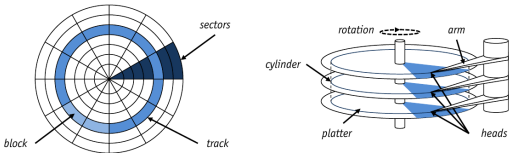
Contents

1 Disk, File, Buffer Management

1.1 Disk

1.1.1 Magnetic Disks:



Blocks: Data is read and written to disk one block at a time. Size is multiple of the sector size.

Sector: A sector is a single part of a track

Track: A track is a ring on a plater

Cylinder: A cylinder is the set of all tracks with the same diameter

Head: The head is mounted at the arm which is controlled by a stepper motor to move from track to track as the disk rotates

The access time t is defined as

$$t = t_s + t_r + t_t$$

where

- t_s seek time (movement of disk head to desired track)
- t_r rotational delay (waiting time until the desired sector has rotated under the disks head)
- t_t transfer time (time taken to read/write the block)

Sequential reads are much faster as one only needs to seek and to wait for the disk to rotate one time per sequential instead of on every access as in random I/O

1.1.2 RAID

Redundant Array of Independent Disks, provide redundancy to improve mean time to failure, improve performance using striping, hardware and software based implementations.
Raid Levels:

0 Striping
Best Write performance, better read performance, no reliability improvements

1 Mirroring
On-line backup, no performance improvements

10 Mirroring + Striping
reasonable performance w extra reliablility; write intensive workloads, small subsystems

2 ECC
inferior to 3

3 Bit-Interleaved parity
Apropriate for large continuous block requests, bad for many small requests

4 Block-Interleaved Parity
Inferior to 5

5 Block-Interleaved Distributed Parity Good general purpose solution, best performance with redundancy for small and large r and large w ops

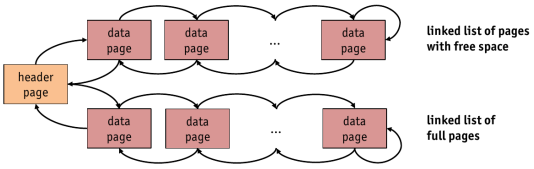
6 P+Q Parity
highest level of reliability, like 5 with 2 parities

Parity Scheme :
Initialization: let $i(n)$ be the number of bits set to 1 at position n on disk; n-th parity bit is set to 1 if $i(n)$ is odd else 0
Recovery: Let $j(n)$ be the no. of bits set to 1 at pos n on the non-failed disks. if $j(n)$ odd and parity bit 1 or $j(n)$ even and parity bit is 0, n-th value on failed disk is 0, else 1

1.2 Files

Free Slot Management: Pages File structure needs to keep track of pages with free space

Linked List of Pages



Operation $f \leftarrow$ `createFile` (n)

1. DBMS allocates a free page (called file **header page**) and stores an entry <*n*, *header page*> to a known location on disk
2. header page is initialized to point to two doubly linked list of pages: one containing **full pages** and one containing **pages with free space**
3. initially, both lists are **empty**

Operation $rid \leftarrow$ `insertRecord` (f, r)

1. try to **find a page** p in the free list with space > $|r|$
2. should this fail ask the disk space manager **to allocate a new page** p
3. record r is **written** to page p
4. since generally $|r| \ll |p|$, p will belong to the **list of pages with free space**
5. a **unique** *rid* for r is computed and returned to the caller

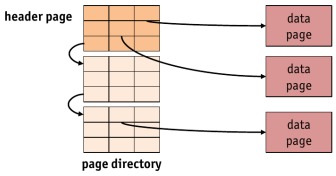
Operation `deleteRecord` (f, r)

1. may result in **moving the containing page** from the list of full pages to the list of pages with free space
2. may even lead to **page deallocation** if the page is completely free after deletion

Operation `openScan` (f)

1. **both** page lists have to be traversed

Directory of Pages



- **Header page** contains first page of a chain of **directory pages**
  - each entry in a directory page identifies a page of the file
  - |page directory| ≪ |data pages|
- Free space management is also done through the directory
  - space vs. accuracy **trade-off:** from *open/closed* flag to exact information
  - for example, entries could be of the form <*page addr p*, *nfree*>, where *nfree* indicates **actual amount of free space** (e.g., in bytes) on page *p*

- Keeping **exact** counter value (e.g., *nfree*) during updates may produce a performance bottleneck in multi-user operations
  - each transaction, whose update changes the amount of available free space, needs to update this meta-information in the directory
  - locking (or some other form of synchronization) needs to be applied to avoid lost updates
  - frequent updates of the same record lead to "hot data item", introducing lock-wait queues and thus reducing degree of parallelism
- Keep **fuzzy information** on available free space in directory, e.g., ⌊nfree/8⌋ (units of 8 bytes or some other granularity)
  - directory only needs to be updates for "large" changes in the available free space on a page
  - page itself contains the exact information (of course)

	Linked List	Directory
+	easy to implement	free space management more efficient
-	most pages in free space list	memory overhead

Free Slot Management: Slots

- Linked List of free blocks:
  1. pointer to first free block
  2. when block is freed pre/append it to free block list
  3. when block is needed take it from the list and adjust pointers
- Bitmap of free blocks
  1. reserve ceil($|blocks|/8$) bytes
  2. interpret bitwise (0 = free, 1 = used)

1.3 BufferManager

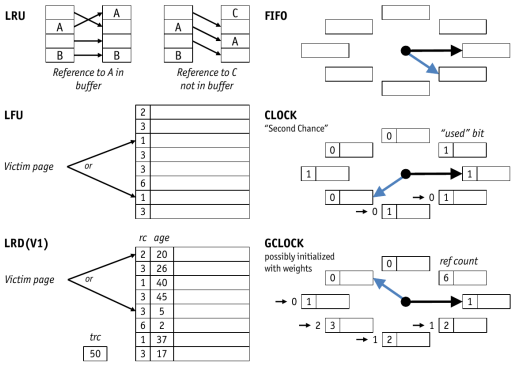
Typical Buffer Replacement Policies

DBMS typically implement more than one replacement policy

- **FIFO** ("first in, first out")
- **LRU** ("least recently used"): evicts the page whose latest `unpin()` is longest ago
- **LRU-*k***: like LRU, but evicts the *k* latest `unpin()` call, not just the latest
- **MRU** ("most recently used"): evicts the page that has been unpinned most recently
- **LFU** ("least frequently used")
- **LRD** ("least reference density")
- **CLOCK** ("second chance"): simulates LRU with less overhead (no LRU queue reorganization on every frame reference)
- **GCLOCK** ("generalized clock")
- **WS, HS** ("working set", "hot set")
- **Random**: evicts a random page

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(V1)
			LRD(V2)	

Criteria matrix for victim selection



Details of LRD

- Record the following 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
- Update these parameters during a transaction's page references (`pin(pageNo)` calls)
- Compute **mean reference density** of a page p at time t as

$$rd(p,t) := \frac{rc(p)}{trc(t) - age(p)}, \text{ where } trc(t) - rc(p) \geq 1$$

- **Strategy** for victim selection
  - chose page with least reference density $rd(p,t)$
  - many variants exist, e.g., for gradually disregarding old references

2 B+Tree and Hash Indexes

Use an auxillary structure to provide support for certain functions. Variants: 1 resembles the sorted file on attribute A, thus only one such index should exist to avoid redundant storage of records. 2 and 3 use rids where 3 groups records that match a search key k.

1. <k, $< \ldots, A = k, \cdots >$>
2. <k, rid>
3. <k, [rid_1, ...]>

2.1 Tree-based Indexes

Containing only one record per page those auxiliary structures recurse until all data fit onto one page in terms of representation. Thus they are very useful for range selections.

Fan-out: The average number of childrean for a non-leaf node.

2.1.1 Indexed Sequential Access Method

1. Sort the file on attribute A and store it
2. for each page maintain a pair $< k_i, p_i >$ conatining the most extreme key wrt. a comparator of a certain Page i and a pointer to that page
3. use them as array to access the right page without scanning the page

2.1.2 B+Tree

- Similar to ISAM but dynamic wrt. updates $\Rightarrow$ no overflow chains/remains balanced
- Search performance is also $\log_F N$
- supports updates efficiently, guaranteed occupancy of 50
- non-leafes have same layout as in ISAM
- leaf nodes contain pointers to records (instead of ISAM: Pages)
- B+Tree index moves data records on split and merge, thus rid changes
  1. $k_i^* =< k_i, < \text{attr}_1, \cdots >>$
  2. $k_i^* =< k_i, \text{rid} >$
  3. $k_i^* =< k_i, [\text{rid}_1, \ldots] >$
- occupancy rule might be relaxed
- duplicates are not supported, supported as normal values (affects search, checking the siblings) or using variant 3 grouped

Searching: As in ISAM, check the key and perform bin search until leaf is reached.

Insert: look for the right place, if not full insert. If full check siblings for redistribution (and update separator if necessary). If not possible split.

Split: Create new node, redistribute keys, take first value of the second leaf as new separator and propagate the spilt upwards.

Delete: Search leaf, delete value from it. If minimal occupancy below limit, check siblings for redistribution. If not possible, merge.

Merge: Merge with sibling node, delete key from parent level pointing to second leaf and propagate upwards

Key Compression uses prefixes or smaller data types to just approximate the actual values in the leafs. Another variant is to store common prefixes only once per node e.g. iri,o,r

Bulk loading If index is created, the tree is traversed |records| times. Most DBMS provide therefore a bulk tree loading utility.

1. for each key k in the data file, create a sorted list of pages of index leaf entries (does not imply sorting the data file itself on key k for variants 2,3; var 1 creates a clustered index)
2. allocate an empty root and let the first pointer p_0 point to the first entry of the sorted list
3. for each leaf level node/list entry insert the index entry $< p_n, \min(val(n)) >$ into the rightmost index node above the leaf level

Invariants

- Order: d
- Occupancy: $d - 1 <$ |values| $< 2d + 1$. Exception: root
- Fan-out: Non-leaf holding m keys has m+1 children
- Sorted Order nodes contain elements in comparator order, all children to the left are smaler wrt. comparator than the value in the parent and the subtrees to the right.
- Balanced: All leaf nodes are on the same level
- Height: $\log_d N$

2.2 Hash-based Indexes

"Unbeatable for Equality operations", no support for range queries

Design of a hash function

1. **By division**: simply define
$$h(k) = k \bmod N$$
   - this guarantees that range of $h(k)$ to be $[0, \ldots, N-1]$
   - **prime numbers** work best for N
   - choosing $N = 2^d$ for some d effectively considers the least d bits of k only
2. **By multiplication**: extract the fractional part of $Z \cdot k$ (for a specific Z) and multiply by hash table size N
$$h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor$$
   - the (inverse) **golden ratio** $Z = {}^{2}/_{\sqrt{5}+1} \approx 0.6180339887$ is a good choice (according to D. E. Knuth, *"Sorting and Searching"*)
   - for $Z = {}^{2}/_{2^w}$ and $N = 2^d$ (w is the number of bits in a CPU word), we simply have $h(k) = msb_d(Z \cdot k)$, where $msb_d(x)$ denotes the d **most significant bits** of x (e.g., $msb_3(42) = 5$)

Hash Function Family

Given an initial hash function h and an initial hash table size N, one approach to define such a family of hash functions $h_0, h_1, h_2, \ldots$ would be

$$h_{level}(k) = h(k) \bmod \left(2^{level} \cdot N\right)$$

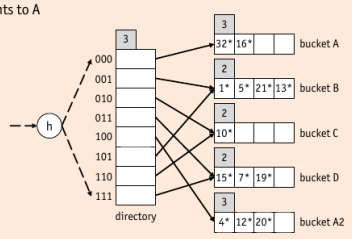
where $level = 0, 1, 2, \ldots$

2.2.1 Extendible Hashing

Uses in-memory bucket directory to keep track of the actual primary buckets by adapting the hash function and the access to the buckets

Example: Insert a record with $h(k) = 20$

2. To address the new bucket, the directory needs to be **doubled** by simply copying its original pages (bucket pointer lookups now use 2 + 1 = 3 bits)
3. Let bucket pointer for 100_2 point to A2, whereas the directory pointer for 000_2 still points to A



Searching: buckets is an array of size 2^{n-1} where each entry points to a corresponding bucket

1. n = global depth
2. b = $h(k)\&(2n-1)$ // mask last n-1 bits
3. bucket = buckets[b]

Insert: If there is free space in the corresponding bucket just insert, else

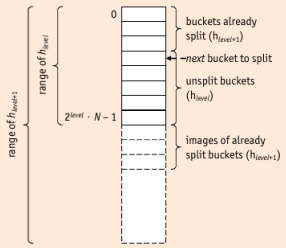
1. Split the bucket by creating a new bucket and use bit position $d+1$ to redistribute the entries where d is the local depth
2. if $d + 1 > n$, $n++$ and double the directory size. The hashing uses now d+1 bits
3. let the old bucket be pointed to by 0[...] and the new be pointed to by 1[...]
4. if no value goes to the new bucket an overflow chain is initialized

Delete: Analog to insertion

2.2.2 Linear Hashing

Rehashing

With every bucket split, next walks down the hash table. Therefore, hashing via h_{level} (search, insert, and delete) needs to take **current** *next* **position** into account.



$h_{level}(k)$ { < *next*: bucket already split, **rehash**: find record in bucket $h_{level+1}(k)$
≥ *next*: bucket not yet split, i.e., **bucket found**

1. Init: level = 0, next = 0
2. current hash fn = h_{level}, active hash buckets: $[0, \ldots, 2^{\text{level}} \cdot N]$
3. whenever a certain criterion is met, split the bucket which the next pointer references (e.g. % occupancy reached in a bucket, overflow chain grew longer than x, ...

Split : All buckets with position ¡ next have been already rehashed

1. Allocate new bucket and append it at position $2^{\text{level}} \cdot N + \text{next}$
2. Redistribute entries in the bucket that next references by rehashing with $h_{\text{level}+1}$
3. next++, if next $> 2^{\text{level}} \cdot N - 1$, next = 0; level++

Insert: like in static hashing plus additional check for split criterion

Delete: like with static hashing plus if bucket$[2^{\text{level}} \cdot N + next]$.empty:

1. remove page pointed to by bucket$[2^{\text{level}} \cdot N + next]$ from hash table
2. next–, if next ¡ 0, level–, next = $2^{\text{level}} \cdot N + next$

3 2 Query Evaluation

3.1 Two-Way Merge Sort

Sorts files of arbitary size (here $N = 2^k$) with three pages of buffer space in multiple passes, producing a certain number of sub-files called **runs**

- **Pass 0**: Sorts each of the 2^k input page individually in main memory, resulting in the same 2^k runs
- **subsequent passes:** Merge pairs of runs into larger runs. Pass n produces $2^k - n$ runs
- pass k produces one overall sorted final run

Pass 0 (**input:** $N = 2^k$ unsorted pages, **output:** 2^k sorted pages)
1. **Read** N pages, one page at a time
2. **Sort** records, page-wise, in main memory
3. **Write** sorted pages to disk (each page results in a **run**)

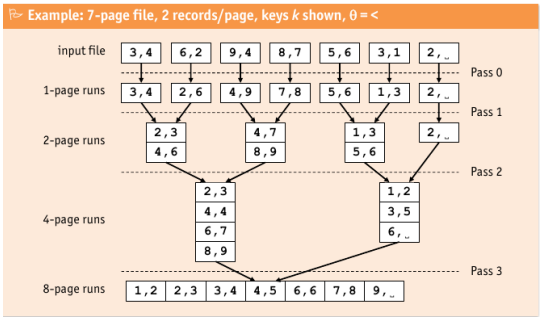
This pass requires **one page** of buffer space

Pass 1 (**input:** $N = 2^k$ sorted pages, **output:** 2^{k-1} sorted runs)
1. **Open** two runs r_1 and r_2 from Pass 0 for reading
2. **Merge** records from r_1 and r_2, reading input page by page
3. **Write** new two-page run to disk, page by page

This pass requires **three pages** of buffer space

Pass n (**input:** $N = 2^{k-n+1}$ sorted runs, **output:** 2^{k-n} sorted runs)
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

This pass requires **three pages** of buffer space



Costs: Each pass needs to read N pages, sort them in-memory and write them out again $\Rightarrow 2N$ I/O ops per pass
in total there are pass 0 and k subsequent pass $\Rightarrow 1 + log_2 N$ In Total Two-way Merge Sort costs

$$2N(1 + log_2 N)\text{I/O ops}$$

3.2 External Merge Sort

like 2-way merge sort, but reduces the number of runs by using more buffer space to avoid creating one page runs in pass 0 and reduces the number of passes by merging more than two runs at a time.

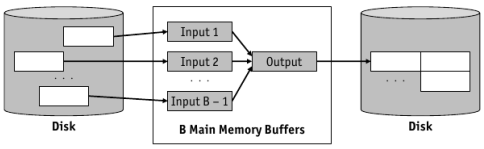
External Merge Sort

Pass 0 (**input:** N = unsorted pages, **output:** $\lceil N/B \rceil$ sorted pages)
1. **Read** N pages, B pages at a time
2. **Sort** records, page-wise, in main memory
3. **Write** sorted pages to disk (resulting in $\lceil N/B \rceil$ **runs**)

This pass uses B **pages** of buffer space

Pass n (**input:** $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs, **output:** $\frac{\lceil N/B \rceil}{(B-1)^n}$ sorted runs)
1. **Open** $B-1$ runs $r_1, \ldots, r_{B-1}$ from Pass $n-1$ for reading
2. **Merge** records from $r_1, \ldots, r_{B-1}$, reading input page by page
3. **Write** new $B \cdot (B-1)^n$-page run to disk, page by page

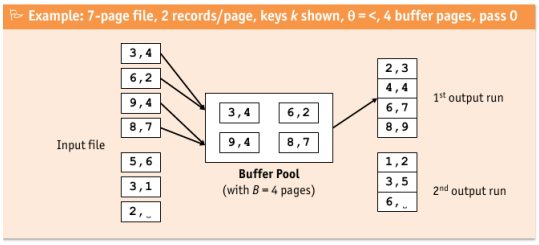
This pass requires B **pages** of buffer space



- Suppose B pages are available in the 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** (leaving one page as a write buffer)

N	$B=3$	$B=5$	$B=9$	$B=17$	$B=129$	$B=257$
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Number of Passes of External Merge Sort



Costs: As in two way: read, sort, write $\Rightarrow 2N$
In pass 0 only $\lceil \frac{n}{B} \rceil$ are written thus only needs $\lceil \log_{B-1} \lceil \frac{n}{B} \rceil \rceil$ where B-1 pages are merged at the same time In Total External Merge Sort costs

$$2N(1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil)\text{I/O ops}$$

3.2.1 Optimizations

Block-grouped I/O: read blocks of b pages at once during merge. This decreases the I/O costs by factor b but decreases the fan-in thus increases the number of passes. In Total:

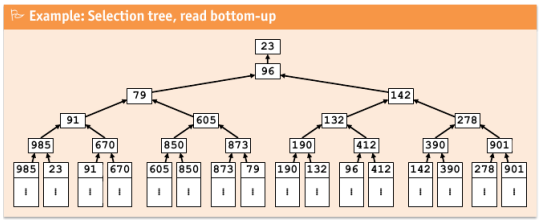
$$2N(1 + \lceil \log_{\lfloor \frac{B}{b} \rfloor - 1} \lceil \frac{N}{B} \rceil \rceil)\text{I/O ops}$$

N	$B=1000$	$B=5000$	$B=10,000$	$B=50,000$
100	1	1	1	1
1000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Number of Passes of External Merge Sort with Block Size $b = 32$

Tree of Losers

i) The tree is initially filled with one tuple from each input by propagating them up the tree. The winner at each level is moved to the parent and we record if the left or right child won. At the end of this process the first element of the array contains the smallest tuple.
ii) Now we can remove the winner and write it to the output run.
iii) After that we have to refill the tree from below. Since all input runs are sorted, the tree currently contains the smallest tuple of all inputs except for the one the removed tuple came from. We can find that input by tracing the markers of who won/lost down the tree. Then we refill the tree with the next tuple from that input.
If an input is drained, `null` can be used as the next tuple instead to mark an empty slot. We just have to make sure that `null` is compared as greater than any other tuple.
When propagating the new tuple up the tree, we again adapt the winner/loser markers.
iv) Repeat steps ii) and iii) until the *winner* becomes `null`, at which point all input runs are drained and the merge is completed.

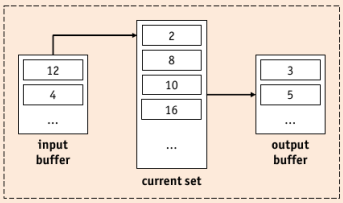


Replacement Sort

Replacement sort
1. Open an empty run file for writing.
2. Load next page of file to be sorted into input buffer. If input file is exhausted, go to 4.
3. While there is space in the current set, move a record from input buffer to current set (if the input buffer is empty, reload it at 2.)
4. In current set, pick record r with smallest key value k such that $k \geq k_{out}$, where k_{out} is the maximum key value in output buffer (if output buffer is empty, define $k_{out} = -\infty$). Move r to output buffer. If output buffer is full, append is 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 writing.
6. If input file is exhausted, stop. Otherwise go to 3.

Replacement sort

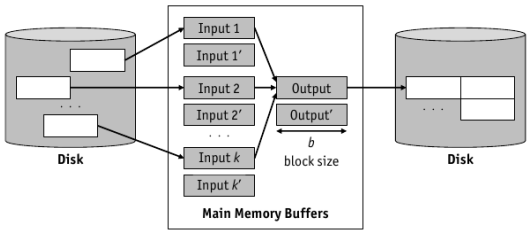
Assume a buffer pool with B pages. Two pages are dedicated **input** and **output buffers**. The remaining $B-2$ pages are called the **current set**.



Double Buffering

Double buffering

To avoid CPU waits, **double buffering** can be used.
1. create a second **shadow buffers** for each input and output buffer
2. CPU **switches** to the "double" buffer, once original buffers is empty/full
3. original buffer is reloaded by **asynchronously** initiating an I/O operation
4. CPU **switches** back to original buffer, once "double" buffer is empty/full, etc.



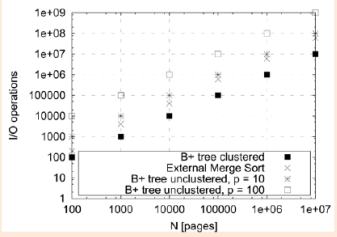
B+ Tree for Sorting Clustered B+Tree Index: just load the N pages as they are already sorted.

Expected page I/O operations for sorting

Let p denote the number of data records per page (typical values are p = 10, ..., 1000). The expected number of page I/O operations to sort using an unclustered B+ tree index is therefore $\boldsymbol{p \cdot N}$ (worst case)

Assumptions in plot

- available buffer space for sorting is $B = 257$ pages
- ignore I/O to traverse B+ tree as well as its sequence set



3.3 Join

Most basic variant is to calculate the cross product between the relations and apply selection according to the join predicate.

3.3.1 Nested Loops Join

- straight foward implementation of cross product and join equivalence
- needs only 3 buffer pages

Nested loops join

```
function ⋈^nl (R1, R2, Rout, p)
  in1 ← openScan (R1) ;
  out ← createFile (Rout) ;
  while (r1 ← nextRecord (in1) ) ≠ ⟨EOF⟩ do
    in2 ← openScan (R2) ;
    while (r2 ← nextRecord (in2) ) ≠ ⟨EOF⟩ do
      if p(r1, r2) then appendRecord (out, ⟨r1, r2⟩) ;
  closeFile (out) ;
end
```

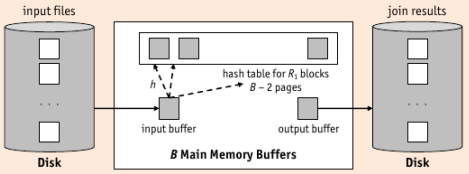
3.3.2 Block Nested Loops Join

Reads both relations in blocks of b_1, b_2 pages respectively

General block nested loops join

```
function ⋈^block-nl (R1, R2, Rout, p)
  in1 ← openScan (R1) ;
  out ← createFile (Rout) ;
  foreach b1-sized block in in1 do
    in2 ← openScan (R2) ;
    foreach b2-sized block in in2 do
      for all matching in-memory tuples r1 ∈ R1 block and r2 ∈ R2 blocks,
      add ⟨r1, r2⟩ to the result out
  closeFile (out) ;
end
```

Block nested loops join with hash table



Block nested loops join with hash table

```
function ⋈^block-nl (R1, R2, Rout, p)
  in1 ← openScan (R1) ;
  out ← createFile (Rout) ;
  repeat
    B' ← min (B – 2, #remaining blocks in R1) ;   (do not read beyond ⟨EOF⟩ of R1)
    if B' > 0 then
      read B' blocks of R1 into buffer, hash record r ∈ R1 to buffer page h(r.A1) mod B' ;
      in2 ← openScan (R2) ;
      while (r2 ← nextRecord (in2) ) ≠ ⟨EOF⟩ do
        compare record r2 with records r1 stored in buffer page h(r2.A2) mod B' ;
        if r1.A1 = r2.A2 then appendRecord (out, ⟨r1, r2⟩) ;
  until B' < B – 2 ;
  closeFile (out) ;
end
```

3.3.3 Index Nested Loops Join

- Uses index on inner Relation to avoid enumerating the corss product
- particularly usefull when index is clustered and join is very selective

Index nested loops join

```
function ⋈^index-nl (R1, R2, Rout, p)
  in1 ← openScan (R1) ;
  out ← createFile (Rout) ;
  while (r1 ← nextRecord (in1) ) ≠ ⟨EOF⟩ do
    probe index on R2 using (key value in) r1 to find matching tuples r2 ∈ R2 ;
    appendRecord (out, ⟨r1, r2⟩) ;
  closeFile (out) ;
end
```

Costs of an Index access :

- Hash Index: $1.2 \begin{cases} 1.2 & clustered \\ n & unclustered \end{cases}$
- B+Tree: $log(|R_2|) \begin{cases} 1 & clustered \\ n & unclustered \end{cases}$

3.3.4 Sort Merge Join

- Uses sorting to partition both input
- best-case is optimal
- integration into external sort, block-grouped I/O, double buffering, replacement sort can be applied to optimize further
- output sorted on join attribute

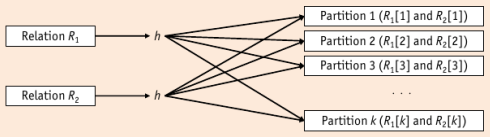
Sort-merge join

```
function ⋈^sort-merge (R1, R2, Rout, R1.A = R2.B)
  if R1 not sorted on A then sort it ;      if R2 not sorted on B then sort it ;
  out ← createFile (Rout) ;
  in1 ← openScan (R1) ;                     in2 ← openScan (R2) ;
  r1 ← nextRecord (in1) ;                   r2 ← nextRecord (in2) ;
  while r1 ≠ ⟨EOF⟩ ∧ r2 ≠ ⟨EOF⟩ do
    while r1.A < r2.B do r1 ← nextRecord (in1) ;
    while r1.A > r2.B do r2 ← nextRecord (in2) ;
    r'2 ← r2 ;                              (remember current position in R2)
    while r1.A = r'2.B do                   (all R1 tuples with the same A value)
      r2 ← r'2 ;                            (rewind r2 to r'2)
      while r1.A = r2.B do                  (all R2 tuples with the same B value)
        appendRecord (out, ⟨r1, r2⟩) ;
        r2 ← nextRecord (in2) ;
      r1 ← nextRecord (in1) ;
  closeFile (out) ;
end
```

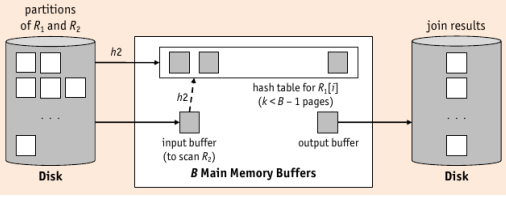
3.3.5 Grace Hash Join

- works only for equality predicates
- Two Phases: Partitioning and matching/probing phase
- follows divide and conquer: partition and do per partition in-memroy joins
- may be accelerated by using second hash function for probing as with the block NLJ
- Needs $B > \sqrt{f \cdot ||R||}$ Buffer pages where f is the factor of increase to maintain a hash table over the partition instead of the partition only, so that partition stays in memory

Partitioning phase of hash join



Probing phase of a hash join



Grace hash join

```
function ⋈^hash-join (R1, R2, Rout, R1.A = R2.B)
  in1 ← openScan (R1) ;                                   (partitioning phase)
  while (r1 ← nextRecord (in1) ) ≠ ⟨EOF⟩ do
    add r1 to partition R1[h(r1.A)] ;                     (flushed as page fills)
  in2 ← openScan (R2) ;
  while (r2 ← nextRecord (in2) ) ≠ ⟨EOF⟩ do
    add r2 to partition R2[h(r2.B)] ;                     (flushed as page fills)
  out ← createFile (Rout) ;
  foreach i ∈ 1, ..., k do                                (probing phase)
    foreach tuple r1 ∈ R1[i] do                           (build in-memory hash table for R1[i], using h2)
      insert r1 into hash table H, using h2(r1.A) ;
    foreach tuple r2 ∈ R2[i] do                           (scan R2[i] and probe for matching R1[i] tuples)
      probe H using h2(r2.B) and append matching tuples ⟨r1, r2⟩ to out ;
    clear H to prepare for next partition ;
  closeFile (out) ;
end
```

3.4 Selection

Definition

The **selectivity** (or **reduction factor**) or a predicate p, denoted by $sel(p)$, is the faction of records in a relation R that satisfy the predicate p.

$$0 \leq sel(p) = \frac{|\sigma_p(R)|}{|R|} \leq 1$$

- Implemented using a combination of **iteration or indexing**

- may be applied on the fly in a pipelined plan
- Complex predicates may be expressed as conjuncts and disjuncts in terms of boolean logic
- three evaluation option for CNF terms:
  1. Single file scan
  2. single index that match a subset of the primary conjuncts, apply others on the fly
  3. multiple indexes each matching a subset of conjuncts, applying intersection over rid on the fly
- similar for DNF but union instead of intersection and only possible when **all** predicates are matched by index. Solution: **Bypass Selection**

No index, unsorted Data:

Selection

```
function σ(p, R_in, R_out)
  in ← openScan(R_in) ;
  out ← createFile(R_out) ;
  while (r ← nextRecord(in)) ≠ ⟨EOF⟩ do
    if p(r) then appendRecord(out, r) ;
  closeFile(out) ;
end
```

B+Tree index with predicate matching sort key

Implementing $\sigma_p(R_{in})$ using a B+ tree

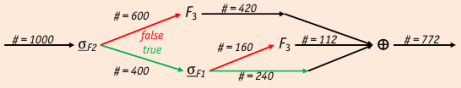
- Descend the B+ tree to retrieve the first index entry that satisfies p
- If the index is **clustered**, access that record on its page in R_{in} and continue to scan inside R_{in}
- If the index is **unclustered** and $sel(p)$ indicates a **large number of qualifying records**, it pays off to
  1. read all matching index entries $k^* = \langle k, rid \rangle$ in the sequence set
  2. sort those entries on the *rid* field
  3. access the pages of R_{in} in sorted *rid* order

Note that the lack of clustering is a minor issue if $sel(p)$ is close to 0

Bypass selection avoids expensive and unselective selections by applying the most selective and the cheap selections first. I.E. the goal is to eliminate tuples early and avoid duplicates.

1. Convert selection condition to CNF
2. apply most selective/cheapest predicate first and safe disjunct tuples (true/false on predicate p_1)
3. repeat step 2 until all conjuncts are applied (when a conjunct contains disjuncts, just chain them)

Example



Mean cost per tuple ($\oplus$ disjoint union): $C_2 + (1 - s_2) \cdot C_3 + s_2 \cdot (C_1 + (1 - s_1) \cdot C_3) = 40.6$

Note that many variations are possible, e.g., for tuning in parallel environments

3.5 Projection

- removes unwanted attributes and eliminates duplicates
- implemented using iteration or partitioning
- without duplicate elimination can be pipelined, with needs to be materialized

1. Do a FileScan, $\forall r \in$ File: cut off uneeded attributes and append to the output
2. do duplicate elimination as follows:

Projection based on sorting

```
function π^sort(l, R_in, R_out)
  in ← openScan(R_in) ;
  out ← createFile(R_tmp) ;
  while (r ← nextRecord(in)) ≠ ⟨EOF⟩ do
    r' ← r with any field not listed in l cut off ;
    appendRecord(out, r') ;
  closeFile(out) ;
  external-merge-sort(R_tmp, R_tmp, θ) ;
  in ← openScan("run␣*␣0") ;
  out ← createFile(R_out) ;
  lastr ← ⟨⟩ ;
  while (r ← nextRecord(in)) ≠ ⟨EOF⟩ do
    if r ≠ lastr then
      appendRecord(out, r) ;
      lastr ← r ;
  closeFile(out) ;
end
```

Marriage of sorting and projection with duplicate elimination

Step ① (**projection**) and Step ② (**duplicate elimination**) can be **integrated** into the passes performed by the external merge sort algorithm.

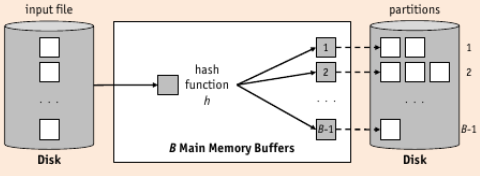
Pass 0
1. read B pages at a time, **projecting unwanted attributes out**
2. use in-memory sort to sort the records of these B pages
3. write a sorted run of B internally sorted pages out to disk

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

Partitioning phase

1. Allocate all B buffer pages: one page will be the **input buffer**, the remaining B – 1 pages will be used as **hash buckets**.
2. Read the file R_{in} page by page: for each record r, **project out** the attributes not listed in list l.
3. For each such record, apply hash function $h_1(r) = h(r) \bmod (B - 1)$, which depends on **all remaining fields of** r, and store r in hash bucket $h_1(r)$.
4. If the hash bucket is **full**, write it to disk, i.e., the overflow chain of a bucket resides on disk

Partitioning phase



Duplicate elimination phase

1. For each partition, read each partition page by page (possibly in parallel), using the same buffer layout as before.
2. To each record, apply a hash function h_2 ($h_2 \neq h_1$) to all record fields.
3. Only if two records collide with respect to h_2, check if $r = r'$ and, if so, discard r'.
4. After the entire partition has been read in, append all hash buckets to the result file, which will be free of duplicates.

4 3-query opt

1. Query Parser: Parse Q and derive a relational algebra expression E
2. **Rewrite optimization (logical level):** From E generate set of logical plans L transforming and simplifying E
3. **Cost-based Optimization (physical level):** Generate a set of physical plans P by annotating the plans in L with access paths and operator algorithms
4. **Plan cost estimator:** Estimate the costs of each plan and chose the best one
5. Query Plan Evaluator: Execute the plan and return the result to the UI

Search space ≡ logical level ∪ physical level

4.1 Relational Algebra Rewriting

1. Break apart conjunctive selections (Rule 1)
2. Move selections down the query tree (Rules 2, 9, 15)
3. Replace selection-cross product pairs with joins (Rule 8)
4. Break list of projections apart & move them down, create new projections where possible (Rules 3, 10, 14)
5. Perform joins with the smallest expected result first

1. **Cascading selections**
$$\sigma_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R) \ldots)$$
2. **commutativity of selections**
$$\sigma_{c_q}(\sigma_{c_p}(R)) \equiv \sigma_{c_p}(\sigma_{c_q}(R))$$
3. **Cascading Projections**
$$\pi_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \pi_{c_1}(\ldots \pi_{c_n}(R) \ldots)$$
4. **Folding selections:** with $a_i \subseteq a_{i+1}$ only the last projection is needed (cutting off step by step vs. doing all cutoffs once)
$$\sigma_{a_1}(R) \equiv \sigma_{a_1}(\ldots \sigma_{a_n}(R) \ldots)$$
5. **Cross-Product and all Joins are associative** with r involves only T and S, p only involves R and S
$$(R \bowtie_p S) \bowtie_{q \wedge r} T \equiv R \bowtie_{p \wedge q} (S \bowtie_r T)$$

6. **Cross-Product and Natural Join are commutative**

$$R \times S \equiv S \times R$$

$$R \bowtie S \equiv S \bowtie R$$

7. **Selection and cross product form a join**

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

8. **Selections and joins can be combined**

$$\sigma_q(R \bowtie_p S) \equiv R \bowtie_{p \wedge q} S$$

9. **Selections commutes with Joins and cross-product** let q only involves R

$$\sigma_q(R \bowtie_p S) \equiv \sigma_q(R) \bowtie_p S$$

10. **Selections and projections distribute over joins and cross products** let p only involve R, q only involve S

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

$$\pi_a(R \bowtie_r S) \equiv \pi_{a_1}(R) \bowtie_r \pi_{a_2}(S)$$

11. **Selections and Projections commute** if the projections keeps all attributes involved in the selection predicates

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

12. **commutativity of Union and Intersection**

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

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

13. **Union and Intersection are associative**

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

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

14. **Projection distributes over Union**

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

15. **Selection, Union, Intersection and Difference are distributive**

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

16. **Selection, Intersection and Difference are commutative**

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

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

4.2 Plan Enumeration

- Plans without Index
  1. heap file scan
  2. selection & join on-the-fly
  3. sort according to group by
  4. apply aggregation and having clauses on-the-fly
- Single-Index Access Path
  1. Choose index that retrieves fewest pages
  2. apply projections and non-primary selections
  3. compute group by and aggregation by sorting
- Multiple-Index Access Path
  1. Retrieve and intersect rid sets and sort result 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
- Sorted Index Access Path
  1. Retreive tuples in order required by group by
  2. apply selection and projection to each retreived tuple on the fly
  3. compute aggregate
- Index-Only Access Path
  1. Do index scan
  2. apply selection and projection on the fly
  3. sort to group and do aggregation

4.3 Dynamic Programming

Principle of Optimality: The Assumption is that in order to find an optimal global plan, it is sufficient to consider optimal plans for all possible sub-plans

Pass 1 (all 1-relation plans)
- find best 1-relation plan for each relation
- this pass mainly consists of selecting access method
- also see discussion on single-relation queries

Keep the best 1-relation plans for each set of physical properties

Pass 2 (all 2-relation plans)
- find best way to join sub-plans from Pass 1 to another relation
- generate left-deep trees with sub-plans from Pass 1 as outer relation in these joins

Again, keep the best 2-relation plans for each set of physical properties

⋮

Pass *n* (all n-relation plans)
- find best way to join sub-plans of 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

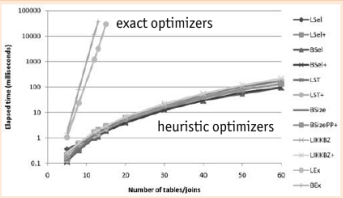
Algorithm to find an optimal *n*-way left-deep join tree

```
function optimize (q(R1, ..., Rn))
  for i = 1 to n do                                          (Pass 1)
    optPlan({Ri}) ← accessPlans (Ri) ;
    prunePlans (optPlan({Ri})) ;
  for i = 2 to n do                                          (Pass 2 to n)
    foreach S ⊆ {R1, ..., Rn} such that |S| = i do
      optPlan(S) ← ∅ ;
      foreach R ⊆ {R1, ..., Rn} do
        optPlan(S) ← optPlan(S) ∪ possibleJoins( ⋈ : optPlan(S\R), optPlan(R) ) ;
      prunePlans (optPlan(S)) ;
  return optPlan({R1, ..., Rn}) ;
end
```

Greedy join enumeration algorithm for an *n*-way join

```
function optimize-greedy (q(R1, ..., Rn))
  worklist ← ∅ ;
  for i = 1 to n do
    worklist ← worklist ∪ bestAccessPlans (Ri) ;
  for i = n downto 2 do                                      (worklist = {P1, ..., Pi})
    find Pj, Pk ∈ worklist ∧ ⋈* such that cost (Pj ⋈* Pk) is minimal ;
    worklist ← worklist \ {Pj, Pk} ∪ {(Pj ⋈* Pk)} ;
  return single plan left in worklist ;                      (worklist = {P1})
end
```

Performance of different optimization alternatives



- Naming scheme
  - Search space: **L** (linear) or **B** (bushy)
  - Ranking: **Sel** (*minSel*), **Size** (*minSize*), or **ST** (*smallestTable*)
  - Merging: **+** (*Switch-HJ* and *Switch- Idx*) or **PP** (*Pull/Push*)

4.4 Histograms

Construct an **equi-width histogram** on attribute **A** of relation **R**

1. divide **active domain** of attribute **A** into B buckets of equal **bucket width** w, such that $w = \frac{High(\mathbf{A}, \mathbf{R}) - Low(\mathbf{A}, \mathbf{R}) + 1}{B}$
2. bucket boundary $b_0 = 0$ and $b_i = b_{i-1} + w$
3. while scanning **R** once sequentially, maintain B **running sums of value frequencies**, one for each bucket

In order to construct an **equi-depth histogram** on attribute **A** in relation **R**

1. divide **active domain** of attribute **A** into B buckets of equal **bucket depth** d, such that $d = NTuples(\mathbf{R})/B$
2. sort **R** by sort criterion on attribute **A**, e.g., natural order
3. bucket boundary $b_0 = Low(\mathbf{A}, \mathbf{R})$, then determine b_i by dividing the sorted relation **R** into blocks of size $\sim d$

– **adaptive sampling** tries to achieve a given precision with minimal sample size
– **double (two-phase) sampling** obtains a course picture from a very small sample first and then computes the necessary sample size for a "useful" sample in a second step
– **sequential sampling**: uses a sliding (continuous) calculation of characteristics and stops once the estimated precision is high enough

4.5 Nested Subqueries

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

Due to the removal of the **CONTAINS** keyword in SQL-2 and higher, Type D nesting is no longer possible

- Algorithm **NEST-N-J** for Type N or Type J nested subqueries
  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 the outermost query block
- Resulting canonical query is **equivalent** to original nested query

Algorithm NEST-N-J

```
SELECT Ri.Ck                          SELECT DISTINCT Ri.Ck
  FROM Ri                               FROM Ri, Rj
 WHERE Ri.Cm IN                       WHERE Ri.Cm = Rj.Cn AND
       (SELECT Rj.Cn                        Ri.Cx = Rj.Cy
          FROM Rj
         WHERE Ri.Cx = Rj.Cy)
```

- Assume R_1 is the relation in the top-level query and R_2 is the relation in the nested subquery
- Algorithm **NEST-JA** for Type JA nested subqueries
  1. create a temporary relation $R_{tmp}(C_1, \ldots, C_n, C_{n+1})$ from relation R_2, such that $R_{tmp}.C_{n+1}$ is the result of applying aggregate function **AGG** on the C_n columns of R_2 that have matching values for $C_1, \ldots, C_n$ in R_1
  2. transform inner query block of the original query by changing all references to R_2 columns in join predicates that also reference R_1 to corresponding R_{tmp} columns
- Result of algorithm NEST-JA is a Type J nested query that can be transformed to its canonical equivalent by algorithm NEST-N-J

Algorithm NEST-JA (Step 1)

```
SELECT R1.Cn+2
  FROM R1
 WHERE R1.Cn+1 = (SELECT AGG(R2.Cn+1)
                    FROM R2
                   WHERE R1.C1 = R2.C1
                     AND R1.C2 = R2.C2
                         ...
                     AND R1.Cn = R2.Cn)

CREATE TABLE Rtmp(C1, ..., Cn, Cn+1) AS
  SELECT C1, ..., Cn, AGG(Cn+1)
    FROM R2
   GROUP BY C1, ..., Cn
```

Algorithm NEST-JA (Step 2)

```
SELECT R1.Cn+2
  FROM R1
 WHERE R1.Cn+1 = (SELECT Rtmp.Cn+1
                    FROM Rtmp
                   WHERE R1.C1 = Rtmp.C1
                     AND R1.C2 = Rtmp.C2
                         ...
                     AND R1.Cn = Rtmp.Cn)
```

- Assume R_1 is the relation in the top-level query and R_2 is the relation in the nested subquery
- Modified algorithm **NEST-JA2** for Type JA nested subqueries
  1. project join column of R_1 and restrict it with any simple predicates applying to R_1
  2. create temporary relation R_{tmp} by joining 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 column(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 (=)
- Result of algorithm NEST-JA2 is again a Type J nested query

EXISTS and NOT EXISTS

```
... WHERE EXISTS                      ... WHERE 0 <
      (SELECT A                             (SELECT COUNT(A)
         FROM R                               FROM R
        WHERE B = C)                         WHERE B = C)

... WHERE NOT EXISTS                  ... WHERE 0 =
      (SELECT A                             (SELECT COUNT(A)
         FROM R                               FEOM R
        WHERE B = C)                         WHERE B = C)
```

ANY and ALL

```
... WHERE const < ANY                 ... WHERE const <
      (SELECT A                             (SELECT MAX(A)
         FROM R                               FROM R
        WHERE B = C)                         WHERE B = C)
```

And ...**> ANY (SELECT A**... is transformed to ...**> SELECT MIN(A)** ..., for reasons of symmetry.

```
... WHERE const < ALL                 ... WHERE const <
      (SELECT A                             (SELECT MIN(A)
         FROM R                               FEOM R
        WHERE B = C)                         WHERE B = C)
```

And ...**> ALL (SELECT A**... is transformed to ...**> SELECT MAX(A)** ..., for reasons of symmetry.

Finally, ...**= ANY**... is transformed to ...**IN**..., whereas ...**<> ANY**... is transformed to ...**NOT IN**....

Recursive algorithm to process a general nested query

```
function nest-g(outer)                          (outer: outer query block)
  foreach p ∈ outer.WHERE do                    (p: predicate)
    if p.inner ≠ ∅ then                         (p contains an inner query block, i.e., p is nested)
      nest-g(p.inner);
      if p.inner.SELECT contains aggregation function then
        if p.inner.WHERE is correlated then     (Type JA nesting)
          nest-ja2(p.inner);
          nest-n-j(outer, p.inner);
        else                                    (Type A nesting)
          nest-a(p.inner);
      else                                      (Type N or Type J nesting)
        nest-n-j(outer, p.inner);
end
```

5 Cost Models

Assumptions

1. **Uniformity and independence assumption**
   All values of an attribute uniformly appear with the same probability (or even distribution). Values of different attributes are independent of each other.
   ↳ *Simple, yet rarely realistic assumption*
2. **Worst case assumption**
   No knowledge about relation contents available at all. In case of a selection σ_p, assume that all records will satisfy predicate p.
   ↳ *Unrealistic assumption, can only be used for computing upper bounds*
3. **Perfect knowledge assumption**
   Details about the exact distribution of values are known. Requires huge catalog or prior knowledge of incoming queries.
   ↳ *Unrealistic assumption, can only be used for computing lower bounds*

5.1 System Catalog

size of buffer pool, page size, information and statistics about tables, views, indexes

Information stored in the system catalog

- **Table metadata**
  - *table name*, *file name* (or some identifier), *file structure* (e.g., heap file)
  - *attribute name* and *type* of each attribute of the table
  - *index name* of each index on the table
  - *integrity constrains* (e.g., primary and foreign key constraints) on the table
- **Index metadata**
  - *index name* and *structure* (e.g., B+ tree)
  - *search key* attributes
- **View metadata**
  - *view name* and *definition*

Statistics

- **Table statistics**
  - *cardinality*: number of tuples $NTuples(R)$ for each table R
  - *size*: number of pages $NPages(R)$ for each table R
- **Index statistics**
  - *cardinality*: number of distinct key values $NKeys(I)$ for each index I
  - *size*: number of pages $INPages(I)$ for each index (for a tree index I, $INPages(I)$ denotes the number of leaf pages)
  - *height*: number of non-leaf levels for each tree index I
  - *range*: minimum present key value $ILow(I)$ and the maximum present key value $IHigh(I)$ for each index I

Example

Tables

name	file	#tuples	size
Tables	...	6	1
Attributes	...	23	1
Views	...	1	1
Indexes	...	0	1
Sailors	...	40,000	500
Reserves	...	100,000	1000

Views

name	text
Captains	SELECT * FROM Sailors WHERE...

Indexes

name	file	type	#keys	size
Boats	...	B+Tree	100	1

Attributes

name	table	type	pos
name	Tables	string	1
file	Tables	string	2
#tuples	Tables	integer	3
size	Tables	integer	4
name	Attributes	string	1
table	Attributes	string	2
type	Attributes	string	3
pos	Attributes	integer	4
⋮	⋮	⋮	⋮
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	date	3
rname	Reserves	string	4

Typical database profile for relation R

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

Selection query $Q := \sigma_{A=c}(R)$

Selectivity $sel(\mathbf{A} = c)$: $\begin{cases} MCF(\mathbf{A}, \mathbf{R})[c] & \text{if } c \in MCF(\mathbf{A}, \mathbf{R}) \\ 1/V(\mathbf{A}, \mathbf{R}) & \textbf{(uniformity assumption)} \end{cases}$

Cardinality $|Q|$: $sel(\mathbf{A} = c) \cdot NTuples(\mathbf{R})$

Record size $s(Q)$: $s(\mathbf{R})$

Number of attribute values $V(\mathbf{A}', Q)$: $\begin{cases} 1, & \text{for } \mathbf{A}' = \mathbf{A} \\ c(|\mathbf{R}|, V(\mathbf{A}, \mathbf{R}), |Q|) & \text{otherwise} \end{cases}$

$$c(n, m, r) = \begin{cases} r, & \text{for } r < \frac{m}{2} \\ \frac{r+m}{3}, & \text{for } \frac{m}{2} \le r < 2m \\ m, & \text{for } r \ge 2m \end{cases}$$

Selection query $Q := \sigma_{A=B}(R)$

Equality between attributes, e.g., $\sigma_{\mathbf{A}=\mathbf{B}}(\mathbf{R})$ can be approximated by

$$sel(\mathbf{A} = \mathbf{B}) = 1/\max(V(\mathbf{A}, \mathbf{R}), V(\mathbf{B}, \mathbf{R}))$$

This formula assumes that each value of the attribute with fewer distinct values has a corresponding match in the other attribute (**independence assumption**).

Selection query $Q := \sigma_{A>c}(R)$

If $Low(\mathbf{A}, \mathbf{R}) \le c \le High(\mathbf{A}, \mathbf{R})$, range selections, e.g., $\sigma_{\mathbf{A}>c}(\mathbf{R})$ can be approximated by

$$sel(\mathbf{A} > c) = \frac{High(\mathbf{A}, \mathbf{R}) - c}{High(\mathbf{A}, \mathbf{R}) - Low(\mathbf{A}, \mathbf{R})}$$

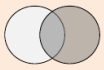
This formula uses the **uniformity assumption**.

Selection query $Q := \sigma_{A \text{ IN } (...)}(R)$

Element tests, e.g., $\sigma_{\mathbf{A} \text{ IN } (...)}(\mathbf{R})$ can be approximated by multiplying the selectivity for an equality selection $sel(\mathbf{A} = c)$ with the number of elements in the list of values.

Selections with composite predicates

- **conjunctive** predicates, e.g., $Q := \sigma_{\mathbf{A}=c1 \wedge \mathbf{B}=c2}(\mathbf{R})$



$$sel(\mathbf{A} = c_1 \wedge \mathbf{B} = c_2) = sel(\mathbf{A} = c_1) \cdot sel(\mathbf{B} = c_2)$$

which gives $|Q| = \frac{|\mathbf{R}|}{V(\mathbf{A}, \mathbf{R}) \cdot V(\mathbf{B}, \mathbf{R})}$

- **disjunctive** predicates, e.g., $Q := \sigma_{\mathbf{A}=c1 \vee \mathbf{B}=c2}(\mathbf{R})$

$$sel(\mathbf{A} = c_1 \vee \mathbf{B} = c_2) = sel(\mathbf{A} = c_1) + sel(\mathbf{B} = c_2) - sel(\mathbf{A} = c_1) \cdot sel(\mathbf{B} = c_2)$$

which gives $|Q| = \frac{|\mathbf{R}|}{V(\mathbf{A}, \mathbf{R}) + V(\mathbf{B}, \mathbf{R}) - V(\mathbf{A}, \mathbf{R}) \cdot V(\mathbf{B}, \mathbf{R})}$

Projection query $Q := \pi_L(R)$

Cardinality $|Q|$: $\begin{cases} V(\mathbf{A}, \mathbf{R}), & \text{for } L = \{\mathbf{A}\} \\ |\mathbf{R}| & \text{if keys of } \mathbf{R} \in L \\ |\mathbf{R}| & \text{no duplicate elimination} \\ \min(|\mathbf{R}|, \Pi_{\mathbf{A}_i \in L} V(\mathbf{A}_i, \mathbf{R})) & \text{otherwise} \end{cases}$

Record size $s(Q)$: $\Sigma_{\mathbf{A}_i \in L} s(\mathbf{A}_i)$

Number of attribute values $V(\mathbf{A}_i, Q)$: $V(\mathbf{A}_i, \mathbf{R})$ for $\mathbf{A}_i \in L$

Union query $Q := R \cup S$

$$\begin{aligned} |Q| &\le |\mathbf{R}| + |\mathbf{S}| \\ s(Q) &= s(\mathbf{R}) = s(\mathbf{S}) \\ V(\mathbf{A}, Q) &\le V(\mathbf{A}, \mathbf{R}) + V(\mathbf{A}, \mathbf{S}) \end{aligned} \quad \text{sch}(\mathbf{R}) = \text{sch}(\mathbf{S})$$

Difference query $Q := R - S$

$$\begin{aligned} \max(0, |\mathbf{R}| - |\mathbf{S}|) \le |Q| &\le |\mathbf{R}| \\ s(Q) &= s(\mathbf{R}) = s(\mathbf{S}) \\ V(\mathbf{A}, Q) &\le V(\mathbf{A}, \mathbf{R}) \end{aligned}$$

Cross-product query $Q := R \times S$

$$\begin{aligned} |Q| &= |\mathbf{R}| \cdot |\mathbf{S}| \\ s(Q) &= s(\mathbf{R}) + s(\mathbf{S}) \\ V(\mathbf{A}, Q) &= \begin{cases} V(\mathbf{A}, \mathbf{R}), & \text{if } \mathbf{A} \in sch(\mathbf{R}) \\ V(\mathbf{A}, \mathbf{S}), & \text{if } \mathbf{A} \in sch(\mathbf{S}) \end{cases} \end{aligned}$$

Special cases of join queries $Q := R \bowtie_p S$

- no common attributes ($sch(\mathbf{R}) \cap sch(\mathbf{S}) = \emptyset$) or join predicate p = true

$$\mathbf{R} \bowtie_p \mathbf{S} = \mathbf{R} \times \mathbf{S}$$

- join attribute, say $\mathbf{A}$, is key in one of the relations, e.g., in $\mathbf{R}$, and assuming the inclusion dependency $\pi_{\mathbf{A}}(\mathbf{S}) \subseteq \pi_{\mathbf{A}}(\mathbf{R})$

$$|Q| = |\mathbf{R}|$$

↳ this inclusion dependency is guaranteed by a foreign key relationship between $\mathbf{R}.\mathbf{A}$ and $\mathbf{S}.\mathbf{A}$ in $\mathbf{R} \bowtie_{\mathbf{R}.\mathbf{A}=\mathbf{S}.\mathbf{A}} \mathbf{S}$.

General join queries $Q := R \bowtie_{R.A=S.B} S$

Assuming inclusion dependencies, the cardinality of a general join query Q can be estimated as

Cardinality $|Q|$: $\begin{cases} \frac{|\mathbf{R}| \cdot |\mathbf{S}|}{V(\mathbf{A}, \mathbf{R})}, & \text{for } \pi_{\mathbf{B}}(\mathbf{S}) \subseteq \pi_{\mathbf{A}}(\mathbf{R}) \\ \frac{|\mathbf{R}| \cdot |\mathbf{S}|}{V(\mathbf{B}, \mathbf{S})}, & \text{for } \pi_{\mathbf{A}}(\mathbf{R}) \subseteq \pi_{\mathbf{B}}(\mathbf{S}) \end{cases}$

Typically, the smaller of these two estimates is used

Cardinality $|Q|$: $\frac{|\mathbf{R}| \cdot |\mathbf{S}|}{\max(V(\mathbf{A}, \mathbf{R}), V(\mathbf{B}, \mathbf{S}))}$

Record size $s(Q)$: $s(\mathbf{R}) + s(\mathbf{S}) - \Sigma\, s(\mathbf{A}_i)$ for all common $\mathbf{A}_i$ of a **natural join**

Number of attribute values $V(\mathbf{A}', Q) \le \begin{cases} \min(V(\mathbf{A}', \mathbf{R}), V(\mathbf{A}', \mathbf{S})), & \mathbf{A}' \in sch(\mathbf{R}) \cap sch(\mathbf{S}) \\ V(\mathbf{A}', X), & \mathbf{A}' \in sch(X) \end{cases}$

5.2 File

Cost Model :

Parameter	Description
b	number of pages per file
r	number of records per page
D	time to read a disk page
C	CPU time needed to process a record
H	CPU time taken to apply a function to a record e.g. comparison or hash

Cost of Scan :

File Org	Description	est. Cost
Heap	read all pages, process each of the records per page	$b \cdot (D + r \cdot C)$
Sorted	Same as for heap files	$b \cdot (D + r \cdot C)$
Hashed	additional free space due to overflow chain avoidance	$(100/80) \cdot b \cdot (D + r \cdot C)$

Cost of Search w. Equality :

File Org	Description	est. Cost
Heap	if equality test is on primary key, adds factor $\frac{1}{2}$	$b \cdot (D + r \cdot C)$ or $\frac{1}{2} b \cdot (D + r \cdot C)$
Sorted	Assuming equality test is on sort criterion, use bin search	$\log_2 b \cdot D log_2 r \cdot C$
Hashed	Assuming equality test on hash attribute. Directly leads to the page containing the hit	$H + D + r \cdot C$ or $H + D + \frac{1}{2} r \cdot C$

Cost of Search w. Range :

File Org	Description	est. Cost
Heap	Can appear everywhere =¿ full scan	$b \cdot (D + r \cdot C)$
Sorted	Search for equality=lower and scan sequentially until the first record with A ¿ upper	$\log_2 b \cdot D + \log_2 r \cdot C + \lfloor \frac{n}{r} \rfloor \cdot D + n \cdot C$
Hashed	Performs worst as additional space needs to be scaned	$(100/80) \cdot b \cdot (D + r \cdot C)$

Cost of Insert :

File Org	Description	est. Cost
Heap	Can be written to an arbitary page, involves reading and writing the page	$2D + C)$
Sorted	Insert into a sepcific place and shift all subsequent	$\log_2 b \cdot D + \log_2 r \cdot C + \frac{1}{2} \cdot b \cdot (2 \cdot D + r \cdot C)$
Hashed	Write to the page that the hash fn indicates	$H + D + C + D$

Cost of Delete :

File Org	Description	est. Cost
Heap	Read, delete and write	$2D + C)$
Sorted	delete and shift all subsequent	$D + \frac{1}{2} \cdot b \cdot (2 \cdot D + r \cdot C)$
Hashed	Access by rid is faster than hashing so same as heap	$D + C + D$

5.3 Access Paths

Cost model for access methods on relation R

Access method	Cost
access primary index **I**	$\begin{cases} height(\mathbf{I})+1 & \text{if I is B+ tree} \\ 1.2+1 & \text{if I is hash index} \end{cases}$
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	$NPages(\mathbf{R})$

If less than 5% are retrieved, a table scan is cheaper.

Hash vs. B+Tree Index: Hash Indexes match if selection contains equality on indexed attribute; B+Trees match if selection contains any condition on an attribute in the trees search prefix. If matches with an index were found in a CNF, those conjuncts are called **primary conjuncts**

5.4 Operators

In Total Two-way Merge Sort costs

$$2N(1 + log_2 N)\text{I/O ops}$$

In Total External Merge Sort costs

$$2N(1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil)\text{I/O ops}$$

Cost of $R_1 \bowtie_p^{nl} R_2$

access path	file scan (**openScan**) of R_1 and R_2
prerequisites	none (p arbitrary, R_1 and R_2 may be heap files)
I/O cost	$\underbrace{\|R_1\|}_{\text{outer loop}} + \underbrace{\|R_1\| \cdot \|R_2\|}_{\text{inner loop}}$

Block Nested Loops Join Costs: $\lceil ||R_1||/b_1 \rceil \cdot \lceil ||R_2||/b_2 \rceil$

Cost of $R_1 \bowtie_p^{index\text{-}nl} R_2$

access path	file scan (**openScan**) of R_1, index access to R_2
prerequisites	index on R_2 that matches join predicate p
I/O cost	$\underbrace{\|R_1\|}_{\text{outer loop}} + \underbrace{\|R_1\| \cdot (\text{cost of } \textbf{one} \text{ index access to } R_2)}_{\text{inner loop}}$

Cost of $R_1 \bowtie_{A=B}^{sort\text{-}merge} R_2$

access path	sorted file scan of R_1 and R_2
prerequisites	p equality predicate $R_1.A = R_2.B$
I/O cost	cost of sorting R_1 and/or R_2, if not sorted already, plus
	best case: $\|R_1\| + \|R_2\|$
	worst case: $\|R_1\| \cdot \|R_2\|$

Cost of $R_1 \bowtie_{A=B}^{hash\text{-}join} R_2$

access path	file scan (**openScan**) of R_1 and R_2
prerequisites	equi-join, i.e., p equality predicate $R_1.A = R_2.B$
I/O cost	$\underbrace{\underbrace{\|R_1\| + \|R_2\|}_{\text{read}} + \underbrace{\|R_1\| + \|R_2\|}_{\text{write}}}_{\text{partitioning phase}} + \underbrace{\|R_1\| + \|R_2\|}_{\text{probing phase}} = 3 \cdot (\|R_1\| + \|R_2\|)$

Definition

The **selectivity** (or **reduction factor**) or a predicate p, denoted by $sel(p)$, is the faction of records in a relation R that satisfy the predicate p.

$$0 \leq sel(p) = \frac{|\sigma_p(R)|}{|R|} \leq 1$$

Reduction Factor: The fraction of tuples that satisfy a certain conjunct. For several conjuncts this factor is approximated, making an independence assumption. For range queries uniformity is assumed: For a tree index the reduction factor is estimated using the system catalouge: $\frac{High(T)-value}{High(T)-Low(T)}$

Cost of $\sigma_p^{scan}(R_{in})$ using a sequential scan

access path	file scan (**openScan**) of R_{in}
prerequisites	none (p arbitrary, R_{in} may be a heap file)
I/O cost	$\underbrace{\|R_{in}\|}_{\text{input cost}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$

Cost of $\sigma_p(R_{in})$ using binary search

access path	binary search, then sorted file scan of R_{in}
prerequisites	R_{in} sorted on sort key k that matches p
I/O cost	$\underbrace{\log_2\|R_{in}\|}_{\text{input cost}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{sorted scan}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$

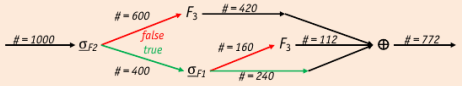
Cost of $\sigma_p(R_{in})$ using a clustered B+ tree index

access path	access of B+ tree on R_{in}, then sequence set scan
prerequisites	clustered B+ tree on R_{in} with key k that matches p
I/O cost	$\underbrace{\approx 3}_{\text{B+ tree access}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{sorted scan}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$

Cost of $\sigma_p(R_{in})$ using a hash index

access path	hash index on R_{in}
prerequisites	R_{in} hashed on key k, p has a term $k = c$
I/O cost	$\underbrace{\approx 1.2}_{\text{B+ tree access}} + \underbrace{sel(p) \cdot \|R_{in}\|}_{\text{output cost}}$

Example



Mean cost per tuple ($\oplus$ disjoint union): $C_2 + (1 - s_2) \cdot C_3 + s_2 \cdot (C_1 + (1 - s_1) \cdot C_3) = 40.6$

Note that many variations are possible, e.g., for tuning in parallel environments

Cost of $\pi_l^{sort}(R_{in})$ using sorting for duplicate elimination

access path	file scan (**openScan**) of R_{in}
prerequisites	none (B available buffer pages)
I/O cost	$\underbrace{\|R_{in}\| + \|R_{tmp}\|}_{\text{projection}} + \underbrace{2 \cdot \|R_{tmp}\| \cdot \left(\left\lceil \log_{B-1} \left\lceil {}^{\|R_{tmp}\|}/_{B} \right\rceil \right\rceil\right)}_{\text{duplicate elimination}}$

Cost of $\pi_l^{hash}(R_{in})$ using hashing for duplicate elimination

access path	file scan (**openScan**) of R_{in}
prerequisites	none (B available buffer pages)
I/O cost	$\underbrace{\|R_{in}\| + \|R_{tmp}\|}_{\text{projection}} + \underbrace{\|R_{tmp}\| + \|R_{tmp}\|}_{\text{duplicate elimination}}$