6.5830 / 6.5831 – Lab 2 Bootcamp

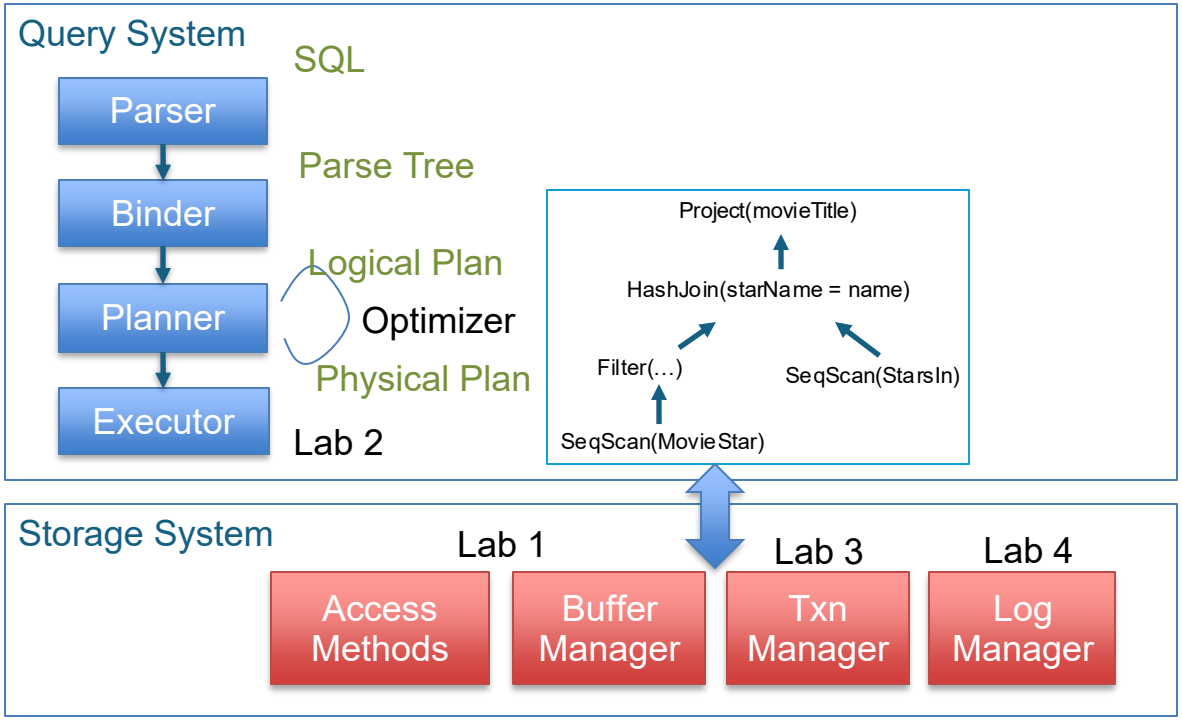
Query Execution

Navigating GoDB

- This may be your first time working with a large codebase
  - Check out all parts of the API, not everything you need will be mentioned in the handout
  - Ctrl/Cmd-click will jump to relevant of the API

- Files to look at:
  - execution/executor.go for operator interface
  - indexing/index.go
  - planner/[operator]_node.go

Query Execution: Overview



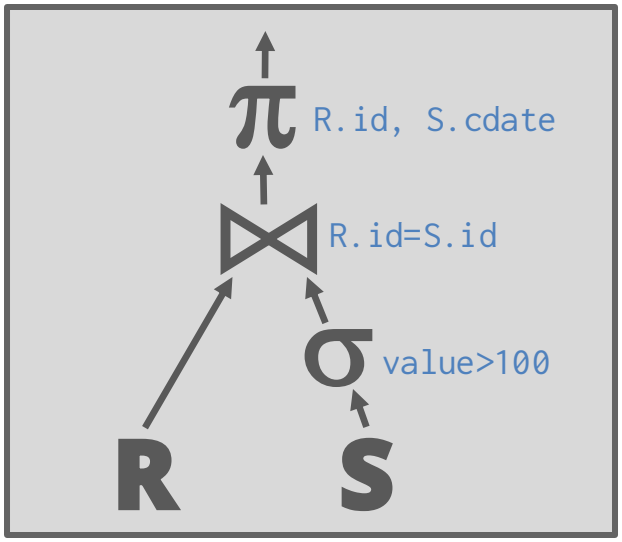
What are we building?

- Operators
  - Seq Scan, Filter, Join(s), Sort, etc.
  - Volcano-Style Iterator Model
- Indexes
  - Scan, Lookup, Insert/Delete/Update

Recap: Query/Execution Plan

- DAG of **operators**. The operators are arranged in a tree
- The iterator of each node uses the child iterators to compute output one tuple at a time
- The output of the root node is the result of the query

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

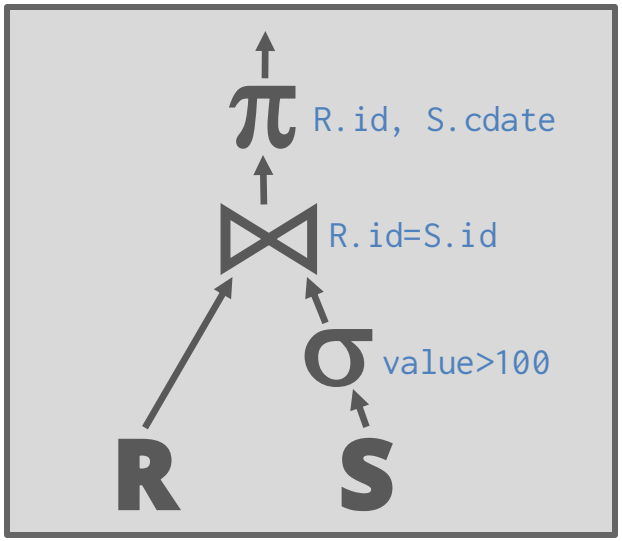


Volcano-Style Iterator Model

- TableHeapIterator sits at the storage layer

- Volcano-Style Iterator Model sits at the query execution layer
  - Used to connect query operators

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Part 1 – Basic Operators

- Sequential Scan
- Filter
- Projection
- Limit

Operator Interface

- **State:** Keep track of current position
- **PlanNode():** Return logical plan, contains schema and configuration
- **Init():** Allocate resources that should be released in Close() (different from constructor)
- **Next():** Advance to next valid element, return true if a next element exist, false if at end
- **Current():** Return the current element
- **Error():** Track errors
- **Close():** Release any resources held by the iterator

Plan Node vs. Executor

- **Plan Node:** WHAT
  - Logical description of query (configuration)
  - Forms the query plan tree

- **Executor:** HOW
  - Created using the plan nodes
  - Runs the query using the Volcano-style Iterator Model
  - Maintains runtime state

Example: Filter

```go
// NewFilter creates a new FilterExecutor executor.
func NewFilter(plan *planner.FilterNode, child Executor) *FilterExecutor {
    panic("unimplemented")
}
```

- Takes in plan and child — store these as fields
- Explore planner/filter_node.go: contains the predicate that you filter on

```go
// FilterNode filters tuples from its child based on a predicate.
type FilterNode struct {
    Child     PlanNode
    Predicate Expr
}
```

Part 2 — Indexes

- Index Access Methods (Scan + Lookup)
  - See indexing/index.go

- Modifications: must maintain index consistency
  - Insert, Delete, Update
  - Return tuple with number of modified records (SQL convention)

Recap: Index Access Methods

- Index Lookups (i.e., point queries)
  - Can be served from either index type
  - E.g., SELECT * FROM emp WHERE id = 1
- Index Scan (i.e., range queries)
  - Requires tree index
  - E.g., SELECT * FROM emp WHERE sal > 10k AND sal < 100k

Example: Insert

- Consumes tuples from the child and adds them to a TableHeap
  - Bugs might carry from Lab 1
- After inserting, remember to update indexes
  - Why? Indexes map key to RecordID (page ID, slot)
- Current() returns the number of tuples inserted so far

Building Output Tuples

- Get the tuple directly from the child iterator
- Build output tuples with storage.FromRawTuple or storage.FromValues
  - Maintain a reusable buffer for storing data

Part 3 – Core Processing Operators

- More operators!
  - Block Nested Loop Join
  - Sort
  - Aggregation

- See handout for NULL handling details

Example: Aggregator

- Materializes/blocks: consume all tuples from child before doing anything
- Sum, Count, Min, Max
  - NULL handled differently for different operations
- Hash Aggregation
  - Hash table already implemented for you in execution/hash_table.go
- The iterator should populate the hash map keyed by the GROUP BY field and iterate through the results

Part 4 — Advanced Operators

- Implement 2/5 more operators
  - Hash Join
  - Sort-Merge Join
  - Index Nested Loop Join
  - Top-N Optimization (heaps)
  - Materialization

Part 5 — GoDB Frontend

- Uncomment the rules you have implemented in physicalRules in main.go

- -explain flag to see what operators are being used
  - Can help with visualizing query plans

```
GoDB> SELECT c_first, c_last, o_id
FROM customer, orders
WHERE c_id = o_c_id AND c_w_id = o_w_id AND c_d_id = o_d_id
AND c_last = 'Smith';   ->    ->    ->
Initial Logical Plan:
LogicalProjection: customer.c_first AS c_first, customer.c_last AS c_last, orders.o_
id AS o_id
└─ LogicalFilter: ((((customer.c_id = orders.o_c_id) AND (customer.c_w_id = orders.o
_w_id)) AND (customer.c_d_id = orders.o_d_id)) AND (customer.c_last = 'Smith'))
   └─ LogicalJoin: CROSS JOIN
      ├─ LogicalScan: customer (OID: 5)
      └─ LogicalScan: orders (OID: 8)

Optimized Logical Plan:
LogicalProjection: customer.c_first AS c_first, customer.c_last AS c_last, orders.o_
id AS o_id
└─ LogicalJoin: INNER JOIN ON (customer.c_id = orders.o_c_id) AND (customer.c_w_id =
 orders.o_w_id) AND (customer.c_d_id = orders.o_d_id)
   ├─ LogicalScan: customer (OID: 5) | Filter: [(customer.c_last = 'Smith')]
   └─ LogicalScan: orders (OID: 8)

Physical Plan:
Projection: {c_first: offset 3 type string} |, {c_last: offset 4 type string} |, {o_
id: offset 7 type int} |
└─ IndexJoin: Table(8) via Index(10), outputSchema=[int int int string string string
 int int int int int int]
   └─ Filter: ({c_last: offset 4 type string} = 'Smith')
      └─ SeqScan: TableOID(5)


c_first  c_last   o_id
'Alice'  'Smith'  1
'Alice'  'Smith'  2
(2 rows) [83.208µs]
```

Query

Index Nested Loop Join

Questions?