Announcements

- Reading assignments for this week
  - “Query Rewrite Optimization Rules in IBM DB2 Universal Database,” by Leung et al. (in red book)
- Homework #3 will be posted later tonight (March 24)
  - Due in 16 days (Wednesday, April 9)
- Recitation session this Friday (March 28)
  - Graded Homework #2 and common problems
  - Help on Homework #3

Physical (execution) plan

- A complex query may involve multiple tables and various query processing 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 tablesstreams and produces a single output tablestream

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-SCAN (name = "Bart")
INDEX-NESTED-LOOP-JOIN (SID)
INDEX-SCAN (Enroll)
INDEX-SCAN (Student)
FILTER (name = "Bart")
SORT (SID)
SORT (CID)
MERGE-JOIN (SID)
MERGE-JOIN (CID)
SCAN (Course)
SCAN (Enroll)
```

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

- **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()**
  - R.open(); S.open(); r = R.getNext();
- **getNext()**
  - do {
      s = S.getNext();
      if (s == null) {
        S.close(); S.open(); s = S.getNext(); if (s == null) return null;
        r = R.getNext(); if (r == null) return null;
      }
    } until (r joins with s);
  - return rs;
- **close()**
  - R.close(); S.close();

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: 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