Unary Query Processing Operators

Not in the Textbook!

A “Slice” Through Query Processing

- We’ll study single-table queries today
  - SQL details
  - Query Executor Architecture
  - Simple Query “Optimization”

Basic Single-Table Queries

- SELECT [DISTINCT] <column expression list>
  FROM <single table>
  [WHERE <predicate>]
  [GROUP BY <column list>]
  [HAVING <predicate>]
  [ORDER BY <column list>]

• Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs

- SELECT DISTINCT <column expression list>
  FROM <single table>
  [WHERE <predicate>]
  [GROUP BY <column list>]
  [HAVING <predicate>]
  [ORDER BY <column list>]

• DISTINCT flag specifies removal of duplicates before output
ORDER BY

- SELECT DISTINCT S.name, S.gpa, S.age*2 AS a2
  FROM Students S
  WHERE S.dept = 'CS'
  [GROUP BY <column list>
  [HAVING <predicate>]
  ORDER BY S.gpa, S.name, a2;

- ORDER BY clause specifies that output should be sorted
  – Lexicographic ordering again!
- Obviously must refer to columns in the output
  – Note the AS clause for naming output columns!

Aggregates

- SELECT [DISTINCT] AVERAGE(S.gpa)
  FROM Students S
  WHERE S.dept = 'CS'
  [GROUP BY <column list>
  [HAVING <predicate>]
  [ORDER BY <column list>]
  • Before producing output, compute a summary (a.k.a. an aggregate) of some arithmetic expression
  • Produces 1 row of output
  – with one column in this case
  • Other aggregates: SUM, COUNT, MAX, MIN
  • Note: can use DISTINCT inside the agg function
  – SELECT COUNT(DISTINCT S.name) FROM Students S
  – vs. SELECT DISTINCT COUNT (S.name) FROM Students S;

HAVING

- SELECT [DISTINCT] AVERAGE(S.gpa), S.dept
  FROM Students S
  [WHERE <predicate>]
  [GROUP BY S.dept]
  [HAVING <predicate>]
  [ORDER BY <column list>]
  • The HAVING predicate is applied after grouping and aggregation
  – Hence it can contain anything that could go in the SELECT list
  – i.e. aggs or GROUP BY columns
  • HAVING can only be used in aggregate queries
  • It’s an optional clause

GROUP BY

- SELECT [DISTINCT] AVERAGE(S.gpa), S.dept
  FROM Students S
  [WHERE <predicate>]
  [GROUP BY S.dept]
  [HAVING <predicate>]
  [ORDER BY <column list>]
  • Partition the table into groups that have the same value on GROUP BY columns
  – Can group by a list of columns
  • Produce an aggregate result per group
  – Cardinality of output = # of distinct group values
  • Note: can put grouping columns in SELECT list
  – For aggregate queries, SELECT list can contain aggs and GROUP BY columns
  – What would it mean if we said SELECT S.name, AVERAGE(S.gpa) above?

Putting it all together

- SELECT S.dept, AVERAGE(S.gpa), COUNT(*)
  FROM Students S
  WHERE S.gender = 'F'
  [GROUP BY S.dept]
  [HAVING COUNT(*) > 5]
  [ORDER BY S.dept];
Postgres Version

- src/backend/executor/nodeSort.c
  - ExecInitSort (init)
  - ExecSort (next)
  - ExecEndSort (close)
- The encapsulation stuff is hardwired into the Postgres C code
  - Postgres predates even C++!
  - See src/backend/execProcNode.c for the code that "dispatches the methods" explicitly!

Iterators

- The relational operators are all subclasses of the class iterator:
  ```
  class iterator {
    void init();
    tuple next();
    iterator &inputs[];
    // additional state goes here
  }
  ```
- Notes:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information

Context

- We looked at SQL
- Now shift gears and look at Query Processing

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

Example: Sort

```
class Sort extends iterator {
  void init();
tuple next();
void close();
iterator &inputs[];
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 Fun" -- if no tuples remain)
- close():
  - deallocate the runs and nextRID arrays

Sort GROUP BY: Naive 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
  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??

### Two Phases

**General Idea**

- **Two phases:**
  - **Partition:** use a hash function \( h_i \) 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 \( h_i \)
    - Then go through each bucket of this hash table to bring together matching tuples

### Analysis

- **How big of a table can we hash in one pass?**
  - B-1 “spill partitions” in Phase 1
  - Each should be no more than B blocks big
  - Answer: B(B-1).
    - Said differently: We can hash a table of size \( N \) blocks in about \( N/B \) steps.
    - Much like sorting!
- **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

---

### Hash GROUP BY: Naive 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 \(<\text{GroupVals}, \text{TransVals}>\)
  - 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 \(<\text{GroupVals}, \text{TransVals}>\) 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?**
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?

![Diagram of hybrid hashing]

A Hash Function for Hybrid Hashing

- Assume we like the hash-partition function h_p
- Define h, operationally as follows:
  - h(x) = 1 if in-memory hashtable is not yet full
  - h(x) = 1 if x is already in the hashtable
  - h(x) = h_p(x) otherwise
- This ensures that:
  - Bucket 1 fits in k pages of memory
  - If the entire set of distinct hashtable entries is smaller than k, we do no spilling!

![Diagram of hybrid hashing]

Context

- We looked at SQL
- We looked at Query Execution
  - Query plans & Iterators
  - A specific example
- How do we map from SQL to query plans?

Query Optimization

- 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!
  - Note: Where did SELECT clause go?
    - Implicit!

Summary

- Single-table SQL, in detail
- Exposure to query processing architecture
  - Query optimizer translates SQL to a query plan
  - Query executor "interprets" the plan
    - Query plans are graphs of iterators
- Hashing is a useful alternative to sorting
  - For many but not all purposes