimization
  - Minimize number of Joins
  - For specific relational algebra queries
Algebraic optimization

- Replacement of terms (of relational algebra) by means of algebra equivalences
- Equivalences are used directed as replacement rules
- Heuristic method: move operations in order to obtain smaller intermediate results; detect redundancies
Principles of algebraic Optimization

Example:

BOOKS = { Title, Author, Publisher, ISBN } 
PUBLISHER = { Name, PublAdr } 
BORROWER = { BorName, BorAdr, BorCard } 
BORROWING = { BorCard, ISBN, Date }
Remove redundant Operations

- Necessary for queries with views

\[ r(\text{LONGAWAY}) = r(\text{BOOKS}) \Join \]

\[ \pi_{\text{ISBN,DATE}}(\ldots \sigma_{\text{DATE}<'31.12.1995}(r(\text{BORROWING}))) \]

- Query to view:

\[ \pi_{\text{TITLE}}(r(\text{BOOKS}) \Join r(\text{LONGAWAY})) \]

- View expansion:

\[ \pi_{\text{TITLE}}(r(\text{BOOKS}) \Join r(\text{BOOKS}) \Join \pi_{\ldots}(\ldots)) \]

- Rule: Idempotency

\[ r = r \Join r, \text{ i.e., } \Join \text{ is idempotent} \]
Move of Selections

\[ \sigma_{\text{AUTHOR}=\text{'Heuer'}}(r(\text{BOOKS})) \bowtie \pi_{\text{ISBN,DATE}}(\ldots) \]

Better:

\[ (\sigma_{\text{AUTHOR}=\text{'Heuer'}}(r(\text{BOOKS}))) \bowtie \pi_{\text{ISBN,DATE}}(\ldots) \]

Rule:

\begin{center}
\text{commutating Selection and Join}
\end{center}

only, if allowed by attributes of selection predicate
Order of Joins

- Knowledge of statistical information of catalogue necessary

\[(r(PUBLISHER) \bowtie r(BORROWING)) \bowtie r(BOOKS)\]

- First Join: Cartesian product, hence:

\[r(PUBLISHER) \bowtie (r(BORROWING) \bowtie r(BOOKS))\]

Rule:

\[\bowtie\text{ is associative and commutative}\]

no distinctive preferred direction by applying this rule (hence, internal optimization and cost-based selection)
Algebraic Rules

- **CommJoin**: Operator $\Join$ is commutative:
  \[
  r_1 \Join r_2 \iff r_2 \Join r_1
  \]

- **AssocJoin**: Operator $\Join$ is associative:
  \[
  (r_1 \Join r_2) \Join r_3 \iff r_1 \Join (r_2 \Join r_3)
  \]

- **ProjProj**: For operator $\pi$ the outer parameter dominates the inner one (in combination of both):
  \[
  \pi_X(\pi_Y(r_1)) \iff \pi_X(r_1)
  \]
Algebraic Rules (II)

- **SelSel**: Combination of predicates for $\sigma$ corresponds to logical AND $\Rightarrow$ Order of formulas can be changed

\[
\sigma_{F_1}(\sigma_{F_2}(r_1)) \iff \sigma_{F_1 \land F_2}(r_1) \iff \sigma_{F_2}(\sigma_{F_1}(r_1))
\]

(Usage of commutativity and of logical AND)
SelProj: Operators $\pi$ and $\sigma$ commutate, if predicate $F$ is defined on projection attributes:

$$\sigma_F(\pi_X(r_1)) \iff \pi_X(\sigma_F(r_1))$$

if $\text{attr}(F) \subseteq X$

otherwise permutation possible, if Projection can be extended by needed attributes:

$$\pi_{X_1}(\sigma_F(\pi_{X_1}X_2(r_1))) \iff \pi_{X_1}(\sigma_F(r_1))$$

if $\text{attr}(F) \supseteq X_2$
SelJoin: Operators $\sigma$ and $\Join$ commute, if all selection attributes originate from one relation:

$$\sigma_F(r_1 \Join r_2) \iff \sigma_F(r_1) \Join r_2$$

if $\text{attr}(F) \subseteq R_1$

if selection predicate can be splitted in a way, that in $F = F_1 \land F_2$ both parts of the conjunction have appropriate attributes:

$$\sigma_F(r_1 \Join r_2) \iff \sigma_{F_1}(r_1) \Join \sigma_{F_2}(r_2)$$

if $\text{attr}(F_1) \subseteq R_1$ and $\text{attr}(F_2) \subseteq R_2$
Algebraic Rules (V)

SelJoin (cont.): in every Case: Separation of \( F_1 \) with attributes of relation \( R_1 \), if \( F_2 \) concerns attributes of \( R_1 \) and \( R_2 \):

\[
\sigma_F(r_1 \bowtie r_2) \iff \sigma_{F_2}(\sigma_{F_1}(r_1) \bowtie r_2)
\]

if \( \text{attr}(F_1) \subseteq R_1 \)
Algebraic Rules (VI)

- **SelUnion**: Commutating of $\sigma$ and $\cup$:
  $$\sigma_F(r_1 \cup r_2) \iff \sigma_F(r_1) \cup \sigma_F(r_2)$$

- **SelDiff**: Commutating of $\sigma$ and $-:
  $$\sigma_F(r_1 - r_2) \iff \sigma_F(r_1) - \sigma_F(r_2)$$
  or (cause tuples are only deleted from first relation):
  $$\sigma_F(r_1 - r_2) \iff \sigma_F(r_1) - r_2$$
Algebraic Rules (VII)

- **ProjJoin**: Commutating of $\pi$ and $\bowtie$:

$$\pi_X(r_1 \bowtie r_2) \iff \pi_X(\pi_{Y_1}(r_1) \bowtie \pi_{Y_2}(r_2))$$

with

$$Y_1 = (X \cap R_1) \cup (R_1 \cap R_2)$$

and

$$Y_2 = (X \cap R_2) \cup (R_1 \cap R_2)$$

Move Projection into one Join, if the computation of $Y_i$ cares, that the join attributes needed for Natural Join remain (Projection on these attributes after Join)
Algebraic Rules (VIII)

- **ProjUnion**: Commutating of $\pi$ and $\cup$:

  $$\pi_X(r_1 \cup r_2) \iff \pi_X(r_1) \cup \pi_X(r_2)$$

- Distributive law for $\Join$ and $\cup$ as well as for $\Join$ and $-$, commutating of Renaming $\beta$ with other operations, etc.
Further Rules

- **Idempotencies**

IdemUnion: \( r_1 \cup r_1 \iff r_1 \)

IdemIntersect: \( r_1 \cap r_1 \iff r_1 \)

IdemJoin: \( r_1 \Join r_1 \iff r_1 \)

IdemDiff: \( r_1 - r_1 \iff \{\} \)

- Concatenation with empty relation:

EmptyUnion: \( r_1 \cup \{\} \iff r_1 \)

EmptyIntersect: \( r_1 \cap \{\} \iff \{\} \)

EmptyJoin: \( r_1 \Join \{\} \iff \{\} \)

EmptyDiffRight: \( r_1 - \{\} \iff r_1 \)

EmptyDiffLeft: \( \{\} - r_1 \iff \{\} \)

- For \( \Join \), \( \cup \) and \( \cap \) the **commutative** and **associative law** apply (Notation: Komm\* and Assoz\*)
Simple Optimization Algorithm

- Resolve complex Selection predicates (rule SelSel, rules for resolution for ¬ and ∨ (if applicable)
- By means of rules SelJoin, SelProj, SelUnion and SelDiff move Selections as far as possible to the direction of the leaves, change Selections according to rule SelSel if applicable
- Move of projections in direction of the leaves using rules ProjProj, ProjJoin and ProjUnion

Particular steps are applied in stated order as long as no replacements are possible anymore
Unoptimized Query Plan

\[
\pi_{\text{Title}} \rightarrow \sigma_{\text{Date}=1.1.88 \land \text{Author}='\text{Heuer'}} \rightarrow \pi_{\text{ProjList}} \\
\bowtie r(\text{Books}) \bowtie r(\text{Borrowing}) \bowtie r(\text{Borrower})
\]

ProjList=Title, Author, Publisher, ISBN, BorAdr, BorName...
Query Plan (II)

After Moving of Selections

\[
\begin{align*}
\pi_{\text{Title}} \\
\sigma_{\text{Author}='\text{Heuer}'} \\
\sigma_{\text{Date}<1.1.88} \\
\sigma_{\text{Date}<1.1.88} \\
r(\text{Borrower}) \\
r(\text{Books}) \\
r(\text{Borrowing})
\end{align*}
\]
Query Plan (III)

With additional Projections

\[\pi_{\text{Title}}\]

\[\pi_{\text{ISBN}}\]

\[\sigma_{\text{Date}<\text{1.1.88}}\]

\[r(\text{BORROWING})\]

\[r(\text{BORROWER})\]

\[\sigma_{\text{Author}='\text{Heuer}'\}\]

\[\pi_{\text{Title,ISBN}}\]
Join Optimization with Tableaus

- Exact optimization (minimize numbers of Joins)
- For a limited class of queries \((\sigma, \pi, \Join)\)

Useful for

- Optimization of queries on views
- Query interpretation of universal relation interfaces
Tableaus — Informal Introduction

Matrix-shaped representation of a relational algebra or calculus query

- Columns of matrix: Attributes of universe
- First row of Tableau: *summary*; consists of *blanks* and *distinguished variables* $a_i$
- Further rows: *blanks*, distinguished variables, constants and *non-distinguished variables* $b_j$; each row possesses tag *tag* (name of basis relation)
Equivalent Classes of Queries

- **Tableau queries**
- **Conjunctive queries** Subset of domain calculus

\[ \{ a_1 \ldots a_n \mid \exists b_1 \ldots \exists b_m : F_1 \land \cdots \land F_k \} \]

whereas \( F_i = R(c_1 \ldots c_r) \), i.e., \( c_1 \ldots c_r \in r(R) \); \( c_j \) distinguished or non-distinguished variables or constants

- **Limited relational algebraic expressions.**

\( \sigma_{A=c}, \sigma_{A=B}, \sigma_F, \pi, \bowtie \) with \( F \) consisting of \( A = c, A = B, \land. \)
Example for Tableau Queries

\( \mathcal{U} = \{A, B, C\} \text{ mit } R_1 = \{A, B\} \text{ und } R_2 = \{B, C\} \)

- Algebra expression: \( \pi_A(\sigma_{C=3}(r(R_1) \bowtie r(R_2))) \)
- Conjunctive query: \( \{a_1 \mid \exists b_1 : R_1(a_1, b_1) \land R_2(b_1, 3)\} \)
- Tableau query:

\[
\begin{array}{c|c|c|c}
\text{tags} & A & B & C \\
\downarrow & & & \\
\text{summary} & w_0 & a_1 & b_1 \\
\rightarrow & w_1 & a_1 & b_1 \\
\rightarrow & w_2 & b_1 & 3 \\
\end{array}
\]

\( R_1 \)
\( R_2 \)
Internal Optimization

- First step: Transform operations of relational algebra in internal operations (see chapter 6)
- Further operations
  - Replace tuple sets in parts by sorted tuple lists
  - Multisets instead of sets
  - Besides tuples, even TID lists are processed
  - Accesses on indices
Selection of Computation Algorithms

- **Projection:**
  - $\pi_{\text{AttList}}^{\text{REL}/\text{with}}$: Projection by full table scan (\textbf{with} indicates a Projection with duplicate elimination)
  - $\pi_{\text{AttList}}^{\text{REL}/\text{wo}}$: Projection by full table scan without duplicate elimination
  - $\pi_{\text{AttList}}^{\text{SORT}/\text{with}}$: Projection by full table scan on a relation, sorted by \textbf{AttList} with duplicate elimination
  - $\pi_{\text{AttList}}^{\text{SORT}/\text{wo}}$: Projektion Projection by full table scan on a relation, sorted by \textbf{AttList} without duplicate elimination
Selection of Computation Algorithms II

- **Selection:**
  - $\sigma_{\phi}^{\text{REL}}$: Selection by full table scan
  - Selection on index (in detail later)

- **Join:**
  - $\Join_{\text{DIRECT}}$: Join by Nested-Loops $\text{DIRECT}$
  - $\Join_{\text{MERGE}}$: Join by merging $\text{MERGE}$ (precondition: input relation sorted by join attribute(s)).
  - $\Join_{\text{HASH}}$: Join by hashing $\text{HASH}$

- Operators for union, difference and intersection analogous to Join; union with or without duplicate elimination
Index Access

- Access over index

\[ \sigma_{A \Theta a}^{\text{IND}}(\mathcal{I}(\mathcal{R}(A))) \rightarrow \text{list}(\text{tid}) \]

returns TID list

- Special case \( \sigma_{\text{true}}^{\text{IND}}(\mathcal{I}(\mathcal{R}(A))) \): Returns all index entries sorted by \( A \)

- Variants:
  - Primary index w/o duplicates: Result for \( A = a \) is a single tuple
  - Tuples directly in index: Result for index access is sorted list of tuples
  - Range query: Predicate \( a_1 \leq A \leq a_2 \)
Index Access II

- Index can support Projection as well
- Result of $\sigma^{\text{IND}}_{\text{true}}(\mathcal{I}(R(A)))$ used as input for $\pi^{\text{SORT}/-}$ operator
- Combined access: $\pi^{\text{IND/with AttList}}$ or $\pi^{\text{IND/wo AttList}}$ respectively
- In $\pi^{\text{IND/-}}_{A}(\mathcal{I}(R(A)))$ access on basis relation can be dispensed
New Operators

- For TID lists: ‘Realization’ operator $\rho$:
  \[ \rho(\langle \text{TID list for } R \text{ tuples} \rangle, r(R)) \]

- On TID lists, $\cup$, $\cap$ and $-$ are defined

- *Sorting* of tuple sets: $\omega$
  \[ \omega_{\text{AttList}}(\langle \text{tuple sequence} \rangle) \]

- Sorting a relation:
  \[ \rho(\sigma^\text{IND}_{\text{true}}(I(R(\text{AttList}))), r(R)) = \omega_{\text{AttList}}(r(R)) \]

Effort for first variant is $O(|R|)$, for second variant $O(|R| \times \log |R|)$
Examples for internal Access Plans

```
select *
from DELIVERER
where Product = 'Tea' and
      ( DName = 'Tom' or DName = 'Jerry')
and Price < 10
```
Two Access Plans
Pipelining of Operations

\[ \pi_A (\sigma_{\varphi} (r(R))) \]
Pipelining of Operations II

 Typical combinations (fusions):

- Combination of Selection and Projection
- Combination of a Selection with Join
- Integration of a Selection in outer loop of Nested-Loop Join
- Integration of Selections in Merge Join
- Coupling of Selection with Realization operator

```sql
select C.CName, Kto
from CUSTOMER C, ORDER O
where C.CName = O.CName and O.Product = 'Tea'
```
Shared Partial Queries

- Task: Detection of *shared partial queries* \( \sim \) the corresponding subtrees (of the operator tree) have to be put on the same level

- Problems:
  - Different syntactical form: \( r_1 \cup r_2 \) identical with \( r_2 \cup r_1 \)
  - Overlapping \( (\sigma_\varphi \text{ overlaps } \sigma_{\varphi \land \psi}) \)
Cost-Based Selection

Considerations (for selection):

- Actual size of database relations
- Existence of indices (primary, secondary) and their size
- Clustering of several relations
- Selectivity of an attribute, an index has been defined on
Relevant Database Parameter

- **System parameter** from catalogue: $s$: length of a page (useful page space in Byte)
- size of database: $S$ as number of reserved pages (is needed, if tuples of relations are stored scattered)
- Statistical data about relations and indices:
  - $T_R$: Number of tuples in relation $R$
  - $L_R$: Average length of a tuple in $R$
  - $W_{A,R}$: Number of different values for attribute $A$ in $R$ (index information or statistic)

Statistics $\rightarrow$ Update realized in case of change operation or by calling the respective command
Selectivity of Attributes

Number of different values for attribute $A$ in $R$: $W_{A,R}$

- Equality:
  
  $$sel(A=c, R) = \frac{1}{W_{A,R}}$$

- Inequality:
  
  $$sel(\neg A=c, R) = 1 - sel(A=c, R) = 1 - \frac{1}{W_{A,R}}$$

- Comparison using $<$, $>$, ...:
  
  $$sel(A<c, R) = sel(A>c) = \frac{1}{2}$$
Selectivity of Attributes II

Refinement:

\[ sel(A \leq c, R) = \frac{A_{max} - c}{A_{max} - A_{min}} \]

Range queries:

\[ sel(c_u \leq A \leq c_o, R) = \frac{c_o - c_u}{A_{max} - A_{min}} \]

Selectivities for Joins:

\[ sel_{\bowtie}(\varphi, R, S) \approx \frac{|R \bowtie_{\varphi} S|}{|R \times S|} \]
Optimization

Estimation of Selectivity

1. **Parametrized Functions**
   Parameter of a function, which reflects well the data distribution, should be specified as exact as possible (e.g., Gaussian distribution)

2. **Histograms**
   Domain is divided into subregions and the values, existing in reality (in these regions) have to be count

3. **Sampling**
   Selectivity is determined with the help of random samples of the stored data records
Histograms for Attribute Values
Height-Balanced Histograms

a) uniformly distributed values

b) unequally distributed values
Computation of Costs: Example

Selections on relation ORDER with attribute Product

- $S = 4\,000$ (the database reserves 4000 pages overall)
- $T_{\text{ORDER}} = 10\,000$ (in the relation ORDER 10,000 tuples are stored)
- $s/L_{\text{ORDER}} = 10$ (in average, 10 tuples fit into one page)
- $W_{\text{Product},\text{ORDER}} = 50$ (50 different products as value for the Product attribute)

Selection $\sigma_\varphi$ mit $\varphi = (A \theta a \land \psi)$
Computation of Costs: Variant A

- **Index** $I(R(A))$ on Selection attribute $A$

  \[
  \langle \sigma_{\psi} \circ \rho \circ \sigma_{A\theta a}^{\text{IND}} \rangle (I(R(A)), r(R))
  \]

- **A1** Index with clustering: $S_R \times sel(A\theta a, R)$.
- **A2** Index without clustering: $T_R \times sel(A\theta a, R)$.
  (Maximum value, if tuples are stored on different pages respectively)
Computation of Costs: Variant B

- **Index** \( I(R(B)) \) with \( B \neq A \)

\[
\left\langle \sigma^\text{REL}_{A \theta a \land \psi} \circ \rho \circ \sigma^\text{IND}_{\text{true}} \right\rangle(I(R(B)), r(R))
\]

**B1** Index with clustering: \( S_R \).

**B2** Index without clustering: \( T_R \).

Costs result from the fact, that the selectivity of the predicate \text{true} is 1.
Computation of Costs: Variant C

- Full table scan
- Assumption: All pages of the DB are read and all tuples, existing on these pages are found
- Costs: Given by the number $S$
Computation of Costs: Queries

1. \( \sigma_{\text{Product}=\text{`Tea'}}(r(\text{ORDER})) \).
   Selectivity \( sel: \frac{1}{50} \)

2. \( \sigma_{\text{Product}>\text{`Tea'}}(r(\text{ORDER})) \).
   Selectivity \( sel: \text{Assumption} \sim \frac{1}{2} \).
## Costs for Realization Variants

<table>
<thead>
<tr>
<th>Variant</th>
<th>Cost formula</th>
<th>Product = 'Tea'</th>
<th>Product &gt; 'Tea'</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>(S_R \times sel(A_{\theta a}, R))</td>
<td>20</td>
<td>500</td>
</tr>
<tr>
<td>A2</td>
<td>(T_R \times sel(A_{\theta a}, R))</td>
<td>200</td>
<td>5.000</td>
</tr>
<tr>
<td>B1</td>
<td>(S_R)</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>B2</td>
<td>(T_R)</td>
<td>10.000</td>
<td>10.000</td>
</tr>
<tr>
<td>C</td>
<td>(S)</td>
<td>4.000</td>
<td>4.000</td>
</tr>
</tbody>
</table>
Optimizer Architecture

- Query
- Translation
  - Query representation (internal representation)
  - logical optimization
    - Catalogue
      - Statistics
    - internal optimization
      - Query processing
      - Query execution
  - internal optimization
    - Query plan
    - Plan computation
      - Query result
Optimizer Variants

- **Heuristic, rule-based optimizer:**
  1. Generation of internal query representation by logical optimization
  2. Query plan is created by internal optimization

- **Cost-based (two-phase) optimizer:**
  1. Generation of different query representations by logical optimization
  2. Transfer to internal optimization (plan generation)
  3. Cost evaluation
  4. Selection of most suitable plan
Optimization: Overview

Query

span the search space

Equivalent plans

search strategy

"Best" plan

transformation rules

cost model

Gunter Saake
Database Implementation Techniques Database

Gunter Saake
Database Implementation Techniques

Last Changes: 21.4.2015 7–493
Optimization: Search Space

- Search space: Set of all equivalent query plans
- Spanned by *transformation rules* (algebraic rules)
- Main focus: *Join Trees*
- For *n* relations: *n!* different Join trees!
- Hence: Limitation of search tree by
  - Heuristics (algebraic optimizations)
  - given ""shape"" of the tree
Optimization: Join Trees

- Linear sequence of operator trees
  - Only $2^n$ Variants
  - *left deep tree* and *right deep tree* \(\sim\) all inner nodes of the tree possess at least one leaf node (basis relation) as child

- *bushy trees*
  - Higher potential for parallelization, though high effort for optimization
Optimization: Search Strategies

- "Traversal" of search space
- Selection of most cost-effective plan
- Foundation: Cost model
- Determines
  - *Which plans* are worth to be considered (complete / partial traversal)
  - The *order* in that alternatives are searched
- Variants: deterministic, random-based
Optimization: Search Strategies /2

- deterministic:
  - Systematic generation of plans
  - Starts with plans for access on basis relations
  - Construction of complex plans by composition of more simple plans
  - Exhausting search; guarantees best plan
  - Example: Dynamic Programming (breadth-first search)
  - State of the art
Random-based:
- One or more starting plans by Greedy strategy (depth-first search)
- Improvement of starting plans by analyzing the "neighbors"
- Neighbor: Application of transformation rules, e.g., exchange of two randomly selected operation
- Example: Simulated Annealing
- Better performance for huge number of relations
- No guarantee for best plan
Optimization in INGRES

- Dynamic technique for optimization
- Recursive partition of a calculus query in sequences of *Mono-Relation-Queries*
- Processing of mono-relation queries by "“One Variable Query Processor”" (OVQP)
- OVQP selects best access method (Index selection)
Optimization in INGRES /2

- Principle:
  - Query $q$:
    
    ```
    select $R_2.A_2$, $R_3.A_3$, $\ldots$, $R_n.A_n$
    from $R_1, R_2, \ldots, R_n$
    where $P_1(R_1.A'_1)$ and $P_2(R_1.A_1, R_2.A_2, \ldots, R_n.A_n)$
    ```

  - Partition in $q'$:
    
    ```
    select $R_1.A_1$ into $R'_1$
    from $R_1$
    where $P_1(R_1.A'_1)$
    ```

  - and $q''$:
    
    ```
    select $R_2.A_2$, $R_3.A_3$, $\ldots$, $R_n.A_n$
    from $R'_1, R_2, \ldots, R_n$
    where $P_2(R_1.A_1, R_2.A_2, \ldots, R_n.A_n)$
    ```
Optimization in INGRES /3

- Handling of unreducible multi-relation queries (esp. Join)
  - Converting to mono-relation queries by *tuple substitution*
  - For query $q$: Selection of a relation $R_1$ and derivation of $\text{card}(R_1)$ queries $q'$ with $n - 1$ relations

$$q(R_1, R_2, \ldots, R_n) \text{ replaced by } \{q'(t_1i, R_2, R_3, \ldots R_n); t_1i \in R_1\}$$
Optimization in System R

- Dynamic Programming
- Bottom-Up construction of a plan
  1. Generation of simple plans (access on one relation)
  2. Generation of more complex plans (2 relations, 3 …) by combination (Join) of simple plans

In the process: Pruning
  - Limitation of solution space by deletion of "unsound" plans, equivalent (alternative) plans exist for
    1. Permutations with cartesian products
    2. Commutative strategies exhibit highest costs
Optimization in System R /2

**Input:** SPJ query $q$ on relations $r_1, \ldots, r_n$

**Output:** Query plan for $q$

```latex
\begin{align*}
\text{for } i & := 1 \text{ to } n \text{ do} \\
& \quad \text{optPlan}([r_i]) := \text{accessPlans}(r_i) \\
& \quad \text{prunePlans}(\text{optPlan}([r_i])) \\
\text{end} \\
\text{for } i & := 1 \text{ to } n \text{ do} \\
& \quad \text{forall } s \subseteq \{r_1, \ldots, r_n\} \text{ such that } |s| = i \text{ do} \\
& \quad \quad \text{optPlan}(s) := \emptyset \\
& \quad \quad \text{forall } t \subset s \text{ do} \\
& \quad \quad \quad \text{optPlan}(s) := \text{optPlan}(s) \cup \\
& \quad \quad \quad \quad \text{joinPlans}(\text{optPlan}(t), \text{optPlan}(s - t)) \\
& \quad \quad \quad \text{prunePlans}(\text{optPlan}(s)) \\
& \quad \text{end end end} \\
\text{return } \text{optPlan}([r_1, \ldots, r_n])
\end{align*}
```
Optimization in System R /3

Example query:

```
select E.EName
from Employees E, Assignment A, Project P
where E.ENr = A.ENo and A.PNr = P.PNr
and P.PName = 'DB development'
```

Indices: Employees (ENr), Assignment (PNr), Project (PNr), Project (PName)

Access plans after 1. iteration:

- Employees: Full table scan
- Assignment: Full table scan
- Project: Index scan on PName
Optimization in System R /4

- Possible order of joins

```
Employees x Project
Employees * Assignment
Assignment * Project
Project x Employees

(Assignment * Employees) * Project
(Project * Assignment) * Employees
```
Oracle9i: Statistics

- Tables: Number of tables and blocks, averaged tuple length
- Columns: Number of different values, number of NULL values, data distribution (Histograms)
- Indices: Number of leaf pages, height of tree, clustering factor
- System: I/O- and CPU performance and workload respectively
Retrieval of Statistics

- Retrieval by
  - Estimation based on samples (*row sampling, block sampling*)
  - Exact computation (Effort: Table scan + sorting)
  - user-defined

- Tools
  - `ANALYZE TABLE ...`
  - `Package DBMS_STATS`: Procedures for extended handling of statistics (recommended by oracle)
Maintenance of Statistics

- Manual update of statistics (possibly as job)
- Automatic update (monitoring)
  - Observation of number of change operations
  - $\geq 10\%$ affected $\Rightarrow$ out-dated data $\Rightarrow$ update
Missing Statistics

- Default values for missing statistics
  - Tables
    - Average size of tuples: 100 Bytes
    - Number of blocks: 1
    - Cardinality: \( \text{num}\_\text{of}\_\text{blocks} \times (\text{block}\_\text{size} - \text{cache}\_\text{layer}) / \text{avg}\_\text{row}\_\text{len} \)
  - Indices
    - Height: 1
    - Number of leaf pages: 25
    - Number of diff. key values: 100
Generation of Statistics: **ANALYZE**

- Invocation:
  
  ```sql
  analyze table table kind-of-statistic
  ```

- Kinds
  - **estimate statistics**: Estimation by sampling; optional parameter specifies size of sample (```sample size rows | percent```)
  - **compute statistics**: Exact determination

- Example:
  
  ```sql
  analyze table emp
  estimate statistics sample 10 percent;
  ```
Generation of Statistics: **DBMS_STATS**

- **Invocation of package procedures:**
  - Amongst others `gather_index_stats`, `gather_table_stats`, `gather_schema_stats`, ...

- **Example:**

  ```
  execute dbms_stats.gather_table_stats(
    ownname => 'scott',
    tabname => 'emp',
    estimate_percent => NULL,
    method_opt => NULL);
  ```
**DBMS_STATS (II)**

- **Parameter:**
  - `estimate_percent`: Estimation based on given sample size (in percent)
  - `method_opt`: Specification of the statistics to be generated, u.a.
    - ★ FOR ALL [INDEXED ] COLUMNS: all columns
    - ★ FOR COLUMNS column list: for given columns
    - ★ AUTO: automatic selection of columns for Histograms
Display of Statistics

- Statistic data in Data Dictionary
- Views: `dba_-`, `user_-`, `all_tables`, `-tab_col_statistics`
- Example: Table statistic (`user_tables`)

```
  TABLE_NAME  NUM_ROWS  BLOCKS  AVG_ROW_LEN
  ------------  --------  -------  ---------  
  EMP          14       1       37
```
Display of Statistics (II)

Example: Column statistic (user_tab_col_statistics)

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>NUM_DISTINCT</th>
<th>NUM_NULLS</th>
<th>NUM_BUCKETS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td>14</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>ENAME</td>
<td>14</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>JOB</td>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>MGR</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>HIREDATE</td>
<td>13</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>SAL</td>
<td>12</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>COMM</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>DEPTNO</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Oracle: Histograms

Kinds

- Height-based (equi-height)
- Value-based (frequency)

- Every value of column has corresponding bucket
- Bucket number according to the frequency of the value
- Applied, if number of different values of the column \( \leq \) number of buckets

Selection depends on frequency of values
Optimization

Histograms: Generation

- **ANALYZE TABLE**
  ```sql
  analyze table emp compute statistics
  for column sal size 10;
  ```

- **DBMS_STATS**
  ```sql
  execute dbms_stats.gather_table_stats(
    ownname => 'scott',
    tabname => 'emp',
    method_opt => 'for columns size 10 sal');
  ```

- Default number of buckets: 75
Output of Histograms

- **Data Dictionary views:** dba_, user_, all_histograms
- **Query:**

```sql
select endpoint_number, endpoint_value
from user_histograms
where table_name='EMP' and column_name='SAL';
```

<table>
<thead>
<tr>
<th>ENDPOINT_NUMBER</th>
<th>ENDPOINT_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>800</td>
</tr>
<tr>
<td>2</td>
<td>950</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>13</td>
<td>3000</td>
</tr>
<tr>
<td>14</td>
<td>5000</td>
</tr>
</tbody>
</table>
Oracle: Output of Plans

- Storage of execution plan (incl. costs) for a certain query in a table plan_table

```
explain plan set statement_id = 'MPLAN'
for select title from movie, budget
where id between 2000 and 40000
  and movie.id = budget.movie
  and budget < 100000 and year = 1998;
```
Oracle: Output of Plans (II)

- Readout of plan by query or specific tools

```sql
select substr(lpad(' ',2*(level-1)),1,8) ||
    substr(operation,1,16) "OPERATION",
    substr(options,1,12) "OPTIONS",
    substr(object_name,1,12) object_name,
    id, parent_id, cost, cardinality, bytes
from plan_table
start with id=0 and statement_id = 'MPLAN'
connect by prior id = parent_id and
    statement_id = 'MPLAN';
```
Oracle: Output of Plans (III)

Output

```
OPERATION OPTIONS OBJECT_NAME ID PARENT_ID ------------ -------- -------- --- -------...
STATEMENT 0 NESTED LOOPS 1 0 TABLE ACCESS BY INDEX ROW MY_MOVIE 2 1 INDEX RANGE SCAN MOVIE_PK
3 2 TABLE ACCESS BY INDEX ROW BUDGET 4 1 INDEX UNIQUE SCAN SYS_C006658 5 4
```
## Oracle: Columns of Plan Table

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement_id</td>
<td>ID from <strong>explain plan</strong></td>
</tr>
<tr>
<td>operations</td>
<td>plan operator (1. row: Statement)</td>
</tr>
<tr>
<td>options</td>
<td>details to plan operator</td>
</tr>
<tr>
<td>object_name</td>
<td>table, index etc.</td>
</tr>
<tr>
<td>id, parent_id</td>
<td>IDs of operations</td>
</tr>
<tr>
<td>position</td>
<td>(1. row: total costs, otherwise: relative pos.)</td>
</tr>
<tr>
<td>cost</td>
<td>costs of operation</td>
</tr>
<tr>
<td></td>
<td>(function on CPU/IO costs)</td>
</tr>
<tr>
<td>cardinality</td>
<td>number of processed tuples</td>
</tr>
<tr>
<td>bytes</td>
<td>size of processed Bytes</td>
</tr>
<tr>
<td>cpu_cost, io_cost</td>
<td>CPU/IO costs (proportional to actual values)</td>
</tr>
<tr>
<td>temp_space</td>
<td>required temp. storage in Bytes</td>
</tr>
</tbody>
</table>
### Oracle: Operations of Plan table

<table>
<thead>
<tr>
<th>operation</th>
<th>option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td></td>
<td>Selection wrt. condition</td>
</tr>
<tr>
<td>hash join</td>
<td></td>
<td>Hash Join</td>
</tr>
<tr>
<td>merge join</td>
<td></td>
<td>Merge Join</td>
</tr>
<tr>
<td>merge join (outer, cartesian)</td>
<td></td>
<td>Nested-Loops Join</td>
</tr>
<tr>
<td>nested loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>unique scan</td>
<td>Access to single Rowid</td>
</tr>
<tr>
<td>index</td>
<td>range scan</td>
<td>Range access on index</td>
</tr>
<tr>
<td>sort</td>
<td>group by</td>
<td>Sorting for grouping</td>
</tr>
<tr>
<td>sort</td>
<td>unique</td>
<td>. . . for duplicate elimination</td>
</tr>
<tr>
<td>sort</td>
<td>join</td>
<td>. . . for Merge Join</td>
</tr>
<tr>
<td>table access</td>
<td>full</td>
<td>table scan</td>
</tr>
<tr>
<td>table access</td>
<td>by index rowid</td>
<td>table access on Rowid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of index access</td>
</tr>
</tbody>
</table>
Oracle: Optimizer Hints

- Specific influence on optimization of queries
- Aspects:
  - Objective: Throughput / response time
  - Query transformation (e.g., star queries, materialized views etc.)
  - Access paths
  - Join order
  - Join operation
- Indication: by comments
  
  /* +hint */
  
  -- +hint
Hints: Throughput vs. Response Time

- Goal of optimization: highest throughput (minimal resource usage)

```sql
select /*+ all rows */ *
from emp;
```

- Goal of optimization: best response time, i.e., first $n$ tuples should be provided as fast as possible
  (not for `group by`, `order by`, set operations and `distinct` queries)

```sql
select /*+ first rows(10) */ *
from emp;
```
Hints: Access Paths

- Selection of different strategies: full, index, ...
- Example: Usage of index for `ename`

```sql
select /*+ index(emp ename_idx) */ *
from emp
where ename = 'JONES';
```
Hints: Joins

- Specification of Join implementation
  - /* +use_nl(inner_tbl) */: Nested-Loops Join
  - /* +use_merge(tbl1 tbl2) */: Merge Join
  - /* +use_hash(tbl1 tbl2) */: Hash Join

- Specification of Join order
  - /* +ordered */: Order like in `from` clause

- Example:
  ```sql
  select /*+ ordered */ *
  from tab1, tab2, tab3
  where tab1.col = tab2.col
  and tab1.col = tab3.col;
  ```