Universität Konstanz
Fachbereich Informatik & Informationswissenschaft
Stephan Heidinger

Database Systems Architecture and Implementation
WS 2014/15
15. Februar 2015

ಠ_ಠ Find any errors? Please send them back, I want to keep them!}

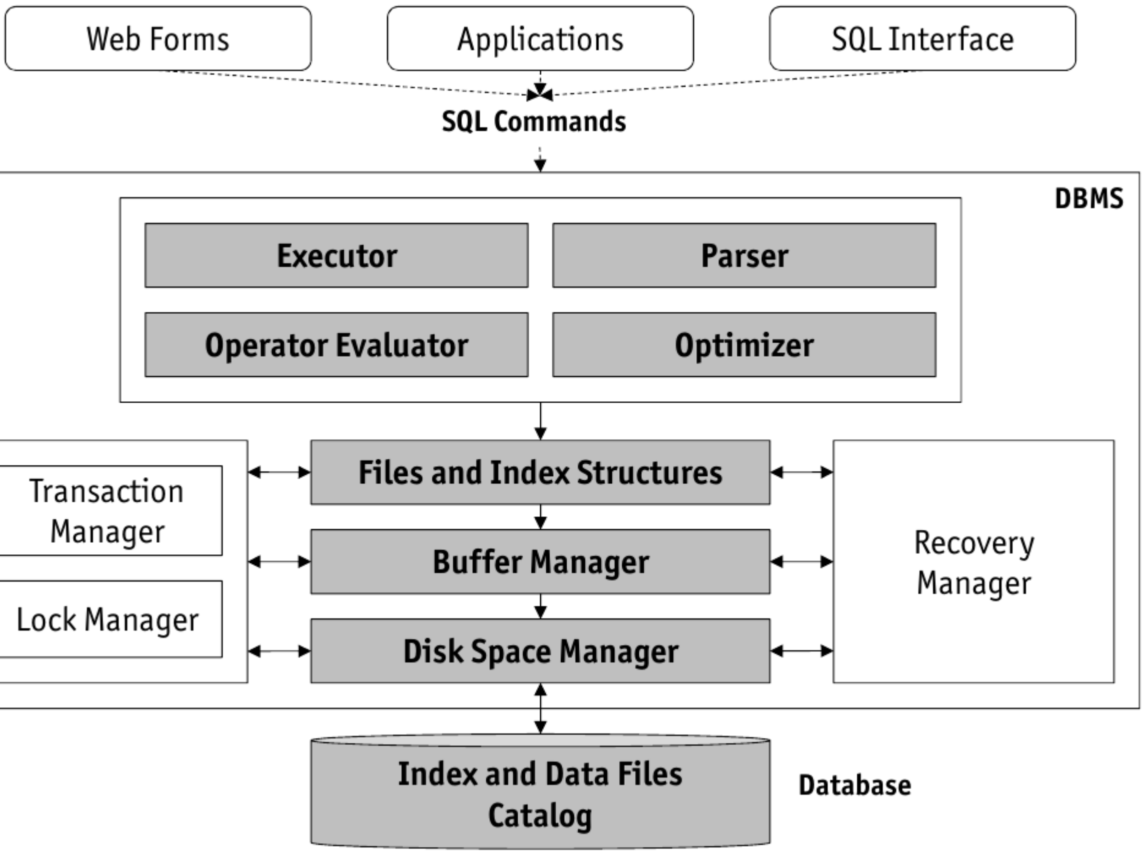


Abbildung 1: Course Structure

1 Disks

- Different types of storage have different speeds/latencies and capacities (Figure~2)

1.1 Magnetic Disk Setup

- **Platters**: One or Two sided
- Concentric Ring: **Tracks**
- Arc-Shaped **sectors**
- **Cylinder**: set of tracks with the same diameter

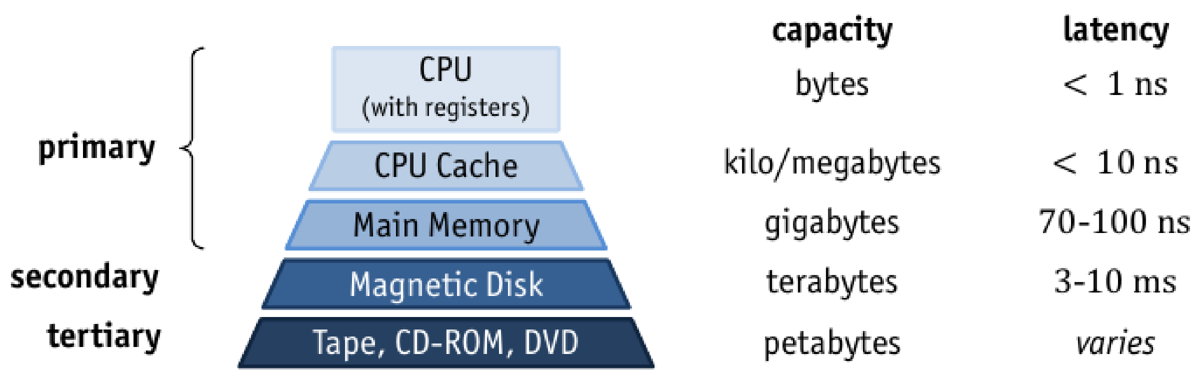


Abbildung 2: Speed of different Storage Types

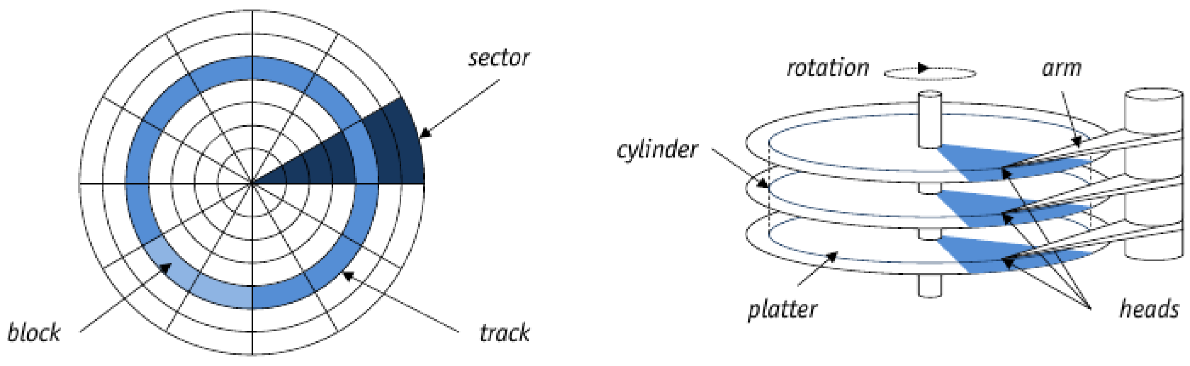


Abbildung 3: Setup of Magnetic Disks

- **block**: multiple of sector size

see Figure 3

1.1.1 Access Time

- To read data, head and disk need to be in correct position.
  => **Access Time**: $\Sigma t = t_s + t_r + t_t$ with:
  - **seek time** (t_s): disk head moves into position (*correct track*)
  - **rotational delay** (t_r): disk rotates under head (*correct block*)
  - **transfer time** (t_t): actual transfer of block data

1.1.2 Sequential vs. Random Access

- **Sequential Access**: Access adjacent blocks: Very fast as there is no additional seek time or rotational delay
- **Random Access**: wait for disk and head to align to correct block.
- => Avoid Random Access wherever possible.

1.1.3 Performance Tricks

- **track skewing**: align sector 0 of each track in a way to avoid rotational dealay
- **request scheduling**: for multiple requests, choose an access path that requires least arm movement
- **zoning**: outer tracks are longer => divide into more sectors

1.1.4 Implications for DBMS

- I/O dominates time for DBMS operations
- Disk block is unit of measurement which DBMS has to work with.
- Reduce *Number* of I/O operations.
- Reduce *Duration* of I/O Operations

1.1.5 RAID Systems

- **R**edundant **A**rray of **I**ndependent **D**isks
- Hardware or Software Implementation
- **Redundancy**: Replicate data over multiple disks => **Reliability**
- **Data Striping**: distribute data over multiple disks => **Performance**
- RAID-Levels
  - **0**: non redundant, just striping
  - **1**: Mirroring, no striping, just replication
  - **10** (0+1): Combination of 0 and 1
  - **2**: Error-Correction Codes (ECC)
  - **3**: Bit-Interleaved parity, one parity disk
  - **4**: Block-Interleaved Parity, like 3, but block wise distribution
  - **5**: Block-Interleaved Distributed Parity, parity distributed over all disks
  - **6**: P+Q Redundancy, like 5, but additional parity blocks

1.2 Solid-State Disks

- very **low latency read access**
- random writes significantly slower
- current topic of research

1.3 Network and Cloud-based Storage

- network not the bottleneck any more
- Storage Area Network:
  - Emulate interface of block structured disks
  - simplify maintainability by abstraction* fault tolerance can be increased

2 Disk Space Manager

- Abstracts from file System
- DBMS issues `allocate/deallocate` and `read/write` commands to disk space manager
- Provides concept of **page**
  - disk block, that has been brought into memory => blocks and pages are of same size
  - *sequences of pages* are mapped to *continuous sequences of blocks*
  - unit of storage for higher layers
  - page number <–> physical location

2.1 Free Space

- Disk Space Manager keeps track of **used** and **free** blocks.
  - *Linked List* of free blocks
  - free block *bitmap*

3 Buffer Manager

- mediate between external storage and main memory
- The **Buffer Pool** is a main memory area managed by the buffer manager
  - organized as collection of **frames**
  - pages loaded and brought into frames as needed
  - **replacement policy** decides, when to evict pages

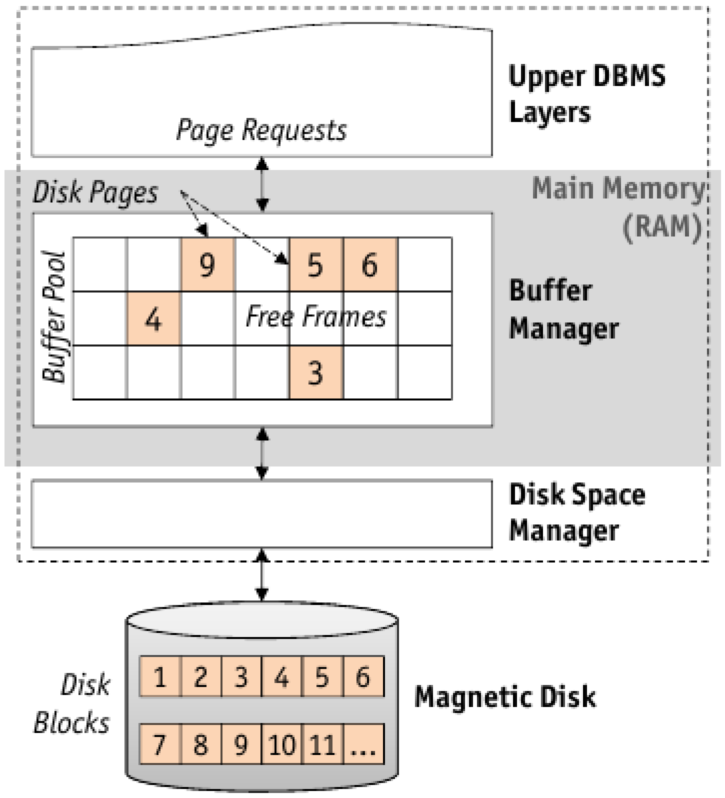


Abbildung 4: Buffer Pool

3.1 Buffer Manager Interface

- Higher-level code request (**pin**s) pages and releases (**unpin**s) pages
  - `pin(pageNo)`
    - request page number *pageNo* from buffer manager
    - if needed, load page and mark as clean ($\neg dirty$)
    - return reference to frame containing page number *pageNo*
  - `unpin(pageNo, dirty)`

    * release page number *pageNo*, make it candidate for eviction
    * if page was modified, *dirty* must be true
  - `pin()` and `unpin()` have to be balanced
  - *clean* pages are not written back to disk
  - `unpin()` does not trigger I/O
  - pages with `pinCount == 0` are candidates for eviction
  - `flushPage(p)` can be used to *force* disk I/O
  - *concurrency* is achieved by **concurrency control**, a higher level

3.1.1 Buffer Allocation Policies

- **Buffer Allocation Problem**: How much buffer space should be allocated to each transactions? *static* vs. *dynamic* assignement
- **Local Policies**:
  - allocate buffer frames to transaction *without* taking reference behaviour into account
  - has to have mechanism for handling allocation for frames for shared pages
  - *Pro*:
    * one transaction cannot hurt other transactions
    * all transactions are treated equally
  - *Cons*:
    * possible *bad* utilization of buffer space
    * some transactions may occupy vast amounts of buffer space while others have to little space (**“internal page thrashing”**)
- **Global Policies**:
  - consider reference behaviour of all other transactions
    * all pages have access to newly loaded pages
    * all other pages are likely to be replaced
    * other transactions may need to reload pages again (**“external page thrashing”**)

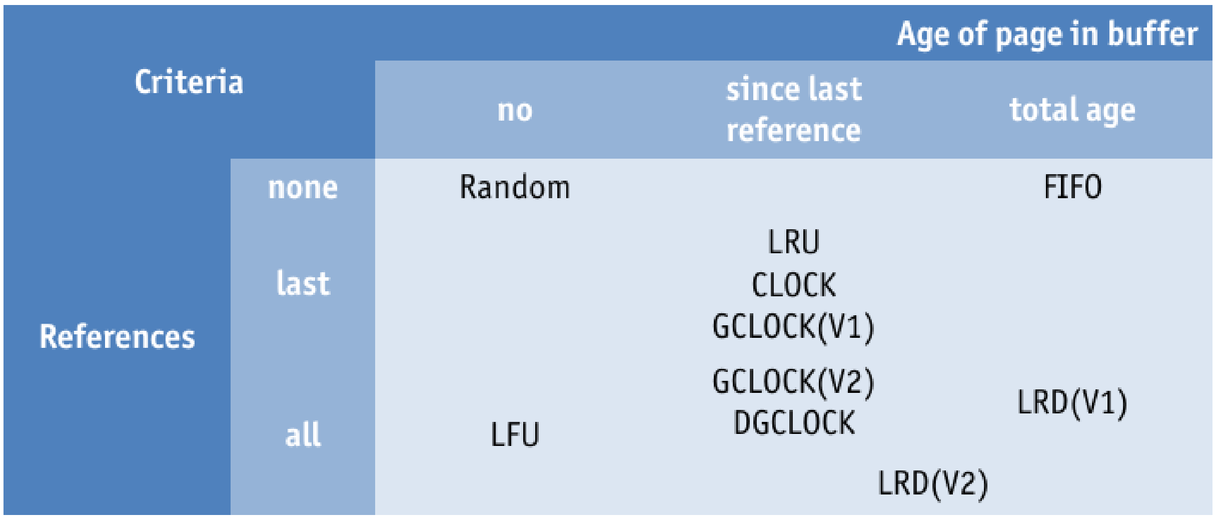
3.1.2 Buffer Allocation Policies

- **Page Replacement Problem**: Which page will be replaced, when a new request arrives into a full buffer pool?
  - *One Buffer Pool*: good space utilization, fragmentation problem
  - *Multiple Buffer Pool*: no fragmentation, worse utilization, global allocation/replacement strategy very complicated
  - Local: using various informations (catalog, index, data, . . .)
  - Local: each transaction gets a certain fraction of the buffer pool
    * **static partitioning**: assign buffer budget once
    * **dynamic partitioning**: adjust buffer budget of transaction according to past reference pattern or semantic information
  - mixed policies are possible, but are quite complex

3.1.3 Dynamic Buffer Allocation Policies

- **Local LRU**
  - Keep separate *LRU-stack* for each transaction
  - Keep global *free list* for pages unpinned by any transaction
  - Strategy:
    1. replace a page from the free list_
    2. replace a page from the LRU-stack of the requesting transaction
    3. replace a page from the transaction with the largest LRU-stack
- **Working Set Model** ($\approx$ virtual memory management of OS)
  - avoid thrashing by allocating "just enough" buffers to each transaction
  - *approach*: observe number of different page requests by each transaction in a certain intervall of time (window τ)
    * => deduce optimal buffer budget, allocate according to ratio between optimal sizes

3.1.4 Buffer Replacement Policies



Criteria		Age of page in buffer		
		no	since last reference	total age
References	none	Random		FIFO
	last		LRU CLOCK GCLOCK(V1)	

	all	LFU	GCLOCK(V2)
			DGCLOCK
			LRD(V2)
				LRD(V1)

Abbildung 5: Buffer Replacement Policies

- **FIFO** (first in, first out)
- **LRU** (least recently used) evicts page with the oldest `unpin()`
- **LRU-k** like LRU but evicts k latest `unpin()` calls
- **MRU** (most recently used) evicts post that has been recently unpinned
- **LFU** (least frequently used)
- **LRD** (least reference density)
  - record three parameters:
    * `trc(t)`: total reference count of transaction t
    * `age(p)`: value of `trc(t)` at the time of loading p into buffer
    * `rc(p)`: reference count of page p

  – compute *mean reference density* of page p at time t as

$$rd(p,t) := \frac{rc(p)}{trc(t) - age(p)}$$

  where $trc(t) - rc(p) \geq 1$

- **Clock** ("second chance") simulates LRU with less overhead
- **GCLOCK** (generalized clock)
- **WS, HS** ("working set", "hot seat")
- **Random**

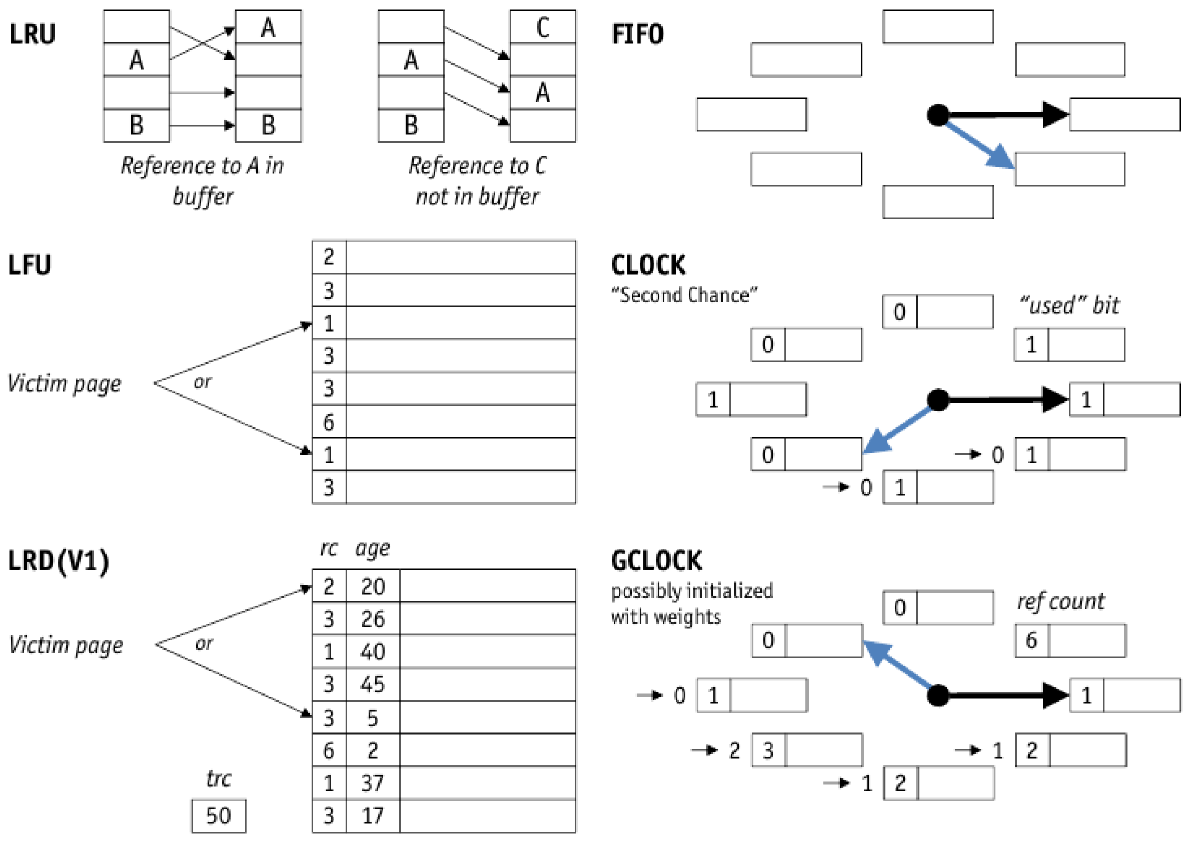


Abbildung 6: Buffer Replacement Policies

These policies are *heuristics* and can fail miserably in certain scenarios.

- Exploiting Semantic Knowledge
  - *Background*: query compiler/optimizer already
    - selects access plan (e.g. sequential scan vs. index)
    - estimates number of page I/O operations (for cost based optimization)
  - Idea: use this information to determine optimal buffer budget ("hot set")
  - => optimize overall system throughput + avoid thrashing
- Hot Set with Disjoint Page Sets
  1. Only queries whose Hot Set can be satisfied are activated
  2. higher demand queries have to wait
  3. each transaction applies a local LRU for its buffer budget

    - *no sharing* of buffer pages
    - risk of *internal thrashing*
    - large Hot Sets *block* other queries, risk of *starvation*

- Hot Set with Non-Disjoint Page Sets

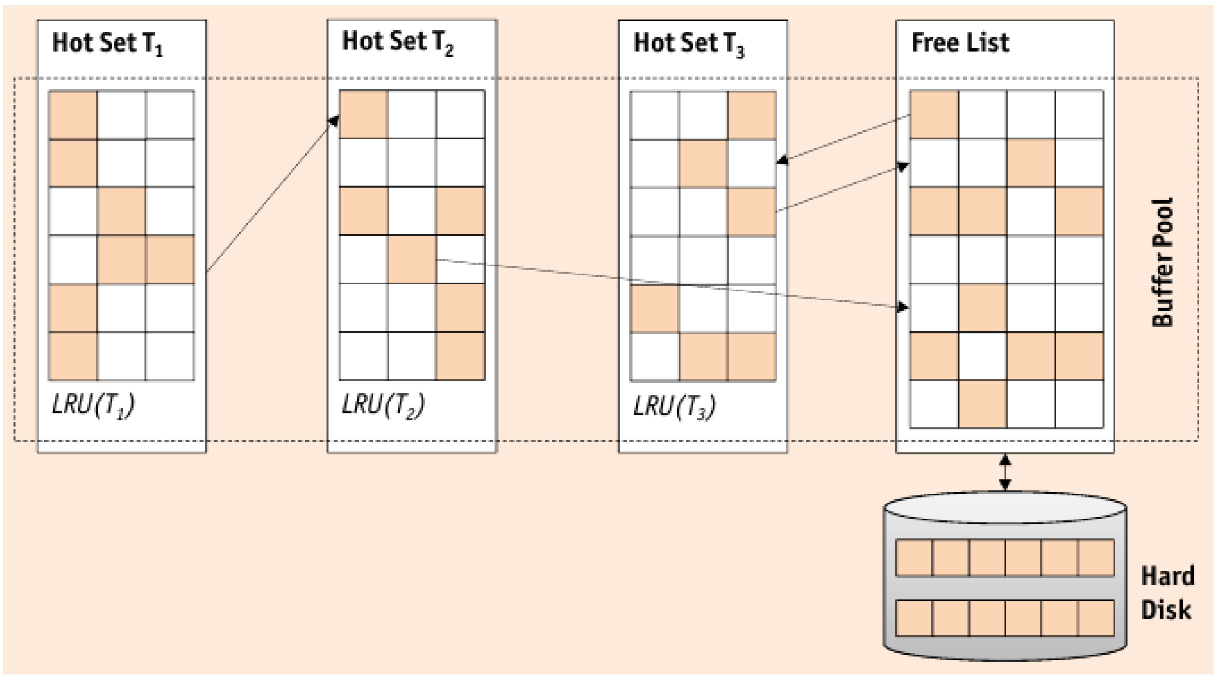


Abbildung 7: Hot Set with Non-Disjoint Page Set

1. queries allocate buffer budget stepwise
2. local LRU used for replacement
3. request new page *p*
    1. if on own LRU stack, update LRU stack
    2. if in another transactions LRU stack, access page, don't update LRU
    3. if in *free list*, push page to own LRU
4. call to `unpin(pageNo)`: push page onto free list
5. filling empty buffer frames: taken from bottom of the free list stack

- Priority Hints
    - with `unpin(pageNo)`, an indication is passed along
        * **preferred page** (managed in transaction-local partition)
        * **ordinary page** (managed in global partition)
    - strategy: when replacing
        1. try to replace ordinary page from global using LRU
        2. replace a preferred page of the requesting transaction using MRU
    - Much simpler than Hot Set, similar Performance
    - easy to deal with "too small" partitions

3.1.5 Prefetching

- Buffer manager tries to *anticipate* page requests => read ahead to improve performance and lower I/O
    - *Prefetch lists*: on-demand, asynchronous read-ahead
    - *heuristic* (speculative) prefetching

3.2 Double Buffering

- DBMS buffer manager can interfere with OS buffer manager => I/O invisible to DBMS can happen or I/O expected might not take place
  - **virtual page fault**: page resides in DBMS buffer, but swapped out of OS
  - **buffer fault**: page does not reside in DBMS buffer, but in OS buffer
  - **double page fault**: page does not reside in DBMS buffer, frame is swapped out of OS => one I/O for frame and one for page
- DBMS buffer => needs to be *memory resident* in OS

4 Database Files

- Conceptual Level: *relational* DBMS manages *tables of rows* and *indexes*
- Physical Level: implemented as *files of record*
  - *file* consists of *one or more pages*
  - *page* contains *one or more records*
  - *record* corresponds to *one row*

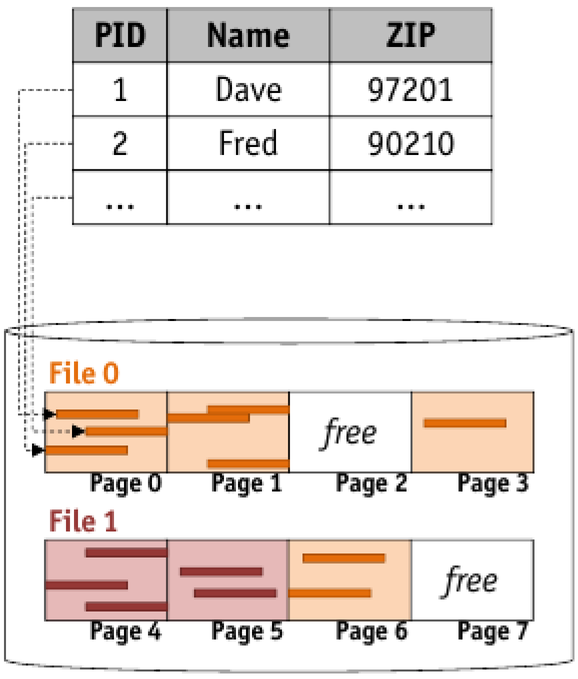


Abbildung 8: Mapping of rows to records to pages to files

4.1 Database Heap Files

- Most simple file structure in database
  - *unordered* collection of records
  - each record as a *unique record identifier* (*rid*)
  - heap file maps given *rid* to the page containing the record
  - heap file interface:

- `f <- create(n)`
  - to implement efficiently: heap file has to keep track of all pages with free space in the file f
- `delete(n)`
- `rid <- insertRecord(f,r)`
- `deleteRecord(f,rid)`
- `r <- getRecord(f,rid)`
- `openScan(f)`
  - Heap file has to keep track of all pages in the file f

- simple structures with those properties:
  - (doubly) *linked list* of pages

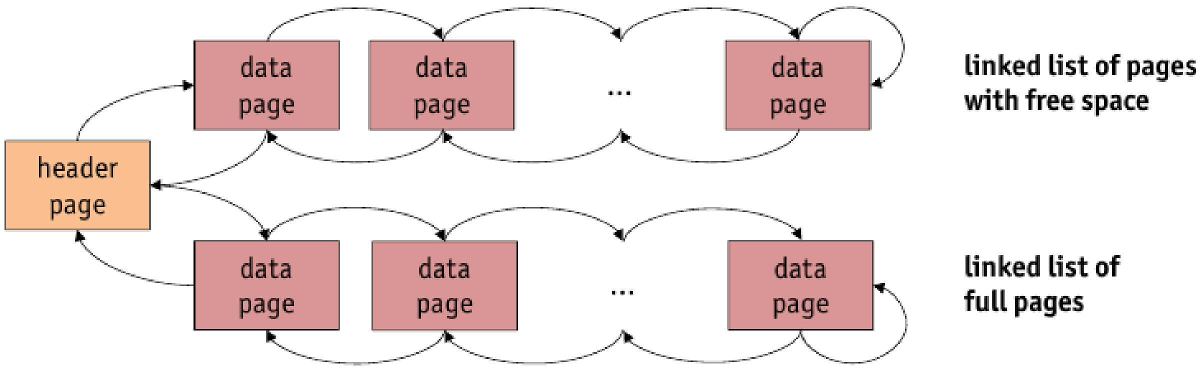


Abbildung 9: Doubly Linked List of Pages

~~~
* ```f <- createFile(n)```

    1. DBMS allocates a free page (_header page_) and stores entry
        ```<n, header page>``` to known location on disk
    2. header page is initialized to point to two doubly linked
       lists of pages: _full pages_ and _pages with free space_
    3. initialize both lists empty

* ```rid <- insertRecord(f,r)```

    1. _find page_ $p$ in the free list with space $> |r|$
    2. should this fail, ask disk manager to _allocate new page_
       $p$
    3. record $r$ is _written_ to page $p$
    4. since generally $|r| << |p|$, $p$ will belong to _list of
       free space_
    5. a _unique_ $rid$ for $r$ is computed and returned

* ```deleteRecord(f,r)```

    * may result in _moving the containing page_ from list of full
        pages to list of free space
    * may also lead to _page deallocation_ if page is completely
       free

* ```openScan(f)```

    * _both_ page lists have to be traversed
~~~

- *directory* of pages
    - *header page* contains first page of chain of *directory pages*
    - each entry in a directory page identifies a page of the file
    - $|pagedirectory| << |datapages|$
    - free space management is managed in directory
    - *exact* counter value: $nfree$
        - each transaction which changes space updates this counter
        - may become bottleneck (locking, etc, necessary)
    - *fuzzy* information: $\lfloor {}^{nfree}/_{8} \rfloor$
        - directory only needs to be updated for "large changes" in free space
        - exact information is kept in the page

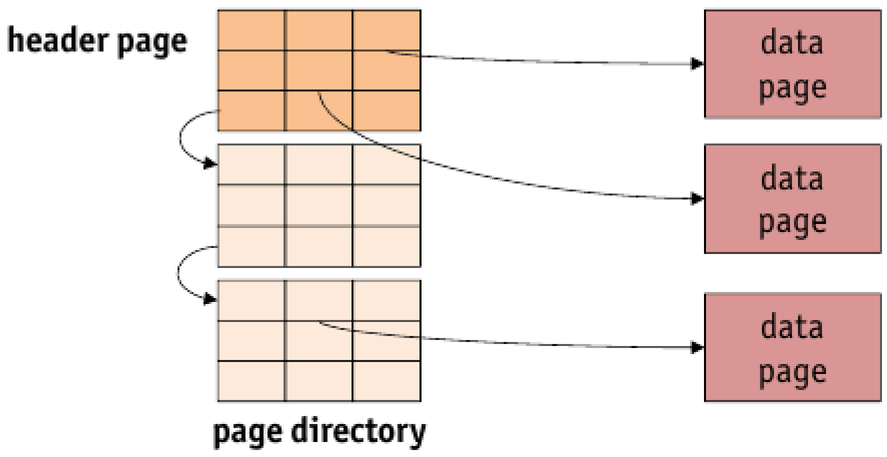


Abbildung 10: Directory Of Pages

4.2 Record Insertion Strategies

- **Append Only**: Always insert into last page, if not possible: create new page
- **Best Fit**: Search all pages and pick page with amount of free space most closely matching required space
- **First Fit**: Search from beginning and pick first sufficient space
- **Next Fit**: maintain running pointer and continue searching from here

4.2.1 Free Space Witnesses

- Accelerate search for free space with *witnesses*
- classify pages into buckets
- for each bucket, remember *witness page*
- only perform standard best/first/next fit search, if not witness page for this bucket present
- populate witness information while searching

4.2.2 Linked List vs. Directory

- Linked List
    - Pro:
        - Easy to implement
    - Con:

Bucket#	%Full	Witness
0	75%–100%	0
1	50%–75%	2
2	25%–50%	–
3	0%–25%	3

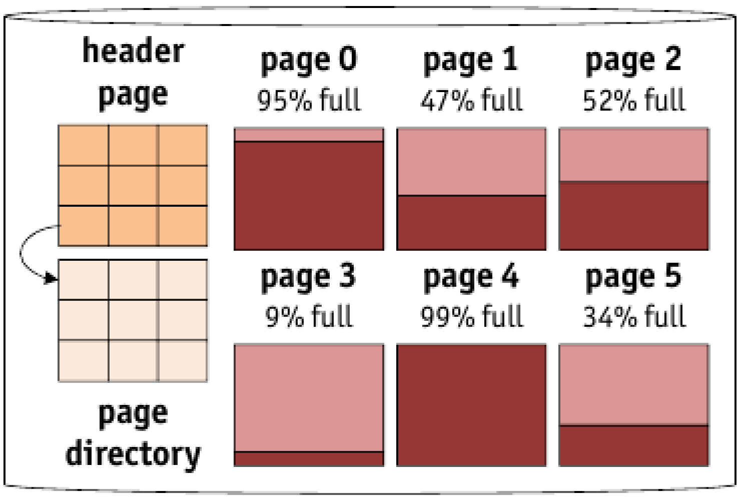


Abbildung 11: Witness Page

    * most pages will end up in free space list
    * searching for space may take long

- Directory
  - Pro: free space management very efficiently
  - Con: memory overhead to host page directory

5 Fixed Length Record

- All records on the page/in the file are the *same size s*
  - `getRecord(f,<p,n>)` given the $rid < p, n >$ => the record is at (byte) offset $n/timess$ on page p
  - `deleteRecord(f,<p,n>)`: copy the bytes of last occupied slot on page p to offset $n/timess$, mark slot as free (=> page is packed, i.e. all occupied slots start at the page)
  - `insertRecord(f,r)`: find page p with free space $\geq s$, copy r to first free slot on p, mark slots as occupied

5.1 Free Slot Bitmap

- Avoid record copying and therefore *rid* modification
- Delete simply sets bit n in bitmap to 0
- bitmap is a not a header, so that bitmap and data grow towards each other

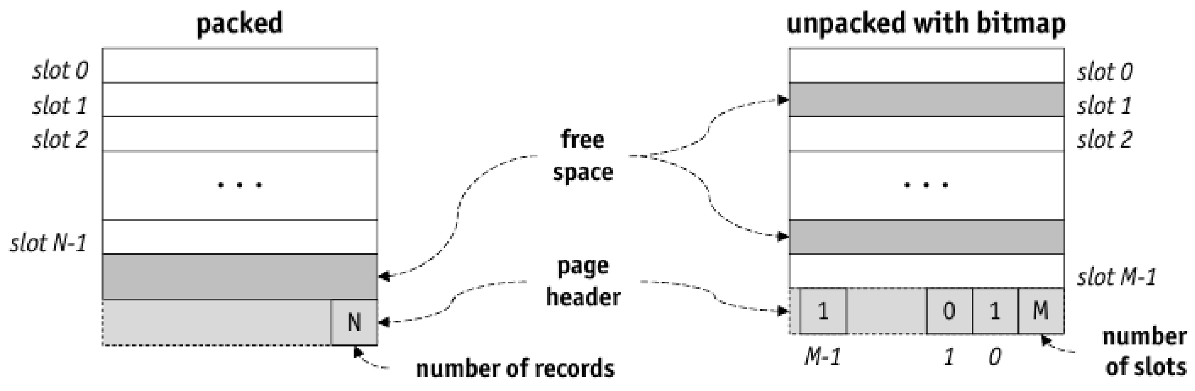


Abbildung 12: Free Slot Bitmap

6 Variable-Length Records

- Records can have variable length (e.g. `VARCHAR(n)`) => page fragmentation
- `insertRecord(f,r)`: find an empty slot of size $\geq |r|$ => wasted space is minimal
- compacting remaining records into *continuous area of free space* => get rid of holes produced by `deleteRecord(f,rid)`
- Solution: maintain *slot directory* on each page
  - free space management similar to heap file
  - contains $< offset, length >$, offset is measured in bytes from the start
  - during `deleteRecord(f,<p,n>)`: set directory entry n to -1 to indicate reusability
  - store length of records, store tombstones on delete

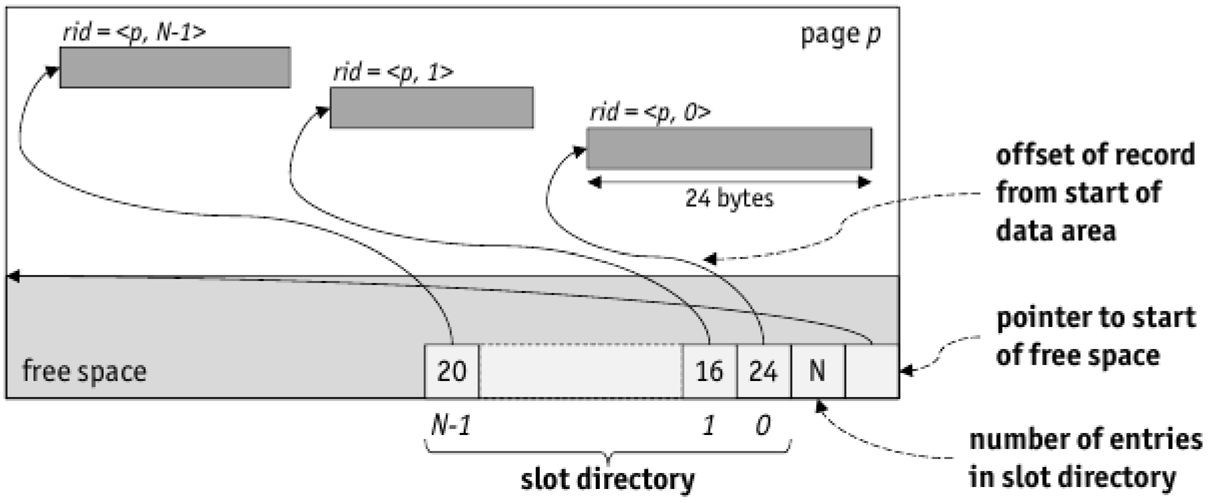

page p
rid = <p, N-1>
rid = <p, 1>
rid = <p, 0>
24 bytes
offset of record from start of data area
pointer to start of free space
free space
20
16
24
N
N-1
1
0
slot directory
number of entries in slot directory

Abbildung 13: Variable Length Record

7 Record Formats

- attributes are considered atomic in RDBMS
- record field type defines length
  - fixed length (Integer, Bigint, Char(n), Date, . . .)

- variable-length (Varchar(n), Clob(n), . . .)

- On `Create Table`: DBMS computes field size information for records of file => recorded in *system catalog*
- **Fixed Length**: each field has fixed length and number of field is also fixed
  - fields can be stored *consecutively*
  - address can be calculated with only number of preceding fields (and information from system catalog)
- **Variable-Length**: multiple variants how to store:

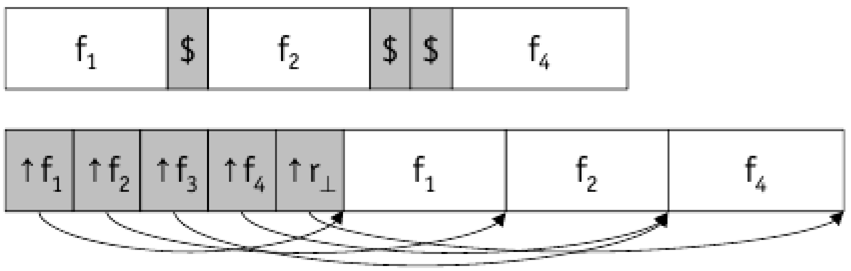


Abbildung 14: Delimiter (upper) and Array (lower) storage approach

  - use delimiter symbol: Special Symbol to sparate record fields, may not be used in fields now
  - use array of $n + 1$offsets pointing to actual location
  - Array approach typically superior:
    - direct access to any field
    - `NULL` values can be recognized by pointer comparision $start = end$
  - *Combination* of delimiter and array approach is also possible, variable length fields stored at end of record, fixed length at the beginning
  - Variable length variant might even be usefull for fixed-length record
    - support for null values
    - schema evolution

8 Addressing Schemes

- Criteria for "good" record ids:
  - given a *rid* => no more than one page I/O to get to record
  - *rid*s should be stable under all circumstances (moving within or across pages)
- Direct Addressing
  - **Relative Byte Address** (RBA): disk file => persistent virtual address space, byte-offset as *rid*
    - Pro: very efficient access to pages and records within pages
    - Con: no stability wrt moving records
  - **Page Pointers** (PP): use disk page numbers as *rid*
    - Pro: very efficient access to page, locating records withing page cheap (in-memory)
    - Con: stable wrt moving records in a page, but not across pages
- Indirect Addressing: **Logical Sequence Numbers** *LSN): Assign logical numbers to record, use address translation table to map LSN to PP or RBA

  - Pro: full stability wrt relocations of records
  - Con: additional I/O operation to translate table (often in the buffer)

- Fancy Indirect Addressing: **LSN with probable Page Pointers** (LSN/PPP): try to avoid extra I/O operations by adding PPP to LSN, where PPP is PP at time of insertion (if record is moved, PP *not* updated)
  - Pro: full stability wrt all record relocations and PPP may save extra I/O, if still correct
  - Con: two additional page I/O operations if PPP no longer valid (read "old" page, read translation table, rew new page)

- **Tuple Identifier** (TID): use $< pageNo, slotNo >$ par as *rid*

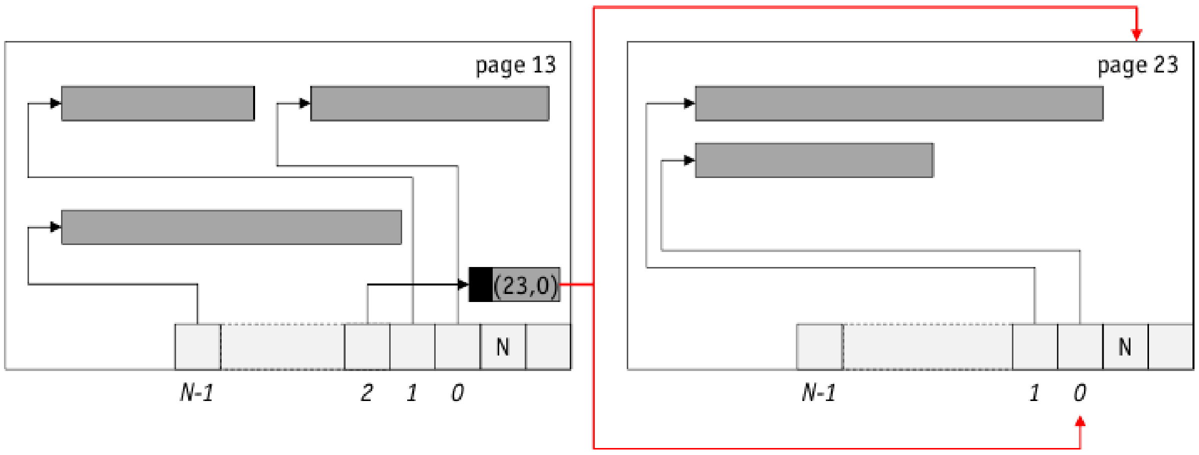


Abbildung 15: Tuple Identifier: Forwarding Address

  - $slotNo$ is an index in page-local offset array
  - => guarantees stability wrt relocation within page
  - stability wrt relocation across => leave *forwarding address* (tombstone)
  - Used by most DBMS

9 File Organisation and Indexes

- Heap File is *just enough* to maintain collection of records => only sequential scan
- For some queries, particular file organisation might be usefull.
- Different File Organisations:
  1. *heap file*: randomly ordered
  2. *sorted* files on one or more record fields
  3. *hashed* files on one or more record fields
- Compare Files with simple cost model, five disciplines:
  1. *scan*: read all records
  2. search with *equality test*
  3. search with *range selection*
  4. *insert* a given record into file's organisation
  5. *delete* a record (by *rid*), maintaining file's organisation

9.1 Cost model

- *block I/O* typically major factor
- *CPU time* also important
- parameters:
  - b: number of pages in file
  - r: number of records on a page
  - D: time to read/write a **d**isk page
  - C: **C**PU time needed to process record
  - H: CPU to,e to apply a function (**H**ash) to a record
- simple **Hash Function** h
  - hash function determines page number only, record placement is stored inside page
  - if a page p is filled to capacity, chain of overflow pages is maintained
  - pages are typically filled up to 80% when heap file is initialized

9.1.1 Cost of *Scan*:

- Involves *reading all b pages* and *processing each record r* on each page
- **Heap File**:
  - $Scan_{heap} = b \cdot (D + r \cdot C)$
- **Sorted File**: sort order does not help, just scan
  - $Scan_{sort} = b \cdot (D + r \cdot C)$
- **Hashed File**: hash function does not help, scan from beginning, hash file will very probably have random access
  $^*Scan_{hash} = \underbrace{{}^{100}/_{80}}_{=1.25} \cdot b \cdot (D + r \cdot C)$

9.1.2 Cost of Search with *Equality Test*

1. **Heap** file:
   - equality test on *primary key*:
     $Search_{heap} = {}^{1}/_{2} \cdot b \cdot (D + r \cdot C)$
   - equality test *not on primary key*:
     $Search_{heap} = b \cdot (D + r \cdot C)$
2. **Sorted** file (sorted on A)
   - assume equality test on A (else its just a scan): use binary search
     $Search_{sort} = \log_2 b \cdot D + \log_2 r \cdot C$
3. **Hashed** file (hashed on A)
   - assume equality test on A => best equality search support (ignoring overflow chains)
   - equality test on *primary key*:
     $Search_{hash} = H + D + {}^{1}/_{2} \cdot r \cdot C$
   - equality test *not on primary key*
     $Search_{hash} = H + D + r \cdot C$ note: no dependence on file size b here

9.1.3 Cost of Search with *Range selection*

1. **Heap** file: Qualifying records can appear anywhere in the file:

   $Range_{heap} = b \cdot (D + r \cdot C)$

2. **Sorted** file (sorted on A): Use equality serach with $A = lower$, then sequential scan until $A > upper$ is found. n denotes number of hits.

   $Range_{sort} = \log_2 b \cdot D + \log_2 r \cdot C + \lfloor n/r \rfloor \cdot D + n \cdot C$

3. **Hashed** File (hashed on A): Hashing scatters the records, no improvement here

   $Range_{hash} = 1.25 \cdot b \cdot (D + r \cdot C)$

9.1.4 Cost of *Insert*

1. **Heap** file: add record to arbitraty page, read and write this page

   $Insert_{heap} = 2 \cdot D + C$

2. **Sorted** file: On average, new record will be in the middle, all subsequent records must be shifted

   $Insert_{sort} = \underbrace{\log_2 b \cdot D + \log_2 r | cdotC}_{search} + \underbrace{1/2 \cdot b \cdot (2 \cdot D + r \cdot C)}_{shiftlatterhalf}$

3. **Hashed** File: Search for record, read and write page determined by hash function (assume there is space to hold new record)

   $Insert_{hash} = \underbrace{H + D}_{search} + C + D$

9.1.5 Cost of *Delete*

1. **Heap** file: Assume file is not compacted after record is found and removed.

   $Delete_{heap} = \underbrace{\ D\ }_{search} + C + D$

2. **Sorted** file: Access record's page and then (on average) shift the latter half of file to compact

   $Delete_{sort} = D + \underbrace{1/2 \cdot D + r \cdot C}_{\text{shift latter half}}$

3. **Hashed** File: Direct access by *rid*, faster than hash.

   $Delete_{hash} = D + C + D$

9.1.6 Performance Comparison

- **Range Selections**: *sorted* file is superior from the beginning
- **Deletions**: *hashed* file is linear, sorted file gets worse with number of records
- Selection of files is depending on what is to be expected

9.2 Indexes

- B+ tree: offer advantages of sorted file *and* support insertions and deletions efficiently
- **Idea**: If basic organisation does not support specific operation => *additionally* maintain an index as auxiliary structure.
  - DBMS uses indexes like *guides*, each guide is specialized on specific attribute => the **search key**.

  1. Query index with **search key**

2. Returned entry is the **index entry** k^* (containing enough info to access the actual record in the file)
3. Read actual record by using information from k^*, record will have an attribute with value k

- Index Entries:

  1. $< k, < \dots, A = k, \cdots >>$
     - 1 => Data stored in the index, no additional storage necessary.
     - If multiple indexes are built, only one should have variant 1.
  2. $< k, rid >$
  3. $< k, [rid_1, rid_2, \dots] >$
     - fewer index entries for multiple matching records
     - index entries have variable length

  - Variant 2 and 3 use rid(s) to point to actual data file

9.2.1 Clustered vs. Unclustered Indexes

- **Clustered** Index: Data is organized in the same way as the index (automatically with variant 1).
- **Unclustered** Index: Data can be organized in any which way.
- Cost for *Range Selection* grows tremendously for unclustered index => Random I/O
- Each data fo;e can have *at most one clustered index* (but any number of unclustered indexes)

9.2.2 Dense vs. Sparse Indexes

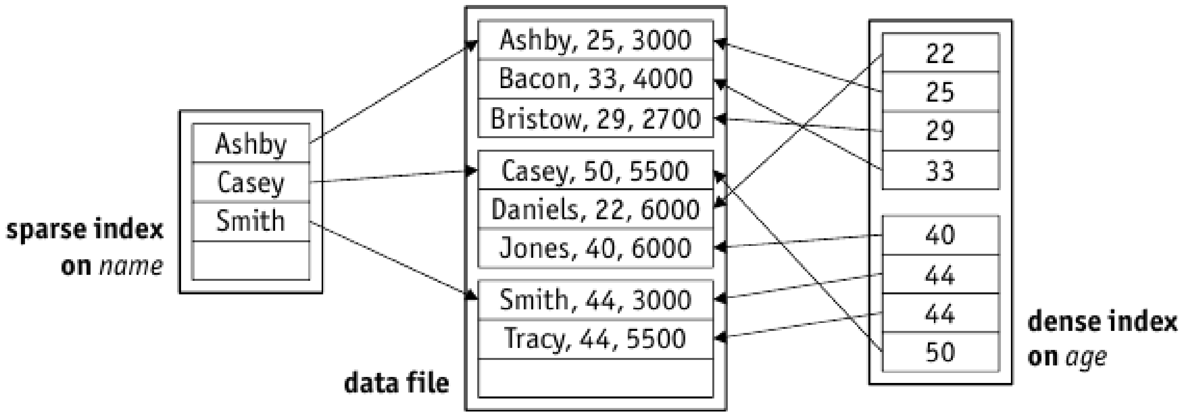


Abbildung 16: Dense vs. Sparse

- *Clustered index* can be **space efficient**
- maintain one index entry k^* **per data file page** => k is the smallest key
- such indexes are called **sparse**, otherwise they are **dense**.
- unclustered sparse index is not possible

9.2.3 Multi-Attribute Indexes

- indexes can use a combination of attribute values as key: `<lastname, firstname> -> searchkey`
- support lookup on whole key or or *prefix* of key.
- *multi-dimensional indexes*: support support for symmetric lookups

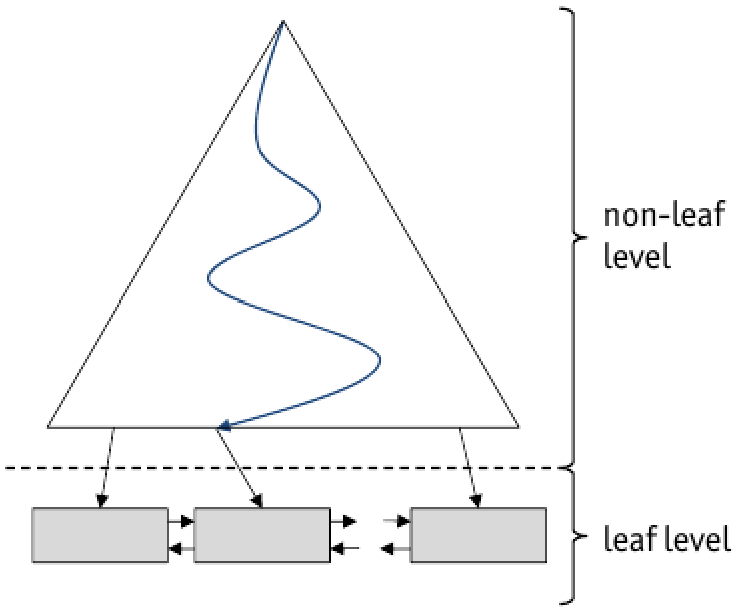


Abbildung 17: Tree-Structured Indexes

10 Tree-Structured Indexes

- **Binary Search**: Needs sorted file
  - *Pro*: During Scan: page accesses are sequential
  - *Con*:
    - During Search: $\log_2(\#tuples)$ need to be read
    - About same number as tuples need to be read
    - large unpredictable jumps => prefetching impossible
- improve binary search by introduction *auxiliary structure* that contains *one record per page of original (data) file* => recursively apply until only one page left
- particularly usefull for *Range Selections*

10.1 Indexed Sequeuntial Access Method (ISAM)

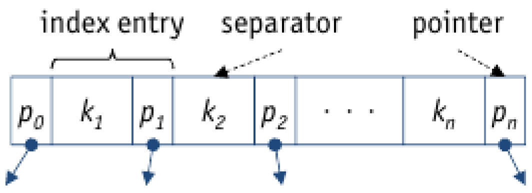


Abbildung 18: ISAM index file structure

- **static** replacement of binary search phase => fewer pages read
- ISAM is a *sparse index*, since in an entry $< k_i, p_i >$, key k_i is the first (minimal) attribute value on the data file.
- **Range Selection**:
  - conduct binary search on index file => do sequential scan of data file

  - index file much smaller than data file

- For large data files, recursively apply index creation step: *multi-level ISAM structure*
- => tree-structured hierarchy of index levels
  - each node corresponds to *one page* (disk block)
  - ISAM structure is created *bottom up*
    1. sort data file on search key
    2. create index leaf level
    3. if topmost level contains more than one page, repeat procedure
  - non-leaf level remain **static**
  - in the leaf level, there may be overflow pages
- ISAM may lose balance after heavy updating
- typically 20% free space to handle overflow problem
- static pages can't be locked, therefore there is no bottleneck here

ISAM is very well suited for relatively static data (were the distribution of data does not change much).

- *Fan-Out*: number of children for non-leaf nodes:
  - n children per page, height h => n^h leaf pages

10.2 B+ Tree Properties

- B+ Tree is a derivation of the ISAM index structure, fully dynamic wrt updates
- Search performance is only dependent on the *height* of the B+ Tree (and height rarely exceeds 3 due to high fan-out)
- B+ Trees **remain balanced**, **no overflow chains** develop
- B+ Trees support efficient **insert/delete operations**, data file can grow/shrink dynamically
- B+ Tree nodes have a minimum occupancy of 50% (typically 66%)
- **Leaf Nodes** are connected using a doubly linked list
- Leaf nodes may contain the actual data (variant 1) or references (variant 2 and 3)
- Non-Leaf nodes have the same structure as ISAM non-leaf nodes
  - **Order d of B+ Tree**:
    inner node: $d \leq n \leq 2 \cdot d$
    root node: $1 \leq n \leq 2 \cdot d$
  - each node contains $n + 1$ pointers
- Leaf Nodes
  - Variant 1: index is data file as well
  - Variant 2 and 3: index may still be clustered or unclustered

10.3 B+ Tree Operations

- Insert: B+ Tree remains balanced
  - All paths from the root to any leaf must be of *equal length*
  - Insertions and Deletions *must* preserve this invariant
- `insert(k)`
  1. start with root node, recursively insert entry into appropriate child node
  2. on leaf node level, recursion stops, page is n, m is the number of entries in n
  3. if $m < 2 \cdot d$ there is room left in n
  4. otherwise: split node
     1. create new node n'
     2. distribute entries of n and k over n and n'
     3. insert a pointer to n' pointing to new node into its parent
- Root node may have occupancy of $< 50\%$
- tree height increases (only), if root node is split
- **Redistribution**: improve average occupancy in B+ Tree
  - before node is plit, entries are **redistributed** with a sibling
  - the **siblings** of a node n is a node that is immediately to the left or right of N and has the same parent as n
- `delete(k)`
  1. start in root node, recursively delete node from appropriate child
  2. stop at leaf level
  3. if $m > d$, n does not have minimum occupancy and k^* can be deleted
  4. otherwise
     - redistribution with siblings, update parent to reflect changes
     - merge node n with an adjacent sibling, update parent to reflect change
       * If last entry in root is deleted, height of tree decreases by 1.
  - In practice, merge/redistribution is often avoided by relaxing occupancy rule

10.4 Duplicates in B+ Tree

- duplicates can be ignored if key is primary key
- using Variant 3: duplicates can be accommodated, but index entry sizes is now variable
- alternatively: change insert, search and delete algorithms

10.5 B+ Tree Performance

- Higher Fan-Out is main factor in search effort (since it depends on height): $s = \log_F N$
- Size of page pointers depends on DBMS pointer representation/hardware specifics
  - $|p_i| << |k_i|$, especially for attributes like `CHAR(\cdot)` or `VARCHAR(\cdot)`
  - **minimize key size**: actual key values not needed, as long as seperator is properly maintained
    * for text attributes: *prefixes of text*
- B+ Tree bulk loading: Build index from bottom up

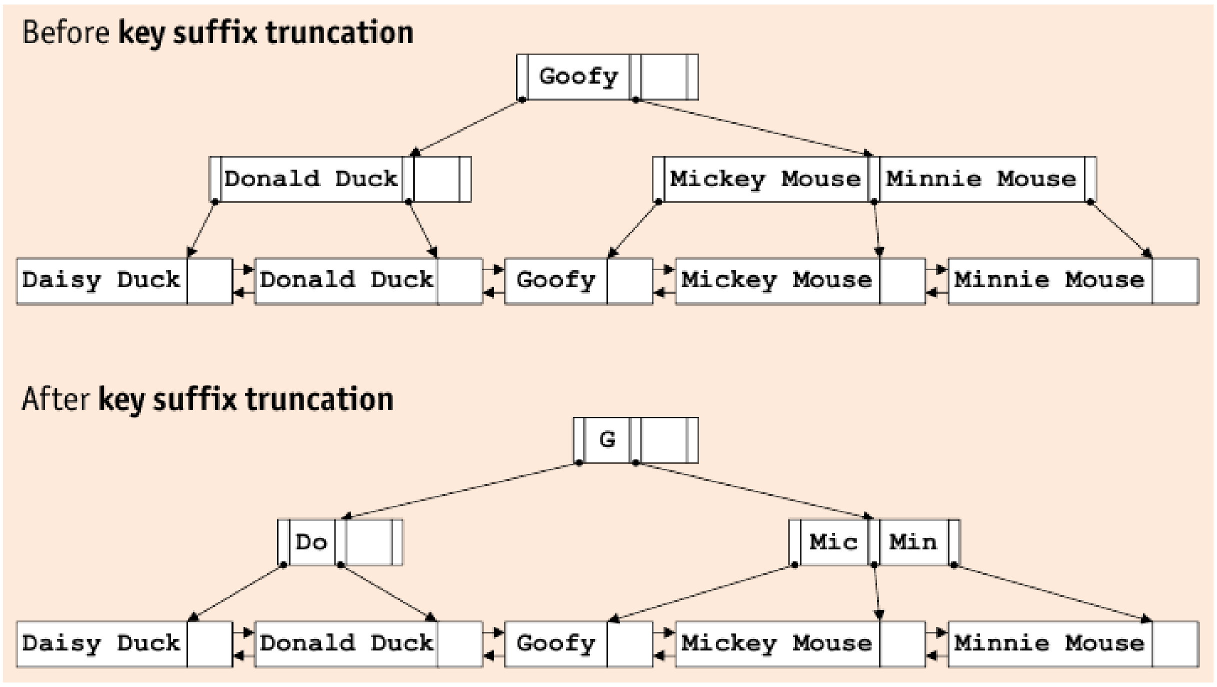


Abbildung 19: Key Suffix Truncation

1. for each record with key k in the data file, create sorted list of pages of index leaf entries k^*

   For variants 2 or 3 this does not imply sorting the data file itself, variant 1 creates a clustered index
2. allocate empy index root node, let its p_0 to the first page of sorted k^* entries
3. for each leaf level node n, inset index entry $<$ min key on $n, *n >$ into right-most index node just above leaf level

- Bulk loading more (time efficient)
  * less tree traversals
  * less page I/O, i.e. buffer pool is used more effectively

- In practice, B+ Tree *order d* most often not used in practice (variable key value lengths, duplicates, variable number of *rid*s, different capacity in nodes, key compression)

  => in practice, use physical space criterion ("Every node needs to be at least half full)

- Note on clustered indexes:
  - splitting and merging leaf nodes *moves data entries*
  - depending on addressing scheme, *rid*s may change if entry is moved

    => May use search key of clustered index as (location independent) record address for non-clustered indexes to avoid *rid* management

10.6 B+ Tree invariants

- **Order** d
- **Occupancy**
  - each non-leaf node holds at least d and at most $2d$ keys (exception, root node may hold down to 1 key)
  - each leaf node holds between d and $2d$ keys
- **Fan-Out**: each non-leaf node holding m keys has $m + 1$ children
- **Sorted Order**:

  - all nodes contain entries in ascending key-order
  - child pointer $p_i(1 \leq i < m)$ if an internal node with m keys $k_1, \ldots, k_m$ leads to sub-tree where all keys k are $k_i \leq k < k_{i+1}$
  - p_0 points to a sub-tree with keys $k < k_1$ and p_m to a sub-tree with keys $k \leq k_m$

- **Balance**: all leaf nodes are on the same level
- **Height**: $\log_F N$
  - N is the total number of index entries/records and F is the average fan-out
  - because of high fan-out, B+ trees generally have low height (typically ≤ 3)

11 Hash-Based Indexes

- Hash-Based Indexes are unrivaled wrt equality selections (in 1.2 operations)
- Operations like equality joins need a large number of equality tests
- Comparison to B+ Trees
  - B+ Tree compares k to other keys k'
  - hash indexes use the bits of k itself to compute the address of the record
- Range Queries: Hash indexes have no support and are not well suited for range queries (scattering)
- Hashing granularity
  - Hashing in DBMS not in-memory => bucket oriented hashing
  - a bucket can *contain several records* and may have an *overflow chain*
  - bucket = (set of) pages on secondary memory

11.1 Static Hashing

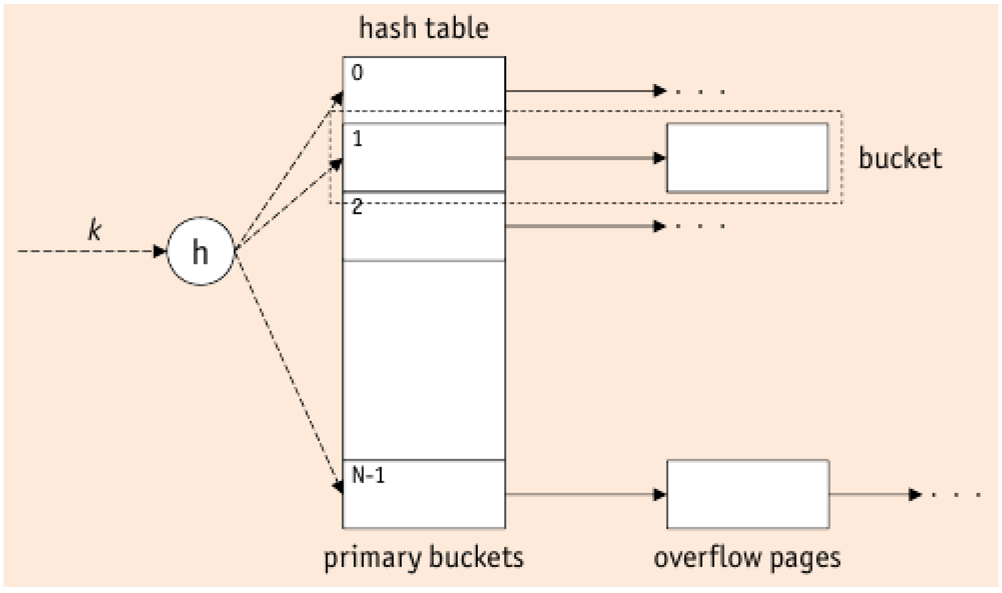


Abbildung 20: Static Hashing Table

- used to illustrate **basic concept** of hashing
- like ISAM, static hashing does not handle updates well

- Build static hash index:

  1. Allocate fixed area of N (successive) disk pages => **primary buckets**
  2. In each bucket => pointer to overflow pages, initially -> `null`
  3. Define *hash function* h with range $[0, \ldots, N-1]$, the **domain** of h is the type of A

- Static Hashing Scheme

  1. apply hashing function h to key value: `h(k)`
  2. access primary bucket page with `h(k)`
  3. search, insert, or delete the record with key k on that page, if necessary, access overflow chain of bucket `h(k)`

- In case of no overflow chains:

  - `hsearch(k)` => single I/O
  - `hinsert(k)` and `hdelete(k)` => two I/O

- don't want to prevent collisions => would need as many buckets as keys

  => it is important, that h scatters the domain of A evenly

- probability of Collisions => Birthday Paradoxon

11.1.1 Hash Functions

- Actual key values are not random, good h

  => division of key value

$$h(k) = k \mod N$$

  - prime numbers work best for N
  - choosing $N = 2^d$ for some d only takes least d bits of k

  => multiplication of key value

$$h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor$$

  - *inverse golden ratio* $Z = {}^{2}/_{\sqrt{5}+1} \approx 0.6180\ldots$ good according to D. E. Knuth
  - for $Z = {}^{\dot{Z}}/_{2^w}$, $N = 2^d$, w number of bits of CPU word) we get $h(k) = msb_d(\dot{Z} \cdot k)$, msb - *most significant bit*

- growing data file => overflow chains => no more predictable I/O behaviour

  - worst case: linear list (one long chain of overflow pages)

- shrinking data file => wasted space

11.2 Dynamic Hashing

- refine the hashing principle and adapt well to record insertions and deletions
- **combine** the use of hash functions **with directories** that guide the way (extendible hashing)
- **adapt the hash function** (linear hashing)

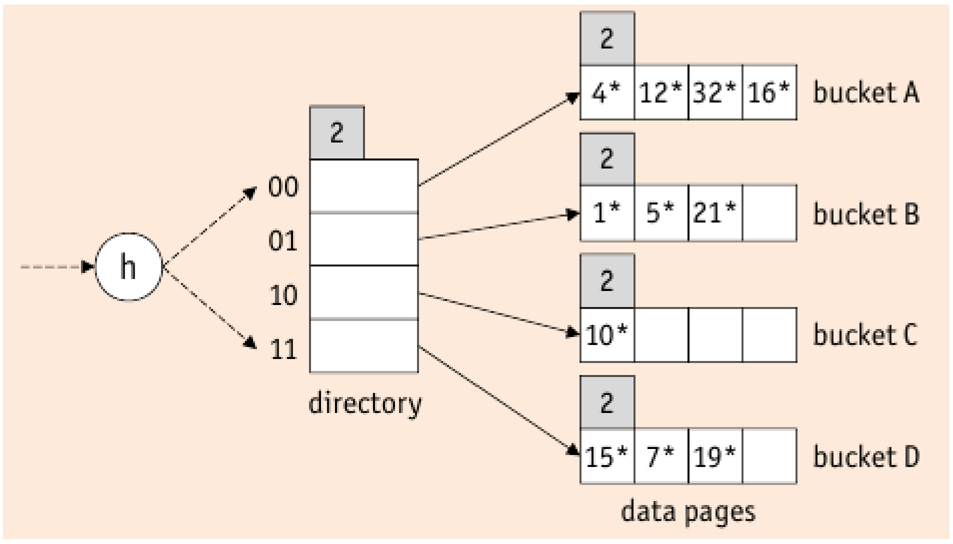


Abbildung 21: Extendible Hashing: hash table setup

11.2.1 Extendible Hashing

- use in-memory bucket directory, primary buckets are contained
- Global Depth n: (gray box at hash directory): use the last n bits of $h(k)$ to look up bucket pointer in directory, directory size is 2^n
- Local Depth d: (gray box at individual buckets): hash values $h(k)$ of values in this bucket agree on their last d bits
- `insert(k)`
    1. compute `h(k)`
    2. use last n bits of `h(k)` to lookup bucket pointer in directory
    3. if *primary bucket* still has capacity, store `h(k)^*` in it
    4. if not
        1. split bucket A by creating new bucket A2 and use bit position $d + 1$ to redistribute entries
        2. local depth is incremented
        3. if $d > n$ double space of directory
            1. pointer of A2 points to new bucket now
            2. all other pointers point to the corresponding “old” bucket
- `hdelete(k*)`: locate and remove entry k^*
    - If bucket is empty after deleting, merge with its split bucket.
    - In practice, this is not done.

11.2.2 Linear Hashing

- No hash directory, only actual hash table buckets
- linear hashing defines flexible criteria to determine when bucket is split
- may perform bad, if key distribution is skewed
- **linear hashing** uses **ordered family of hash functions**
    - $h_0, h_2, h_3, \ldots$ with hash function for each *level*
    - range of $h_{level+1}$ is *twice as large* as range of h_{level}

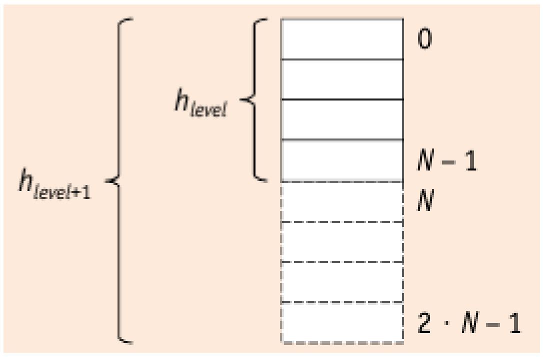


Abbildung 22: Linear Hashing

- Basic Hashing Scheme

  1. Initialize `level <- 0`, `next <- 0`
  2. *current hash function* in use for searches is h_{level}
     active hash buckets are those in the range of h_{level}
  3. Whenever
     - current hash table overflows
     - Insertions filled a primary bucket beyong $c\%$ occupancy
     - Overflow chain of a bucket grew longer than p pages
     - or

     the bucket at hash table position `next` is split

  - *Note* the bucket that triggers the split is generally not split.

- `splitBucket(next)`

  1. allocate new bucket and append it to hash table at position $2^{level} \cdot N = next$
  2. redistribute entries in bucket next by rehashing them via $h_{level+1} (some will remain in bucket `next`, some will move to new bucket)
  3. increment `next`

  - All Buckets with positions < `next` have already been rehashed
  - With every bucket split, next walks down the hash table. Searching needs to take current `next` position into account
    $$h_{level}(k) \begin{cases} < next: & \text{bucket already split, rehash: find record in bucket } h_{level+1}(k) \\ \geq next: & \text{bucket not yet split, i.e. bucket found} \end{cases}$$
  - rehashing means rehashing overflow chain as well

11.2.3 Extendible vs. Linear Hashing

linear hashing => directory can be avoided by cleverness

12 System Catalog

- *two kinds of information* in DBMS:
  - Data
    * alternative file structures for *tables* and _indexes
    * files contain either *tuples of a table* or *index entries*
    * collection of user tables and indexes represent the data of the database

  - Metadata
    * DBMS maintains information that describes tables and indexes
    * descriptive information => stored in *special table*
      => common names: catalog table, data dictionary, *system catalog*

- System Catalog
  - size of buffer pool, page size, etc
  - information about tables, views, indexes
  - statistics about tables and indexes
  - statistics are updated periodically, not every time tables are modified

- System Catalog is stored as a collection of tables itself
  - existing techniques for implementing and managing tables
  - same query language used for catalog tables

13 Relational Operator Evaluation

- *alternative algorithms* can be used to implement each operator
  - performance depends on query, cardinalities, indexes, sort order, size of buffer pool, replacement policiy, ...

- **Common implementation techniques**:
  - **indexing**: an index is used to only process tuples that satisfy a selection or join condition
  - **iteration**: process all tuples of an input table, one after the other or scan all index data entries (index only scan)
  - **partitioning**: decompose an operation into less expensive collection of operations by partitioning tuples on sort key (e.g. sorting and hashing)

13.1 Access Path

- *access path* = way of retrieving tuples from table
  - *file scan*
  - *index* plus *selection condition*

- Relational Operator accept *one or more tables* as input

- Access path significantly contributes to *cost of operator*

- **Conjunctive Normal Form** (CNF): A conjunction of conditions of the form `attribute operator value`, where `operator` is one of the comparison operators $<, \leq, =, \neq \geq, >$, each condition is called a *conjunct*

- An index **matches** a selection criterion if it can be used to retrieve just the tuples that satisfy the condition
  - A hash index matches a CNF selection => *attribute* = *value* for each attribute in the index' search key
  - A tree index matches a CNF celection => $attributeopvalue$ for each attribute in the index' search key
  - A search key prefix is the prefix of a search key: $< a >, < a, b >$ are prefixes of key $< a, b, c >$
    * An index can match a subset of a CNF selection: **primary conjuncts**: the conjucnts that the index matches
    * When only prefix of index is used, produced tuples need to be checked for missing conditions.

- **Selectivity** of an access path: number of index and data entries it retrieves
  - the *most selective* access path is the one that retrieves fewest pages
  - selectivity depends on primary conjuncts
    - each conjunct acts as a filter on the table
- **Reduction Factor**: the fraction of tuples in the table that satisfy a conjunct
  - for several primary conjuncts, the fraction satisfying them all can be approximated by multiplying their reduction factors
  - Reduction Factor Estimation: Assume uniform distribution of data, catalog information $ILow$ and $IHigh$ can be used
  - for tree index T, reduction factor for $attr < value$ is: $\frac{IHigh(T)-value}{IHigh(T)-ILow(T)}$

14 Relational Operator Evaluation

- Focus on $\sigma, \pi, \bowtie$ operator as used in SPJ-queries (select, project, join)
- Cost analysis: only I/O cost considered in number of page I/O operations.

14.1 Selection

- Selections are of the form: $\sigma_{R.attr\,\mathbf{op}\,value}(R)$
  - If there is no index on *R.attr*, R has to be scanned.
  - if *one or more* indexes match the selection, this index can be used
- Projection are of the form $\pi_{\{R.attr1,R.attr2,...\}}(R)$
  - dropping attributes is easy
  - **duplicate elimination** is expensive
  - Without duplicate elimination (no `DISTINCT`)
    - scan of the table, retrieve projected subset
    - scan of an index whose key contains all necessary attributes
  - With duplicate elimination
    1. retrieve subset of projected attributes via scan
    2. sort retrieved tuples with projected attributes as sort key
    3. scan tuples and discard adjacent duplicates
    - typically requires two or three passes
      1. scan entire table, only write out subset of projected attributes
      2. maybe intermediate pass which reads from and writes to disk
      3. final pass of sorting: scans all tuples but only one copy of each subset is written out.
  - presence of appropriate indexes can lead to *less expensive* plans than sorting for duplicate elimination
    - if an index whose search key contains all attributes exists => sort index entries
    - if prefix of search key contains retains all attributes
- Joins are of the form $R \bowtie_{R.attr=S.attr} S$
  - Joins are **common and expensive** => widely studied
  - DBMS typically have *several algorithms* to compute joins
- Other Operators:
  - `GROUP BY`: typically through sorting or tree index with attributes as search key
  - aggregation: *temporary counters* in main memory
  - set operations (union, difference, intersection): as in projection, *duplicate elimination* is expensive, partitioning approach can be adapted

14.2 Query Optimization

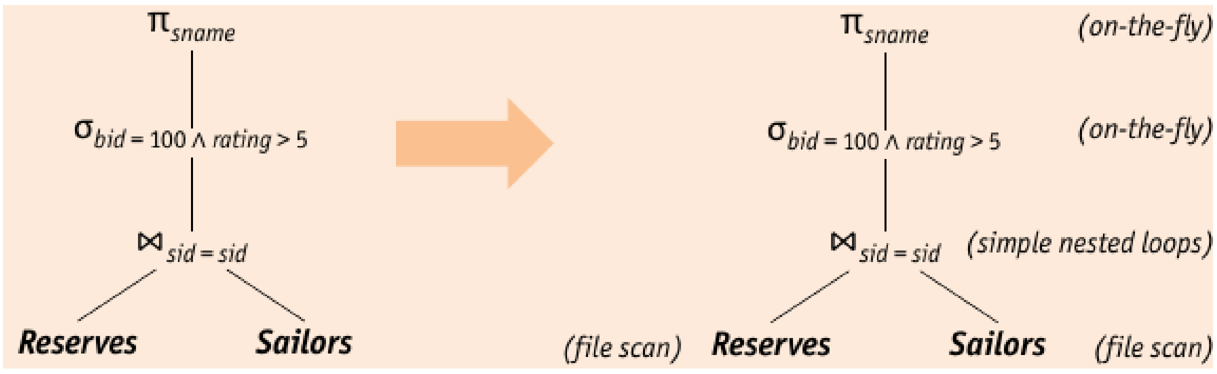


Abbildung 23: Query Evaluation Plan

- SQL gives great flexibility => DBMS can choose from many differeny strategies
  - Quality of *query optimizer* greatly influences performance
  - *Plan Enumerator* enumerates number of alternative plans
  - *Plan Cost Estimator* assigns a cost to each alternative plan
  - plan with least estimated cost is chosen
- Queries are treated as $\sigma - \pi - \bowtie$ algebra expressions, remaining operators are applied to result
- Query Optimization has three steps:
  1. **enumeration**: enumerate alternative plans
  2. **estimation**: estimate the cost of all plans
  3. **choosing**: choose the plan with the lowest cost

  - Typically only a subset of all plans is evaluated because of search space size
- Query evaluation plan consists of relational algebra tree extended with annotations at nodes

14.2.1 Multi-Operator queries

- If a query is composed of several operators, query optimizer can decide on how the plans
  - communicate and pass results
    - *materialized*: output is written to temporary table
    - *pipelined*: results are directly "streamed" from one operator to another
    - pipeline has lower overhead cost, but might not be applicable for all applications (sorted output)
  - are scheduled and interleaved
    - *bracket model*: explicit scheduling
      - central scheduler can fit single operator into bracket on demand
      - each operator runs on its own thread and controls it
      - Synchronization points and communication needs must be known to bracket implementation
    - *iterator model*: implicit scheduling
      - all operators implement *uniform iterator interface*
      - hides internal implementation details
      - assumes pipelined evaluation

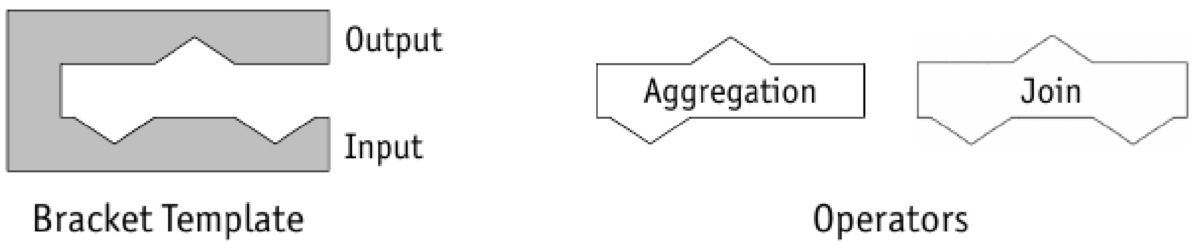
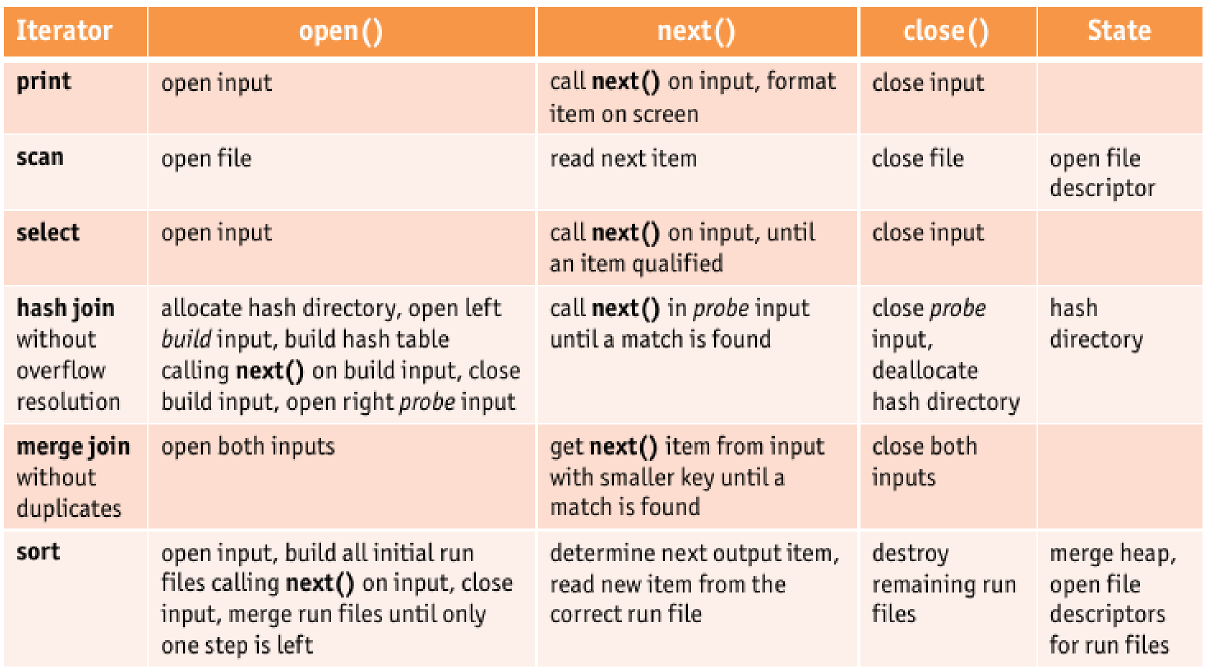


Abbildung 24: Bracket Model



Iterator	open()	next()	close()	State
print	open input	call **next()** on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call **next()** on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left *build* input, build hash table calling **next()** on build input, close build input, open right *probe* input	call **next()** in *probe* input until a match is found	close *probe* input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get **next()** item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling **next()** on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Abbildung 25: Iterator Model

· requires operations
`open()`: initalizes iterator, opens input/output buffers, used to pass modifiers (selection criterion, . . .)
`next()`: pulls next tuple from each input, executes operator-specific code, place result in output-buffer, updates interior state
`close()` deallocates state information

14.2.2 Pushing Operators

- Joins are expensive: size of two inputs determine cost
- reducing input size will speed up join
  => **push selections** to early stages to reduce input size
- **pushing projections** is also helpfull

14.2.3 Plan Enumeration

1. Transform SQL query into relational algebra

FROM $t_1, t_2, \ldots, t_n$	$\rightarrow$	$((t_1 \times t_2) \times \ldots) \times t_n$
WHERE $a = v$ **AND** ...	$\rightarrow$	$\sigma_{a=v \wedge \ldots}(\cdot)$
SELECT a, b, c	$\rightarrow$	$\pi_{\{a,b,c\}}(\cdot)$

Abbildung 26: Transform SQL query into relational algebra

2. Apply relational algebra equivalences to convert initial expression into equivalent expression
   - *selections* (σ) and *cross-products* ($\times$) can be combined into *joins* ($\bowtie$)
   - *selections* (σ) and *projections* (π) can be pushed ahead of *joins* ($\bowtie$) to reduce size of inputs
   - *joins* ($\bowtie$) can be extensively reordered

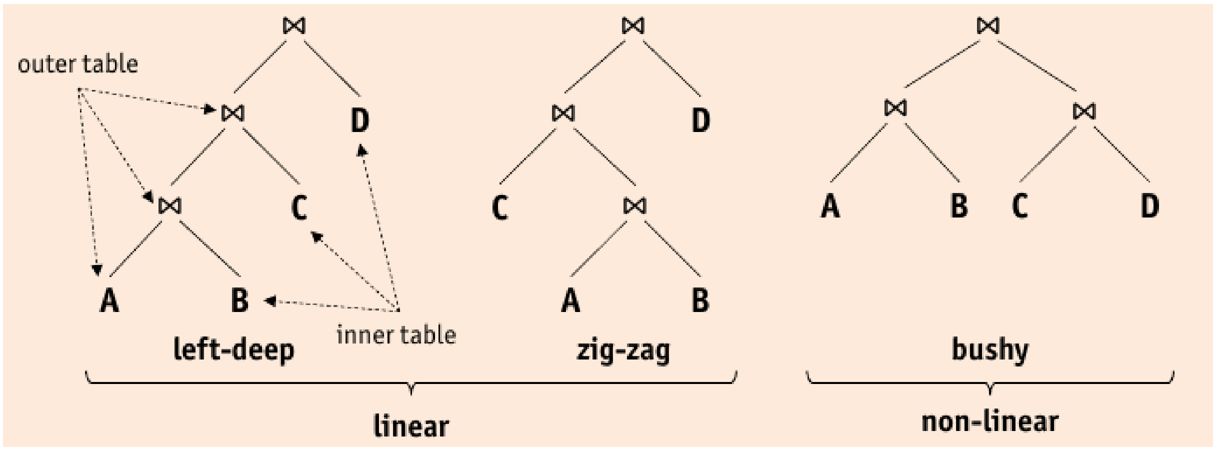


Abbildung 27: Join Reordering

- **Left-Deep Plans** => typically the only plans considered
  - use **dynamic programming** to efficiently search this class
  - apply selections and projections as early as possible

  => query optimizer will *not* find the best plan, if the best plan is not left-deep

14.2.4 Cost Estimation

- Cost Estimation consists of:

  1. reading input tables (for some join and sort algorithms multiple times)
  2. writing intermediate results
  3. sorting the final result (for duplicate elimination or output order)

  - If a plan produces sorted order, this does not have to be done explicitly.
  - If fully pipelined (common) no intermediate results are written.
  - Cost of fully pipelined plans is dominated by reading input tables
  - Not fully pipelined:
    * Cost may greatly yield from intermediate results.
    * Intermediate tables need to be read *and* written.

- Operator Result Size:

  σ: estimated by multiplying with reduction factor

  π equal to input size (no duplicate elimination)

  $\bowtie$ estimated by multiplying maximum result size (product of input sizes) with reduction factor (of join condition) # External Sorting

- sorted wrt to *sort key* k and ordering θ, if for any two records r_1, r_2 their keys are in θ-order:

$$r_1 \theta r_2 \qquad \Leftrightarrow \qquad r_1.k \theta r_2.k$$

- key may be a ordered list of attributes: order is defined *lexicographically*: $k = (A, B), \theta \leq$

$$r_1 < r_2 \qquad \Leftrightarrow \qquad r_1.A < r2.A \lor (r_1.A = r_2.A \land r_1.B < r2.B)$$

- If data to be sorted is to large for available buffer pool => external sorting is required

14.3 Two-Way Merge Sort

- Two-Way Merge Sort can sort files of arbitrary size with only *three pages* of available buffer space.
- Two-Way Merge Sort uses multiple **passes** which create subfiles called **runs**

  1. **Pass 0** sorts each of the 2^k input pages individually in *main memory*, resulting in 2^k sorted runs.
  2. **Subsequent Passes** merge pairs of runs into larger runs. Pass n produces 2^{k-n} runs.
  3. **Pass** k produces one final run, the complete sorted file.

- During each pass, every page of the file is read and written => $2 \cdot k \cdot N$ page I/O operations are needed.
- **Pass 0**:

  1. Read N pages, one page at a time.
  2. Sort records, page-wise, in main memory.
  3. Write sorted pages to disk (each file => in a *run*).

  - This pass requires *one page* of buffer space.

- **Pass** n:

  1. Open two runs r_1 and r_2 from Pass $n-1$ for reading.
  2. Merge records from r_1 and r_2, reading input page by page.
  3. Write new 2^n-page run to disk, page by page

- Restriction to three buffer pages is voluntary. Two-Way Merge Sort can be used for worst case scenarios.

14.4 External Merge Sort

- Improvements: over Two-Way Merge Sort
  - **Reduce number of runs** by using full buffer space to avoid creating one-page runs in Pass 0.
  - **Reduce number of passes** by mergin more than two runs at a time.
- If B pages are available in buffer pool:
  - B pages can be read at a time during Pass 0 and sorted in memory.
  - $B-1$ pages can be merged at a time (one page is the write buffer)
- **Pass 0**:
  1. Read N pages, B *pages at a time.*
  2. Sort records, page-wise, in main memory
  3. Write sorted pages to disk (=> $\lceil N/B \rceil$ runs)

  - This pass uses B pages of buffer space.
- **Pass** n
  1. Open $B-1$ runs $r_1, \dots, r_{B-1}$ from Pass $n-1$ for reading.
  2. Nerge records from $r_1, \dots, r_{B-1}$ from Pass $n-1$ for reading.
  3. Write new $b \cdot (B-1)^n$ - page run to disk, page by page

  - This pass requires B pages of buffer space.
- Each pass reads, processes and writes *all N pages.*
- Number of initial runs determines number of passes.
  - in Pass 0: $\lceil N/B \rceil$ are written.
  - number of additional passes: $\lceil \log_2 \lceil N/B \rceil \rceil$
  - With B pages of buffer space, we can to a $(B-1)$-way merge: $\lceil \log_{B-1} \lceil N/B \rceil \rceil$ additional passes instead
  - => total number of page I/O:

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

14.4.1 Blocked I/O

- improve I/O pattern by using *blocked I/O*
  - allocate b pages for each input => read *blocks of b pages* during merge
    - reduces I/O cost per page
    - decreases fan-in => increases number of I/O operations
  - => total number of page I/O:

$$2 \cdot N \cdot (1 + \lceil \log_{\lfloor B/b \rfloor} \lceil N/B \rceil \rceil)$$

- in practice, main memory is enough to sort in just one merge pass, even with blocked I/O

14.4.2 CPU load

- External merge sort is much more CPU intensive => I/O is still dominating factor.
- Use selection tree (Figure~28)
- Cuts down comparison cost from $B-2$ to $\log_2(B-1)$.

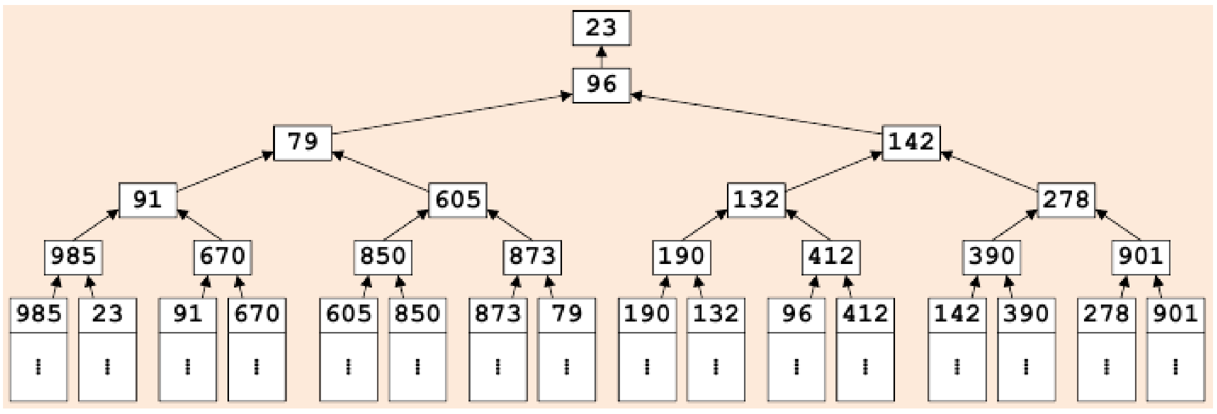


Abbildung 28: Selection tree, read bottom-up

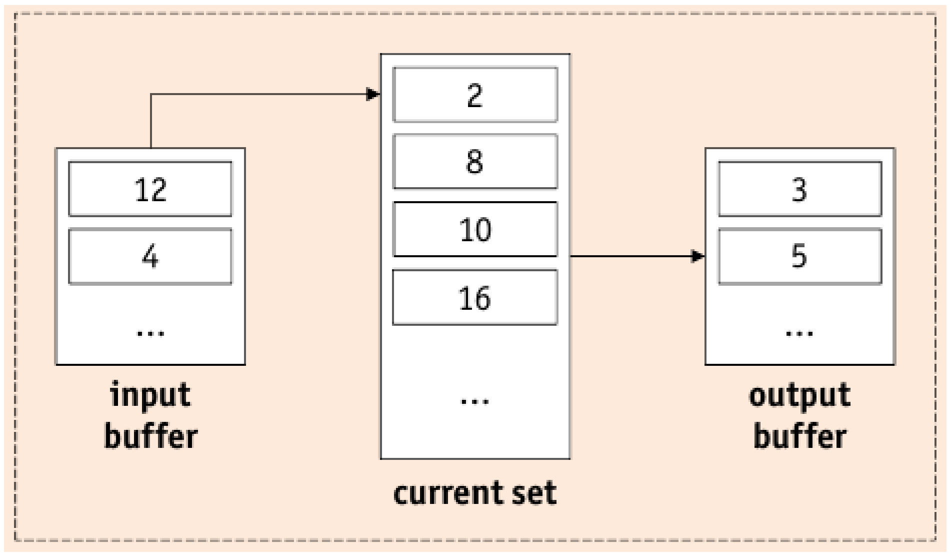


Abbildung 29: Replacement Sort Buffer Layout

14.5 Replacement Sort

- Number of runs still determines number of required passes.
- Produce initial runs with more than B pages. (Figure~)
- Replacement Sort:
    1. Open empty run file for writing.
    2. Load next page of file to be sorted into input buffer.
        - If input file is exhausted, goto 4
    3. While there is space in the current set, move a record from input buffer to current set.
        - If input buffer is empty, goto 2.
    4. In current set, pick record r with smallest key value k such tath $k \geq k_{out}$, where k_{out} is the maximum key value in output buffer. Move r to output buffer. If output buffer is full, append output buffer to current run.
    5. If all k in current set are $< k_{out}$, append output buffer to current run, close current run. Open new empty run file for writing.

    - Step 4 will benefit from techniques like selection tree (Figure~28), esp. if $B - 2$ is large.
- External Sort follows *divide and conquer* principle => independent sub-tasks => execute in parallel
- CPU should never remain idle: avoid waiting for input to be reloaded or output to be appended.
- **Double Buffering**:
    1. create **shadow buffers** for each input and output buffer
    2. switch to "double buffer" one original buffer is empty/full
    3. original buffer is **asynchronously** read or written

14.6 B+ Tree Sorting

- If B+ Tree index matches a sorting task, using the index *may* be better.
    - If index is clustered: data file is already θ-sorted, sufficient to read all pages from index.
    - If index is unclustered: worst case: one page I/O per record.

15 Evaluating Relational Operators

- Operators can be implemented with different algorithms, exploiting physical system:
    - presence or absence of indexes on input files
    - sortedness of input-files
    - size of input files
    - available buffer-space in the pool
    - buffer replacement policy
- Specific operator => **physical operator**
    - implement **logical operators** (of the relational algebra)
    - Common implementation techniques:
        * **indexing**: use an index that satisfies a selection or join condition
        * **iteration**: process all tuples of input table or scan all index data entries (of an index that contains all attributes, not using the index structure)
        * **partitioning**: decompose operation into less expensive collections of operations by partitioning tuples on sort key (e.g. sorting and hashing)

15.1 Join

- semantics of **join operator** $\bowtie_p$ => equivalent to combination of **cross product** $\times$ and **selection** σ_p
- naive way would be enumeration of all pairs in the cross product and pick satisfying predicates afterwards

15.1.1 Nested Loops Join

$$R_1 \bowtie^{NL} R_2 \mathrel{\hat{=}} \sigma(R_1 \times R_2)$$

- R_1 is the *outer*, R_2 the *inner* relation
- I/O cost: $\underbrace{\|R_1\|}_{\text{outer loop}} + \underbrace{|R_1| \cdot \|R_2\|}_{\text{inner loop}}$
- **Pro**: only needs *three pages of buffer space* (two to read R_1 and R_2, one to write)
- **Con**: huge I/O cost (effectively enumerates cross product)

15.1.2 Block Nested Loops Join

- saves random access cost by reading R_1 and R_2 in blocks of b_1 and b_2 pages
  - R_1 is read once with $\lceil \|R_1\|/b_1 \rceil$ disk seeks
  - R_2 is scanned $\lceil \|R_1\|/b_1 \rceil$ with $\lceil \|R_1\|/b_1 \rceil \cdot \lceil \|R_2\|/b_2 \rceil$ disk seeks
- Building **Hash Table** on R_1 speed up *in-memory join*=> works only for equi-joins
  - compare r_2 with r_1: $r_2 \overset{?}{=} h(r_2.A_2)$

15.1.3 Index Nested Loops Join

- index on the inner relation (swap outer <–> outer if necessary)
- index must match join condition p
- avoids enumeration of cross-product
- For each tuple in R_1 query the index for matching R_2-tuples
- Cost-breakdown:
  1. **Access** of index to find first matching entry.
  2. **Scan** index to retrieve *all* n matching *rid*s (sequential I/O, typically negligible)
  3. **fetching** the n matching tuples R_2-tuples
- Cost depends on size of **join result**
  - $\|R_1\| + |R_1| \cdot$ (cost of one index access to)R_2

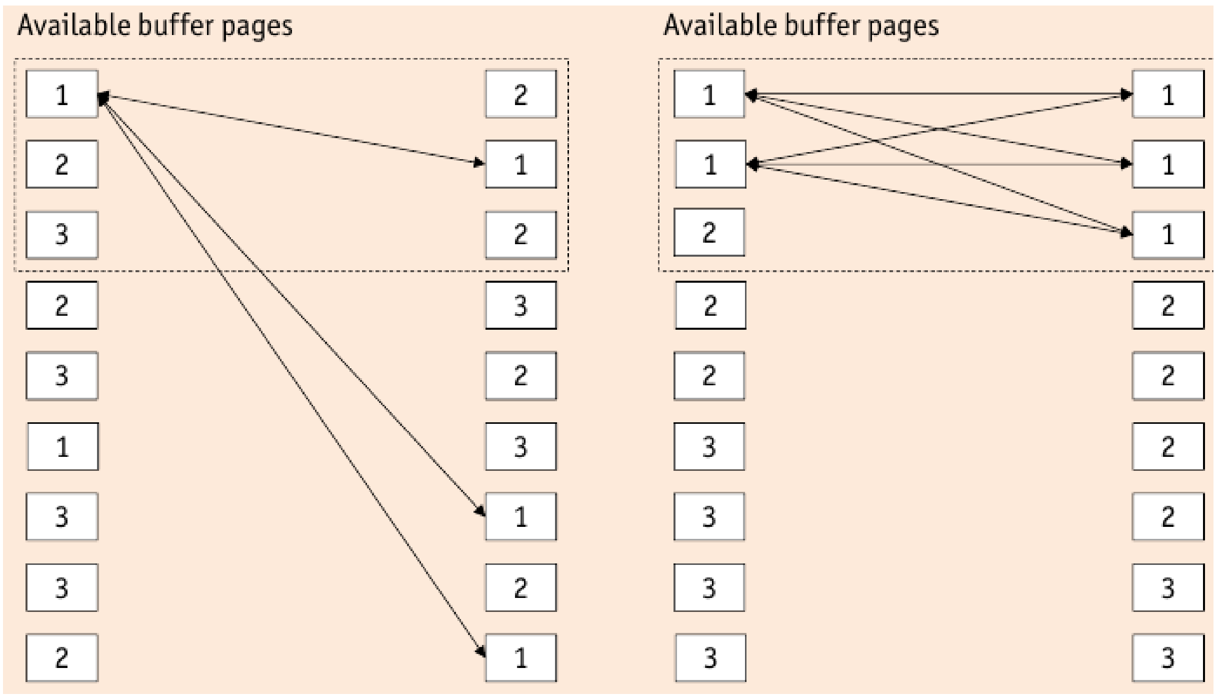


Abbildung 30: Partitioning for Joins

15.1.4 Sort-Merge Join

- use sorting on join attribute to partition both inputs
- typically used for *equi-joins*
- Merge Phase similar to merge phase of external sort => input relations are two runs that have to be merged
  - duplicate elimination has to be done
- Sort Merge Join
  1. Sort both inputs, if not already sorted.
  2. open output buffer, input buffers for R_1 and R_2
  3. while both inputs not empty pull a record depending on which is smaller
  4. put record into output buffer
- I/O cost:
  - best-case: $\|R_1\| + \|R_2\|$
  - worst-case: $\|R_1\| \cdot \|R_2\|$
- Sort-Merge Join can be integrated into final pass of external sort.
- blocked I/O, double buffering, replacement sort => apply to speed up sorting and merging
- output is sorted on join attribute

15.1.5 Grace Hash Join

- **family of Hash Joins** work in two phases
  1. **partitioning** (building) phase: divide R_1 and R_2 into k partitions $R_1[k], R_2[k] (Figure~31)
  2. **probing** (matching) phase: only compare tuples in partition $R_1[i]$ to those in $R_2[i]$ (Figure~32)

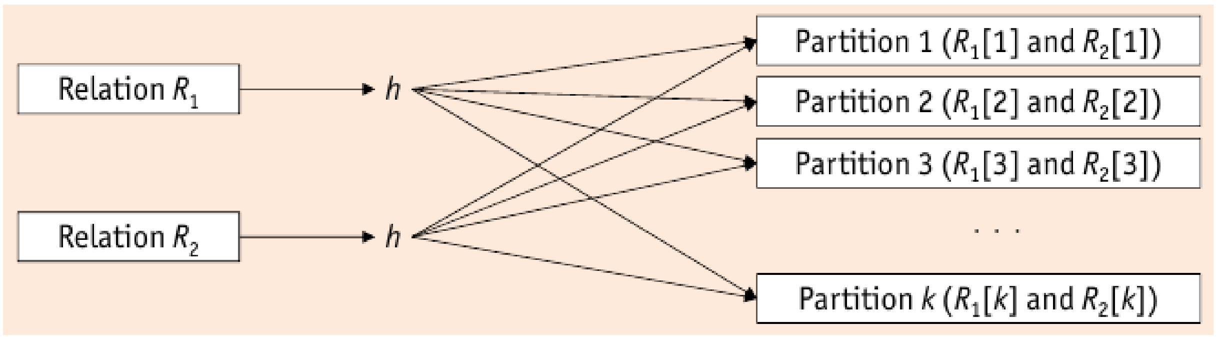

Relation R_1
Relation R_2
h
h
Partition 1 ($R_1[1]$ and $R_2[1]$)
Partition 2 ($R_1[2]$ and $R_2[2]$)
Partition 3 ($R_1[3]$ and $R_2[3]$)
. . .
Partition k ($R_1[k]$ and $R_2[k]$)

Abbildung 31: Grace Hash Join: Partitioning Phase

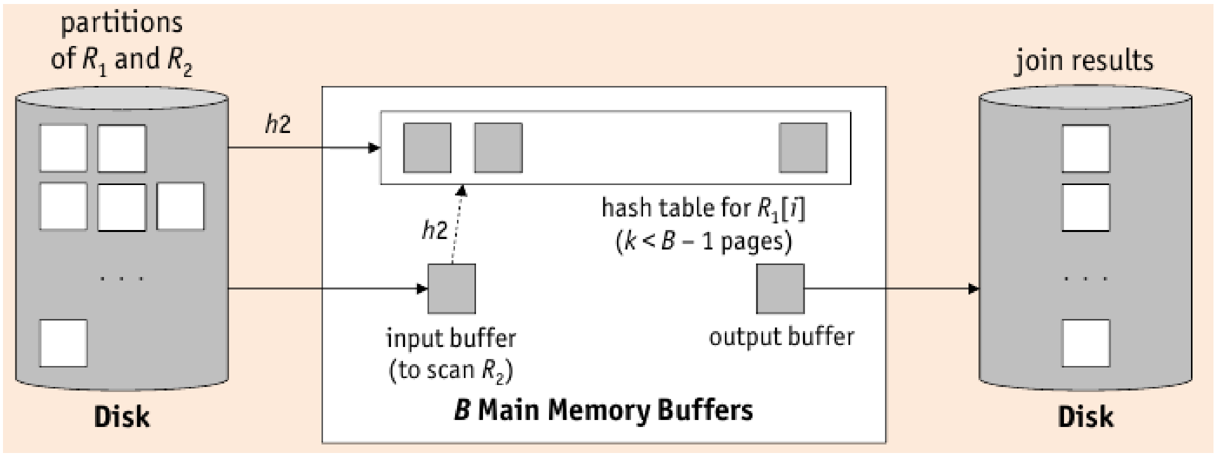

partitions of R_1 and R_2
$h2$
$h2$
hash table for $R_1[i]$ ($k < B - 1$ pages)
input buffer (to scan R_2)
output buffer
B Main Memory Buffers
join results
Disk
Disk

Abbildung 32: Grace Hash Join: Probing Phase

  - divide and conquer => many small in-memory joins
  - $R_1[i] \bowtie R_2[i] = \varnothing$ for all $i \neq j$

- I/O cost: $\underbrace{\underbrace{\|R_1\| + \|R_2\|}_{\text{read}} + \underbrace{\|R_1\| + \|R_2\|}_{\text{write}}}_{\text{partitioning phase}} + \underbrace{\|R_1\| + \|R_2\|}_{\text{probing phase}} = 3 \cdot (\|R_1\| + \|R_2\|)$

- Partitions have to fit into memory for probing phase => maximize number of partitions
  - B buffer pages => $B - 1$ partitions
  - assumes equal distribution, each partition has size $\frac{\|R\|}{B-1}$
  - size of in-memory hash table is $\frac{f \cdot \|R\|}{B-1}$, where $f > 1$ is a *fudge factor*
  - probing phase needs to keep hash table and in/output buffer: $B > \frac{f \cdot \|R\|}{B-1} + 2$
  - algorithm needs *approximately* $B > \sqrt{f \cdot \|R\|}$ buffers to perform well
  - sometimes requires multiple passes (recursive partitioning)
    - ∗ input table R has more than $(B-1)^2$ pages
    - ∗ values of R not uniforml enough over partitions

15.1.6 Hybrid Hash Join

- If more than $B > \sqrt{f \cdot \|R\|}$ buffer pages are available, **hybrid hash join** has better performance
- Suppose $B > f \cdot (\|R\|/k)$ for some int k
  - if R is divided into k partitions of size $\|R\|/k$ an in-memory hash table for each partition can be created

  - k output buffers, one input buffer are needed to partition R (and S) => $B - (k+1)$ extra buffers

- Suppose $B - (k+1) \ > \ f \cdot ({}^{\|R\|}/_{k})$, i.e. there is enough space to hold an in-memory hash table for a parition of R

  - *partitioning of* R: build hash table for first partition of R
  - *partitioning of* S: probe hash table with tuples of first partition of S
  - after partitioning phase, first partitions of R and S already joined.

- Saves I/O cost for writing and reading first partitions.

- *Hash Join vs Block Nested Loops Join*:

  - if hash table of entire smaller relation fits into memory, both are the same
  - if both relations are large relative to buffer size => Hash Join is much more effective

- *Hash Join vs. Sort Merge Join*:

  - $3 \cdot (N + M)$ page I/O$
    - cost of *hash join* if there are $B \ > \ \sqrt{\|M\|}$ buffer pages (M is the smaller relation)
    - cost of *sort-merge join*, if there are $B \ > \ \sqrt{\|N\|}$ buffer pages (N is the larger relation)
  - Sort-Merge Join is less sensitive to skew
  - Hash-Join costs less, if buffer pages between $\sqrt{\|M\|}$ and $\sqrt{\|N\|}$
  - Sort-Merge Join produces *sorted result*

15.1.7 General Join Conditions

- *Equalities* over combination of attributes

  - *index nested loops join* can build new index on combination of attributes or use existing index
  - *hash join* and *sort-merge join* partition over combination of attributes

- *Inequalities*

  - *index nested loops join* requires a B+ Tree index
  - *hash join* and *sort-merge join* are not applicable

- other algorithms remain unaffected

15.2 Selection

- selection σ_p => *iteration* and *indexing*
- physical operator depends on *sortedness* and presence/absence of *indexes*:

15.2.1 no index, unsorted data

- read whole input file R_{in}, writes records that satisfy predicate p to output file R_{out}

- no condition on input file (sortedness, ...)

- predicate p can be arbitrary

- I/O cost: $\underbrace{\|R_{in}\|}_{\text{input cost}} + \underbrace{sel(p) \cdot \|R_in\|}_{\text{output cost}}$

  - for pipelined plans, selection applied without output costs

15.2.2 no index, sorted data

- use binary search to find first input, scan for rest
- I/O cost: $\underbrace{\log_2 \|R_{in}\|}_{\text{input cost}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{sorted scan}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$
- binary search cannot be used in pipelined plan, without additional I/O

15.2.3 B+ tree index

- Cost depends on
  - number of qualifying tuples
  - clusteredness
- Implementation:
  - Descend B+ Tree to first entry satisfying p
  - if index is clustered: continue to scan inside R_in
  - if index is unclustered:
    1. gather the *rid*s of all matching entries
    2. sort entries on *rid* field
    3. access pages in sorted *rid* order
- lack of clustering only minor issue, if $sel(p) \approx 0$
- I/O cost: $\underbrace{\approx 3}_{\text{B+ tree access}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{sorted scan}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$

15.2.4 hash index, equality selection

- matching selection predicate p *matches hash index* on $R_{in}.A iff it contains a term of the form$A=c, $where c$ is a constant
- `h(c)` returns the address of the bucket containing qualifying records
- additional cost due to overflow chains may apply
- $sel(p)$ most likely close to 0
- I/O cost: $\underbrace{\approx 1.2}_{\text{Hash Access}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$

15.2.5 General Selection Conditions

- selection predicates like $\sigma_{A\theta c}(R_in)$ are only special case
- More complex predicates need to be supported: use of $\wedge$ (`AND`) and $\vee$ (`OR`) to combine *simple comparisons* of form $A\theta c$, where $\theta \in \{<, \leq, =, \geq, >\}$
- conjunctive predicate p matches a (multi-attribute)
  - hash index, if p *covers* the key
  - B+ tree index, if p is a *prefix* of the key
- **conjunctive Normal Form** CNF => conjuncts of the form $A\theta c$, connected by $\wedge$ (`AND`)
- In practice: conjuncts are connected by $\vee$ (`OR`) (=> disjunctive)
- **Disjunctive Normal Form** (DNF): disjuncts of the form $A\theta c$ connected by $\vee$ (`OR`)
- If the selection predicate is a *conjunction of terms*:

1. *single file scan*
2. *single index*: that matches a subset of (primary) conjuncts and apply all non-primary conjuncts to each retrieved tuple (on-the-fly)
3. *multiple indexes* that each match a subset of conjuncts

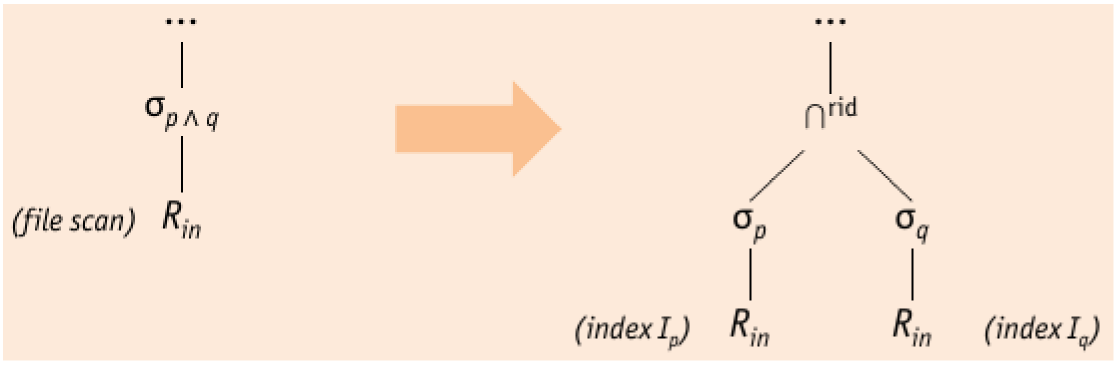
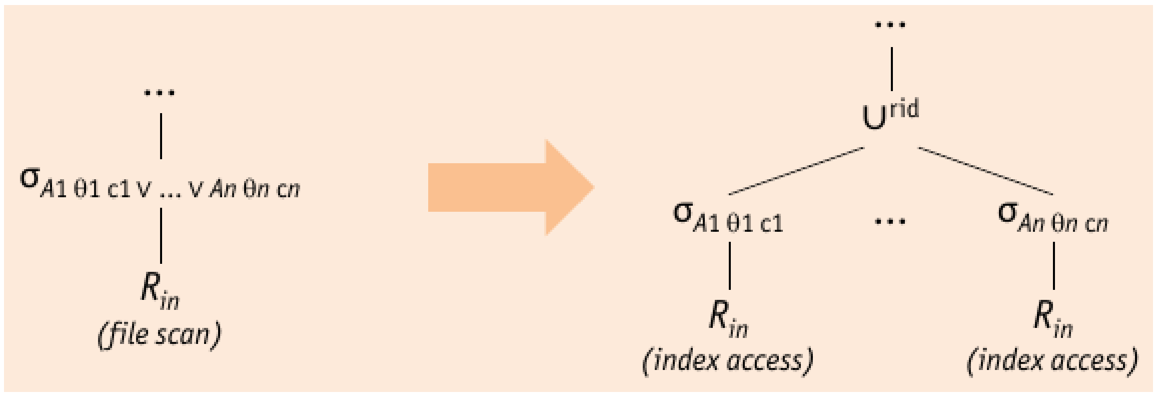


Abbildung 33: Selction without Disjunction: Intersecting *rid* sets

15.2.5.1 Selection without Disjunction

- predicate does not match a single index, both conjuncts p and q match indexes I_p and I_q
  - transform $\sigma_{p \wedge q}(R_{in})$ to $\sigma_p(R_{in}^{I_p}) \cap^{rid} \sigma_q(R_{in}^{I_q})$



...
$\sigma_{A1\,\theta 1\,c1 \vee \ldots \vee An\,\theta n\,cn}$
R_{in}
(file scan)
...
$\cup^{rid}$
$\sigma_{A1\,\theta 1\,c1}$
...
$\sigma_{An\,\theta n\,cn}$
R_{in}
(index access)
R_{in}
(index access)

Abbildung 34: Selection with Disjunction: Unioning *rid* sets

15.2.5.2 Selections with Disjunction

- Choosing reasonable execution plan for *disjunctive selection predicates* is *much harder*
  - if only *one single* term does not match any index => naive scan
  - i *all* terms are supported by indexes, _*rid*-based set union $\cup^{rid}$ can be used

15.2.5.3 Bypass Selections

- Parts of Selection Predicate may be *expensive* to check or *very unselective*
- good strategy to evaluate cheap and selective predicates first
  - $true \vee P \hat{=} true$ (evaluating P is not necessary)
  - $false \vee P \hat{=} P$ (only evaluate P)
- First Method:

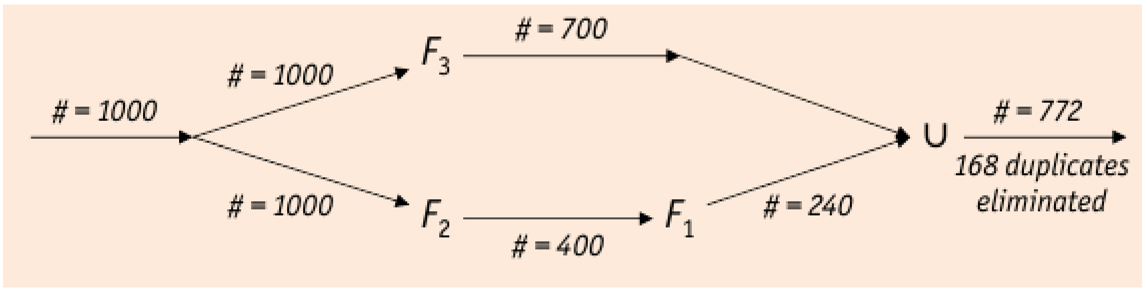


Abbildung 35: Bypass Selection First Method

- convert selection condition to DNF
- Push each tuple from input through each disjunct *in parallel*
- *collect* matching tuples from each disjunct (duplicate elimination)

- Second Method:

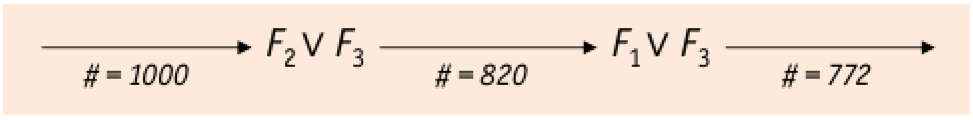
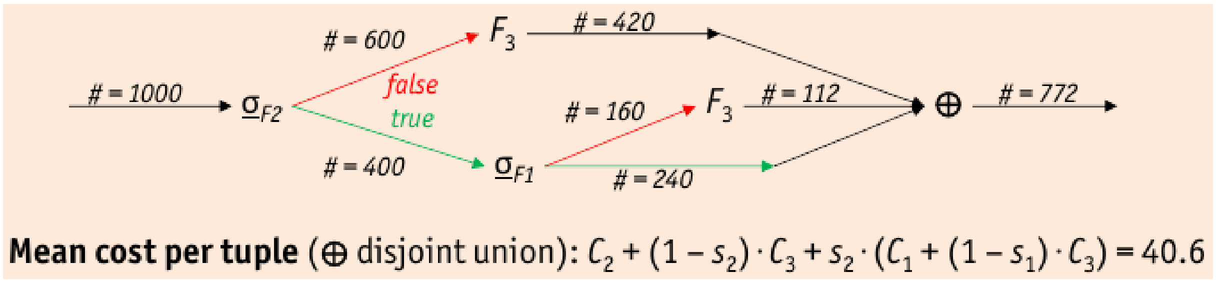


Abbildung 36: Bypass Selection Second Method

- convert selection condition to CNF: $CNF[(F_1 \wedge F_2) \vee F_3] = (F_1 \vee F_3) \wedge (F_2 \vee F_3)$
- push each tuple from input through each conjunct *sequentially*
- matching tuples "survive" conjuncts (*no* duplicate elimination)

- **Goal**: eliminate tuples early, avoid duplication



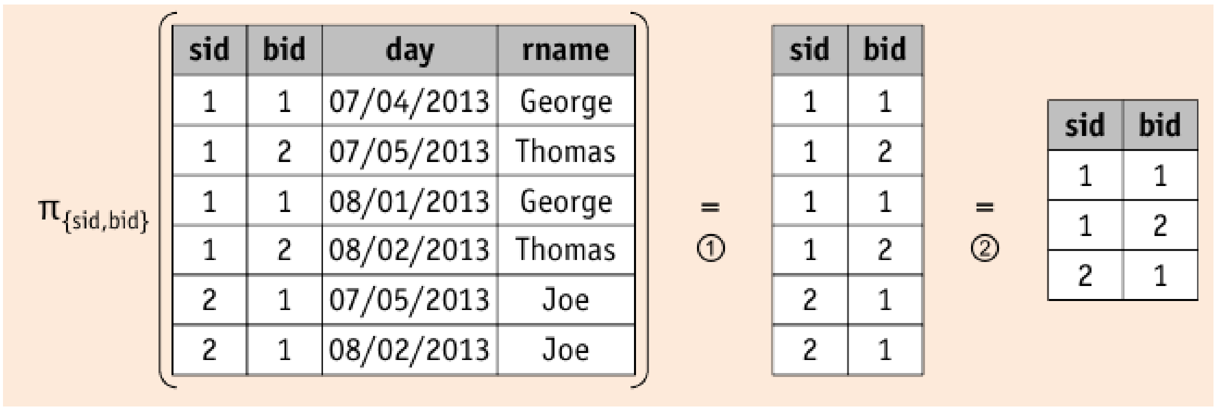
# = 1000
$\underline{\sigma}_{F2}$
# = 600
false
true
# = 400
F_3
# = 420
# = 160
F_3
# = 112
$\underline{\sigma}_{F1}$
# = 240
⊕
# = 772
Mean cost per tuple (⊕ disjoint union): $C_2 + (1 - s_2) \cdot C_3 + s_2 \cdot (C_1 + (1 - s_1) \cdot C_3) = 40.6$

Abbildung 37: Bypass Selection

- Bypass Selection Operator: $\underline{\sigma}_F$
  - produces two disjoint outputs: `true` and `false`
  - Bypass results derived from conjunctive normal form

15.3 Projection

- projection π_l modifies each record in its input file:
  1. *removes* unwanted attributes (not in attribute list l)
  2. *eliminates* any duplicate tuples (SQL `DISTINCT`)
- resulting file will be only fraction of original size
- projection uses *iteration* and *partitioning*
- projection *without duplicate elimination* can be fully *pipelined*



$\pi_{\{sid,bid\}}$
= ①
= ②

sid	bid	day	rname
1	1	07/04/2013	George
1	2	07/05/2013	Thomas
1	1	08/01/2013	George
1	2	08/02/2013	Thomas
2	1	07/05/2013	Joe
2	1	08/02/2013	Joe

sid	bid
1	1
1	2
1	1
1	2
2	1
2	1

sid	bid
1	1
1	2
2	1

Abbildung 38: Projection

- projection *with duplicate eliminiation* hat to *materialize* intermediate results
- Duplicate elimination:
  - compare *each tuple* to *all other tuples* to check whether its a duplicate
  - generally its not possible to fit all other tuples in memory
  - => *partitioning*
    - given B buffer pages, group "similar" tuples into partition
    - load partitions into memory one after another
    - only compare a tuple to other tuples in that partition

15.3.1 Projection Based on Sorting

- Records will fields equal will be adjacent after sorting
  1. scan R_{in} and output set of tuples contain *only* desired attributes
  2. scan this set of tuples using *all* of its attributes as sort key
  3. scan the sorted result, comparing adjacent tuples and discard duplicates
- π_l^{sort} has sorted result
- combine sorting and projection with duplicate elimination
  - integrate *projection* and *duplicate elimination* into merge sort algorithm
  - **Pass 0**:
    1. read B pages at a time, *projecting unwanted attributes out*
    2. use in-memory sort to sort records of these B pages
    3. write a sorted run of B internally sorted pages to disk
  - **Pass 1, . . . , n**
    1. select $B-1$ runs from previous pass, read a page from each run
    2. perform a $(B-1)$-way merge, *eliminating duplicates*
    3. use the B-th page as an output buffer
- I/O cost: $\underbrace{\|R_{in}\| + \|R_{tmp}\|}_{\text{projection}} + \underbrace{2 \cdot \|R_{tmp}\| \cdot \left(\lceil \log_{B-1} \cdot \lceil \|R_{tmp}\|/B \rceil \rceil\right)}_{\text{duplicate elimination}}$

15.3.2 Projection Based on Hashing

- $\pi_l^{hash}(R_{in})$: needs large number of buffer pages B relative to total number of pages in R_{in}
- two phases:
    1. **partitioning**:
        1. Allocate B buffer pages: one *input buffer*, $B-1$ *hash buckets*
        2. Read file R_{in} for each record r: *project out* attributes not listed in l.
        3. For each record, apply hash function $h_1(r) = h(r) \mod (B-1)$, which depends on *all remaining fields of* r, store in hash bucket $h_1(r)$.
        4. If hash bucket is *full*, write it to disk
    2. **duplicate elimination**
        1. For each partition, read each partition page by page (possibly in parallel), using same buffer layout as before.
        2. each record: apply hash function h_2 ($h_2 \neq h_2$) to all record fields
        3. Only if two records collide with respect to h_2, check if $r = r'$, if yes, discard r'.
        4. After entire partition has been read in, append all hash buckets to result file, free of duplicates.
- I/O cost: $\underbrace{\|R_{in}\| + \|R_{tmp}\|}_{\text{projection}} + \underbrace{\|R_{tmp_2}\| + \|R_{tmp_2}\|}_{\text{duplicate elimination}}$
    - R_{tmp_2} may be smaller than R_{tmp}

15.3.3 Sorting versus Hashing

- Sorting is standard approach for duplicate elimination.
    - superior to hashing, if there are many duplicates
    - superior if distribution of (hash) values very non-uniform
    - sorting already required for other operations and therefore already implemented in most systems
- For $B > \sqrt{\|R_{tmp}\|}$ buffer pages, both approaches have same cost.
- Choice depends on *CPU cost*, desireability of *sort order*, *skew* in value distribution, …

15.3.4 Using Indexes for Projection

- Index contains *all attributes* retained in projection
    - use index-only plan (don't access data records)
    - apply hashing or sorting to eliminate duplicates from (much smaller set of pages)
    - Index key contains projected attributes as *prefix* and is *sorted*

15.4 Set Operations

- **Intersection** and **Cross-Product**: special case of *join*
    - *equality on all attributes* as join condition => *intersection*
    - `true` as join condition computes *cross-product*
- **Union** and **Difference**: selection with complex selection condition
    - *union*: main problem is *duplicate elimination*
    - *difference*: variantion of *duplicate elimination*
    - use *sorting* or *hashing* for *duplicate elimination*

15.4.1 Union and Difference using Sorting

- $R \cup S$
  1. sort both R and S using combination of *all fields*
  2. scan sorted R and S in parallel and merge them, eliminating duplicates

  - For difference $R \setminus S$, keep records from R if they do *not appear* in S
- integrate with *external sort operator*

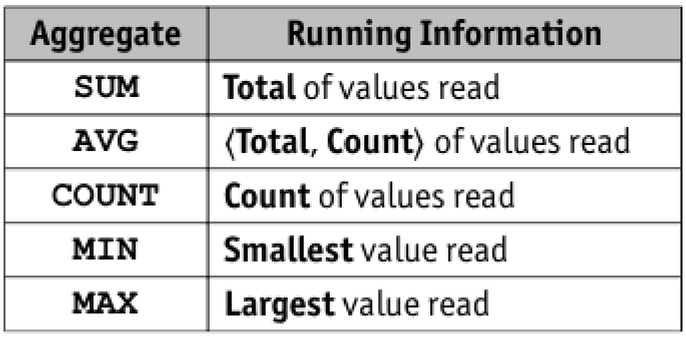
15.4.2 Union and Difference using Hashing

- $R \cup S$
  1. Partition both R and S using a hash function $h(\cdot)$ over combination of *all* fields.
  2. Process each record i as follows:
     - build in-memory hash table using hash function $h_2(\cdot) \neq h(\cdot)$ for $S[i]$
     - scan $R[i]$, for each tuple probe hash table for $S[i]$, if tuple is in hash table, discard it, otherwise add it to table
     - Write out hash table, clear it to prepare for next partition.

  - For difference $R \setminus S$, partition differently. After building in-memory hash table for $S[i]$, $R[i]$ is scanned and $S[i]$ is probed for every tuple in $R[i]$. If tuple is *not in tuple*, it is written to result.

15.5 Aggregate Operations

- `SUM`, `AVG`, `COUNT`, `MIN`, `MAX`
- basic algorithm



Aggregate	Running Information
SUM	**Total** of values read
AVG	**⟨Total, Count⟩** of values read
COUNT	**Count** of values read
MIN	**Smallest** value read
MAX	**Largest** value read

Abbildung 39: Aggregate Operations and their running information

  - scan whole relation (on-the-fly ?), maintain *running information*
  - upon completion of scan, compute aggregate value from running information
- For *aggregation combined with grouping*, two evaluation algorithms based on sorting and hashing respectively
  - **Sorting Approach** (cost equal to I/O cost of sorting)
    * sort on grouping attributes
    * scan again to compute result of aggregate operation for each group
    * refinement => aggregation can be done as part of sorting
  - **Hashing Approach** (if hash table fits into memory, cost is $\|R\|$)

    * build (in-memory) hash table on grouping attributes with entries `<grouping value, running information>`
    * for each tuple of relation, probe hash table to find entry of group to which tuple belongs and update running information
    * when hash table is complete, its entries for grouping value can be used to compute the corresponding result tuples in a straightforward way

- index is not applicable to select subset of tuples
- under certain conditions, index entries instead of data records can be used to evaluate aggregate operations eeficiently
  - if index search key includes *all attributes* needed for aggregation => use index only plan
  - if `GROUP BY` clause attribute list forms a prefix of index search key, index is a tree => sorting can be avoided => index only

15.6 Impact of Buffering

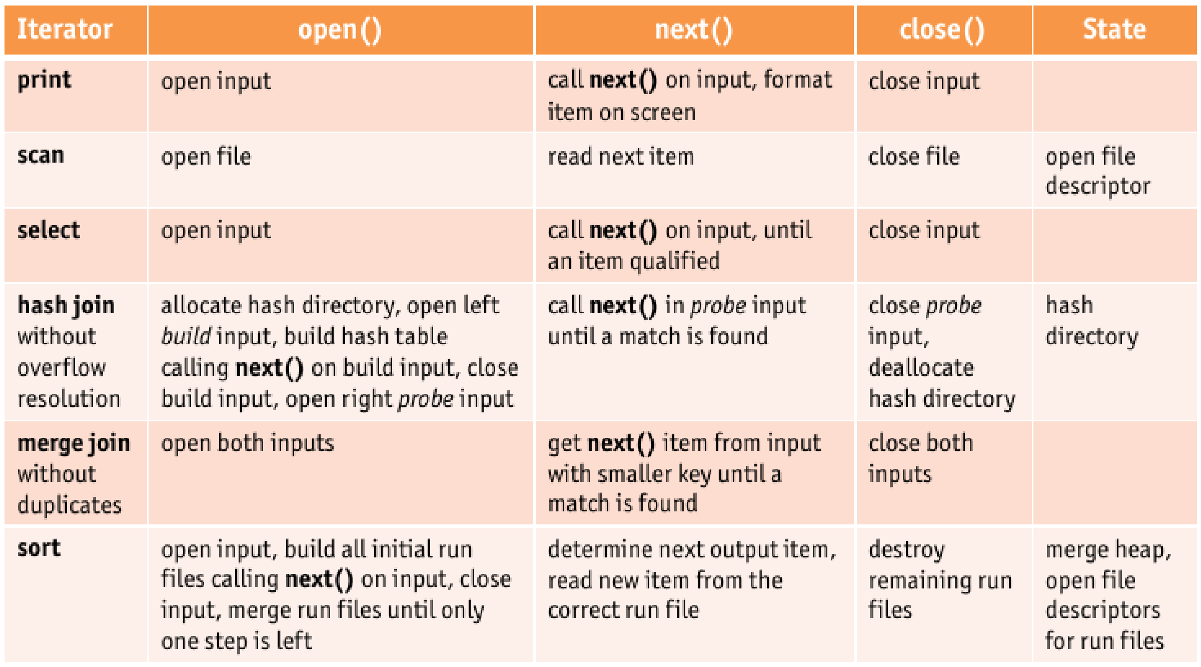
- Effective use of buffer pool is very important.
  - crucial for efficient implementation of relational query engine
  - several operators use size of available buffer space as parameter
- operators that execute concurrently have to **share** buffer pool
- if tuples are accessed through an index, likelihood of finding a page in buffer pool becomes (unpredictably) dependent on its *size* and _replacement policy
- if tuples are accessed through unclustered index, buffer pool *fills up quickly*, as each tuple likely requires new page to be brought into buffer pool
- if an operation has a *repeated pattern of page access*, replacement policy and/or number of buffer pages can speed up operation significantly

15.7 Materialization vs. Pipelining

- Implementations assume materialization so far
  - Communication through secondary storage => lots of I/O
  - operator cannot start, until inputs are fully materialized
  - operators have to be executed in sequence
  - first result is only available after last operator has executed
- pipelined evaluation => avoid writing temporary files whenever possible
  - each operator passes its results directly to next operator
  - results propagate, as soon as they are available
  - as soon as input is available, computation can start
  - operators can execute in parallel
  - typically implemented in *open-next-close* interface or *"Volcano iterator"* model
    * each operator implements the functions:

    `open()`: *initialize* the operators internal state
    `next()`: produce and return *next result tuple* or `<EOF>`
    `close()`: *free* any internal ressources, typically after all tuples have been processesd

    * *state* is kept within each operators instance
      1. upon call to `next()`, operator produces a tuple, updates internal states, then pauses



Iterator	open()	next()	close()	State
print	open input	call **next()** on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call **next()** on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left *build* input, build hash table calling **next()** on build input, close build input, open right *probe* input	call **next()** in *probe* input until a match is found	close *probe* input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get **next()** item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling **next()** on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Abbildung 40: Pipelined Evaluation

2. upon subsequent calls to `next()`, operator uses internal state to resume and produce another tuple

 – iterator implements *demand drive* processing infrastructure

- blocking operators cannot be implemented in pipelined fashion
- Alternative processing infrastructure is *data-driven*
  – Producers and consumers are connected by *queue*
  – operators execute *asynchronously* and in *parallel*
  – produce results as fast as possible and *enqueue* them
- queues buffer data that is pipelined between operators
- queues suspend operators using blocking queue/enqueue calls
- Data-driven pipelines can *exploit more parallelism* that demand-drive

16 Query Optimization

- To evaluate given SQL query Q
  1. parse and analyse Q
  2. derive *relational algebra* expression E that computes Q
  3. generate set of *logical plan* L by transforming and simplifying E
  4. generate a set of *physical plans* P by annotating each plan in L with access methods and operator algorithms
  5. for each plan in P, estimate quality (cost) of plan and choose the best plan as final plan
- Query optimizer does last three steps:
  – plan enumeration in steps 3 and 4
  – algebraic (or rewrite) query Optimization in step 3
  – non-algebraic (or cost-based) query optimization in steps 4 and 5

- Parser: creates internal representation of input query
  - each `SELECT-FROM-WHERE` clause is translated into *query block*
- Query Optimizer: consider each query block and choose evaluation plan for this block
  - focus on single block queries for now
  - optimization of *nested queries* discussed separately
- To find best plan, query optimizer explores *search space*
  - *logical level*: relational algebra equivalences
  - *physical level*: access methods and operator algorithms

16.1 Relational Algebra Equivalences:

- Each query block is a relational algebra expression consisting of cross-product, selection and projection
- rewrite optimization applies *relational algebra equivalences* to transform query blocks
  - cross products => join
  - choose different join orders
  - push selections and projections ahead of joins
- Two relational algebra expressions E_1, E_2 are *equivalent* if they generate the same set of tuples on every legal database instance
  - $E_1 \equiv E_2$
  - Rules may be applied in both directions
- Selections
  1. conjunctive selections can be deconstructed into sequence of individual selections (*cascading selections*)
  $$\sigma_{c1 \wedge c2 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\dots(\sigma_{cn}(R))\dots))$$
  2. selections are *commutative*
  $$\sigma_p(\sigma_q(R)) \equiv \sigma_q(\sigma_p(R))$$
- Projections
  1. only the last projection in a sequence of projections is needed, the others can be omitted (*cascading projections*)
  $$\pi_{a1}(\pi_{a2}(\dots(\pi_{an}(R))\dots)) \equiv \pi_{a1}(R)$$
- Cross-Products and Natural Joins
  1. *commutative*
  $$R \times S \equiv S \times R$$
  $$R \bowtie S \equiv S \bowtie R$$
  2. *associative*
  $$R \times (S \times T) \equiv (R \times S) \times T$$
  $$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$
- General Joins

1. *associative*

$$(R \bowtie_p S) \bowtie_{q \wedge r} T \equiv R \bowtie_{p \wedge q} (S \bowtie_r T)$$

where r involves only attributes of S and T

- Selections, Cross-Product, Join

1. selections can be *combined* with cross-product to form join

$$\sigma_p(R \times S) \equiv R \bowtie_p S$$

2. selection can be combined with join

$$\sigma_p(R \bowtie_q S) \equiv R \bowtie_{p \wedge q} S$$

3. selections *commute* with cross-products and joins, if predicate p involves attributes of R only

$$\sigma_p(R \times S) \equiv \sigma_p(R) \times S$$
$$\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$$

4. selections *distribute* over cross-product and join, if predicates p only involves attributes of R and predicate q only involves attributes of S

$$\sigma_{p \wedge q}(R \times S) \equiv \sigma_p(R) \times \sigma_p(S)$$
$$\sigma_{p \wedge q}(R \bowtie_r S) \equiv \sigma_p(R) \bowtie_r \sigma_p(S)$$

- Projections, Selections, Cross-Product, Join

1. projection and selection *commute*, if the selection predicate p only involves attributes retained by the projection list a

$$\pi_a(\sigma_p(R)) \equiv \sigma_p(\pi_p(R))$$

2. projection *distributes* over cross-product, where a_1 is the subset of attributes in a that appear in R and a_2 is the subset of attributes that appear in S

$$\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$$

3. projection *distributes* over join where $a1$ is the subset of attributes in a that appear in R, $a2$ is a subset of attributes that appear in S and predicate p only involves attributes in $a_1 \cup a_2$

$$\pi_a(R \bowtie_p S) \equiv \pi_{a1}(R) \bowtie_p \pi_{a2}(S)$$

- Union and Intersection

1. are *commutative*

$$R \cup S \equiv S \cup R$$
$$R \cap S \equiv S \cap R$$

2. are *associative*

$$(R \cup S) \cup T \equiv R \cup (S \cup T)$$
$$(R \cap S) \cup T \equiv R \cap (S \cap T)$$

- Projection and Union

1. projections *distribute* over union

$$\pi_a(R \cup S) \equiv \pi_a(R) \cup \pi_a(S)$$

- Selection, Union, Intersection, Difference

  1. are *distributive*

$$\sigma_p(R \cup S) \equiv \sigma_p(S) \cup \sigma_p(R)$$
$$\sigma_p(R \cap S) \equiv \sigma_p(S) \cap \sigma_p(R)$$
$$\sigma_p(R \setminus S) \equiv \sigma_p(S) \setminus \sigma_p(R)$$

  2. Intersect and Difference are *commutative* (does *not* apply to Union)

$$\sigma_p(R \cap S) \equiv \sigma_p(S) \cap R$$
$$\sigma_p(R \setminus S) \equiv \sigma_p(S) \setminus R$$

16.2 Rewrite Optimization

- Optimization is guided by heuristics:

  1. **Break apart conjunctive selections** into a sequence of simpler selections
  2. **Move selections down the query tree** to reduce the cardinality of intermediate results as early as possible
  3. **Replace Selection-Cross-Product pairs with join** to avoid large intermediate results
  4. Break lists of projection attributes apart, move them down the query tree** and **create new projections where possible** to reduce tuple widths as early as possible
  5. **Perform joins with the smallest expected result first** (cost based heuristic)

- Implicit join predicates can be turned into explicit one to enable more join orders.

- Plan enumeration

16.3 Plan Enumeration

- Search space given by:

  – relational algebra equivalences
  – choice of implementation technique for relational operator
  – choice of access method based on presence of indexes
  – other available resources, e.g. number of buffer pages, etc.

- explicit enumeration is impossible in general => prohibitively large space

- important cases:

  – single-relation queries
  – multiple relation queries

16.3.1 Single-Relation Queries

- Optimizer enumerates *all possible plans* and assess cost

- main decision is access method used

  – plan without indexes
  – plan utilizing and index

- different single-relation queries might have different *physical properties*, i.e. sort order

- for each set of *physical properties*, plan with least cost is retained

- plans without index
    1. heap file scan on (single) relation
    2. apply selection and projection (without duplicate elimination) on-the-fly
    3. sort according to `GROUP BY` clause
    4. apply aggregation and `HAVING` clause on-the-fly to each group
- Single-index access path
    1. choose index that is estimated to retrieve the fewest pages
    2. apply projections and non-primary selections
    3. proceed to compute grouping and aggregation operations by sorting
- Multiple-index access path (for index entry variants 2 and 3)
    1. retrieve and intersect *rid* sets and sort results by page id
    2. retrieve tuples that satisfy primary condition of all indexes
    3. apply projections and non-primary selection terms, followed by grouping and aggregation operations
- Sorted-index access path
    1. retrieve tuples in order required by `GROUP BY` clause
    2. apply selection and projection to each retrieved tuple on-the-fly
    3. compute aggregate operations for each group on-the-fly
- Index-Only access path
    1. retrieve matching tuples or perform an index-only scan
    2. apply (non-primary) selection conditions and projections to each retrieved tuple on-the-fly
    3. possibly, sort the result to achieve grouping
    4. compute aggregate operations for each group on-the-fly

16.3.2 Multiple Relation Queries

- Queries over multiple relations require joins (or cross-products)
- since joins are expensive, good plan is very important
- *size of final result* can be estimated by taking product of (size of relations) and (reduction factors of all selection predicates)
- *size of intermediate results* can vary substantially, depending on join order
- number of possible plans much to large (catalan numbers)
    - only consider **left-deep plans**
        - Can be fully pipelined, because inner relation is already materialized (i.e. read)
        - indexes on inner relations can be utilized
        - number of possible left-deep joins is "only" $n!$
    - Optimizer has to make sure it selects the best *left-deep* plan
- Enumerate candiate plans:
    - directly prune cross-products
    - For each candidate plan: enumerate possible join algorithm
        - For each candidate plan: enumerate access methods
            - For each candidate plan: estimate cost
- Cost estimation will enumerate multiple identical sub-plans

  - use **dynamic programming** to memoize already considered sub-plans
  - find cheapest plan for n-way join in n *passes*
  - in each pass k , find the best plan for all *k-relation sub-plans*
  - construct the plans in pass k by joining another relation to the best $(k-1)$-relation sub-plans found in earlier passes

- Dynamic Programming Algorithm
  - **Pass 1**: (all 1-relation plans)
    * find best 1-relation plan for each relation
    * this pass mainly consists of seleccting access methods
    * also see discussion on single-relation queries
    * keep the best 1-relation plans for each set of physical properties
  - **Pass** n: (all n-relation plans)
    * find best way to join sub-plans from Pass $n-1$ to the nth relation
    * sub-plans of Pass $n-1$ appear as outer relation in this join
    * Return the *overall best* plan

- Dynamic Programming record *cost* and *result size estimates* for each retained plan

- pruning for each subset of joined relations:
  - keep cheapest *overall* sub-plan
  - keep cheapest sub-plan that generates and intermediate result with an *interesting order* of tuples
    * presence of SQL `ORDER BY` clause in query
    * presence of SQL `GROUP BY` clause in query
    * join attributes of subsequent equi-joins (prepare for merge join)
  - discard sub-plans that involve *cross-products* immediately
  - Subroutines in dynamic programmin algorithm:
    * `accessPlans(` R `)`: enumerates all access plans for a single relation R
    * `possibleJoins(` $R \bowtie S$ `)`: enumerates the possible joins between relations R and S, e.g. $\bowtie^{NL}$, $\bowtie^{SM}$, $\bowtie^{H}$
    * `prunePlans(` *set* `)`: discard all but the best plans from *set*

- `optimize()` draws it advantage from *filtering* candidate plans early

- heuristics are used to reduce the search space and balance *plan quality* and *optimizer runtime*

16.3.3 Joining Many Relations

- Star: Join all relations to on center relation.

- Chain: Join subsequent relations to temporary table.

- Join enumeration has *exponential* resource requirements
  - time complexity: $\mathcal{O}(3^n)$
  - space complexity: $\mathcal{O}(2^n)$

- might still be too expensive
  - for many relations (~10-20 and more)
  - for simple queries with many indexes (where the choice should have been easy)

- **greedy join enumeration** tries to close this gap
  - in each iteration, choose *cheapest* join that can be made over remaining sub-plans (greedy)

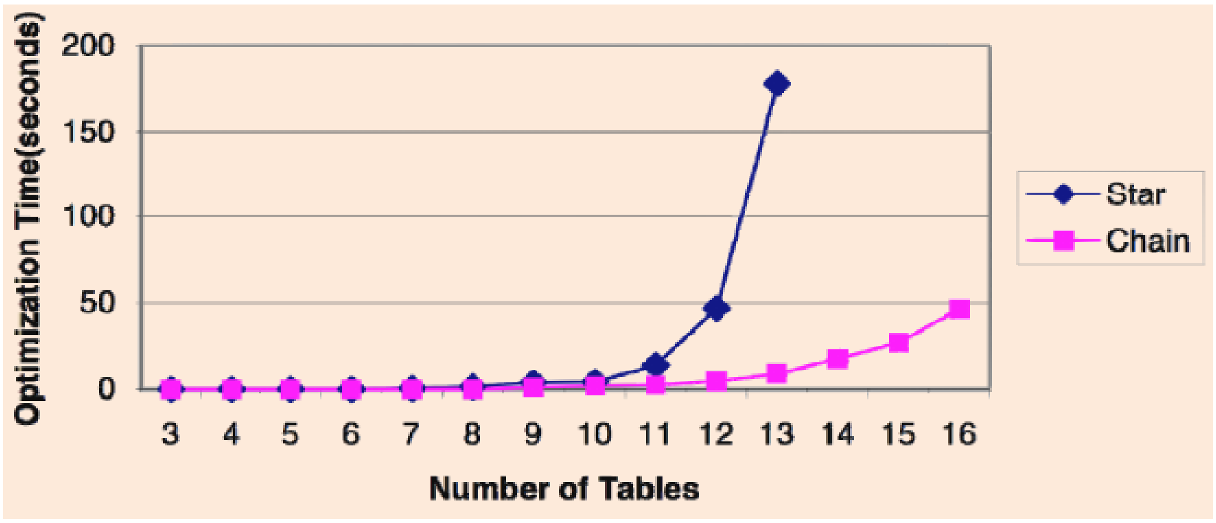
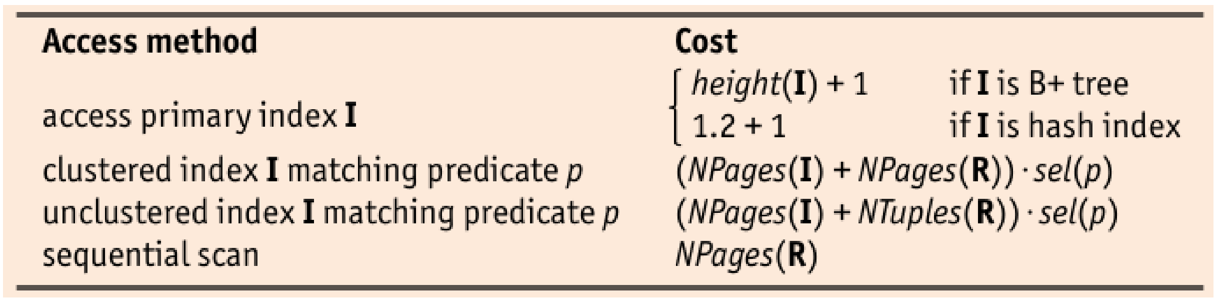


Abbildung 41: Optimizer Runtime for Joining many relations

  - `optimize-greedy()` operates similar to finding binary tree for *Huffman coding*
  - time complexity: $\mathcal{O}(n^3)$
    * each loop with n iterations: $\mathcal{O}(n)$
    * each iteration looks at all remaining pairs of plans in *worklist*: $\mathcal{O}(n^2)$
  - consider all types of join trees: mitigate risk of returning a really bad one

- other possible ways to replace greedy algorithm:
  - **randomized algorithm**: randomly rewrite join tree, using *hill-climbing* or *simulated annealing* to find optimal plan
  - **genetic algorithms**: explore plan space by *combining* plan ("offspring") and *altering* some plans randomly ("mutations")

16.3.4 Cardinality Estimation



Access method	Cost	
access primary index **I**	$\begin{cases} height(\mathbf{I}) + 1 \\ 1.2 + 1 \end{cases}$	if **I** is B+ tree / if **I** is hash index
clustered index **I** matching predicate p	$(NPages(\mathbf{I}) + NPages(\mathbf{R})) \cdot sel(p)$	
unclustered index **I** matching predicate p	$(NPages(\mathbf{I}) + NTuples(\mathbf{R})) \cdot sel(p)$	
sequential scan	$NPages(\mathbf{R})$	

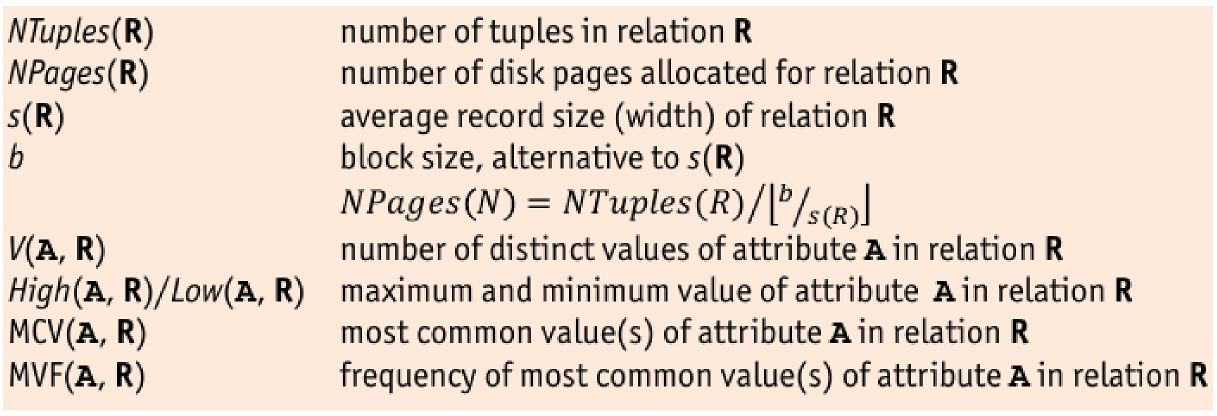
Abbildung 42: Cost model for access methods on relation R

- Cost estimation of multi-relation plans additionally involves **cardinality estimation** of intermediate query results
  - cost is dominated by page I/O
  - I/O is determined by size of inputs
- Cardinality estimates are used to allocate buffer pages and to determine blocking factor b for blocked I/O
- A **database profile** is one of two principal approaches to query result cardinality estimation.

  - **base relations**: maintain statistical information, e.g. number and sizes of tuples, distribution of attribute values, . . . in *database catalog*
  - **intermediate query results**: derive this information based on a simple statistical model during query optimization.
  - statistical model assumes *uniformity* and *independence*
  - both are typically *not valid*, but allow for simple calculations
  - system can record *histograms* that approximate value distribution to provide better cardinality estimates
  - Alternatively, *sampling techniques* can be used to estimate query result set cardinality
    1. run query on *small sample* of database
    2. *gather statistics* about query plan
    3. *extrapolate* to full input size

    ∗ balance between sample size (performance) and resulting accuracy of estimation must be found

- Keep **profiling information** in *database catalog*



NTuples(**R**)	number of tuples in relation **R**
NPages(**R**)	number of disk pages allocated for relation **R**
s(**R**)	average record size (width) of relation **R**
b	block size, alternative to *s*(**R**)
	$NPages(N) = NTuples(R) / \lfloor {}^{b}/_{s(R)} \rfloor$
V(**A**, **R**)	number of distinct values of attribute **A** in relation **R**
High(**A**, **R**)/*Low*(**A**, **R**)	maximum and minimum value of attribute **A** in relation **R**
MCV(**A**, **R**)	most common value(s) of attribute **A** in relation **R**
MVF(**A**, **R**)	frequency of most common value(s) of attribute **A** in relation **R**

Abbildung 43: Examples for Database Profiling Information

- in order to obtain simple and tractable estimation, assume one of:

  1. **Unifmormity and independence assumption**: values uniformly appear with same probability, values of different attributes are independent of each other
  2. **Word Case assumption**: no knowledge about relation contents available => calculate upper bounds
  3. **Perfect Knowledge assumption**: details about exact distribution are known => huge catalog or prior knowledge of incoming queries is needed

16.3.5 Histograms

- uniformity not realistic

- histograms need to be maintained

  1. divide active domains of A into *adjacent intervals* (=> buckets) by selecting boundary values $b_i \in dom(A)$
  2. collect *statistical parameters* for each interval such as the number of tuples $b_{i-1} < t[A] \leq b_i$ or the number of distinct A values in that interval

- Histograms allow for more exact estimates of both *equality* and *range selections*

- two common types:

1. **Equi-Width Histograms** All histogram buckets have the *same width.*
2. **Equi-Depth Histograms** All histogram buckets contain the *same number of tuples* => varying width
   - better adaptable to data skew

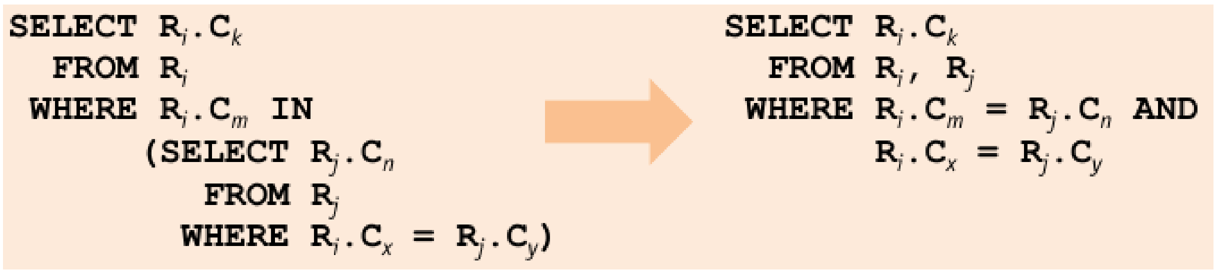
- Number of histogram buckets can be used to control *resolution* <–> *size* tradeoff
- Maintaining histogram under insertions and deletions => update histogram
- inside bucket: assume uniformity

16.3.6 Sampling

- Maintaining histograms can be *costly* and *error prone*
- sampling might be better instead
- parameters: one parameter can be chosen, the other is then calculated
  - sample size
  - extrapolation precision
- usual approaches
- *adaptive sampling*: tries to achieve given precision with minimal sample size
- *double (two-phase) sampling*: obtain a course picture from very small sample => compute sample size in second step
- *sequential sampling*: uses sliding (continuous) calculation of characteristics => stop estimation once precision is high enough

16.4 Nested Subqueries

- consists of multiple query blocks
- Optimization options depend on
  - form of `WHERE` predicate
  - presence or absence of *aggregates* in subquery
  - whether subquery is *uncorrelated* or *correlated*
  - other characteristics
- **SQL Canonical Form**: if the `FROM` clause only contains table names and its `WHERE` clause only contains simple and join predicates
- **correlated subquery**: if it references table or tuple variable of the top-level query
- Types of Nesting:
  - **Type A**: uncorrelated, aggregated subquery
  - **Type N**: uncorrelated, not aggregated subquery
  - **Type J**: correlated, not aggregated subquery
  - **Type JA**: correlated, aggregated subquery
  - **Type D**: correlated, division subquery => no longer possible, as `CONTAINS` was removed
- Nested Subquery Rewrite:
  1. rewrite query to **Canonical Form**
  2. Based on type of subquery, one of two algorithms:
     - `NEST-N-J`: transform nested subquery of Type N and Type J to a query in canonical form

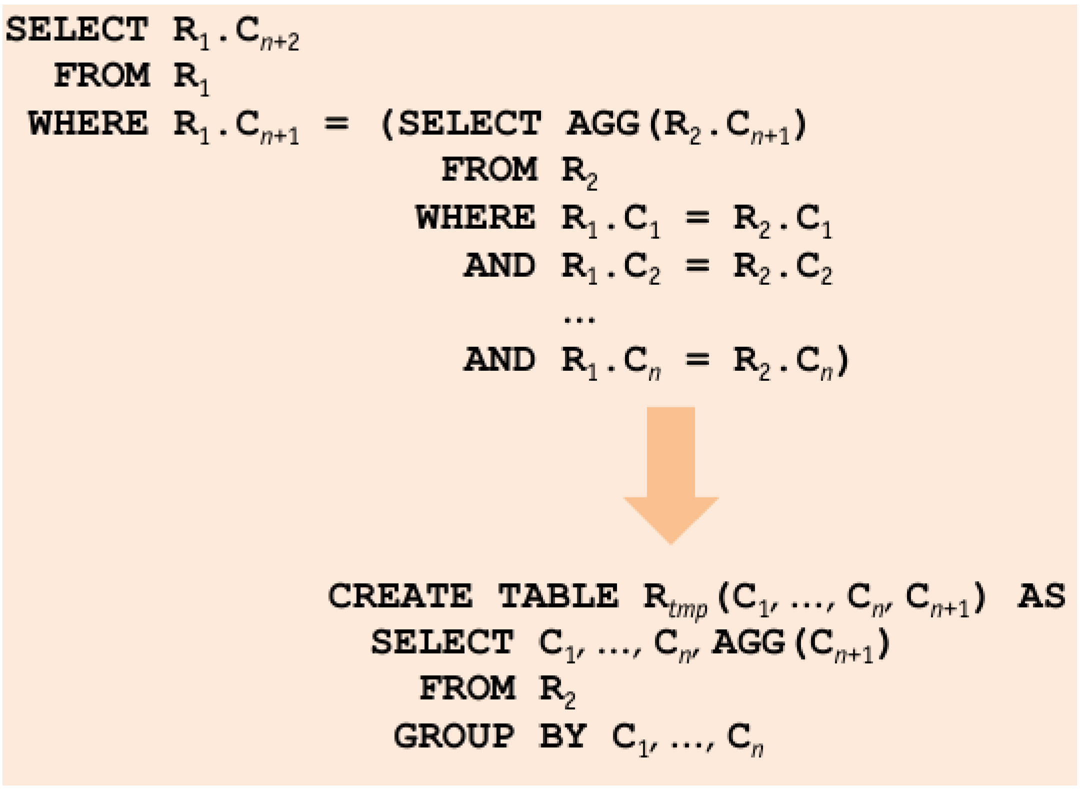


```
SELECT R_i.C_k                               SELECT R_i.C_k
  FROM R_i                                     FROM R_i, R_j
 WHERE R_i.C_m IN                             WHERE R_i.C_m = R_j.C_n AND
      (SELECT R_j.C_n                               R_i.C_x = R_j.C_y
         FROM R_j
        WHERE R_i.C_x = R_j.C_y)
```

Abbildung 44: `Nest-N-J` algorithm

1. combine `FROM` clauses of all query blocks into one `FROM` clause
2. combine `WHERE` clauses of all query blocks using `AND`, replacing element test (`IN`) with equality (`=`)
3. retain `SELECT` clause of outermost query block

– `NEST-JA`: use temporary tables to remove one level of nesting for subqueries of type JA



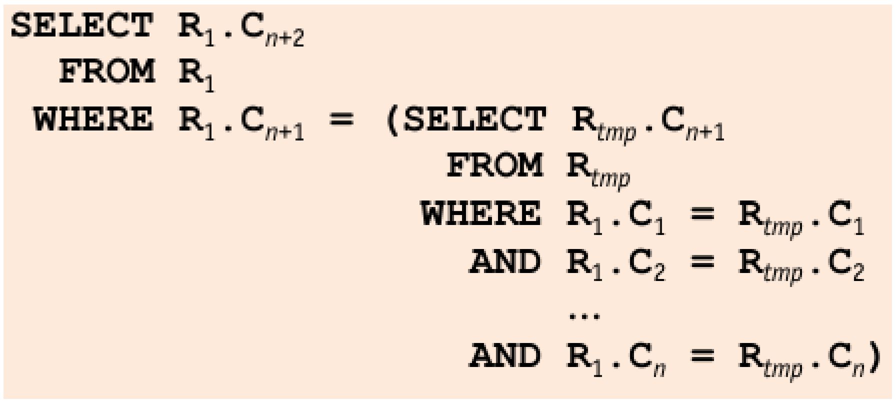
```
SELECT R_1.C_{n+2}
  FROM R_1
 WHERE R_1.C_{n+1} = (SELECT AGG(R_2.C_{n+1})
                        FROM R_2
                       WHERE R_1.C_1 = R_2.C_1
                         AND R_1.C_2 = R_2.C_2
                         ...
                         AND R_1.C_n = R_2.C_n)

CREATE TABLE R_tmp(C_1, ..., C_n, C_{n+1}) AS
  SELECT C_1, ..., C_n, AGG(C_{n+1})
    FROM R_2
   GROUP BY C_1, ..., C_n
```

Abbildung 45: `NEST-JA` algorithm part 1

1. project join column of R_1 and restrict it with any simple predicates applying to R_1
2. create temporary relation $R_{tmp}byjoining$R_1$ and R_2 using same operator as join predicate in original query
   * if aggregate function is `COUNT`, join must be outer join
   * if aggregate function is `COUNT(*)`, compute aggregate over join column
   * include join column(s) and aggregated column in `SELECT` clause
   * include join columns(s) in `GROUP BY` clause
3. join R_1 to R_{tmp} according to transformed version of original query by changing join predicate to equality (`=`)

* transform resulting Type J query with `NEST-N-J`



```
SELECT R_1.C_{n+2}
  FROM R_1
 WHERE R_1.C_{n+1} = (SELECT R_{tmp}.C_{n+1}
                        FROM R_{tmp}
                       WHERE R_1.C_1 = R_{tmp}.C_1
                         AND R_1.C_2 = R_{tmp}.C_2
                             ...
                         AND R_1.C_n = R_{tmp}.C_n)
```

Abbildung 46: `NEST-JA` algorithm part 2

16.5 Example Query Optimizers

- **Bottom-Up**: start with individual relations of query
  - global cost of a plan *must* be computed bottom-up as the cost of each operator depends on the cost of its inputs
  - dynamic programming *requires* breadth-first enumeration to pick the best plan
  - impossible to pick best plan until its cost has been computed
- **Top-Down**: start with entire expression of query
  - operators may *require* certain properties (e.g. order or partitioning)
  - limit exploration based upon context of use
  - prune based on upper and lower bound
- **Starburst**: *query graph model* (QGM)

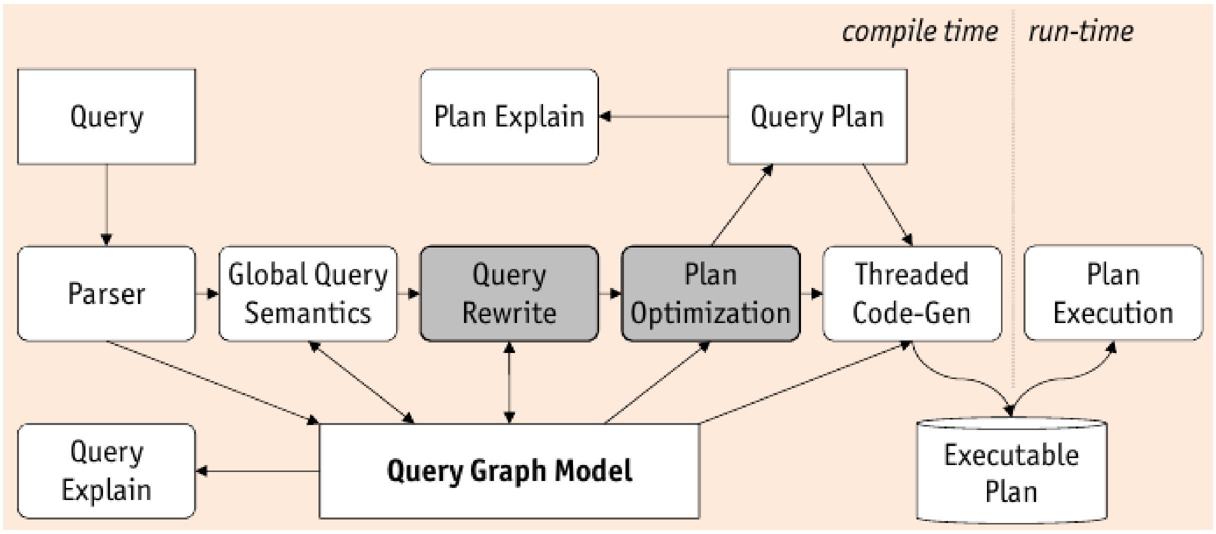


Abbildung 47: Query Graph Model: Query Compiler Overview

1. *query rewrite* uses rules to transform a QGM representation of a query into another equivalent QGM representation.
2. *plan optimization* dreives a query execution plan from QGM representation of a query bottom up using production rules

- first phase performs rewrite (heuristic) optimization, second phase uses cost-based optimization

- **Cascades**: transformation-based, top-down approach:
  - *depth-first search*: beginning with original query, consider subqueries and optimize
  - *goal-driven* no need to maintain bottom-up interesting orders
  - Fully Cost-Based
  - flexible and extendible:
    * **operators**: expressions consisting of logical, physical, and item operators => represent query trees and execution plans
    * **rules**: plan search space ise defined by set of transformation, implementation and enforcer rules
    * **strategy**: search space exploration strategy is guided by sequence of optimization tasks
- Query trees and execution plans are represented as expressions
  - Logical Operators:
    * matching rule pattern to expressions
    * hashing for detecting duplicate expressions
    * deriving logical properties (e.g. schema) from input
  - Physical Operators:
    * computing cost from opertor algorithm and input cost
    * checking cost limit between optimizatin of two inputs
    * translation optimization goals to input operators
  - Item Operators:
    * represent selection predicates for easy manipulation by rules

16.5.1 Starburst vs. Cascades

- Phases:
  - Starburst optimizes queries in *two phases*
    * (heuristic) query rewrite phase
    * (cost-based) plan optimization phase
  - Cascades uses one phase with only cost-based transformations
- Mapping logical to physical operators
  - Starburst expands expressions *step-by-step*
  - Cascades does *one single step*
- Rule application
  - Starburst applies rule by *forward chaining* in query rewrite phase
  - Cascades performs *goal-driven* application of rules