Carnegie Mellon Univ.  
Dept. of Computer Science  
15-415 - Database Applications

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

Cost-based Query Sub-System

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

Query Parser

Query Optimizer

Plan Generator

Plan Cost Estimator

Catalog Manager

Cost-based Query Sub-System

Queries

Select * From Blah B
Where B.blah = blah

Query Plan Evaluator

Schema

• What would you store?

• How?

• 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?

• A: To estimate cost of query plans

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
**Iterator Interface**

```sql
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

```
select name
from STUDENT, TAKES
where c-id='415' and
STUDENT.ssn=TAKES.ssn
```

 Canonical form

Q-opt - example

```
Canonical form
```

```
### Q-opt - example

```
\begin{tikzpicture}
  \node (STUDENT) at (0,0) {STUDENT};
  \node (TAKES) at (1,0) {TAKES};
  \path[->] (STUDENT) edge node {$\sigma$} (TAKES);
  \path[->] (STUDENT) edge node {Hash join; merge join; nested loops;} (TAKES);
  \path[->] (TAKES) edge node {\(\sigma\rightarrow\text{Index; seq scan}\)} (TAKES);
\end{tikzpicture}
```

### 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 \( h_p \) 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

Two Phases

- Partition:

  ![Diagram of two phases](image-url)
General Idea

- Two phases:
  - Phase 2: 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

- Rehash:

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?

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
    \( B \approx \sqrt{f \times N} \)

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
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?
Hashing vs. Sorting

- **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 $\sqrt{N}$ space
      - Same as hashing!

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

- 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
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?

- 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

---

### Hashing vs. Sorting revisited

<table>
<thead>
<tr>
<th>Group Size or Reduction Factor</th>
<th>I/O (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>600</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
</tr>
<tr>
<td>10</td>
<td>200</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

- Sorting without early aggregation
- Sorting with early aggregation
- Hashing with early aggregation
- Hashing with hybrid hashing

---

**Notes:**
1. based on analytical (not empirical) evaluation
2. numbers for sort do not reflect heapsort optimization
3. assumes even distribution of hash buckets


---

So, hashing’s better … right?

- Any caveats?
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

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
  - Both are valuable techniques for a DBMS