Midterm Review

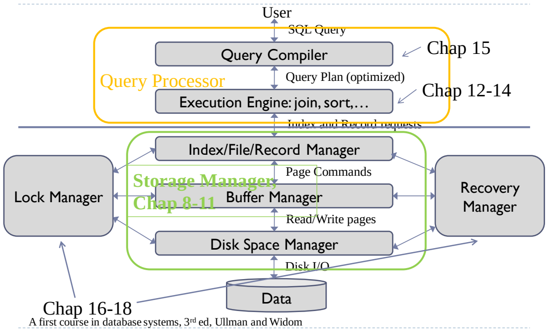
CS634

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Coverage

- Text, chapters 8 through 15 (hw1 – hw4)
- PKs, FKs, E-R to Relational: Text, Sec. 3.2-3.5, to pg. 77 inclusive, createdb.sql
- Basics of RAID: Sec. 9.2, Slides of Lecture 3
- SQL for creating and dropping tables (standardized), Not standardized: create indexes, commands for bulk loading big tables (Oracle and mysql cases).

Architecture of a DBMS



A first course in database systems, 3rd ed, Ullman and Widom

Disks

Accessing a Disk Block

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate for up to about 1MB transfers, and DB pages are smaller than that
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays!

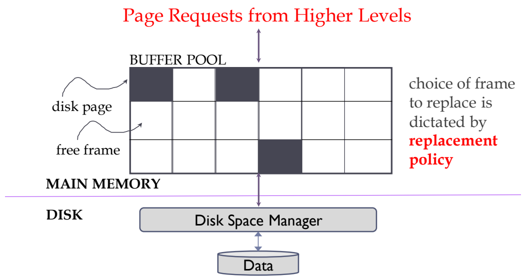
Arranging Pages on Disk

- \`*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
  - Logical block numbers of current disks follow this sequence
- Blocks that are accessed together frequently should be sequential on disk (by \`next'), to minimize access time
- Use newly-initialized file systems for DB files to avoid OS file fragmentation
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Important RAID Levels

- Level 0: Striping but no redundancy
  - Maximum transfer rate = aggregate bandwidth
  - Stripe size can be many blocks, example 256KB
  - With *N* data disks, read/write bandwidth improves up to *N* times
- Level 1: Mirroring ←strongly recommended for redo log files
  - Each data disk has a mirror image (check disk)
  - Parallel reads possible, but a write involves both disks
- Level 0+1: Striping and Mirroring (AKA RAID 10)
  - Maximum transfer rate = aggregate bandwidth
  - With *N* data disks, read bandwidth improves up to *N* times
- Level 5: Block-Interleaved Distributed Parity (in wide use)
  - Every disk acts as data disk for some blocks, and check disk for other blocks
  - Most popular of the higher RAID levels (over 0+1).
- Dbs3 has RAID 5, even for redo log file (so not best performance for actions that change the database)

Buffer Management



- *Size of buffer pool is a huge factor in performance! And under DBA control.*

File Organization

1. Unsorted, or heap file
   - Records stored in random order
2. Sorted according to set of attributes
   - E.g., file sorted on <*age*>
   - Or on the combination of <*age*, *salary*>

- No single organization is best for all operations
  - E.g., sorted file is good for range queries
  - But it is expensive to insert records
  - We need to understand trade-offs of various organizations

Unordered Files: Heap

- Heap
  - simplest file structure
  - contains records in no particular order
  - as file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page

Data Organization

- Index/File/Record Manger provides abstraction of file of records (or short, file)
  - File of records is collection of pages containing records
  - A File can be a heap table, a heap table accessed via a certain index, a sorted table, or a certain index
- File operations
  - read/delete/modify a record (specified using *record id*)
  - insert record
  - scan all records, search with equality selection, search with range selection
- Record id functions as data locator
  - contains information on the address of the record on disk
  - e.g., page and record offset in page
  - "search-by-address"

QP to Storage Engine API

- Storage Engine works on one "File" at a time, that is, in one call from the QP, which could be in the middle of doing a join of two tables, or a sort, or …
- Table scan and index scan are just scans of two kinds of Files
- Cost models are based on the costs of the various calls into the Storage Engine, since it does all the disk i/o.
- See Figure 8.4 for various costs.

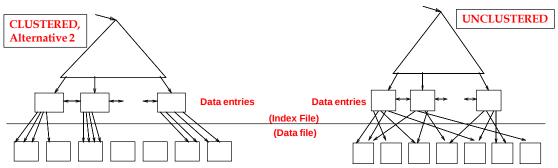
Indexing, starts in Chap. 8, then continues in 10 and 11

Alternatives for Data Entry *k** in Index (pg. 276 in Chap 8)

1. Data record with key value *k*
   - Leaf node stores actual record
   - Only one such index can be used (without replication)
2. *<k, rid>* rid of data record with search key value *k*
   - Only a pointer (rid) to the page and record are stored
3. *<k, list of rids>* list of rids of records with search key *k*
   - Similar to previous method, but more compact
   - Disadvantage is that data entry is of variable length
   - Don't worry about this case for exams

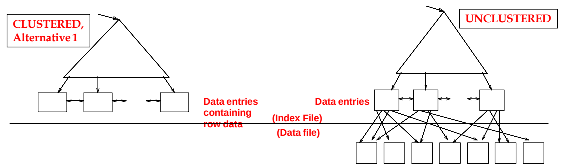
- Several indexes with alternatives 2 and 3 may exist

Clustered vs. Unclustered Index



- Clustered index: order of data records is close to the sort order
- Here: loaded from ordered data, so records fall in order naturally.
- However, the most common kind of clustered index uses Alternative 1, not Alternative 2 as shown above, see next slide for picture
- Unclustered: must be Alternative 2 (or 3, but we're not worrying about that case)

Clustered vs. Unclustered Indexes



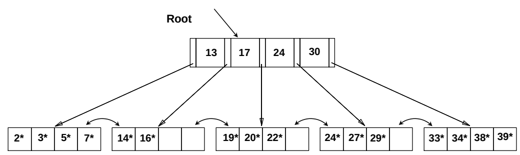
- Clustered index: order of data records is close to the sort order
- The most common kind of clustered index uses Alternative 1, as shown above
- If see "clustered index" without Alternative specified, assume Alternative 1.

B+ Tree

- Most Widely Used Index
- Dynamic data structure (as opposed to ISAM)
- Tree is *height-balanced*
  - Height is $\log_F N$ (F = fanout, N = # leaf pages)
- Minimum 50% occupancy constraint
  - Each node (except root) contains **d** <= m <= 2**d** entries
  - Parameter **d** is called the *order* of the tree
- Search just like in ISAM
  - But insert/delete more complex due to occupancy constraint
  - Insert/delete may trigger re-structuring at all levels of tree

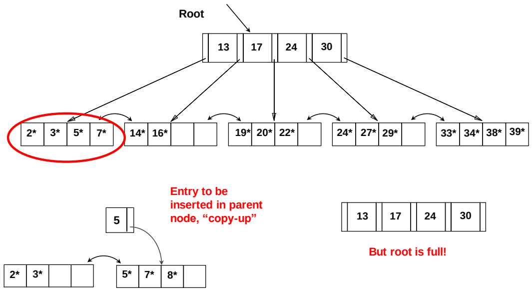
B+ Tree Example

- Search begins at root, key comparisons direct it to a leaf

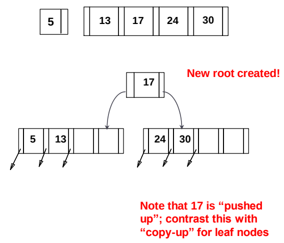


Based on the search for 15, we know it is not in the tree!*

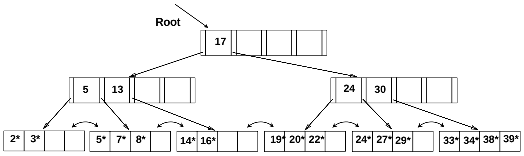
B+ Tree Example: Insert 8* (d=2)



B+ Tree Example: Insert 8* (d=2)



Example B+ Tree After Inserting 8*



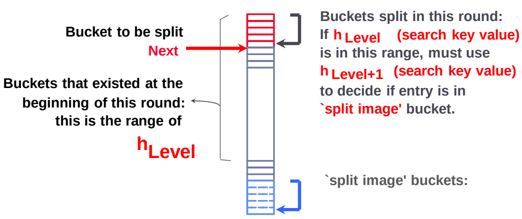
Root was split, leading to increase in height

We can avoid split by re-distributing entries, but his is usually not done in practice for insertions

Linear Hashing

- Dynamic hashing scheme
- Handles the problem of long overflow chains
  - But does not require a directory!
  - Deals well with collisions!
- Main Idea: use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$
    - N = initial number of buckets
  - If $N = 2^{d_0}$, for some d_0, $\mathbf{h}_i$ consists of applying **h** and looking at the last d_i bits, where $d_i = d_0 + i$
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

Overview of Linear Hashing

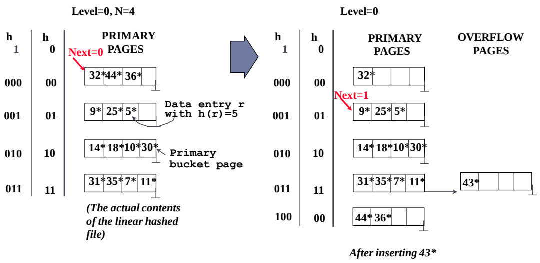


Linear Hashing Properties

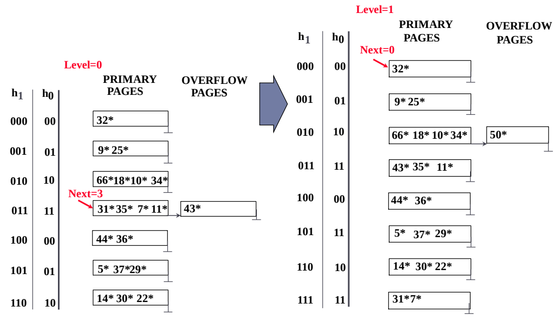
- Directory avoided in LH by using overflow pages
- Buckets are split round-robin
  - Splitting proceeds in `rounds'
    - Round ends when all N_R initial buckets are split (for round R)
    - Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number referred to as *Level*
- Search for data entry *r* :
  - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', search bucket $\mathbf{h}_{Level}(r)$
  - Otherwise, apply $\mathbf{h}_{Level+1}(r)$ to find bucket

Example of Linear Hashing

- On split, $h_{Level+1}$ is used to re-distribute entries.



End of a Round



Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

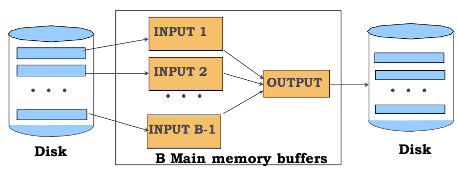
Notes on these costs

- B = # pages of data in file
- D = time for one (random) i/o
- R = # records/page
- Without further info on index size, use 10% table size
- Here, scan of clustered table takes 1.5 BD, where B = #pages data would take in a heap file, but if B = no of data pages of B-tree, average of 2/3 full, then should be just BD.
- Similarly, scan of FILE by unclustered tree index = .1(size of table) + RBD
- Later, we said Size(Clustered index) = 1.1(Size of data), which really only should be used if the B-tree nodes are full.
- B-tree nodes can be full after bulk load from sorted data.

Sorting

General External Merge Sort

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 2, …, etc.: merge *B-1* runs.



Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{B-1}\lceil N / B\rceil\rceil$
- Cost = 2N * (# of passes), assuming we need to read the input from a FILE and write the output to a FILE.
- Can save some i/o using pipelining in and out.
- Example: with 5 buffer pages, sort 108 page file:
  - Pass 0: ceil(108/5) = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4\rceil$= 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Query Evaluation

Executing Selections

- Find the *most selective access path*, retrieve tuples using it
  - Then, apply any remaining terms that don't match the index
- *Most selective access path:* index or file scan estimated to require the fewest page I/Os
  - Consider *day<8/9/94 AND bid=5 AND sid=3*
- If we have B+ tree index on *day*, use that access path
  - Then, *bid=5* and *sid=3* must be checked for each retrieved tuple
  - *day* condition is primary conjunct
- Alternatively, use hash index on *<bid, sid>* first
  - Then, *day<8/9/94* must then be checked

Example of matching indexes

Pg. 399: fix error Sailors→ Reserves on line 8

Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string) ← rname column added here

with indexes:

- Index1: Hash index on (rname, bid, sid)
  - Matches: rname='Joe' and bid = 5 and sid=3
  - Doesn't match: rname='Joe' and bid = 5
- Index2: Tree index on (rname, bid, sid)
  - Matches: rname='Joe' and bid = 5 and sid=3
  - Matches: rname='Joe' and bid = 5, also rname = 'Joe'
  - Doesn't match: bid = 5
- Index3: Tree index on (rname)
- Index4: Hash index on (rname)
  - These two match any conjunct with rname='Joe' in it

Using an Index for Selections

- Cost influenced by:
  - Number of qualifying tuples
  - Whether the index is clustered or not
  - Cost of finding qualifying data entries is typically small
  - E.g.,

```
SELECT  *
FROM    Reserves R
WHERE   R.rname < 'C%'
```

- Assuming uniform distribution of names, 10% of tuples qualify, that is 10000 tuples
  - With a clustered index, cost is little more 100 I/Os
  - If not clustered, up to10K I/Os!

Hw4 **Problem 2 (15.2, question 1)**

NPages(R) = 10000, 8000 usable bytes/page, *20 bytes/data entry*

NTuples(R) = 800,000, so 800,000 *20 bytes/secondary index, = 2000 pgs

Reduction Factor (RF) = 0.1

For unclustered indexes, NPages(UI) = 2000

Cost(UI) = (NPages(UI)+NTuples(R)) * product of RF's of matching predicates

For clustered indexes, the rows lie in the leaf pages, 2000 of them, and the level above that has 2000 data entries, or 20*2000 bytes = 25 pages. The level above that is the root.

Alt. 1: NPages(CI) = NLeafPages + NIndexPages = 10,000 + 25 = 10,025.

Alt. 2: NPages(CI) = index size + table size = 2000 + 10,000 = 12,000

Cost(CI) = NPages(CI) * product of RF's of matching predicates

Index #1: Unclustered hash index on *eid*

Index #2: Unclustered B+ Tree index on *sal*

Index #3: Unclustered hash index on *age*

Index #4: Clustered B+ Tree index on *<age, sal>*

- See the hw4 solution for further info on this.

Projection with Sorting

- Modify Pass 0 of external sort to eliminate unwanted fields
  - Runs of about 2B pages are produced
  - Tuples in runs are smaller than input tuples
  - Size ratio depends on number and size of fields that are dropped
- Modify merging passes to eliminate duplicates
  - Thus, number of result tuples smaller than input
  - Difference depends on number of duplicates
- Cost
  - In Pass 0, read original relation (size M), write out same number of smaller tuples
  - In merging passes, fewer tuples written out in each pass. Using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25

Projection with Hashing

- *Partitioning phase*:
  - Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function *h1* to choose one of B-1output buffers
  - Result is B-1 partitions (of tuples with no unwanted fields), tuples from different partitions guaranteed to be distinct
- *Duplicate elimination phase*:
  - For each partition, read it and build an in-memory hash table, using hash *h2* on all fields, while discarding duplicates
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition
- Cost
  - Read R, write out each tuple, but fewer fields. Result read in next phase

Discussion of Projection

- Sort-based approach is the standard
  - better handling of skew and result is sorted.
- If index on relation contains all wanted attributes in its search key, do *index-only* scan
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as *prefix* of search key, can do even better:
  - Retrieve data entries in order (index-only scan)
  - Discard unwanted fields, compare adjacent tuples to check for duplicates

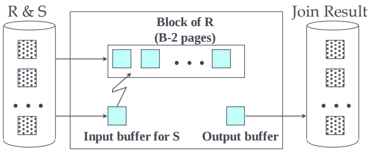
Simple Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S do
    if $r_i == s_j$ then add <r, s> to result

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: M + p_R * M * N = 1000 + 100*1000*500 I/Os
- Page-oriented Nested Loops join:
  - For each *page* of R, get each *page* of S, and write out matching pairs
  - Cost: M + M*N = 1000 + 1000*500
  - If smaller relation (S) is outer, cost = 500 + 500*1000

Block Nested Loops Join

- one page input buffer for scanning the inner S
- one page as the output buffer
- remaining pages to hold ``block" of outer R
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Example of NLJ

SELECT *
FROM Reserves R1, Sailors S1
WHERE R1.sid=S1.sid

- Reserves:
  - 40 bytes long tuple, 100K records, 100 tuples per page, 1000 pages, suppose B+ Tree index on bid, another on sid
- Sailors:
  - 50 bytes long tuple, 40K tuples, 80 tuples per page, 500 pages, suppose B+ tree index on sid

Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \# \text{ of pages of outer} / \text{blocksize} \rceil$
- With Reserves (R) as outer, and 100 pages of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - Total 6000 i/os
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.

    Total 5500 I/Os
- With 50-page block of S as outer (hw4 #1 part 2, S has 200 pages, R has 1000)
  - Cost of scanning S is 200 I/Os; a total of 4 blocks.
  - Per block of S, we scan Reserves; 4*1000 I/Os.

    Total 4,200 I/Os

Executing Joins: Index Nested Loops

foreach tuple r in R do
foreach tuple s in S where $r_i == s_j$ do
add <r, s> to result

- Cost = M + (M*p_R) * (cost of finding matching S tuples)
- M = number of pages of R, p_R = number of R tuples per page
- If relation has index on join attribute, make it inner relation
  - For each outer tuple, cost of probing inner index is 1.2 for hash index, 2-4 (say 2 for simplicity) for B+, plus cost to retrieve matching S tuples
  - Clustered index:
    - Alt 1: typically no more I/Os (data entry has whole row)
    - Alt 2: typically single I/O (data entry has RIDs, but target rows are clustered in table)
  - Unclustered index 1 I/O per matching S tuple

Example of Index Nested Loops (1/2)

Case 1: B+tree-index on *sid* of Sailors

- Choose Sailors as inner relation
- Scan Reserves: 100K tuples, 1000 page I/Os
- For each Reserves tuple
  - 2 I/Os to get data entry in index (simplifying 2-4 in text)
  - No more i/o if clustered Alt 1, 1 I/O to get (the exactly one) matching Sailors tuple (primary key), clustered Alt 2 or unclustered
- Total: 201,000 or 301,000 I/Os, terrible. 3010 s = 50 min.

Example of Index Nested Loops (2/2)

Case 2: B+ tree-index on *sid* of Reserves

- Choose Reserves as inner
- Scan Sailors: 40K tuples, 500 page I/Os
- For each Sailors tuple
  - 2 I/Os to find index page with data entries (simplified from 2-4)
  - Assuming uniform distribution, 2.5 matching records per sailor
  - Cost of retrieving records is nothing (Alt 1 clustered), single I/O (Alt. 2 clustered index) or 2.5 I/Os (unclustered index)
- Total: 80,500 I/Os (clustered Alt 1), 120,500 I/Os (clustered Alt 2) or 180,500 I/Os (unclustered) All bad.
- Better to use block NLJ here, if required to do NLJ.

Sort-Merge Join

- Sort R and S on the join column
- Then scan them to do a merge on join column:
  - Advance scan of R until current R-tuple >= current S tuple
  - Then, advance scan of S until current S-tuple >= current R tuple
  - Repeat until current R tuple = current S tuple
  - At this point, all R tuples with same value in *Ri* (*current R group*) and all S tuples with same value in *Sj* (*current S group*) **match**
  - Output <r, s> for all pairs of such tuples
  - Resume scanning R and S

Sort-Merge Join Cost

- R is scanned once
- Each S group is scanned once per matching R tuple
  - Multiple scans per group needed only if S records with same join attribute value span multiple pages
  - Multiple scans of an S group are likely to find needed pages in buffer
- Cost: (assume B buffers)
  - $2M (1+\log_{B-1}(M/B)) + 2N (1+\log_{B-1}(N/B)) + (M+N)$
  - The cost of scanning, M+N, could be M*N worst case (very unlikely!)
  - In many cases, join attribute is primary key in one of the tables!

Sort-Merge Join Cost: hw4 #1, part 3

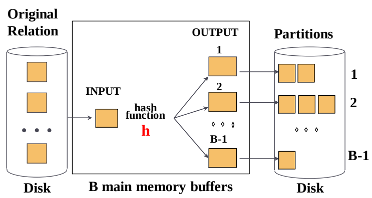
- R JOIN S on R.a = S.b, where b is the PK of S. B=1500.
- 1. Sort R and S on the join column:
  - Sort R on a: 100K pages. S on b: 20K pages
    - Pass 0: 100K/1500 = 68 runs of 1500 pages; 20K/1500 = 14
    - Pass 1: merge 68 runs into one merge 14 into one
  - So both are 2-pass sorts, unlike original-problem setup
  - Cost = 2*2*M+2*2*N = 4*(100K+20K) = 480K i/os.
- 2. Then scan them to do a merge on join column:
  - R is scanned once, each row matching one row of S
  - Cost = M+N reads (ignore output costs by problem) = 120K i/os
- Total cost = 600K i/os (not 100x old answer!)
- Note: this is not using the optimization which yields 3(M+N) for 2-pass sorts: by that algorithm, cost = 3*120K = 360K i/os.

2-Pass Sort-Merge Join

- With enough buffers, sort can be done in 2 passes
  - First pass generates N/B sorted runs of B pages each
  - If one page from each run + output buffer fits in memory, then merge can be done in one pass; denote larger relation by L
  - 2L/B + 1 <= B, holds if (approx) B > $\sqrt{2L}$
- One optimization of sort allows runs of 2B on average
  - First pass generates N/2B sorted runs of 2B pages each
  - Condition above for 2-pass sort becomes B > $\sqrt{L}$
- Merge can be combined with filtering of matching tuples
  - The cost of sort-merge join becomes 3(M+N)

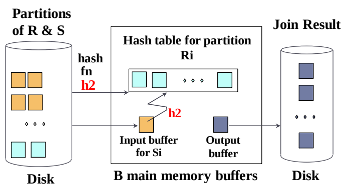
Hash-Join: Partitioning Phase

- Partition both relations using hash function **h**
- R tuples in partition *i* will only match S tuples in partition *i*



Hash-Join: Probing Phase

Read in a partition of R, hash it using **h2 (<> h!)**
Scan matching partition of S, search for matches.



Hash-Join Properties

- #partitions k <= B-1 because one buffer is needed for scanning input
- Assuming uniformly sized partitions, and maximizing k:
  - k= B-1, and M/(B-1) <= B-2, i.e., B > $\sqrt{M}$
  - M is smaller of the two relations!
- If we build an in-memory hash table to speed up the matching of tuples, slightly more memory is needed
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory
  - Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- In partitioning phase, read+write both R and S: 2(M+N)
- In matching phase, read both R and S: M+N
- So Cost with partitioning = 3(M+N)
  - With sizes of 1000 and 500 pages, total is 4500 I/Os
  - (not counting materialization of result)
- If hash table of one table's data fits in memory, cost = M+N
- If hash table for a partition doesn't fit in memory, cost exceeds above estimate.

Hash-Join vs Sort-Merge Join

- Given sufficient amount of memory both have a cost of 3(M+N) I/Os
- Hash Join superior on this count if relation sizes differ greatly
- Hash Join shown to be highly parallelizable
- Sort-Merge less sensitive to data skew, and result is sorted

Query Optimization: Chap. 15

CS634
Lecture 12, Mar 9, 2016

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Block Optimization

- Block = Unit of optimization
- For each block, consider:
  1. All available access methods, for each relation in FROM clause
  2. All **left-deep join trees**
     - all ways to join the relations one-at-a-time
     - all relation permutations and join methods
- Recall:
- Left table = outer table of a nested loop join
- Left table of NLJ can be pipelined: rows used one at a time in order
- But need to consider other join methods too, giving up pipelining in many cases

$\sigma \pi \times$ Expressions

- Query is simplified to a selection-projection-cross product expression
  - Aggregation and grouping can be done afterwards
- Optimization with respect to such expressions
- Cross-product includes conceptually joins
  - Will talk about equivalences in a bit

Size Estimation and Reduction Factors

SELECT attribute list
FROM relation list
WHERE term_1 AND … AND term_k

- Maximum number of tuples is cardinality of cross product
- Reduction factor (RF) associated with each *term* reflects its impact in reducing result size
  - Implicit assumption that *terms* are independent!
  - *col = value* has RF =1/NKeys(*I*), given index *I* on *col*
  - *col1 = col2* has RF = 1/max(NKeys(*I1*), NKeys(*I2*))
  - *col > value* has RF = (High(*I*)-value)/(High(*I*)-Low(*I*))

Single-Relation Plans

- FROM clause contains single relation
- Query is combination of selection, projection, and aggregates (possibly GROUP BY and HAVING, but these come late in the logical progression, so usually less crucial to planning)
- Main issue is to select best from all available access paths (either file scan or index)
- Access path involves the table and the WHERE clause
- Another factor is whether the output must be sorted
  - E.g., GROUP BY requires sorting
  - Sorting may be done as separate step, or using an index if an indexed access path is available

Plans Without Indexes

- Only access path is file scan
- Apply selection and projection to each retrieved tuple
  - Projection may or may not use duplicate elimination, depending on whether there is a DISTINCT keyword present
- GROUP BY:
- Write out intermediate relation after selection/projection
- (or pipeline into sort)
- Sort intermediate relation to create groups
- Apply aggregates on-the-fly per each group
  - HAVING also performed on-the-fly, no additional I/O needed

Plans With Indexes

- There are four cases:

1. Single-index access path
   - Each index offers an alternative access path
   - Choose one with lowest I/O cost
   - Non-primary conjuncts, projection, aggregates/grouping applied next
2. Multiple-index access path
   - Each index used to retrieve set of rids
   - Rid sets intersected, result sorted by page id
   - Retrieve each page only once
   - Non-primary conjuncts, projection, aggregates/grouping applied next

Plans With Indexes (contd.)

3. Index-only access path
   - If all attributes in query included in index, then there is no need to access data records: **index-only scan**
   - If index matches selection, even better: only part of index examined
   - Does not matter if index is clustered or not!
   - If GROUP BY attributes prefix of a tree index, no need to sort!
   - Example: With tree index on rating

```
SELECT max(rating),count(*)
FROM Sailors S
```

   - Note count(*) doesn't require access to row, just RID.

Plans With Indexes (contd.)

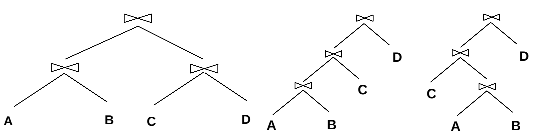
3. Index-only access path
   - If all attributes in query included in index, then there is no need to access data records: **index-only scan**
   - If index matches selection, even better: only part of index examined
   - Does not matter if index is clustered or not!
   - If GROUP BY attributes prefix of a tree index, no need to sort!
   - Example: With tree index on rating

```
SELECT max(rating),count(*)
FROM Sailors S
```

   - Note count(*) doesn't require access to row, just RID.

Queries Over Multiple Relations

- In System R *only left-deep join trees* are considered
  - In order to restrict the search space
  - Left-deep trees allow us to generate all *fully pipelined* plans
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., sort-merge join)



Left-deep

Enumeration of Left-Deep Plans

- Among all left-deep plans, we need to determine:
  - the order of joining relations
  - the access method for each relation
  - the join method for each join
- Enumeration done in N passes (if N relations are joined):
  - Pass 1: Find best 1-relation plan for each relation
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation - result is the set of all 2-relation plans
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation - result is the set of all N-relation plans
- Speed-up computation using dynamic programming