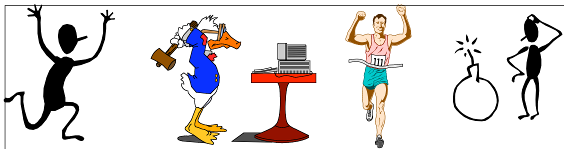


Midterm Review

Spring 2003

Overview

- **Sorting**
- **Hashing**
- **Selections**
- **Joins**

Two-Way External Merge Sort

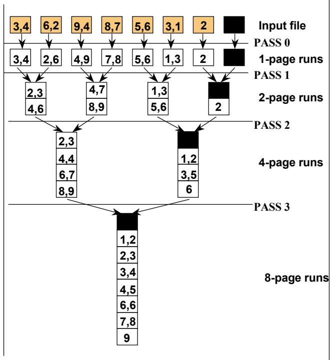
- **Each pass we read + write each page in file.**
- **N pages in the file => the number of passes**

$$= \lceil \log_2 N \rceil + 1$$

- **So total cost is:**

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

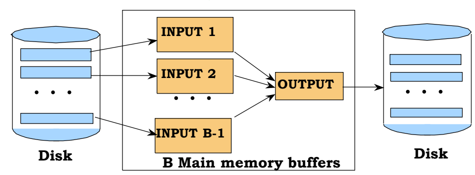
- ***Idea:*** *Divide and conquer:* **sort subfiles and merge**



General External Merge Sort

☛ *More than 3 buffer pages. How can we utilize them?*

- **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 1, 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)**
- **E.g., with 5 buffer pages, to sort 108 page file:**
  - Pass 0: $\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
- **Now, do four-way (B-1) merges**
  - 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

Sorting warnings

- **Be able to run the general external merge sort!**
  - Careful use of buffers in pass 0 vs. pass *i*, *i*>0.
  - Draw pictures of runs like the "tree" in the slides for 2-way external merge sort (will look slightly different!)
- **Be able to compute # of passes correctly for file of *N* blocks, *B* buffers!**
  - Watch the number of buffers available in pass 0
  - tournament sort (heapsort) vs. quicksort
  - Be able to count I/Os carefully!

More tips

- How to sort any file using 3 memory Pages
- How to sort in as few passes given some amount of memory
- I have a file of **N** blocks and **B** buffers
  - How big can **N** be to sort in **2** phases ?

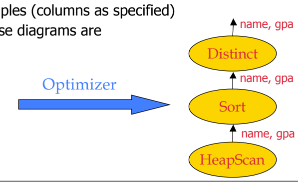
B-1 >= N/B

So, N <= B^2 .. approx of course

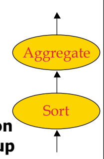
Query Processing Overview

- **The *query optimizer* translates SQL to a special internal "language"**
  - Query Plans
- **The *query executor* is an *interpreter* for query plans**
- **Think of query plans as "box-and-arrow" *dataflow* diagrams**
  - Each box implements a *relational operator*
  - Edges represent a flow of tuples (columns as specified)
  - For single-table queries, these diagrams are straight-line graphs

```
SELECT DISTINCT name, gpa
  FROM Students
```



Sort GROUP BY: Naïve Solution



- **The Sort iterator (could be external sorting, as explained last week) naturally permutes its input so that all tuples are output in sequence**
- **The Aggregate iterator keeps running info ("transition values") on agg functions in the SELECT list, per group**
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVERAGE it keeps `sum-so-far` *and* `count-so-far`
- **As soon as the Aggregate iterator sees a tuple from a new group:**
  1. It produces an output for the old group based on the agg function
     E.g. for AVERAGE it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple's info

An Alternative to Sorting: Hashing!

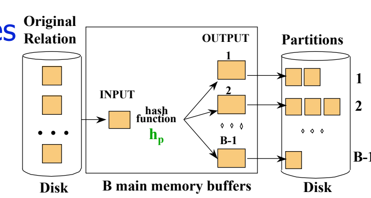
- **Idea:**
  - Many of the things we use sort for don't exploit the *order* of the sorted data
  - E.g.: forming groups in GROUP BY
  - E.g.: removing duplicates in DISTINCT
- **Often good enough to match all tuples with equal field-values**
- **Hashing does this!**
  - And may be cheaper than sorting! (Hmmm…!)
  - But how to do it for data sets bigger than memory??

General Idea

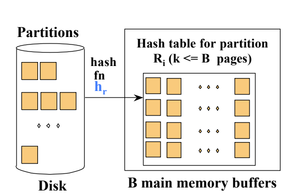
- **Two phases:**
  - Partition: use a hash function h_p to split tuples into partitions on disk.
    - We know that all matches live in the same partition.
    - Partitions are "spilled" to disk via output buffers
  - ReHash: for each partition on disk, read it into memory and build a main-memory hash table based on a hash function h_r
    - Then go through each bucket of this hash table to bring together matching tuples

Two Phases

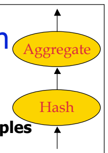
- **Partition:**



- **Rehash:**

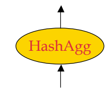


Hash GROUP BY: Naïve Solution (similar to the Sort GROUPBY)



- **The Hash iterator permutes its input so that all tuples are output in sequence**
- **The Aggregate iterator keeps running info ("transition values") on agg functions in the SELECT list, per group**
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVERAGE it keeps `sum-so-far` *and* `count-so-far`
- **When the Aggregate iterator sees a tuple from a new group:**
  1. It produces an output for the old group based on the agg function
     E.g. for AVERAGE it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple's info

We Can Do Better!



- **Combine the summarization into the hashing process**
  - During the ReHash phase, don't store tuples, store pairs of the form <GroupVals, TransVals>
  - When we want to insert a new tuple into the hash table
    - If we find a matching GroupVals, just update the TransVals appropriately
    - Else insert a new <GroupVals,TransVals> pair
- **What's the benefit?**
  - Q: How many pairs will we have to handle?
  - A: Number of distinct values of GroupVals columns
    - Not the number of tuples!!
  - Also probably "narrower" than the tuples
- **Can we play the same trick during sorting?**

Hashing for Grouped Aggregation

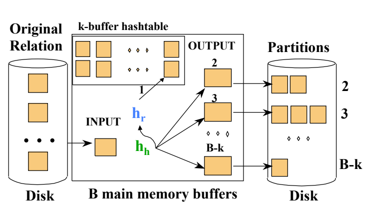
- **How big can a partition be ?**
  - As big as can fit into the hashtable during rehash
  - For grouped aggs, we have one entry per group !
  - So, the key is : **the number of unique groups !**
  - **A partition's size is *only* limited by the number of unique groups in the partition**
- Similar analysis holds for duplicate elimination
  - Note: Can think of dup-elem as a grouped agg
  - All tuples that contribute to the agg are identical
  - So any tuple of a "group" is a "representative"

Analysis

- **How big of a table can we process?**
  - B-1 "spill partitions" in Phase 1
  - Each limited by the number of unique tuples per partition and that can be accommodated in the hash table (U_H)
- **Have a bigger table?** Recursive partitioning!
  - In the ReHash phase, if a partition *b* has more unique tuples than U_H, then recurse:
    - pretend that *b* is a table we need to hash, run the Partitioning phase on *b*, and then the ReHash phase on each of its (sub)partitions

Even Better: Hybrid Hashing

- **What if the set of <GroupVals,TransVals> pairs fits in memory**
  - It would be a waste to spill it to disk and read it all back!
  - Recall this could be true even if there are *tons* of tuples!
- **Idea: keep a smaller 1st partition in memory during phase 1!**
  - Output its stuff at the end of Phase 1.
  - Q: how do we choose the number k?



Analysis: Hybrid Hashing, GroupAgg

- **H** buffers in all:
  - In Phase 1: **P** "spill partitions", **H-P** buffers for hash table
  - Subsequent phases: **H-1** buffers for hash table
- **How big of a table can we process ?**
  - Each of the **P** partitions is limited by the number of unique tuples per partition and that can be accommodated in the hash table ($\mathbf{U_H}$)
  - Note that that $\mathbf{U_H}$ depends on the phase !
    - In Phase 1 $\mathbf{U_H}$ is based on **H-P** buffers
    - In subsequent phases $\mathbf{U_H}$ is based on **H-1** buffers

Simple Selections (cont)

- **With no index, unsorted:**
  - Must essentially scan the whole relation
  - cost is M (#pages in R). For "reserves" = 1000 I/Os.
- **With no index, sorted:**
  - cost of binary search + number of pages containing results.
  - For reserves = 10 I/Os + ⌈selectivity*#pages⌉
- **With an index on selection attribute:**
  - Use index to find qualifying data entries,
  - then retrieve corresponding data records.
  - Cost?

Using an Index for Selections

- **Cost depends on #qualifying tuples, and clustering.**
  - Cost:
    - finding qualifying data entries (typically small)
    - plus cost of retrieving records (could be large w/o clustering).
  - In example "reserves" relation, if 10% of tuples qualify (100 pages, 10000 tuples).
    - With a *clustered* index, cost is little more than 100 I/Os;
    - If *unclustered*, could be up to 10000 I/Os!
      - Unless you get fancy…

Projection (DupElim)

```
SELECT  DISTINCT
        R.sid, R.bid
FROM    Reserves R
```

- **Issue is removing duplicates.**
- **Basic approach is to use sorting**
  - 1. Scan R, extract only the needed attrs (why do this 1st?)
  - 2. Sort the resulting set
  - 3. Remove adjacent duplicates
  - Cost: Reserves with size ratio 0.25 = 250 pages. With 20 buffer pages can sort in 2 passes, so 1000 +250 + 2 * 2 * 250 + 250 = 2500 I/Os
- **Can improve by modifying external sort algorithm (see chapter 12):**
  - Modify Pass 0 of external sort to eliminate unwanted fields.
  - Modify merging passes to eliminate duplicates.
  - Cost: for above case: read 1000 pages, write out 250 in runs of 40 pages, merge runs = 1000 + 250 +250 = 1500.

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.**
- **How much does this Cost?**
- **(p_R * M) * N + M = 100*1000*500 + 1000 I/Os.**
  - At 10ms/IO, Total: ???
- **What if smaller relation (S) was outer?**

- **What assumptions are being made here?**

Q: What is cost if one relation can fit entirely in memory?

Page-Oriented Nested Loops Join

```
foreach page b_R in R do
  foreach page b_S in S do
    foreach tuple r in b_R do
      foreach tuple s in b_S do
        if r_i == s_j then add <r, s> to result
```

- **For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.**

- **What is the cost of this approach?**

- **M*N + M= 1000*500 + 1000**
  - If smaller relation (S) is outer, cost = 500*1000 + 500

Question from midterm fall 1998

- **Sorting:** Trying to sort a file of 250,000 blocks with only 250 buffers available.
  - How many initial runs will be generated with quicksort ? **N/B = 250,000/250 = 1000**
  - How many total I/O will the sort perform, *including* the cost of writing out the output ?
    $2N(\log_{B-1}[N/B] + 1)$
  - How many runs (on average) with heapsort ?
    Avg size = 2(B-2) = 2(248) = 496
    Num runs = N/2(B-2) = 250 = 504

Question from midterm fall 1998

- **Sorting:** Trying to sort a file of 250,000 blocks with only 250 buffers available.
  - How many initial runs will be generated with quicksort ?
  - How many total I/O will the sort perform, *including* the cost of writing out the output ?
  - How many runs (on average) with heapsort ?