CAS CS 460/660
Introduction to Database Systems

Query Evaluation I

Slides from UC Berkeley
We’ve covered the basic underlying storage, buffering, and indexing technology.

Now we can move on to query processing.

Some database operations are EXPENSIVE

Can greatly improve performance by being “smart”

e.g., can speed up 1,000x over naïve approach

Main weapons are:

1. clever implementation techniques for operators
2. exploiting “equivalencies” of relational operators
3. using statistics and cost models to choose among these.
Cost-based Query Sub-System

Queries

Select * 
From Blah B 
Where B.blah = blah

Usually there is a heuristics-based rewriting step before the cost-based steps.
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
A deep subject, focuses on multi-table queries

- We will only need a cookbook version for now.

Build the dataflow bottom up:

- Choose an Access Method (HeapScan or IndexScan)
  - Non-trivial, we’ll learn about this later!
- Next apply any WHERE clause filters
- Next apply GROUP BY and aggregation
  - Can choose between sorting and hashing!
- Next apply any HAVING clause filters
- Next Sort to help with ORDER BY and DISTINCT
  - In absence of ORDER BY, can do DISTINCT via hashing!
The relational operators are all subclasses of the class `iterator`:

```c++
class iterator {
    void init();
    tuple next();
    void close();
    iterator inputs[];
    // additional state goes here
}
```

**Note:**
- Edges in the graph are specified by inputs (max 2, usually 1)
- Encapsulation: any iterator can be input to any other!
- When subclassing, different iterators will keep different kinds of state information
Example: Scan

class Scan extends iterator {
  void init();
  tuple next();
  void close();

  iterator inputs[1];
  bool_expr filter_expr;
  proj_attr_list proj_list;
}

■ init():
  ➔ Set up internal state
  ➔ call init() on child – often a file open

■ next():
  ➔ call next() on child until qualifying tuple found or EOF
  ➔ keep only those fields in “proj_list”
  ➔ return tuple (or EOF -- “End of File” -- if no tuples remain)

■ close():
  ➔ call close() on child
  ➔ clean up internal state

Note: Scan also applies “selection” filters and “projections”
(without duplicate elimination)
Example: Sort

class Sort extends iterator {
    void init();
    tuple next();
    void close();
    iterator inputs[1];
    int numberOfRuns;
    DiskBlock runs[];
    RID nextRID[];
}

- **init():**
  - generate the sorted runs on disk
  - Allocate runs[] array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to NULLs

- **next():**
  - nextRID array tells us where we’re “up to” in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- “End of File” -- if no tuples remain)

- **close():**
  - deallocate the runs and nextRID arrays
Streaming through RAM

Simple case: “Map”. (assume many records per disk page)
- Goal: Compute \( f(x) \) for each record, write out the result
- Challenge: minimize RAM, call read/write rarely

Approach
- Read a chunk from INPUT to an Input Buffer
- Write \( f(x) \) for each item to an Output Buffer
- When Input Buffer is consumed, read another chunk
- When Output Buffer fills, write it to OUTPUT

Reads and Writes are not coordinated (i.e., not in lockstep)
- E.g., if \( f() \) is Compress(), you read many chunks per write.
- E.g., if \( f() \) is DeCompress(), you write many chunks per read.
Rendezvous

- Streaming: one chunk at a time. Easy.
- But some algorithms need certain items to be co-resident in memory
  - not guaranteed to appear in the same input chunk

- *Time-space Rendezvous*
  - in the same place (RAM) at the same time

- There may be many combos of such items
- Out-of-core algorithms orchestrate rendezvous.

- Typical RAM Allocation:
  - Assume B pages worth of RAM available
  - Use 1 page of RAM to read into
  - Use 1 page of RAM to write into
  - B-2 pages of RAM as workspace
Phase 1

“streamwise” divide into $N/(B-2)$ megachunks

output (write) to disk one megachunk at a time
Phase 2

Now megachunks will be the input
process each *megachunk* individually.
**Better: Double Buffering**

- Main thread runs $f(x)$ on one pair of I/O buffers
- 2nd “I/O thread” fills/drains simultaneously fills and drains the other pair of I/O bufs
- Main thread ready for a new buf? All set!
- Why better??????

**Usable in any of the following discussion**

- Assuming you have RAM buffers to spare!
- For simplicity we won’t use this trick in our examples
Sometimes you need order

- Rendezvous
  - Eliminating duplicates
  - Summarizing groups of items

- Sorted Data
  - Output must be ordered
  - E.g., return results in decreasing order of relevance

- Upcoming fundamentals:
  - *Sort-merge join* algorithm involves sorting (rendezvous)
  - First step in bulk loading *B+ tree index* (ordering)

- Problem: sort 100GB of data with 1GB of RAM.
  - Why not virtual memory?
• **Pass 0:**
  - read a page, sort it, write it.
  - only one buffer page is used
  - a repeated “batch job”
Pass 1, 2, 3, …, etc. (merge):

- Requires **3 buffer pages**
  - Note: this has nothing to do with double buffering!
- *Merge pairs* of runs into runs **twice as long**
- A streaming algorithm, as in the previous slide!
**Two-Way External Merge Sort**

- **Sort subfiles and Merge**
- How many passes?
- N pages in the file
  => the number of passes = \[ \lceil \log_2 N \rceil + 1 \]

- Total I/O cost? (reads + writes)
- Each pass we read + write each page in file. So total cost is:

\[
2N \left( \lceil \log_2 N \rceil +1 \right)
\]
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 $\left\lfloor \frac{N}{B} \right\rfloor$ sorted runs of $B$ pages each.
General External Merge Sort

Pass 1, 2, …, etc.: merge B-1 runs.

Creates runs of (B-1) * size of runs from previous pass.
Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \rceil$
- Cost = $2N \times (\# \text{ of passes})$
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil \frac{108}{5} \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil \frac{22}{4} \rceil = 6$ 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

Formula check: $1 + \lceil \log_4 22 \rceil = 1 + 3 \rightarrow 4 \text{ passes} \quad \checkmark$
# of Passes of External Sort

(I/O cost is 2N times number of passes)

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
How big of a table can we sort in two passes?

- Each “sorted run” after Phase 0 is of size $B$.
- Can merge up to $B-1$ sorted runs in Phase 1.

Answer: $B(B-1)$.

Sort $N$ pages of data in about $\sqrt{N}$ space.
Alternative: Hashing

■ Idea:
   🔵 Many times we don’t require order
   🔵 E.g.: removing duplicates
   🔵 E.g.: forming groups

■ Often just need to *rendezvous* matches

■ Hashing does this
   🔵 And may be cheaper than sorting! (Hmmm…!)
   🔵 But how to do it out-of-core??
Streaming Partition (divide):
Use a hash f’n $h_p$ to stream records to disk partitions

- All matches rendezvous in the same partition.
- *Streaming* alg to create partitions on disk:
  - “Spill” partitions to disk via output buffers
Divide & Conquer

- Streaming Partition (divide):
  Use a hash function $h_p$ to stream records to disk-based partitions
    - All matches rendezvous in the same partition.
    - *Streaming* alg to create partitions on disk:
      - “Spill” partitions to disk via output buffers

- ReHash (conquer):
  Read partitions into RAM-based hash table one at a time, using hash function $h_r$
    - Then go through each bucket of this hash table to achieve rendezvous in RAM

- Note: Two different hash functions
  - $h_p$ is coarser-grained than $h_r$
Two Phases

Partition:

Original Relation

Disk

B main memory buffers

OUTPUT

INPUT

hash function $h_p$

Partitions

1

2

B-1

Disk
Two Phases

Partition:

Rehash:
Cost of External Hashing

\[ \text{cost} = 4 \times N \text{ IO's} \]
1.30 Memory Requirement

■ How big of a table can we hash in two passes?
  ✤ B-1 “partitions” result from Phase 0
  ✤ Each should be no more than B pages in size
  ✤ Answer: B(B-1).

  ▪ We can hash a table of size N pages in about $\sqrt{N}$ space
  ✤ Note: assumes hash function distributes records evenly!

■ Have a bigger table? Recursive partitioning!
How does this compare with external sorting?
So which is better??

- Simplest analysis:
  - Same memory requirement for 2 passes
  - Same I/O cost
  - But we can dig a bit deeper…

- Sorting pros:
  - Great if input already sorted (or almost sorted) w/heapsort
  - Great if need output to be sorted anyway
  - Not sensitive to “data skew” or “bad” hash functions

- Hashing pros:
  - For duplicate elimination, scales with # of values
    - Not # of items! We’ll see this again.
  - Can exploit extra memory to reduce # IOs (stay tuned…)
Unordered collection model

Read in chunks to avoid fixed I/O costs

Patterns for Big Data
- Streaming
- Divide & Conquer
- also Parallelism (but we didn’t cover this here)
Summary Part 2

- **Sort/Hash Duality**
  - Sorting is Conquer & Merge
  - Hashing is Divide & Conquer

- **Sorting is overkill for rendezvous**
  - But sometimes a win anyhow

- **Sorting sensitive to internal sort alg**
  - Quicksort vs. HeapSort
  - In practice, QuickSort tends to be used

- **Don’t forget double buffering (with threads)**