TECHNISCHE UNIVERSITÄT DRESDEN

Dresden Database Research Group

Dirk Habich

Scalable Data Management (SDM)

3

Traditional Database Architecture

What is in the Lecture?

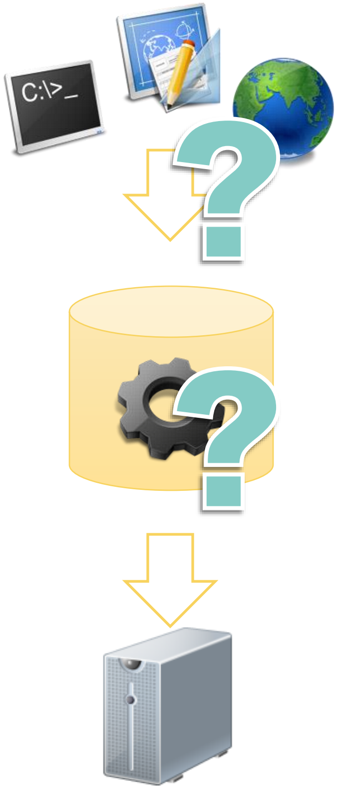
Dresden Database Research Group

Database Usage (Previous Lecture)

- Query
- Programming
- Design

(Traditional) Database Architecture

- Data Storage
- Indexes
- Query Processing
- Query Optimization



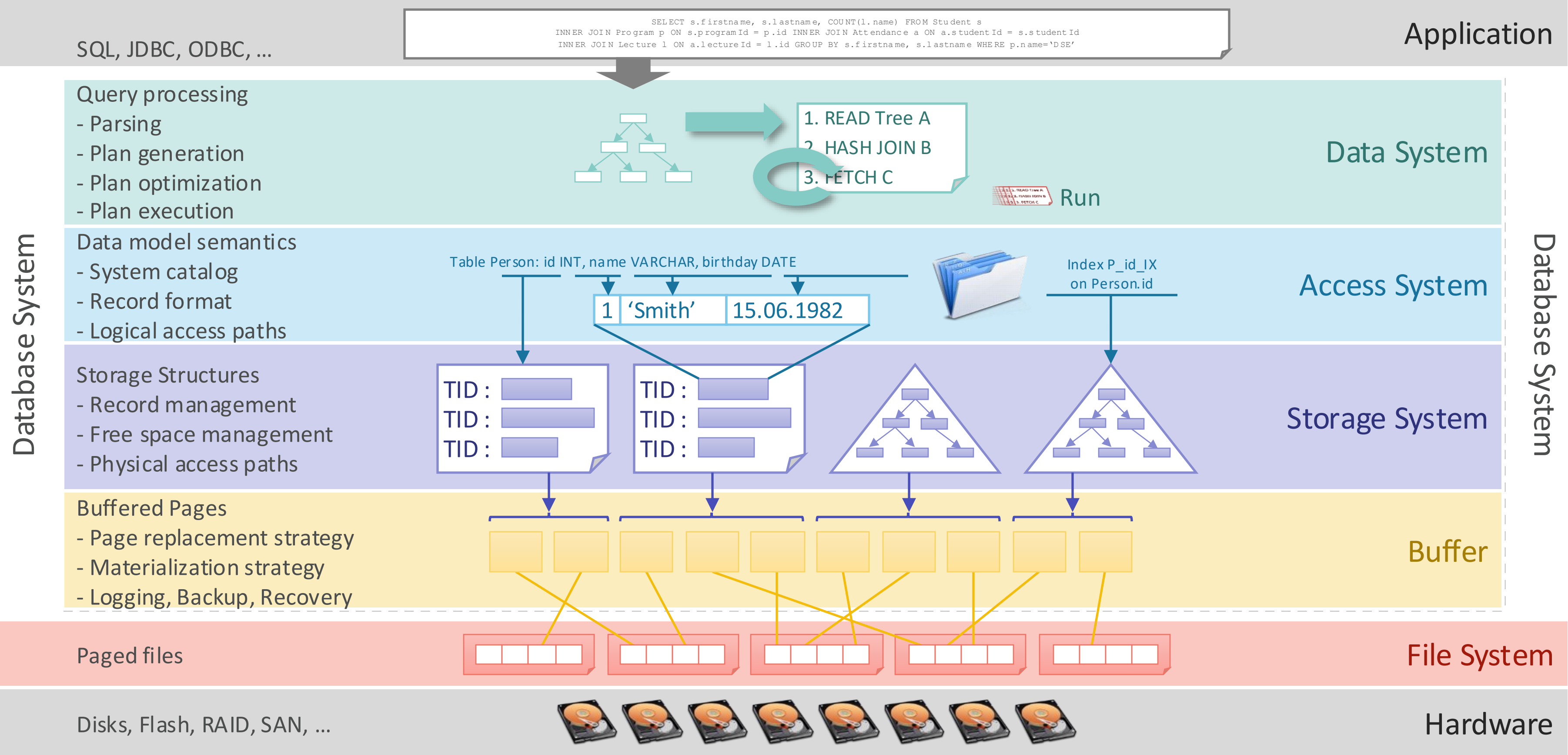
How is Database System build?


```
SELECT s.firstname, s.lastname, COUNT(l.name)
FROM Student s
INNER JOIN Program p ON s.programId = p.id
INNER JOIN Attendance a ON a.studentId = s.studentId
INNER JOIN Lecture l ON a.lectureId = l.id
GROUP BY s.firstname, s.lastname WHERE p.name='DSE'
```

```
byte[] b = read(File f, int pos, int length)
```

Architectural Blue Print

Dresden Database Research Group



TECHNISCHE UNIVERSITÄT DRESDEN

Buffer Management

Buffer Management

Setup

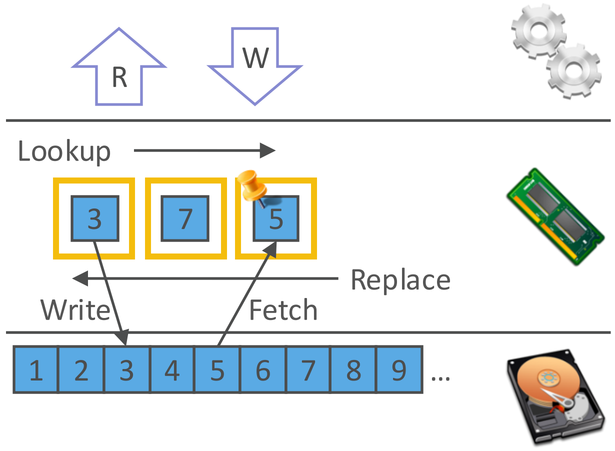
- All reads/writes go through buffer
- Buffer frames cache disk pages
- Only small fraction of pages
- Use of application knowledge for management

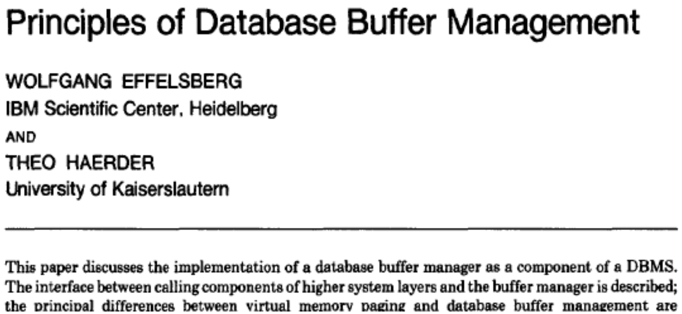
Operations

- Get page: Lookup, Fetch, Replace
- Pin/unpin: Fix page in buffer
- Mark dirty: Page must be written back if replaced
- Write: Immediate write back to disk

Strategies in buffer management

- Finding a buffered page in frames
- Allocating of buffer space among queries
- Replacement strategy





Principles of Database Buffer Management

WOLFGANG EFFELSBERG
IBM Scientific Center, Heidelberg
AND
THEO HAERDER
University of Kaiserslautern

This paper discusses the implementation of a database buffer manager as a component of a DBMS. The interface between calling components of higher system layers and the buffer manager is described; the principal differences between virtual memory paging and database buffer management are

[W. Effelsberg, T. Haerder: Principles of Database Buffer Management. ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984]

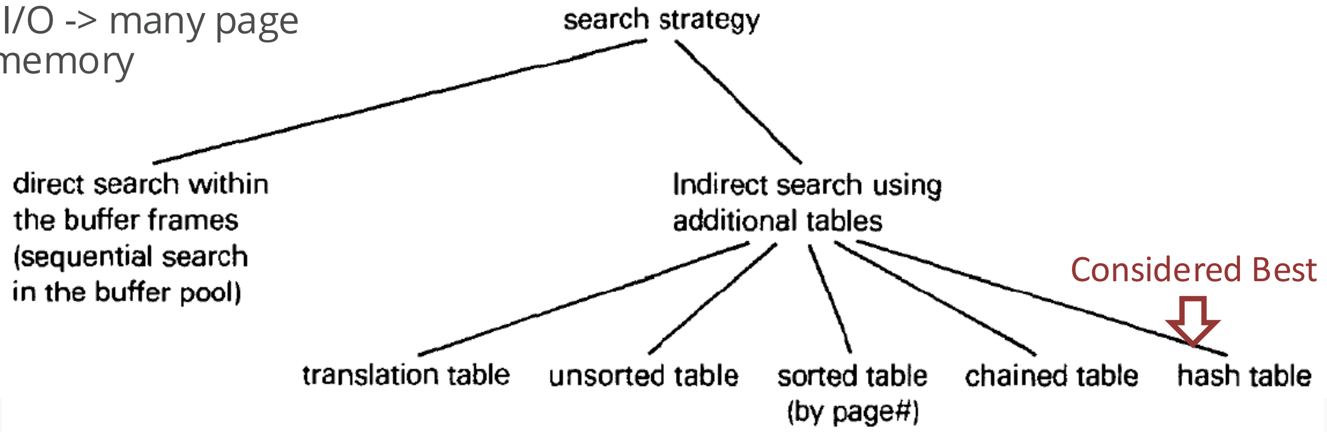
Finding a Buffered Page

Requirement

- Highly efficient, since extreme frequent operation!

Search strategies

- Direct search in buffer
  - Sequential scanning buffer frames
  - Check header of each page if it is the requested page
  - Very expensive
  - Much memory I/O -> many page fault in virtual memory
- Indirect search with auxiliary structures
  - Auxiliary structures maps page id to frame number
  - Less memory I/O
  - More efficient search algorithms



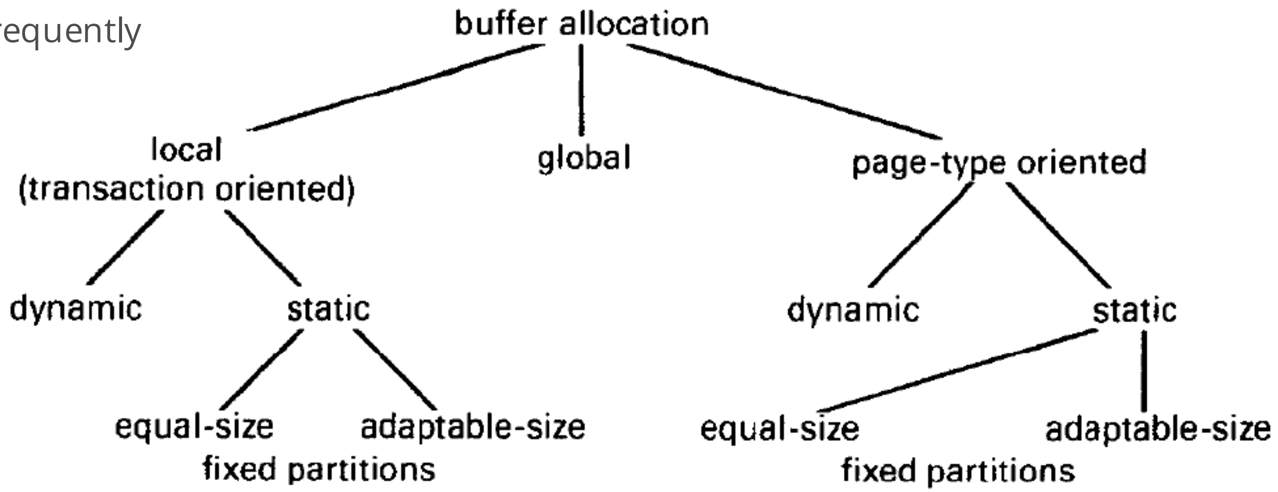
Allocating Buffer Space

Task

- Distribute available buffer frames among the concurrent database queries/transactions
- Closely related to the page replacement algorithm
- Reference behavior of database transactions is predictable,
  since it is based on existing access path structures

Classification

- Static allocation is inflexible in situations
  where the DBMS load changes frequently
- Dynamic allocation looks
  at reference history
  (of transaction or page type)
- Global allocation
  coincides with
  replacement algorithms



Replacement Strategy

Dresden Database Research Group

Fetching

- Prefetching: reduces the overall I/O costs, may increase overhead
- Demand fetching: only reacts on request, no overhead

Bounds of applicable algorithms

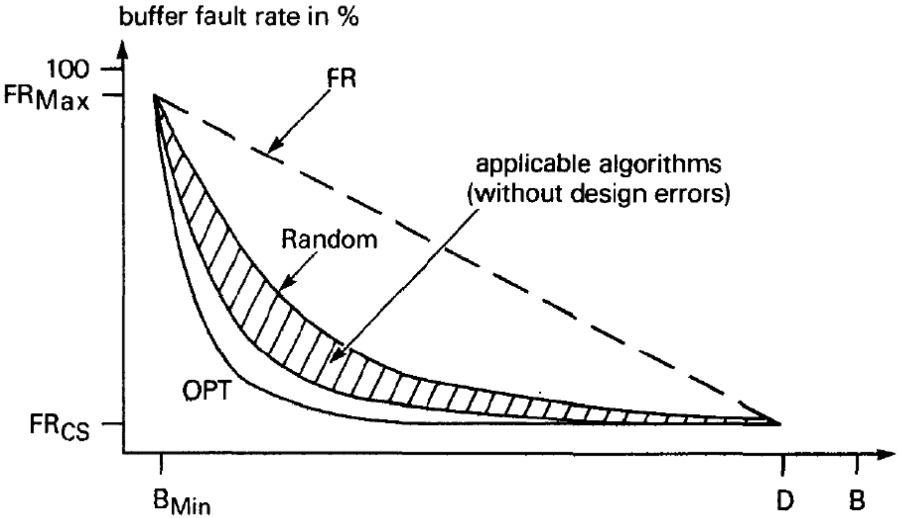
- Worst strategy: RANDOM
- Best strategy: OPT (not achievable)

Criteria and Horizon

	Complete History	Recent History
Age	Since first reference	Since last reference
References	All references	Recent reference(s)

Goal

- Get as close as possible to OPT

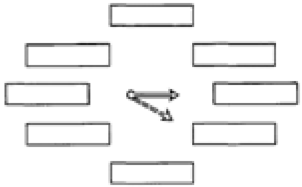


B_{Min} = Minimal buffer size
D = Buffer holding entire database
FR_{CS} = Cold start buffer fault rate, i.e. the minimal fault rate for a given reference string due to the initially empty buffer (no. of different page/no. of references × 100%)

Replacement Strategy (2)

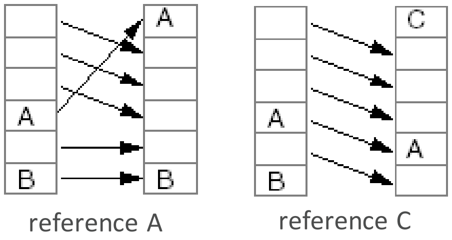
First-In-First-Out (FIFO)

- Age since last reference
- Replace oldest page
- Simple: Less maintenance than LRU



Least-Recently Used (LRU/LRU-K)

- Age of last/k-recent reference
- Replace page with oldest last/k-recent reference
- Very common policy: intuitive, simple and effective

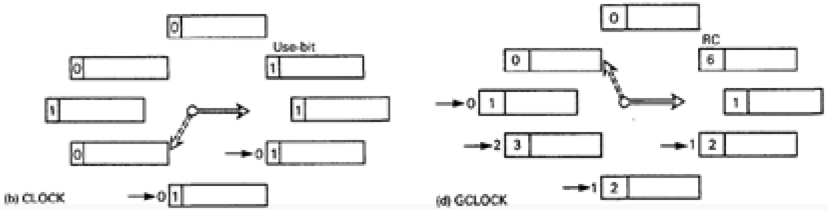


Least-Frequently-Used (LFU)

- Number of all references
- Replace page with lowest number of references
- Multiple pages may have same counter

CLOCK/GCLOCK

- Mimics LRU with FIFO like implementation
- Use-Bits or generalized with reference counters



Storage Management

Dresden Database Research Group

Record

- Package of fields that together describe a thing, a person, a fact, etc.
- Each fields represents on property of the entity described by the record
- Similar to a struct in C
- Variable length (in contrast to pages)

StudentID	LastName	FirstName	Birthday	City
1001	Schmidt	Hans	24.2.1990	Würzburg
1002	Meisel	Dirk	17.8.1989	Schweinfurt
1003	Schmidt	Amelie	19.9.1992	Rimpar
1004	Krause	Christian	3.5.1990	Würzburg
1005	Schäfer	Julia	30.3.1993	Kitzingen
1006	Rasch	Lara	30.3.1992	Würzburg
1007	Bakowski	Juri	15.7.1988	Schweinfurt

Single Record
Single Tuple

```
struct Person {
    int StudentID;
    char *LastName;
    char *FirstName;
    Date Birthday;
    char *City;
}
```

Each tuple/record of a table
is organized in a contiguous way

Record

Record

- Package of fields that together describe a thing, a person, a fact, etc.
- Each fields represents on property of the entity described by the record
- Similar to a struct in C
- Variable length (in contrast to pages)

Record Manager

- Organizes physical storage of records in pages
- Operations: Get, Insert, Update, Delete, Scan
- Agnostic to record structure and semantic; records considered as byte strings of variable length
- Structure and content of record is defined be Access System and application

Challenges

- Record addressing
- Free space management

Record Addressing

Dresden Database Research Group

Record address

- Identifier for records, used to address records, e.g., in indexes or query processing
- Assigned during insert of a record

Goals

- Stability of identifier
- Fast and direct access
- Less organizational overhead

Direct addressing

- Byte address or position number in file or page
- Instable
  - Byte address: If record grows in length, following records would get new address
  - Position number: Insert and delete operations change series or records

Indirect addressing

- Surrogate with mapping table (complete indirection)
- Tuple Identifier (TID concept)

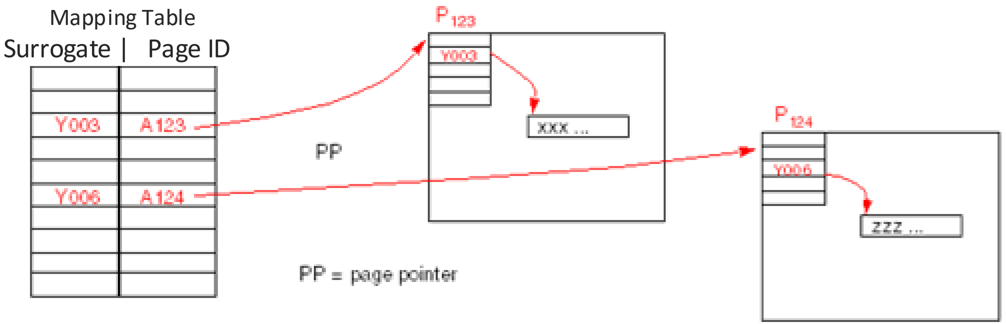
Surrogate with Mapping Table

Surrogate

- Record type + serial number
- Serial number remains constant during record's life time

Mapping table

- Maps surrogate to page



Problems

- Where to store mapping table?
- How can it be extended?
- How to search mapping table efficiently?

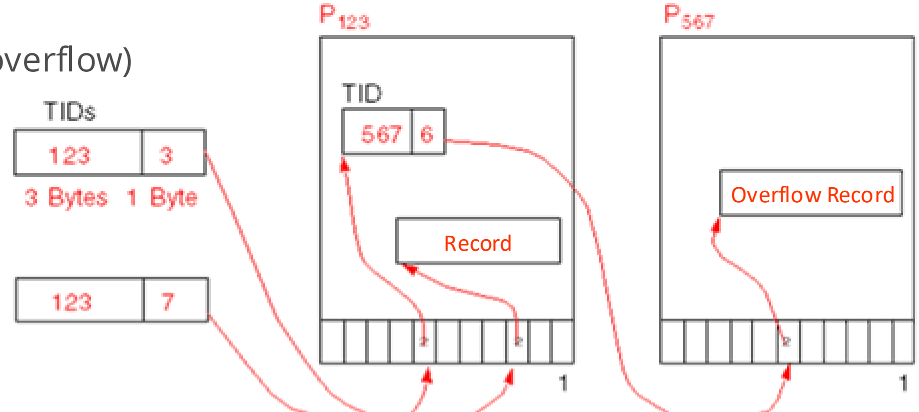
TID Concept

Record addressing with indirection inside the page

- Each page contains an array with record positions
- TID of a record consist of page id and index in position array

Pros

- Access with one page access (two pages in case of overflow)
- Stable
- No mapping table required



Operations

- Insert: Reuse unused position or add position
- Delete: Mark position as unused in array
- Update: Update all positions in array
- Update with overflow: Store record as overflow record and store TID of overflow record at original position (No double overflow: Update TID at original position)

Free Space Management

Problem

- In which page is enough space for new record?

Solution

- Free space table lists for all pages how much space is left

Free space value

- Precise value: Ceil(Log_2(page size)) => 2 bytes for common page size of 4K
- Rough value: use less bytes, free space = (value / page size)*2^(bits per value)

Free space table

- With direct page addressing
  - Assuming a single page can take n free space entries
  - First page and each (n+1)-th page takes free space entries
- With indirect page addressing
  - Free space information stored in page table

Access Layer – Index Structures

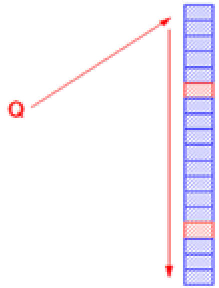
Overview Indexes

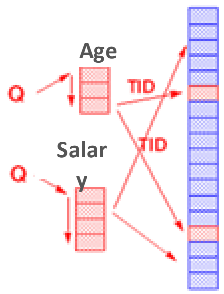
Dresden Database Research Group

Pers(PID, NAME, AGE, SALARY)

Table scan

- Read all pages and for each record evaluate the search criteria
- Pre-fetching



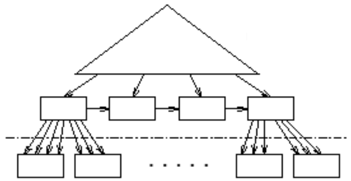


Index Scan

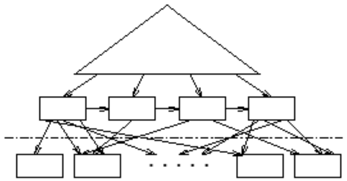
- Use index for search criteria
  on one or more attributes
- Fast access to single values or value ranges of index attributes
- Logical/physical sorting of values of key attributes (depending on index structure)
- Enforcing uniqueness

Types if indexes

- Primary (Clustered) Index, determines physical organization; use for PK
- Secondary (Non-Clustered) Index, redundant access path



Primary Index



Secondary Index

Overview Indexes (2)

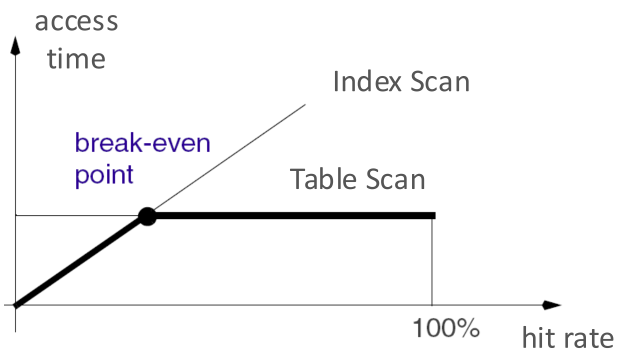
Dresden Database Research Group

Choice of Access Paths

- Index scan
  - Only useful for low selectivity (low number of result tuples)
  - Break even-point according to the output ratio of the number of tuples (usually max. 5%)
  - Requires statistics about data
  - Additional costs for index storage and updating
- Table Scan
  - adequate/efficient for small tables (e.g., 5 pages)
  - Queries with high selectivity (large result sets)
  - 100-200MB/s sequential read ~ 100 disk seeks/s

Reasons

- Logical order and physical order of tuples differ (non-clustered index)
- Random vs. sequential access to pages
- Page read per tuple vs. per page

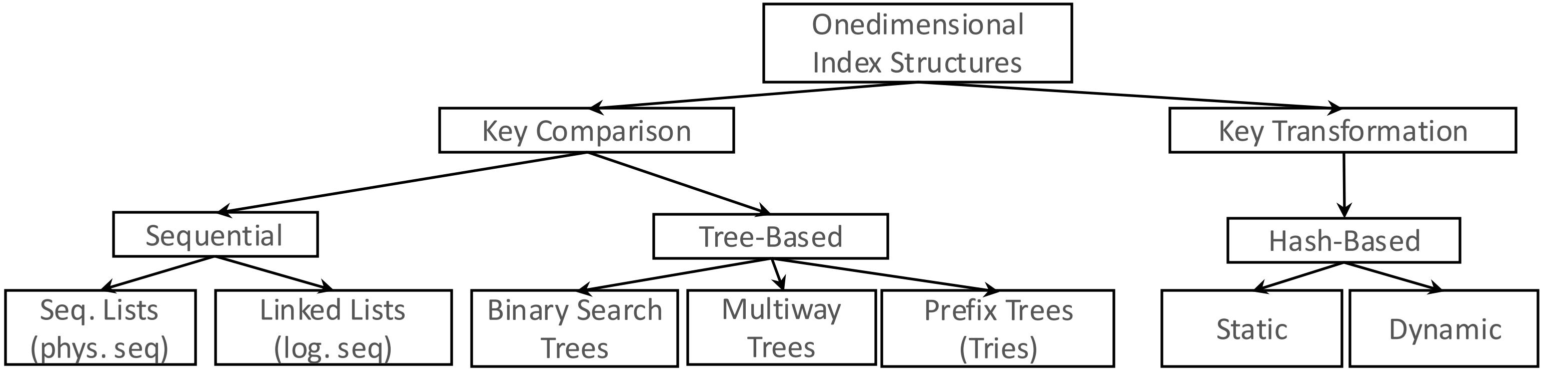


- Probability page is not read: $(1 - s)^b$
- Example:
  10% selectivity (s), 20 tuples/page (b)
  → 12% chance page is not read

Classification of Index Structures

Dresden Database Research Group

Classification



Example: B-Tree

Multiway Trees

- Tree structure with multiple children per node
- Idea: chose fan out so that node size suits page size

B-Tree

Dresden Database Research Group

History

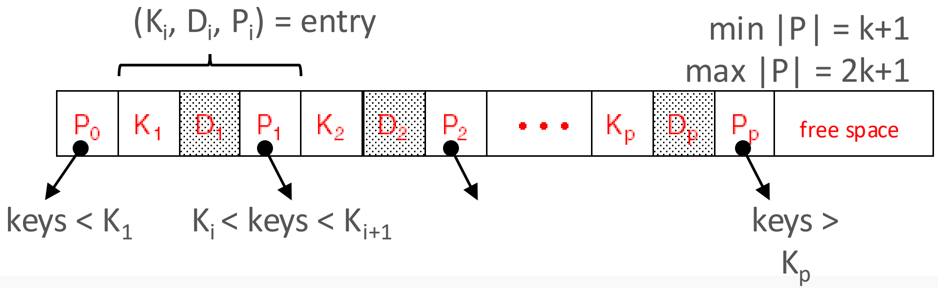
- ($\mathbf{B}$ayer, McCreight, 1972), **B**lock-based, **B**alanced, **B**oeing
- Multiway tree (node size = page size); designed for DBMS

B-tree of type (k,h) is a tree with the following three properties

- Each node (except for root and leaves) has at least k + 1 successors;
  root node is either a leaf or has at least 2 successors
- Each node has at most 2k + 1 successors; 2k entries
- Each path from the root to the leaf has the same length h (balanced)
  $\rightarrow \lceil \log_{2k+1}(n+1) \rceil \leq h \leq \left\lfloor \log_{k+1}\left(\frac{n+1}{2}\right) \right\rfloor + 1$

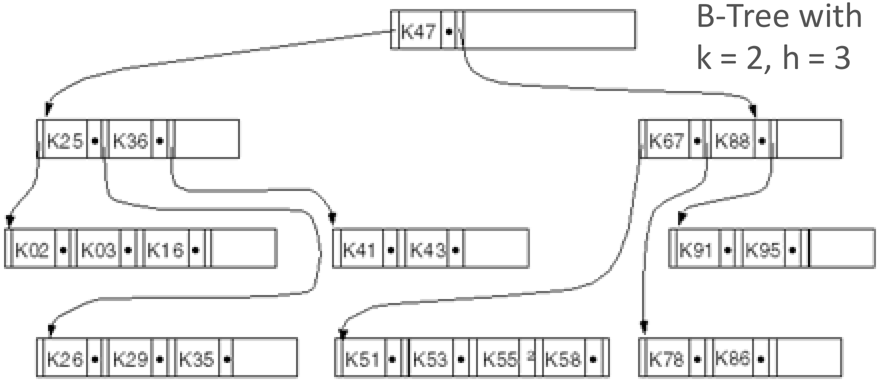
Page format

- K_i = key
- D_i = data (payload)
- P_i = pointer to a successor page



B-Tree (2)

Example



B-Tree with
k = 2, h = 3

Keys

- Agnostic to specific key semantic
- Only defined complete order required
- Could be of fixed or variable length

Payload

- Agnostic to specific data semantic
- Can be record or reference (TID) or mix

Operations

- Search for data for given key value
- Insertion and deletion of key-data pair

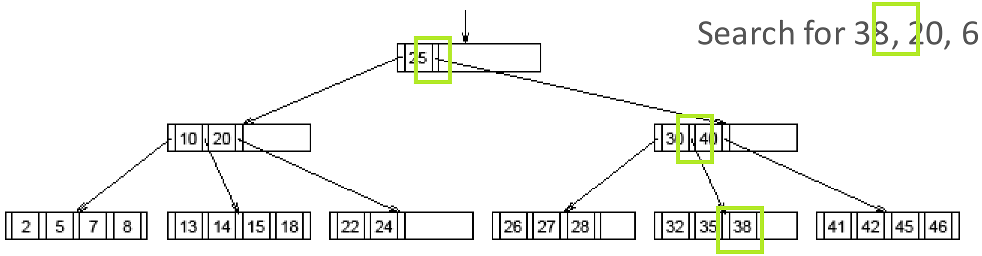
Search in the B-Tree

Starting at the root node, each node is searched from left to right

- 1) if K_i matches the desired key value, the data record has been found (further records with the same key value might be located in a sub-tree to which P_{i-1} points)
- 2) if K_i is smaller than the desired value, the search will be continued in the root of the sub-tree identified by P_{i-1}
- 3) if K_i is larger than the desired value, the comparison with K_{i+1} is repeated
- 4) if K_{2k} is also smaller than the desired value, the search will be continued in the sub-tree of P_{2k}

If it‘s impossible to descend further into a sub-tree (2. or 4.) (leaf node):

- The search is aborted, no record with the desired key value is found



Search for 38, 20, 6

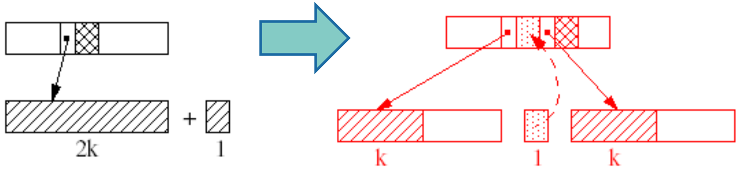
Insertions in the B-Tree (1)

Insertion

- Rule: insert only into leaf nodes!
- At Non-Leaf Nodes: descend down the tree as for the search
  - $S \leq K_i$: follow P_{i-1}
  - $S > K_i$: check K_{i+1}
  - $S > K_{2k}$: follow P_{2k}
- At Leaf Node
  - Insert the data record according to the sorting order
  - Special case: leaf node is full (2k records) → split the leaf node

Splitting

- Generate a new leaf node
- Split the 2k+1 entries (in order) into two leaf nodes
  - first k entries → left node
  - last k entries → right node
- middle entry (k+1-th) is used as new "discriminator" (branching) and inserted into the parent node



Insertions in the B-Tree (2)

Node Splitting during Insertion

- Two possible situations after a split
  - The parent node is full → repeat split on this level
  - Enough space → FINISHED
- Special case: root split
  - Split of the root node → New root with two successor nodes
  - Height of a tree grows by 1
  - The tree has been split from the bottom to the top

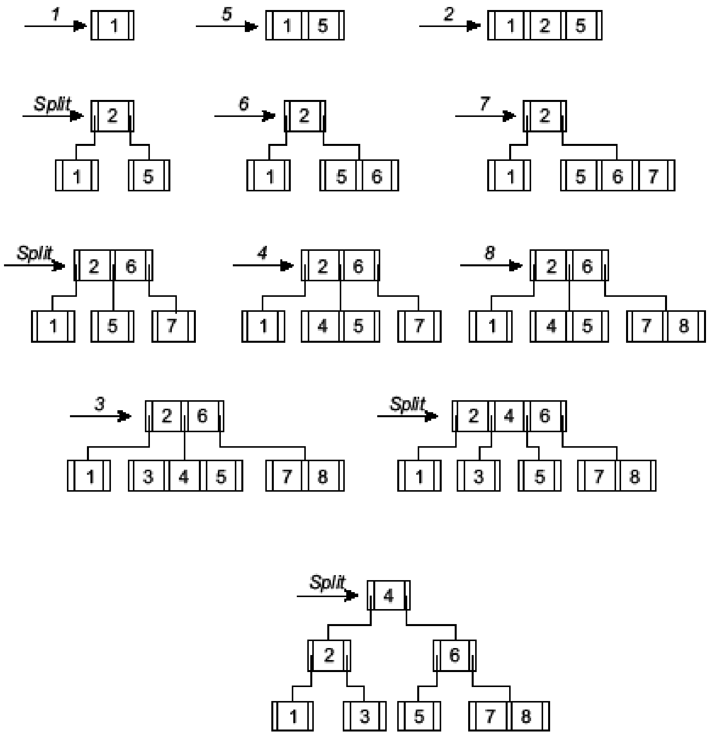
Dynamic reorganization (self-balancing)

- No unloading or loading necessary
- Tree is always balanced
- But: In case of many insertions / deletions reorganization can be beneficial

Insertions in the B-Tree (3)

Insertion Example

- Order k = 1, n=2k
- Keys: 1, 5, 2, 6, 7, 4, 8, 3



- Finally, h=3

Insertion and Deletion in the B-Tree

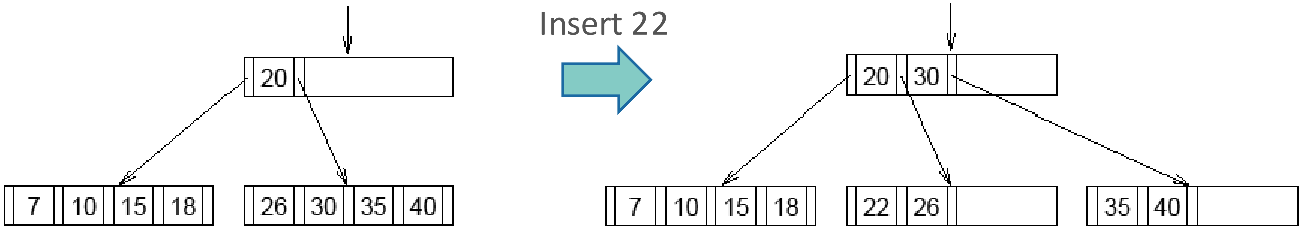
Dresden Database Research Group

Problem

- Insertion can create overflow
- Deletion can create underflow and overflow

Example:

- Insertion of key 22
  → Overflow → Split



- Deletion of key 22?
  → Underflow, need to access all four nodes, finally same as input

B-Trees, B$^+$-Trees, and B*-Trees

Dresden Database Research Group

B$^+$-Trees and B*-Trees

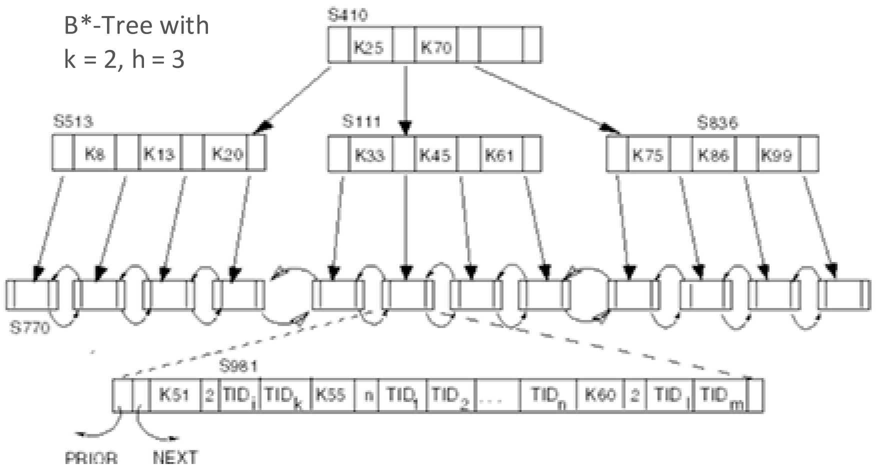
- Data is only in leaf nodes
  - Key redundancy, but higher fan-out → lower tree high, less I/O
  - Simpler delete procedure → requires only merging of nodes
- Double linked list of all leaf nodes

B*-Trees

- Modified valid node sizes: from [k,2k] to [4/3k,2k] → better node utilization, but more splits/merges

Example

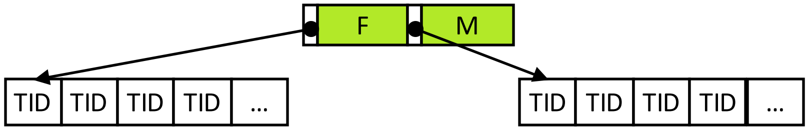
- Secondary index
- Non unique



Indexing Low Cardinality Columns

Problem

- Example: B-tree on the sex of customers for a table with 1,000,000 tuples results in two lists with approximately 500,000 tuples each



- Query for all female customers requires 500,000 random page accesses (secondary index!)
  → Table scan would be much faster

Conclusion

- B-trees (and also hashing) are useful for predicates with low selectivity (output/input cardinality ratio)
- Rule of thumb: margin hit rate is approx. 5%
- higher hit rates do not justify the efforts for an index access

Bitmap Index

Idea

- (Long history since the 1960s)
- Create a bitmap/bitlist for each attribute value
- Each tuple in the table is assigned to one bit in the bitmap (by position/ sequential TID)
- Bit values
  - 1 → attribute value set
  - 0 → attribute value not set
- Necessary condition:
  Sequential numbering of the tuples (TIDs)

Name	Sex	Region	Race
Carol	f	n	white
Harold	m	e	black
Anne	f	e	asian
Iris	f	ne	white
…	m	se	hisp
…	f	e	white
…	f	sw	asian
…	f	w	black
…	f	n	asian
…	m	e	hisp
…	m	se	black
…	f	s	white
…	m	nw	black
…	f	s	white
…	f	w	black

Sex

F	M
1	0
0	1
1	0
1	0
0	1
1	0
1	0
1	0
1	0
0	1
0	1
1	0
0	1
1	0
1	0

Querying Bitmap Indexes

Main advantage of bitmap indexes

- Simple and efficient logical join possible
- Read only data that is relevant for predicates
- Example:
  - $\sigma_{Sex='f' \wedge Region='n'}$ R
  - Bitmaps B1 and B2 in conjunction:

```
for (i=0; i<B1.length; i++)
  B = B1[i] & B2[i];
```

F		N		A		*
1		0		0		0
0		1		0		0
1	AND	1	AND	1	=	1
1		0		0		0
0		0		0		0
1		1		0		0
1		0		1		0
1		0		0		0
1		0		1		0
0		1		0		0
0		0		0		0
1		0		0		0
0		0		0		0
1		0		0		0
1		0		0		0

Example I/O Costs Estimation

- $\sigma_{Sex='f' \wedge Region='n' \wedge Race='Asian'}$ R(“Asian women of region North”)
- Selectivity: $1/2 \cdot 1/8 \cdot 1/4 = 1/64$
- N=10,000 tuples, with length of 400 bytes each
  (~ 10 tuples per page for 4kB pages)
- Table scan: 1000 pages
- Bitmap access: 10000/64 ➔ 156 pages (worst case: each tuple in a different page), +1 page for bitmaps

Access Layer – Record Format

Record Format

Example

StudentID^	LastName	FirstName	Birthday	City
1001	Schmidt	Hans	24.2.1990	Würzburg
1002	Meisel	Dirk	17.8.1989	Schweinfurt
1003	Schmidt	Amelie	19.9.1992	Rimpar
1004	Krause	Christian	3.5.1990	Würzburg
1005	Schäfer	Julia	30.3.1993	Kitzingen
1006	Rasch	Lara	30.3.1992	Würzburg
1007	Bakowski	Juri	15.7.1988	Schweinfurt

```
struct Person {
    int StudentID;
    char *LastName;
    char *FirstName;
    Date Birthday;
    char *City;
}
```

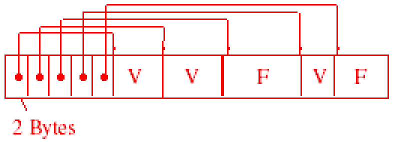
Record Format

Fixed-length fields

- Space inefficient
- Inflexible

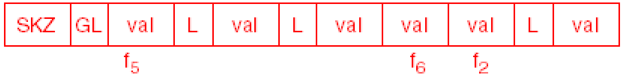
Pointer in header

- Pointer resolution also for fixed-length field necessary



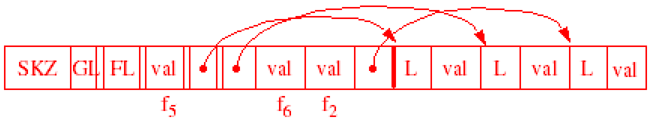
Length fields

- Pointer resolution also for fixed-length field necessary if not at the beginning of the record



Pointer and Length fields

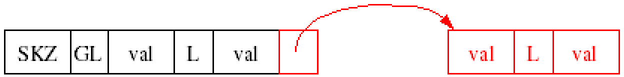
- Variable fields do not affect fixed-length fields
- No pointer resolution for fixed-length fields necessary



Record Format

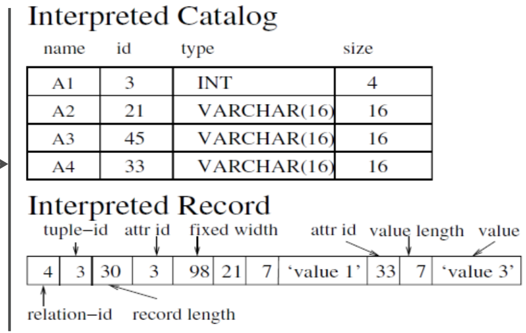
Splitting records

- Record identifier (e.g., TID) points to next part of record
- Necessary for records longer than a single page
- Necessary for records with LOB field (picture, text, etc.)
- Could also be used in case updated record does fit in original page



Null values

- Bit list (1 means field is NULL) in header of record (most common approach)
  - Field value is omitted
- For variable-length field: length can be used
  - Attention: depends on data type, e.g., an empty string != NULL
- For vary sparse data: Store pairs of field id and value

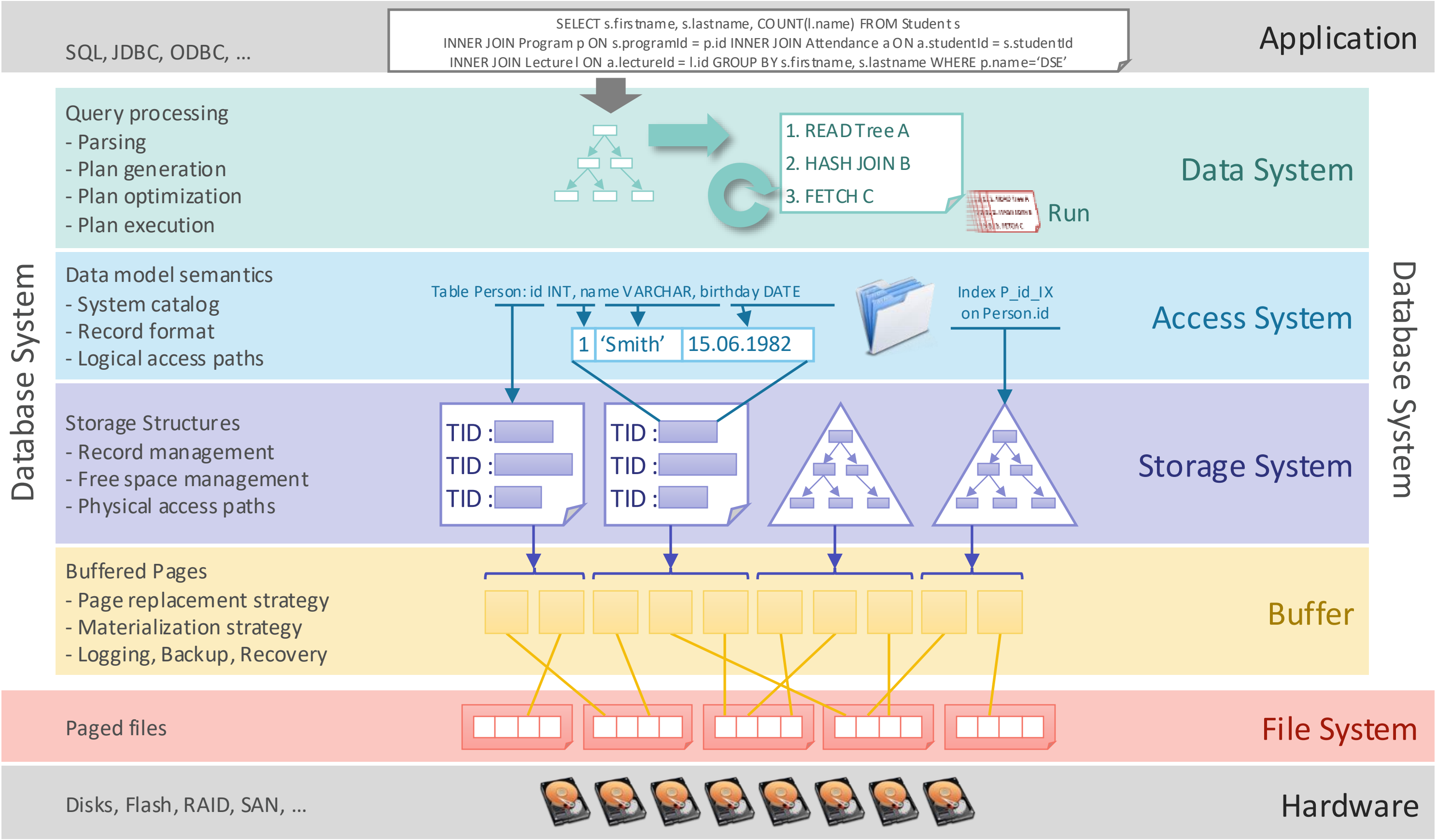


- [Beckmann et al.: Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format, *ICDE'06, IEEE Computer Society*, **2006**]

Data System

Architectural Blue Print

Dresden Database Research Group



Example Query

Dresden Database Research Group

SQL Query (What!)

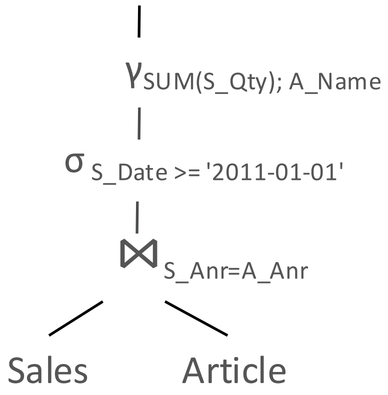
```
SELECT A_Name, SUM(S_Qty)
FROM Article, Sales
WHERE A_Anr = S_Anr AND
      S_Date >= '2011-01-01'
GROUP BY A_Name
```

YES, but HOW do we get there (efficiently)?

Result

A_Name	SUM
Article A	12
Acticle B	1

Query Plan (How!)



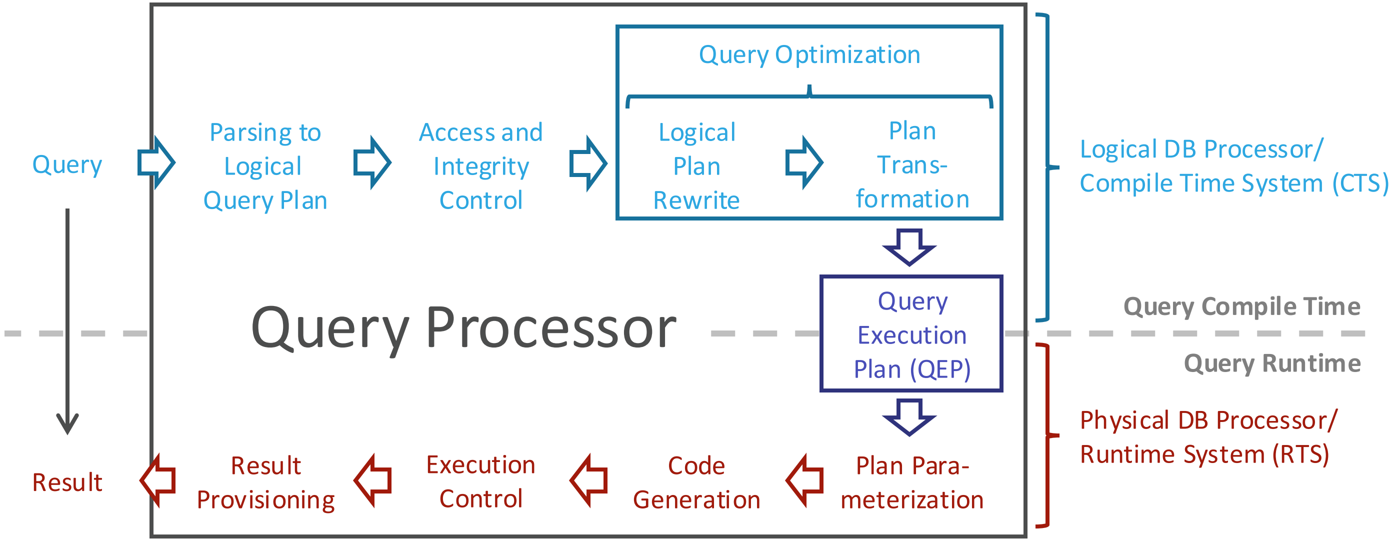
Sales

...	S_Anr	S_Date	S_Qty
...	1	2010-09-20	7
...	1	2011-01-17	2
...	1	2011-02-21	5
...	2	2011-02-22	1
...	1	2011-03-07	5

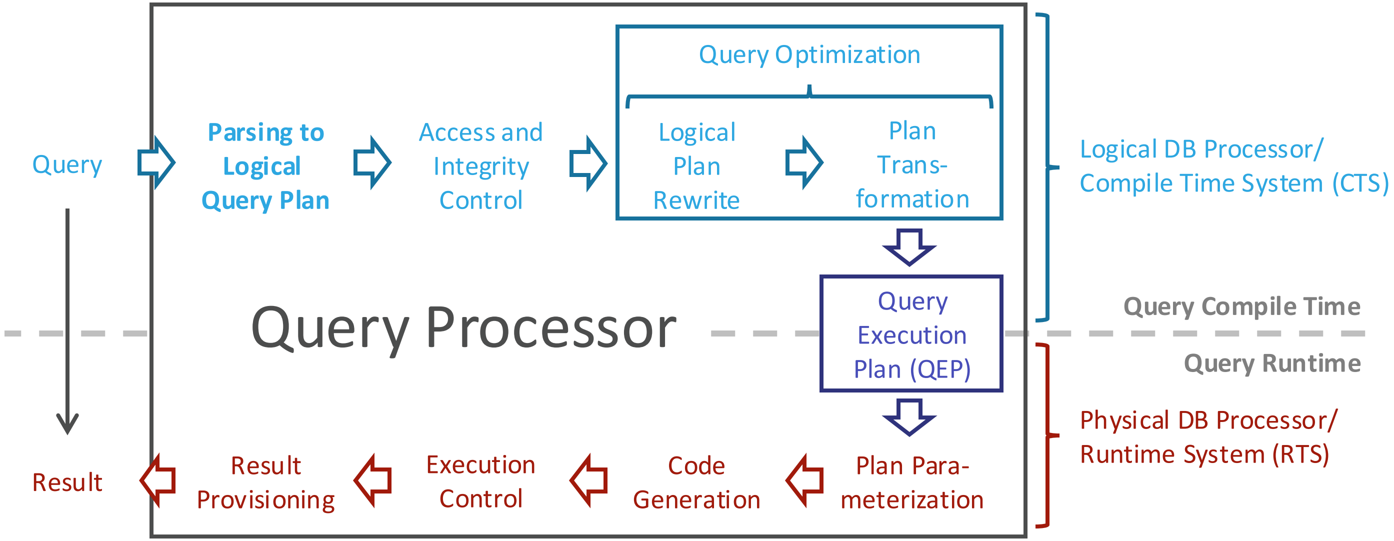
Article

A_Anr	A_Name	...
1	Article A	...
2	Acticle B	...

Query Processing



Query Processing



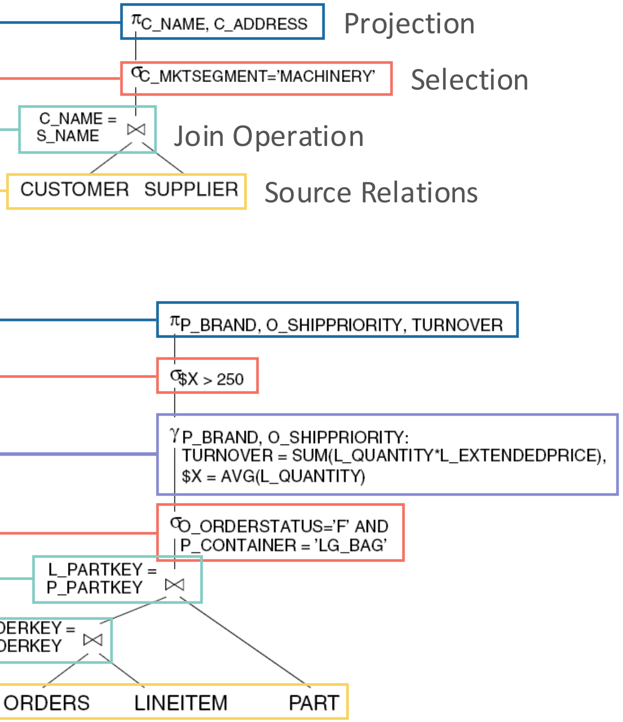
Logical Query Plan

Mono-Block Query

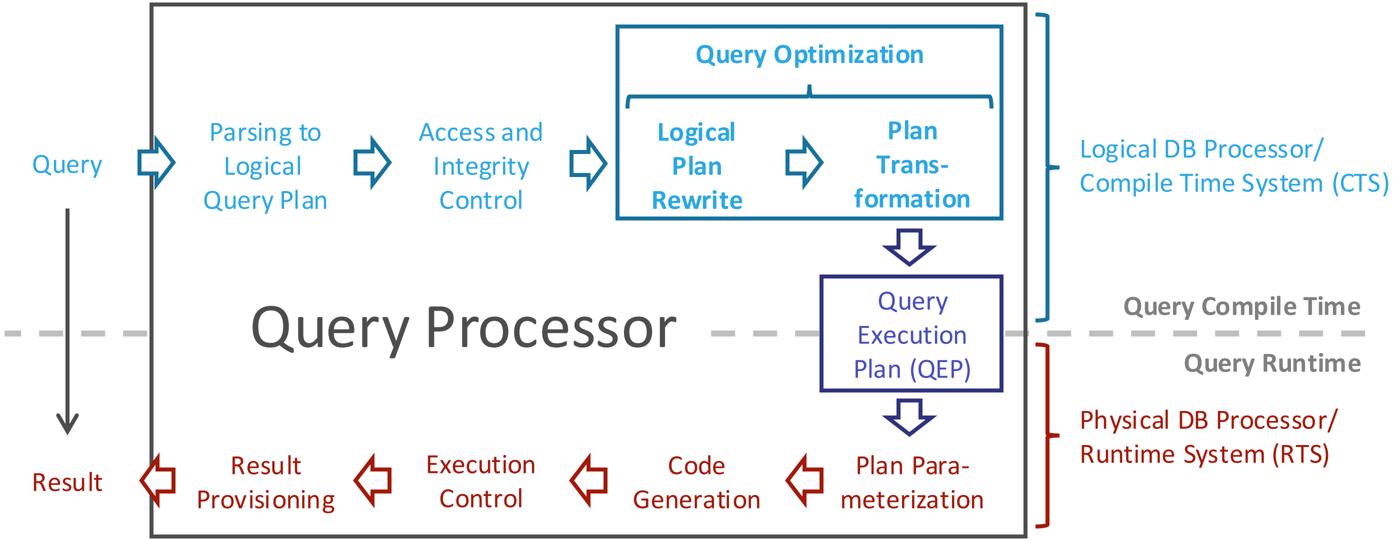
```
SELECT C_NAME, C_ADDRESS
  FROM TPCD.CUSTOMER, TPCD.SUPPLIER
 WHERE C_NAME = S_NAME
   AND C_MKTSEGMENT = 'MACHINERY';
```

Star Query

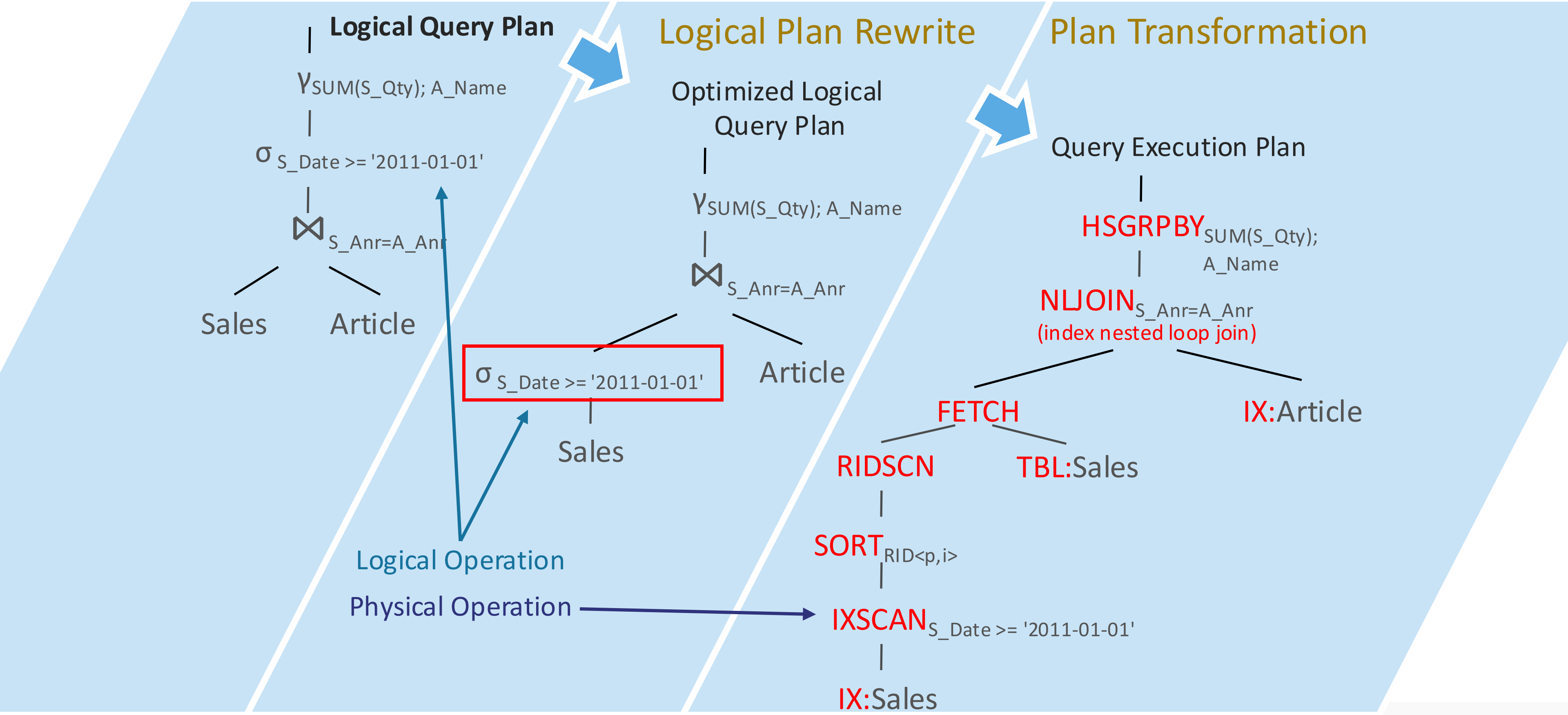
```
SELECT
  P_BRAND, O_SHIPPRIORITY,
  SUM(L_QUANTITY*L_EXTENDEDPRICE)
     AS TURNOVER
FROM
  TPCD.LINEITEM, TPCD.ORDERS, TPCD.PART
WHERE L_ORDERKEY = O_ORDERKEY
  AND L_PARTKEY = P_PARTKEY
  AND O_ORDERSTATUS = 'F'
  AND P_CONTAINER = 'LG_BAG'
GROUP BY
  P_BRAND, O_SHIPPRIORITY
HAVING
  AVG(L_QUANTITY) > 250;
```



Query Processing

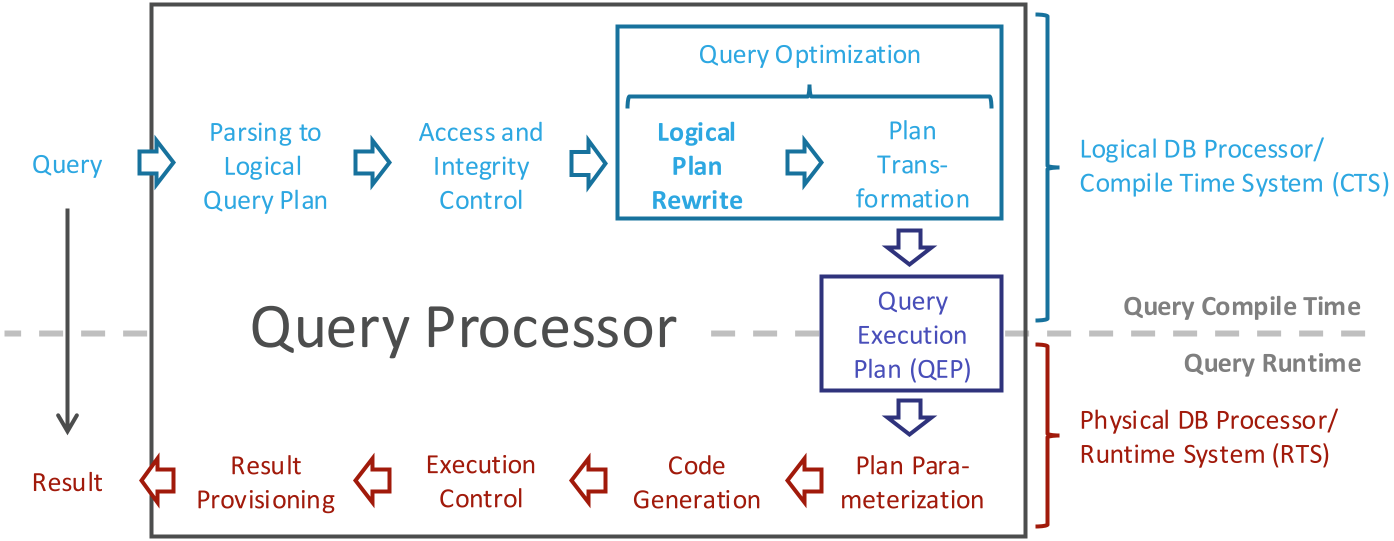


Query Optimization – Example



Query Processing

Dresden Database Research Group



Logical Plan Rewrite

Goal

- Get a better but logically equivalent plan
- OR: Solve the query as far as possible without reading data

Basis

- Equivalences in relational algebra and logic
- Knowledge from integrity rules and data statistics

Steps

- 1) Standardization and simplification
- 2) Un-nesting sub-queries
- 3) Query rewriting rules

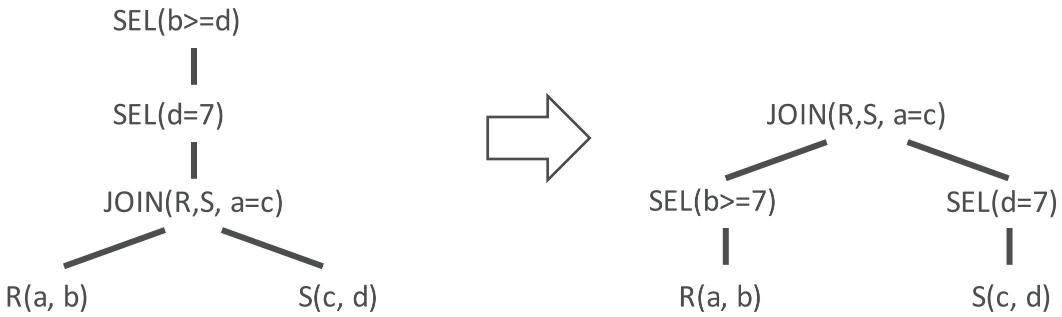
Standardization and Simplification

Dresden Database Research Group

Standardization

- Conjunctive normal form (P_{11} OR ... OR P_{1n}) AND ... AND (P_{m1} OR ... OR P_{mp})
- Disjunctive normal form (P_{11} AND ... AND P_{1q}) OR ... OR (P_{r1} AND ... AND P_{rs})
- Others: e.g., Prenex normal form (quantifiers are shifted)

Example:



Standardization and Simplification

Simplification

- Equivalent expressions can have a different degree of redundancy
  - Idempotency rules, Expressions with “empty relations”
- Treatment/elimination of common sub-expressions
  - $(A_1 = a_{11} \text{ OR } A_1 = a_{12}) \text{ AND } (A_1 = a_{12} \text{ OR } A_1 = a_{11})$
- Propagation of constants (closure of the qualification predicates)
  - $A \geq B \text{ AND } B = \text{const.} \Rightarrow A \geq \text{const. AND } B = \text{const.}$
- Expressions that cannot become true
  - $A \geq B \text{ AND } B > C \text{ AND } C \geq A \Rightarrow A > A \rightarrow \text{false}$
- Use of information on semantic integrity requirements
  - A is primary key/unique: $\pi_A \rightarrow$ no duplicate elimination necessary
  - Rules: MAR_STATUS = ‘married’ AND TAX_CLASS ≥ 3
    $\Rightarrow$ (MAR_STATUS = ‘married’ AND TAX_CLASS = 1) → false

Standardization and Simplification

Dresden Database Research Group

Simplification (cont.)

- Transformation and idempotency rules for Boolean expressions

Rule Name	Examples	
Commutativity rules	A OR B ⇔ B OR A	A AND B ⇔ B AND A
Associativity rules	(A OR B) OR C ⇔ A OR (B OR C) (A AND B) AND C ⇔ A AND (B AND C)	
Distributivity rules	A OR (B AND C) ⇔ (A OR B) AND (A OR C) A AND (B OR C) ⇔ (A AND B) OR (A AND C)	
De Morgan's rules	NOT (A AND B) ⇔ NOT (A) OR NOT (B) NOT (A OR B) ⇔ NOT (A) AND NOT (B)	
Double-negation rules	NOT(NOT(A)) ⇔ A	
Idempotency rules	A OR A ⇔ A A OR NOT(A) ⇔ TRUE A AND (A OR B) ⇔ A A OR FALSE ⇔ A A AND FALSE ⇔ FALSE	A AND A ⇔ A A AND NOT (A) ⇔ FALSE A OR (A AND B) ⇔ A A OR TRUE ⇔ TRUE

Standardization and Simplification

Simplification (cont.)

- Example equivalences in relational algebra used for simplification

Original	Simplified
$R \bowtie R$	R
$R \cup R$	R
$R - R$	$\emptyset$
$R \bowtie (\sigma_p R)$	$\sigma_p R$
$R \cup (\sigma_p R)$	R
$R - (\sigma_p R)$	$\sigma_{\neg p} R$
$(\sigma_{p1} R) \bowtie (\sigma_{p2} R)$	$\sigma_{p1 \wedge p2} R$
$(\sigma_{p1} R) \cup (\sigma_{p2} R)$	$\sigma_{p1 \vee p2} R$
$(\sigma_{p1} R) - (\sigma_{p2} R)$	$\sigma_{p1 \wedge \neg p2} R$

$\bowtie$ … natural join
$\cup$ … union distinct

Un-Nesting of Sub-Queries

Dresden Database Research Group

Un-Nesting

- Transformation rules for quantified expressions
- Un-nesting of sub-queries

Case 1: Type-A Nesting

- Inner block is not correlated and computes the single aggregate value
- Solution: Computation of the aggregate value and insertion into the outer query

```
SELECT OrderNo
 FROM Order
 WHERE ProdNo = (SELECT MAX(ProdNo)
                  FROM Product
                  WHERE Price < 100)
```

⇒

```
$X = SELECT MAX(ProdNo) FROM Product
     WHERE Price < 100

SELECT OrderNo
     FROM Order
     WHERE ProdNo = $X
```

Case 2: Type-N Nesting

- Inner block is not correlated and returns a set of tuples
- Solution: Transformation into a symmetric form

```
SELECT OrderNo
 FROM Order
 WHERE ProdNo IN (SELECT ProdNo
  FROM Product WHERE Price < 100)
```

⇒

```
SELECT OrderNo
 FROM Order O, Product P
 WHERE O.ProdNo = P.ProdNo
   AND P.Price < 100
```

Un-Nesting of Sub-Queries (2)

Dresden Database Research Group

Case 3: Type-J Nesting

- Un-nesting of correlated sub-queries

```
SELECT OrderNo
  FROM Order O
 WHERE ProdNo IN
       (SELECT ProdNo FROM Project P
         WHERE P.ProjNo = O.OrderNo
           AND P.Budget > 100,000)
```

⇨

```
SELECT OrderNo
  FROM Order O, Project P
 WHERE O.ProdNo = P.ProdNo
   AND P.ProjNo = O.OrderNo
   AND P.Budget > 100,000
```

Case 4: Type-JA Nesting

- Un-nesting of correlated sub-queries with aggregation

```
SELECT OrderNo
  FROM Order O
 WHERE ProdNo IN
       (SELECT MAX(ProdNo)
          FROM Project P
         WHERE P.ProjNo = O.OrderNo
           AND P.Budget > 100,000)
```

⇨

```
SELECT OrderNo
  FROM Order O
 WHERE ProdNo IN
       (SELECT ProdNo FROM
          (SELECT ProjNo, MAX(ProdNo)
           FROM Project
           GROUP BY ProjNo) P
        WHERE P.ProjNo = O.OrderNo
          AND P.Budget > 100.000)
```

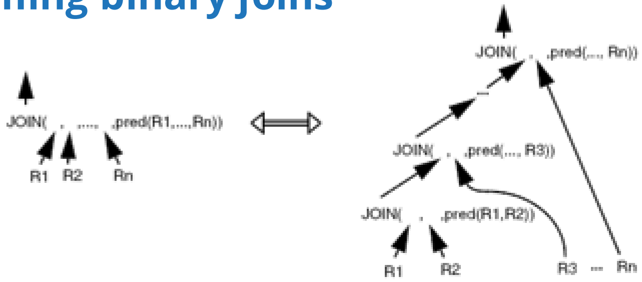
Type-J Nesting

Type-A Nesting

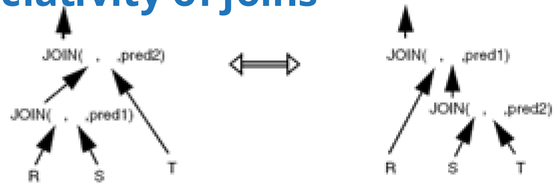
- Further un-nesting analogously to Case 3 and Case 1

Query Rewriting Rules

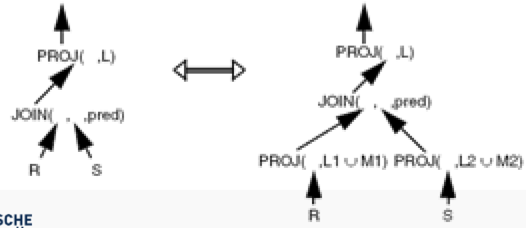
Forming binary joins



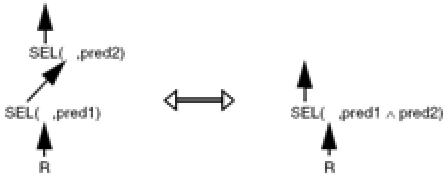
Associativity of joins



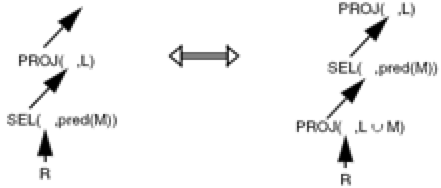
Exchange of projection and join



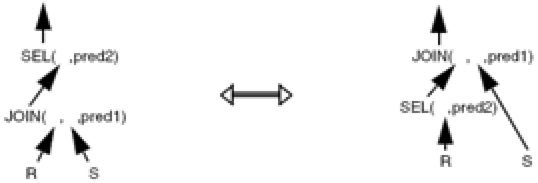
Grouping selections



Exchange selection and projection



Exchange of selection and join



Simplified Restructuring Algorithm

Dresden Database Research Group

Pre-Condition

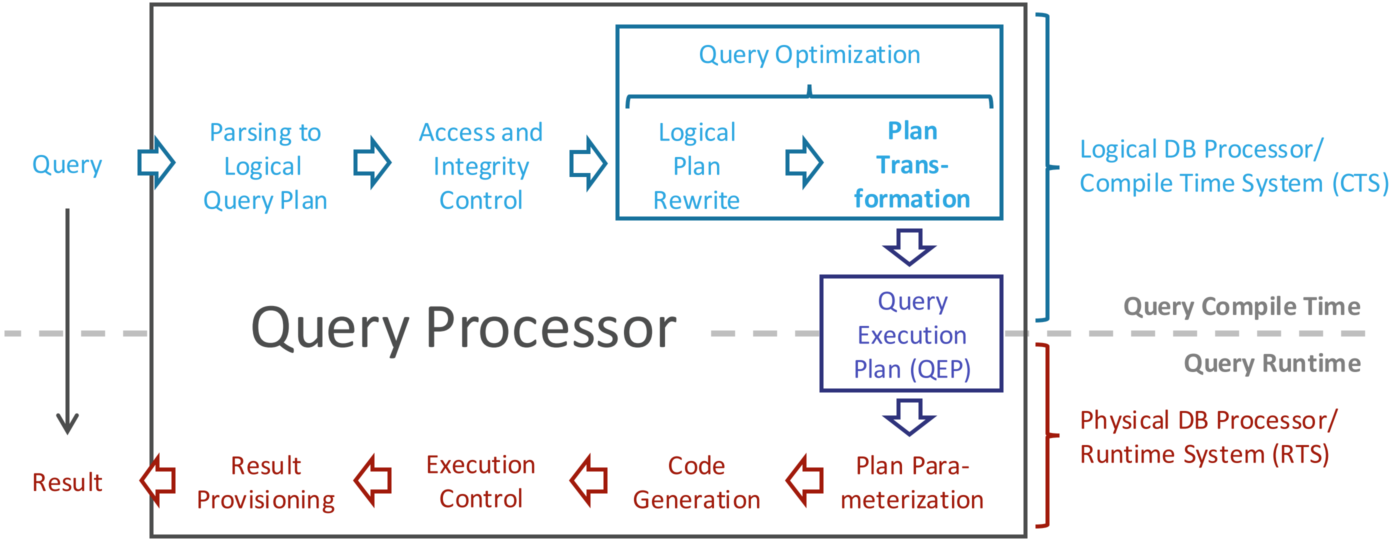
- Standarized, simplified, and unnested query graph

Algorithm

- Applies restructuring rules in order to prepare the query graph for the optimizer

- 1) Split complex n-ary join operations into binary joins
- 2) Split complex multi-term selections into single-term selections
- 3) Push-down selections to the leafs (as far as possible)
- 4) Group adjacent single-term selections again (e.g., predicates on single relation)
- 5) Push-down projection to the leafs (as far as possible) but try to avoid duplicate elimation operations

Query Processing

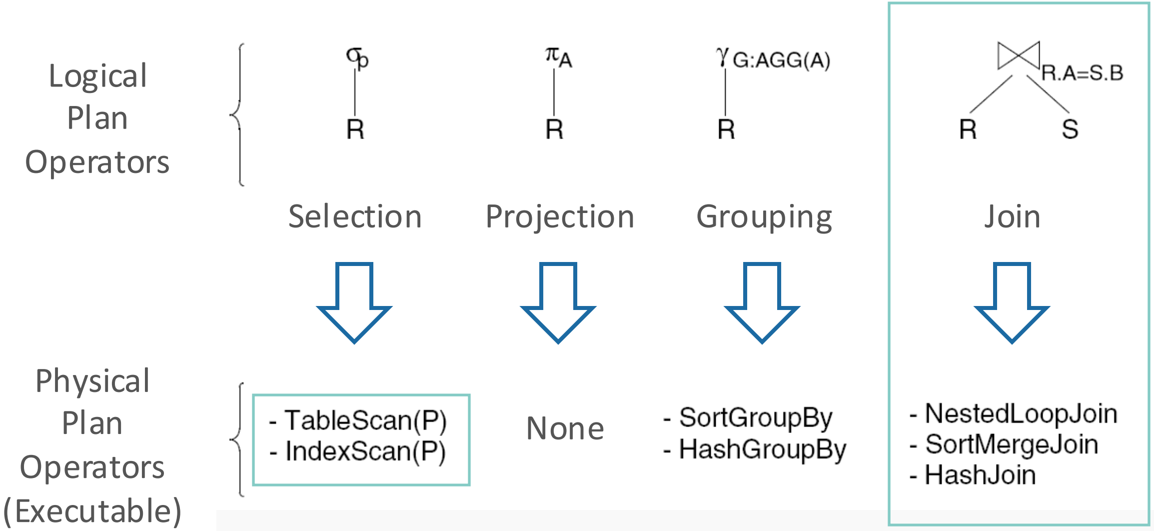


Plan Operators

Overview

- Multiple physical plan operators per relational operator (different use cases)
- Additional physical operators (e.g., TEMP)

Examples (supported in most DBMS)



Nested Loop Join

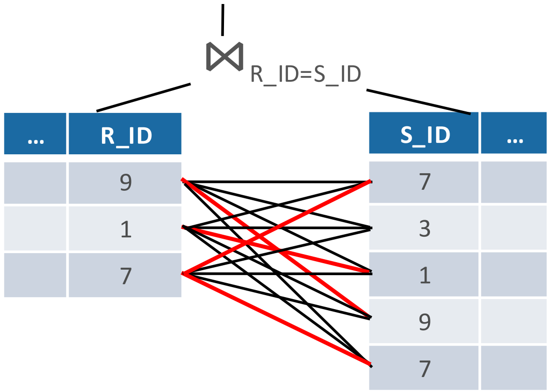
Dresden Database Research Group

Overview

- General case: no sorting order, no index structures, arbitrary join predicates

Algorithm / Example

```
Scan R
For each r in R
  Scan S
  For each s in S
  if ( R_ID ⊝ S_ID )
    output concat(r,s)
```



N = |R|
M = |S|

Note

- Block Nested Loop Join; Index Nested Loop Join (O(N log M))
- Natural Join vs. Equi Join

Complexity

- Time: $O(N \cdot M)$
- Space: $O(1)$

Sort-Merge Join

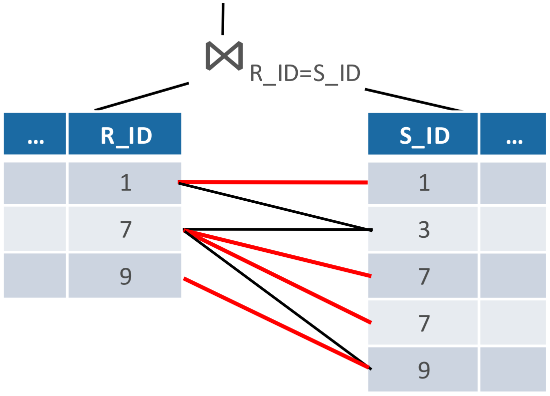
Dresden Database Research Group

Overview

- Exploit sorting order/index structures, only equality predicates

Algorithm

- Phase 1: Sort
```
Scan R, Sort R
Scan S, Sort S
```
- Phase 2: Merge
```
Step-by-step Scan (R,S)
if ( R_ID ⊖ S_ID )
  output concat(r,s)
```



Notes

- Produces sorted output
  → affecting following operators

Complexity

	unsorted	sorted
▪ Time:	O(N log N + M log M)	O(N + M)
▪ Space:	O(N + M)	O(1)

Hash Join

Dresden Database Research Group

Overview

- No sorting order, no index structures, only equality predicates

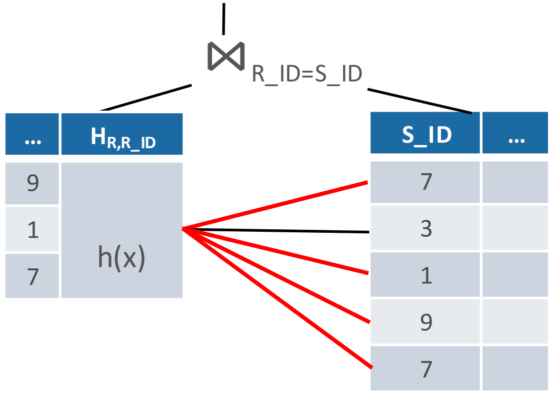
Algorithm

- Phase 1: Building

```
Scan R
Build HR
```

- Phase 2: Probing

```
Scan S
For each s in S
  For each r in HR.get( s.S_ID )
    output concat(r,s)
```



Notes

- Classic hashing (p tbl fragments);
- simple hashing (p value-based tbl fragments)

Complexity

- Time: O(N + M)
- Space: O(N)

Plan Transformation

Non-algebraic / physical optimization

- Selecting the access paths
- Mapping relational operators to physical plan operators
- Selecting the execution order of plan operators

Relevant Sub-Problems

- Different implementations (e.g. join) or mapping variants (e.g. index use)
- Grouping of directly adjacent operators into a single plan operator
- Join sequence for join operations
  - Goal: minimal costs for the operation sequence
  - Heuristics: minimization of the size of intermediate results, i.e. the smallest (intermediate) relations are always joined first
- Detection of common sub-trees
  - Compute only once
  - Necessary for this: Buffering of the intermediate result relation

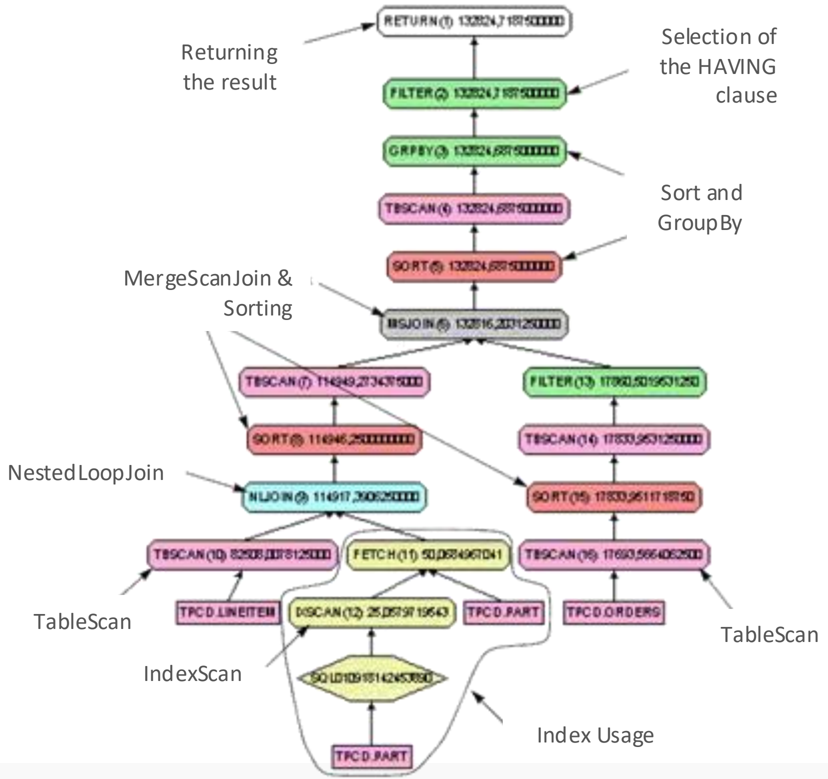
Example: Query Execution Plan

IBM DB2

Dresden Database Research Group

Query

```
SELECT
  P_BRAND, O_SHIPPRIORITY,
  SUM(L_QUANTITY*L_EXTENDEDPRICE)
     AS TURNOVER
FROM
  TPCD.LINEITEM,
  TPCD.ORDERS,
  TPCD.PART
WHERE L_ORDERKEY = O_ORDERKEY
  AND L_PARTKEY = P_PARTKEY
  AND O_ORDERSTATUS = 'F'
  AND P_CONTAINER = 'LG_BAG'
GROUP BY
  P_BRAND, O_SHIPPRIORITY
HAVING
  AVG(L_QUANTITY) > 250;
```



Plan Transformation (2)

Input

- Algebraically optimized logical query plan
- Existing access paths and meta data
- Cost model and statistics

Output

- Optimal (or at least: good) query execution plan

Framework conditions

- Fatal assumptions (in general wrong)
  - No Skew: Attribute values are equally distributed
  - Independence/No correlation: Predicates in queries are independent
  - Example: 10 Manufacturers, 100 Models, 10000 Cars
- Limited resources
  - Costs of query optimization should not exceed query execution cost improvements

	estimated	real
$\sigma_{Model=\text{'Golf'}}$	10	980
$\sigma_{Make=\text{'VW'}}$	1000	5000
Cars	10000	10000

Cost Model

Dresden Database Research Group

Cost model

- Basis for cost estimation of a query execution plan
- Highly influences the quality of query optimization
- Operator-specific cost formulas
- Uses real and estimated statistics (e.g., cardinalities, selectivities)

Different cost types (not independent, often weighted)

- Computation costs
  - CPU costs
  - Path lengths
- I/O costs
  - Number of physical references
- Memory costs
  - Temporary memory allocation in the DB buffer and in external memory
- Communication costs (distributed DBS)
  - Number of messages
  - Number of data to be transferred

Cost Model Example

Assumed Cost Model

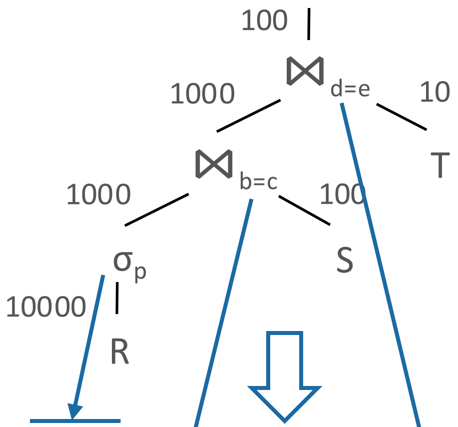
- Selection: $C(\sigma_p(e_1)) = |e_1|$
- Nested Loop Join: $C(e_1 \bowtie e_2) = |e_1| + f \cdot |e_1| \cdot |e_2|$

Cost Computation

- Use known base relation cardinalities
- Estimate size of intermediate results (via selectivities)
- Compute total plan costs as sum of operator costs

Example

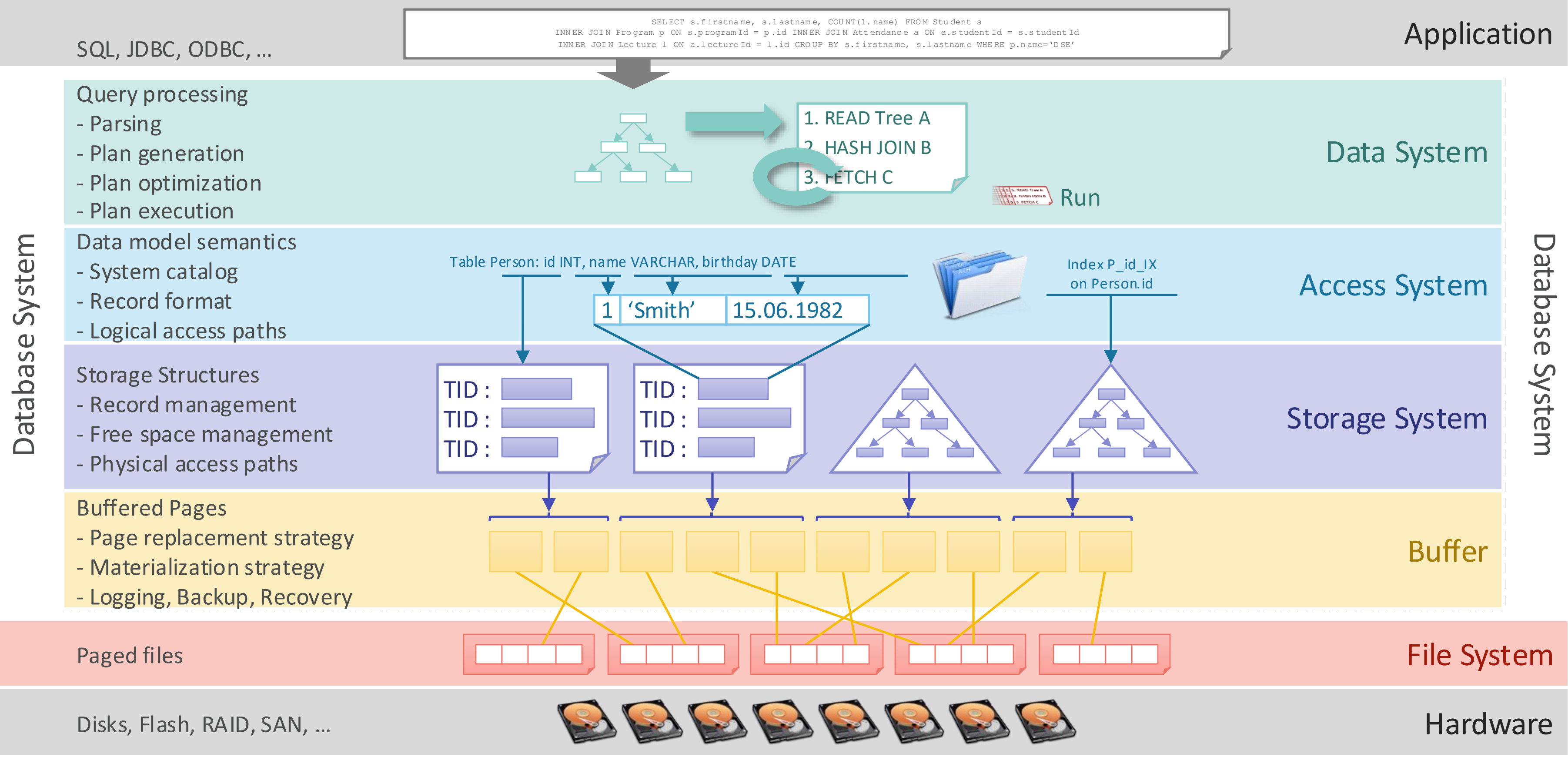
- Query Q: $((\sigma_p(R) \bowtie S) \bowtie T)$
- Statistics:
  $|R|=10000$, $|S|=100$, $|T| = 10$,
  $f_p=0.1$, $f_{b,c}=0.01$, $f_{d,e}=0.01$



Cost(Q) $= 10000$
$+ (1000 + 0.01 \cdot 1000 \cdot 100)$
$+ (1000 + 0.01 \cdot 1000 \cdot 10)$
$= 13100$

Summary

Architectural Blue Print



Homework

Query Optimizers: Time to Rethink the Contract?

Read the Paper

- 03-QueryOptimizer.pdf (in OPAL)

Query Optimizers: Time to Rethink the Contract?

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Query Optimization is expected to produce good execution plans for complex queries while taking relatively small optimization time. Moreover, it is expected to pick the execution plans with rather limited knowledge of data and without any additional input from the application. We argue that it is worth rethinking this prevalent model of the optimizer. Specifically, we discuss how the optimizer may benefit from leveraging rich usage data and from application input. We conclude with a call to action to further advance query optimization technology.

While there are no easy solutions to these problems, one line of thinking that has not been explored is revisiting the contract with the optimizer. The contract, as defined in [1], is well-intentioned as it imposed the least burden on applications: The optimizer will produce *high-quality execution plans* for all queries while taking *relatively small optimization time* with *limited additional input* such as histograms. But, by virtue of this contract, optimizers are also by design “closed” to additional information that can potentially help lessen the difficulties of the challenges mentioned above.