Query Processing: A Systems View

CompSci 316
Introduction to Database Systems

Announcements (Tue. Nov. 20)
- Project milestone #2 feedback will be emailed to you by Thursday
- Homework #4 due in 2 weeks

A query’s trip through the DBMS

SQL query → Parser → Parse tree
Parser: SQL → parse tree
- Good old lex & yacc will do
- Detect and reject syntax errors

Validator: parse tree → logical plan
- Detect and reject semantic errors
  - Nonexistent tables/views/columns?
  - Insufficient access privileges?
  - Type mismatches?
    - Examples: AVG(name), name + GPA, Student UNION Enroll
- Also
  - Expand *
  - Expand view definitions
- Information required for semantic checking is found in system catalog (contains all schema information)

Logical plan
- Nodes are logical operators (often relational algebra operators)
- There are many equivalent logical plans

Physical (execution) plan
- A complex query may involve multiple tables and various query processing algorithms
  - E.g., table scan, index nested-loop join, sort-merge join, hash-based duplicate elimination…
- A physical plan for a query tells the DBMS query processor how to execute the query
  - A tree of physical plan operators
  - Each operator implements a query processing algorithm
  - Each operator accepts a number of input tables/streams and produces a single output table/stream

Parsing and validation
- Parser: SQL → parse tree
- Validator: parse tree → logical plan
- Logical plan → Optimizer → Physical plan
- Physical plan → Executor → Result
Examples of physical plans

```
SELECT Course.title
FROM Student, Enroll, Course
WHERE Student.name = 'Bart'
AND Student.SID = Enroll.SID
AND Enroll.CID = Course.CID;
```

- **Project**: (title)
- **Index-Nested-Loop-Join**: (CID)
  - Index on Enroll(SID)
- **Index-Nested-Loop-Join**: (SID)
  - Index on Student(name)
- **Index-Scan** (name = "Bart")
- **Index-Scan** (Enroll)
- **Index-Scan** (Course)
- **Sort** (CID)
- **Merge-Join** (SID)
- **Scan** (Enroll)
- **Scan** (Course)
- **Filter** (name = "Bart")

Many physical plans for a single query
- Equivalent results, but different costs and assumptions!
- DBMS query optimizer picks the “best” possible physical plan

Physical plan execution

- How are intermediate results passed from child operators to parent operators?
  - Temporary files
    - Compute the tree bottom-up
    - Children write intermediate results to temporary files
    - Parents read temporary files
  - Iterators
    - Do not materialize intermediate results
    - Children pipeline their results to parents

Iterator interface

- Every physical operator maintains its own execution state and implements the following methods:
  - **open()**: Initialize state and get ready for processing
  - **getNext()**: Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  - **close()**: Clean up

An iterator for table scan

- State: a block of memory for buffering input $R$; a pointer to a tuple within the block
- **open()**: allocate a block of memory
- **getNext()**
  - If no block of $R$ has been read yet, read the first block from the disk and return the first tuple in the block
  - Or the null pointer if $R$ is empty
  - If there is no more tuple left in the current block, read the next block of $R$ from the disk and return the first tuple in the block
    - Or the null pointer if there are no more blocks in $R$
  - Otherwise, return the next tuple in the memory block
- **close()**: deallocate the block of memory

An iterator for nested-loop join

- $R$: An iterator for the left subtree
- $S$: An iterator for the right subtree
- **open()**
- **getNext()**
  - If called for the first time
    - Call **getNext()** on child to fill all blocks, sort the tuples, and output a run
    - Repeat until **getNext()** on child returns null
    - Read one block from each run into memory, and initialize pointers to point to the beginning tuple of each block
    - Return the smallest tuple and advance the corresponding pointer; if a block is exhausted bring in the next block in the same run
- **close()**
  - Call **close()** on child
- Deallocate sorting memory and delete temporary runs

An iterator for 2-pass merge sort

- **open()**
  - Allocate a number of memory blocks for sorting
  - Call **open()** on child iterator
- **getNext()**
  - If called for the first time
    - Call **getNext()** on child to fill all blocks, sort the tuples, and output a run
    - Repeat until **getNext()** on child returns null
    - Read one block from each run into memory, and initialize pointers to point to the beginning tuple of each block
    - Return the smallest tuple and advance the corresponding pointer; if a block is exhausted bring in the next block in the same run
- **close()**
  - Call **close()** on child
  - Deallocate sorting memory and delete temporary runs
Blocking vs. non-blocking iterators

- A blocking iterator must call `getNext()` exhaustively (or nearly exhaustively) on its children before returning its first output tuple
  - Examples: sort, aggregation
- A non-blocking iterator expects to make only a few `getNext()` calls on its children before returning its first (or next) output tuple
  - Examples: dup-preserving projection, filter, merge join with sorted inputs

Execution of an iterator tree

- Call `root.open()`
- Call `root.getNext()` repeatedly until it returns null
- Call `root.close()`

- Requests go down the tree
- Intermediate result tuples go up the tree
- No intermediate files are needed
  - But maybe useful if an iterator is opened many times
    - Example: complex inner iterator tree in a nested-loop join; “cache” its result in an intermediate file