Announcement (Thu., Nov. 18)

- Homework 4 due in 1½ week
  - Except Problem X2, which will be due in two weeks
- Weekly project progress update today (every Thu.)
- We are still wrapping up Homework 3 and Project Milestone 3 grading
- Mini-Hackathon this weekend!
  - Time: 10AM-3PM
  - Date: Saturday, Nov. 20
  - Location: North 232
  - We will bring drink & snacks

A query’s trip through the DBMS
Parsing and validation

- **Parser:** SQL → parse tree
  - 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 + pop, User UNION Member
  - Also
    - Expand *
    - Expand view definitions
- Information required for semantic checking is found in
  system catalog (which contains all schema information)

Logical plan

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

\[
\pi_{\text{Group.name}} \left( \sigma_{\text{Name} = \text{“Bart”}} \left( \text{User} \bowtie \text{Member} \right) \right) \bowtie \text{Group} = \text{Member}.\text{gid} \]

An equivalent plan:

\[
\pi_{\text{Group.name}} \left( \text{Member}.\text{gid} = \text{Group}.\text{gid} \right) \bowtie \text{Member} \left( \text{User}.\text{uid} = \text{Member}.\text{uid} \right) \left( \sigma_{\text{Name} = \text{“Bart”}} \right) \text{User} \]

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
Examples of physical plans

SELECT Group.name
FROM User, Member, Group
WHERE User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;

• 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 null 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 null 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()`:
  - while true:
    - if $R$ is empty:
      - null
    - else:
      - $r = R$.getNext()
    - if $S$ is empty:
      - null
    - else:
      - $s = S$.getNext()
    - if $r$ is null:
      - $R$.close()
      - null
    - if $s$ is null:
      - $S$.close()
      - null
    - else:
      - if `joins(r, s)`:
        - return
      - else:
        - `concat(r, s)`
- `close()`

Is this tuple-based or block-based nested-loop join?
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:

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
Iterators are showing their age...

While iterators are an elegant way of pipelining execution, their implementation tends to be inefficient on modern architectures

- Too many (virtual) function calls
- Poor data locality—in memory instead of CPU registers
- Fail to take advantage of
  - Compiler loop unrolling
  - CPU pipelining
  - SIMD (single instruction, multiple data)

Which one do you think runs faster?

```python
class NLJ:
    def open(self):
        R.open()
        S.open()
    def close(self):
        R.close()
        S.close()
    def __call__(self):
        while True:
            r = R.getNext()
            if r is None:
                return
            s = S.getNext()
            if s is None:
                return None
            if joins(r, s):
                return
            return concat(r, s)
```  

```python
class Aggr:
    def open(self):
        R.open()
    def close(self):
        R.close()
    def __call__(self):
        state = init()
        while True:
            r = R.getNext()
            if r is None:
                return finalize(state)
            state = accumulate(state, r)
```

Whole-stage “codegen”

- Given a physical plan, fuse operators together to generate query-specific code, with loops instead of iterator function calls
- Instead of “interpreting” the physical plan, give generated code to an optimizing compiler

☞ Functionality of a general-purpose execution engine; performance as if system is hand-built to run your specific query
- This approach has been adopted by newer systems, such as Spark