Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 – DB Applications

Lecture #13: Query Evaluation
(R&G ch. 12 and 14)

Cost-based Query Sub-System

Queries

query

Catalog Manager

Schema

Statistics

Plan Generator

Plan Cost Estimator

Query Optimizer

Query Plan Evaluator

Outline

• (12.1) Catalog
• (12.2) Intro to Operator Evaluation
• (12.3) Algo’s for Relational Operations
• (12.6) Typical Q-optimizer
• (14.3.2) Hashing
Cost-based Query Sub-System

Queries

- Query Parser
- Query Optimizer
- Plan Generator
- Plan Cost Estimator
- Catalog Manager
- Query Plan Evaluator
- Schema
- Statistics

Schema

- What would you store?
- How?

Schema

- What would you store?
- A: info about tables, attributes, indices, users
- How?
- A: in tables! eg.,
  - Attribute_Cat (attr_name: string, rel_name: string, type: string, position: integer)
Statistics
• Why do we need them?
• What would you store?
  - NTuples(R): # records for table R
  - NPages(R): # pages for R
  - NKeys(I): # distinct key values for index I
  - INPages(I): # pages for index I
  - IHeight(I): # levels for I
  - ILow(I), IHigh(I): range of values for I

Outline
• (12.1) Catalog
• (12.2) Intro to Operator Evaluation
• (12.3) Algo’s for Relational Operations
• (12.6) Typical Q-optimizer
• (14.3.2) Hashing
Operator evaluation

3 methods we’ll see often:

• indexing
• iteration (= seq. scanning)
• partitioning (sorting and hashing)

``Access Path’’

• Eg., index (tree, or hash), or scanning
• Selectivity of an access path:
  – % of pages we retrieve
• eg., selectivity of a hash index, on range query: 100% (no reduction!)
Outline

- (12.1) Catalog
- (12.2) Intro to Operator Evaluation
- (12.3) Algo’s for Relational Operations
- (12.6) Typical Q-optimizer
- (14.3.2) Hashing

Algorithms

- selection:
- projection
- join
- group by
- order by

Algorithms

- selection: scan; index
- projection (dup. elim.):
- join
- group by
- order by
Algorithms

• selection: scan; index
• projection (dup. elim.): hashing; sorting
• join
• group by
• order by

Algorithms

• selection: scan; index
• projection (dup. elim.): hashing; sorting
• join: many ways (loops, sort-merge, etc)
• group by
• order by

Algorithms

• selection: scan; index
• projection (dup. elim.): hashing; sorting
• join: many ways (loops, sort-merge, etc)
• group by: hashing; sorting
• order by: sorting
Iterator Interface

SELECT DISTINCT name, gpa
FROM Students

Iterators

- Relational operators: subclasses of iterator:
  class iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[];
    // additional state goes here
  }
- iterators can be cascaded

Outline

- (12.1) Catalog
- (12.2) Intro to Operator Evaluation
- (12.3) Algo’s for Relational Operations
- (12.6) Typical Q-optimizer
- (14.3.2) Hashing
Q-opt steps

- bring query in internal form (e.g., parse tree)
- … into ‘canonical form’ (syntactic q-opt)
- generate alt. plans
- estimate cost; pick best

Q-opt - example

```sql
SELECT name
FROM STUDENT, TAKES
WHERE c-id = '415' AND
      STUDENT.ssn = TAKES.ssn
```

Q-opt - example

Canonical form
Q-opt - example

Outline

- (12.1) Catalog
- (12.2) Intro to Operator Evaluation
- (12.3) Algo’s for Relational Operations
- (12.6) Typical Q-optimizer
- (14.3.2) Hashing

Grouping; Duplicate Elimination

select distinct ssn
from TAKES

- (Q1: what does it do, in English?)
- Q2: how to execute it?
An Alternative to Sorting: Hashing!

- Idea:
  - maybe we don’t need the order of the sorted data
  - e.g.: forming groups in GROUP BY
  - e.g.: removing duplicates in DISTINCT
- Hashing does this!
  - And may be cheaper than sorting! (why?)
  - But what if table doesn’t fit in memory??

General Idea

- Two phases:
  - Phase1: Partition: use a hash function \( h \) 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

Two Phases

- Partition:
General Idea

- Two phases:
  - Phase 2: ReHash: for each partition on disk
    - (assuming it fits in memory)
    - 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

- Rehash:

<table>
<thead>
<tr>
<th>Partitions</th>
<th>Hash table for partition $R_i$ (size B pages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i$</td>
<td></td>
</tr>
<tr>
<td>Disk</td>
<td></td>
</tr>
<tr>
<td>hash to $R_i$</td>
<td></td>
</tr>
</tbody>
</table>

  B main memory buffers

  - $B - 1$ "spill partitions" in Phase 1
  - Each should be no more than $B$ blocks big

Analysis

- How big of a table can we hash using this approach?
  - $B - 1$ "spill partitions" in Phase 1
  - Each should be no more than $B$ blocks big
Analysis

• How big of a table can we hash using this approach?
  – B-1 "spill partitions" in Phase 1
  – Each should be no more than B blocks big
  – Answer: B(B-1).
    • i.e., a table of N blocks needs about sqrt(N) buffers
    – What assumption do we make?

  – B ~ sqrt(f * N)

Analysis

• How big of a table can we hash using this approach?
  – B-1 "spill partitions" in Phase 1
  – Each should be no more than B blocks big
  – Answer: B(B-1).
    • i.e., a table of N blocks needs about sqrt(N) buffers
    – Note: assumes hash function distributes records evenly!
    • use a 'fudge factor' f>1 for that: we need
      B ~ sqrt(f * N)

Analysis

• Have a bigger table? Recursive partitioning!
  – In the ReHash phase, if a partition b is bigger than B, 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
Recursive partitioning

Real story

- Partition + Rehash
- Performance is very slow!
- What could have gone wrong?

Real story

- Partition + Rehash
- Performance is very slow!
- What could have gone wrong?
- Hint: some buckets are empty; some others are way over-full.
Hashing vs. Sorting

• Which one needs more buffers?

Recall: can hash a table of size $N$ blocks in $\sqrt{N}$ space

• How big of a table can we sort in 2 passes?
  – Get $N/B$ sorted runs after Pass 0
  – Can merge all runs in Pass 1 if $N/B \leq B-1$
    • Thus, we (roughly) require: $N \leq B^2$
    • We can sort a table of size $N$ blocks in about space $\sqrt{N}$
    – Same as hashing!

Choice of sorting vs. hashing is subtle and depends on optimizations done in each case …
  – Already discussed some optimizations for sorting:
Hashing vs. Sorting

- Choice of sorting vs. hashing is subtle and depends on optimizations done in each case…
  - Already discussed some optimizations for sorting:
    - (Heapsort in Pass 0 for longer runs)
    - Chunk I/O into large blocks to amortize seek+RD costs
    - Double-buffering to overlap CPU and I/O
  - Another optimization when using sorting for aggregation:
    - "Early aggregation" of records in sorted runs
  - We will discuss some optimizations for hashing next…

Hashing: We Can Do Better!

- Combine the summarization into the hashing process - How?
  - During the ReHash phase, don’t store tuples, store pairs of the form <GroupVals, RunningVals>
  - When we want to insert a new tuple into the hash table
    - If we find a matching GroupVals, just update the RunningVals appropriately
    - Else insert a new <GroupVals, RunningVals> pair

```sql
select ssn, sum(credits) as groupVal, runningVal
from takes
group by ssn
(12345, 12)
(45678, 18)
```
Hashing: We Can Do Better!

- Combine the summarization into the hashing process
- What’s the benefit?
  - Q: How many entries will we have to handle?
  - A: Number of distinct values of GroupVals columns
    - Not the number of tuples!!
  - Also probably “narrower” than the tuples

Even Better: Hybrid Hashing

- What if $B > \sqrt{N}$?
- e.g., $N=10,000$, $B=200$
- $B=100$ (actually, 101) would be enough for 2 passes
- How could we use the extra 100 buffers?

A: 1ph for first partition;
2 for all others
Even Better: Hybrid Hashing

- Idea: hybrid! … keep 1st partition in memory during phase 1!
  - Output its stuff at the end of Phase 1.

Even Better: Hybrid Hashing

- What if B=300? (and N=10,000, again)
  - i.e., 200 extra buffers?

Even Better: Hybrid Hashing

- What if B=300? (and N=10,000, again)
  - i.e., 200 extra buffers?
  - A: keep the first 2 partitions in main memory
Even Better: Hybrid Hashing

• What if \( B=150 \)? (and \( N=10,000 \), again)
• i.e., 50 extra buffers?

A: keep half of the first bucket in memory

Hybrid hashing

• can be used together with the summarization idea
So, hashing’s better … right?

- Any caveats?

- A1: sorting is better on non-uniform data
- A2: ... and when sorted output is required later.

Hashing vs. sorting:
- Commercial systems use either or both

Notes:
1. Based on analytical (not empirical) evaluation
2. Numbers for sort do not reflect heap sort optimization
3. Assumes even distribution of hash buckets

Summary

- Query processing architecture:
  - Query optimizer translates SQL to a query plan = graph of iterators
  - Query executor "interprets" the plan
- Hashing is a useful alternative to sorting for dup. elim / group-by
  - Both are valuable techniques for a DBMS