Today's Class

- Catalog (12.1)
- Intro to Operator Evaluation (12.2-3)
- Typical Query Optimizer (12.6)
- Projection: Sorting vs. Hashing (14.3.2)
Cost-based Query Sub-System

Catalog: Schema

• What would you store?
  – Info about tables, attributes, indices, users

• How?
  – In tables!
    Attribute_Cat (attr_name: string, rel_name: string, type: string, position: integer)

See INFORMATION_SCHEMA discussion from Lecture #7
Catalog: Statistics

• Why do we need them?
  – 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
Query Plan Example

```sql
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

Relational Algebra:

\[ \pi_{\text{cname, amt}} (\sigma_{\text{amt}>1000} (\text{customer} \bowtie \text{account})) \]

---

Query Plan Example

```sql
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

CUSTOMER ACCOUNT

---

Query Plan Example

```sql
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

CUSTOMER ACCOUNT

---

Query Plan Example

```sql
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

CUSTOMER ACCOUNT

---

File Scan

Nested Loop

"On-the-fly"
Query Plan Example

```sql
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
AND account.amt > 1000
```

The output of each operator is the input to the next operator. Each operator iterates over its input and performs some task.

Operator Evaluation

- Several algorithms are available for different relational operators.
- Each has its own performance trade-offs.
- The goal of the query optimizer is to choose the one that has the lowest “cost”.

Next Class: How the DBMS finds the best plan.

Operator Execution Strategies

- Indexing
- Iteration (= seq. scanning)
- Partitioning (sorting and hashing)
Access Paths

• How the DBMS retrieves tuples from a table for a query plan.
  – File Scan (aka Sequential Scan)
  – Index Scan (Tree, Hash, List, …)

• Selectivity of an access path:
  – % of pages we retrieve
  – e.g., Selectivity of a hash index, on range query: 100% (no reduction!)

Operator Algorithms

• Selection:
• Projection:
• Join:
• Group By:
• Order By:
Operator Algorithms

- **Selection**: file scan; index scan
- **Projection**: hashing; sorting
- **Join**: many ways (loops, sort-merge, etc)
- **Group By**: hashing; sorting
- **Order By**: sorting
Today's Class

• Catalog (12.1)
• Intro to Operator Evaluation (12.2-3)
• Typical Query Optimizer (12.6)
• Projection: Sorting vs. Hashing (14.3.2)

Query Optimization

• Bring query in internal form (eg., parse tree)
• … into “canonical form” (syntactic q-opt)
• Generate alternative plans.
• Estimate cost for each plan.
• Pick the best one.

Query Plan Example

```
SELECT cname, amt
FROM customer, account
WHERE customer.acctno = account.acctno
  AND account.amt > 1000
```
**Query Plan Example**

```
SELECT cname, amt 
FROM customer, account 
WHERE customer.acctno = account.acctno 
AND account.amt > 1000
```

**Today's Class**

- Catalog (12.1)
- Intro to Operator Evaluation (12.2,3)
- Typical Query Optimizer (12.6)
  - Projection: Sorting vs. Hashing (14.3.2)

**Duplicate Elimination**

```
SELECT DISTINCT bname 
FROM account 
WHERE amt > 1000
```

- What does it do, in English?
- How to execute it?
  
  \[ \pi_{DISTINCT} (\sigma_{amt>1000}(account)) \]
Duplicate Elimination

```
SELECT DISTINCT bname
FROM account
WHERE amt > 1000
```

Two Choices:
- Sorting
- Hashing

---

Sorting Projection

```
ACCOUNT

DISTINCT bname
amt>1000

<table>
<thead>
<tr>
<th>acctno</th>
<th>bname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-123</td>
<td>Redwood</td>
<td>1800</td>
</tr>
<tr>
<td>A-789</td>
<td>Downtown</td>
<td>2000</td>
</tr>
<tr>
<td>A-123</td>
<td>Perry</td>
<td>1500</td>
</tr>
<tr>
<td>A-456</td>
<td>Downtown</td>
<td>1300</td>
</tr>
</tbody>
</table>
```

---

Alternative to Sorting: Hashing!

- What if we don’t need the order of the sorted data?
  - Forming groups in `GROUP BY`
  - Removing duplicates in `DISTINCT`
- Hashing does this!
  - And may be cheaper than sorting! (why?)
  - But what if table doesn’t fit in memory?
Hashing Projection

- Populate an ephemeral hash table as we iterate over a table.
- For each record, check whether there is already an entry in the hash table:
  - `DISTINCT`: Discard duplicate.
  - `GROUP BY`: Perform aggregate computation.
- Two phase approach.

Phase 1: Partition

- Use a hash function \( h_1 \) 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.
- Assume that we have \( B \) buffers.
Phase 2: ReHash

• For each partition on disk:
  – Read it into memory and build an in-memory hash table based on a hash function $h_2$
  – Then go through each bucket of this hash table to bring together matching tuples

• This assumes that each partition fits in memory.

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).
    • A table of N blocks needs about sqrt(N) buffers
  – What assumption do we make?

• A table of N blocks needs about \sqrt{N} buffers
  – Assumes hash distributes records evenly!

• Use a “fudge factor” f > 1 for that: we need
  \[ B = \sqrt{f \cdot N} \]

Analysis

• Have a bigger table? Recursive partitioning!
  – In the ReHash phase, if a partition i is bigger than B, then recurse.
  – Pretend that i is a table we need to hash, run the Partitioning phase on i, and then the ReHash phase on each of its (sub)partitions
Recursive Partitioning

Hash the overflowing bucket again

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: We 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 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
Hashing vs. Sorting

• Choice of sorting vs. hashing is subtle and depends on optimizations done in each case
• Another optimization when using sorting for aggregation:
  – “Early aggregation” of records in sorted runs
• Let’s look at some optimizations for hashing next…

Hashing: We Can Do Better!

• Combine the summarization into the hashing process - How?

Hashing: We Can Do Better!

• During the ReHash phase, store pairs of the form <GroupKey, RunningVal>
• When we want to insert a new tuple into the hash table:
  – If we find a matching GroupKey, just update the RunningVal appropriately
  – Else insert a new <GroupKey, RunningVal>
SELECT acctno, SUM(amt) FROM account GROUP BY acctno

- What’s the benefit?
- How many entries will we have to handle?
  - Number of distinct values of GroupKeys columns
  - Not the number of tuples!!
  - Also probably “narrower” than the tuples

So, hashing is better…right?

- Any caveats?
So, hashing is 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 for duplicate elimination / group-by
  - Both are valuable techniques for a DBMS