

# Iterators and Relational Operators

Alvin Cheung  
Fall 2022

Reading: R&G, Chapters 4.1 - 4.2



# Announcements



- Nested loops join re-lecture
- Project 3 is out
- Exam accommodations
  - For health-related issues we will require a documentation
    - Dr's note / covid test / DSP letter
- Anonymous feedback
  - Posting lectures on youtube
  - Gap between assignments and lectures
- We need more break videos!

# Recall from Relational Algebra



SQL Query

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

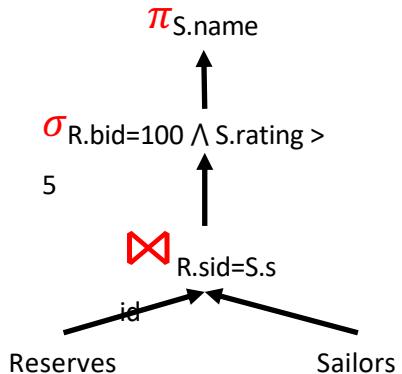
Query Parser  
& Optimizer

Relational Algebra

$$\pi_{S.name}(\sigma_{R.bid=100 \wedge S.rating>5}(Reserves \bowtie_{R.sid=S.sid} Sailors))$$

How are relational algebra operators implemented?

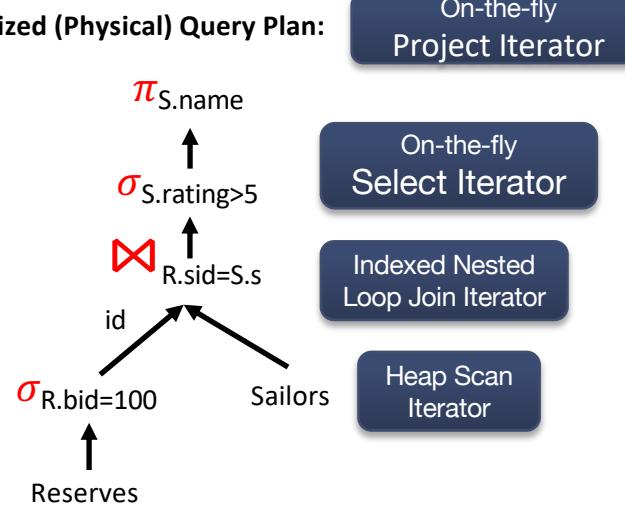
(Logical) Query Plan:



But actually will produce...

Operator Code  
B+-Tree  
Indexed Scan  
Iterator

Optimized (Physical) Query Plan:



# Relational Operators and Query Plans

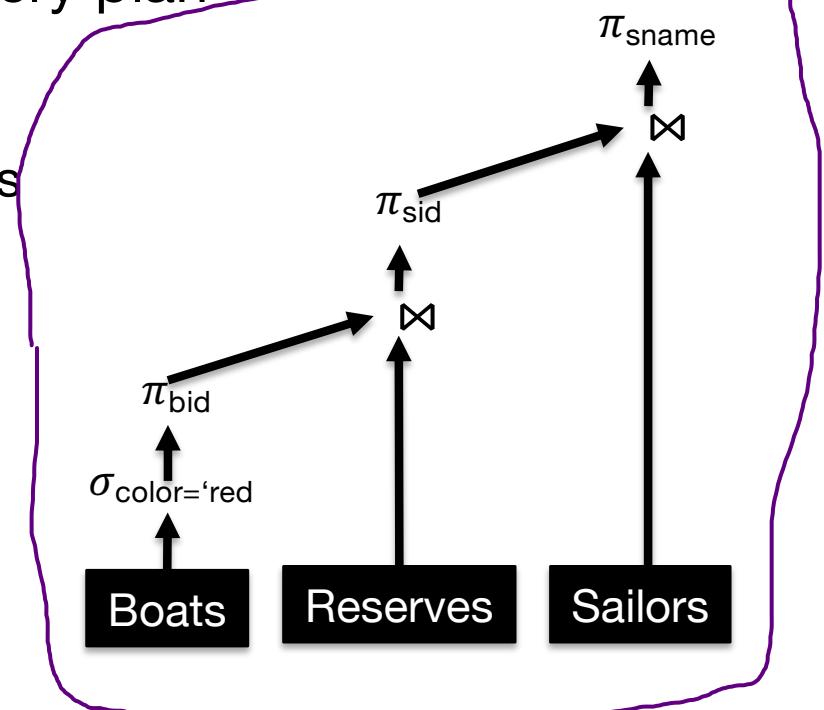

$$\pi_{\text{sname}}(\pi_{\text{sid}}(\pi_{\text{bid}}(\sigma_{\text{color}=\text{'red'}}(\text{Boats}) \bowtie \text{Reserves}) \bowtie \text{Sailors}))$$

- Expression Tree Representation = Query plan

- Edges encode “flow” of tuples
- Vertices = Relational Alg Operators
- Source vertices = table access operators

- Also called dataflow graph

- Here, “flow of tuples”
- Not specific to DBMSs
  - E.g., “big data systems”, ML systems

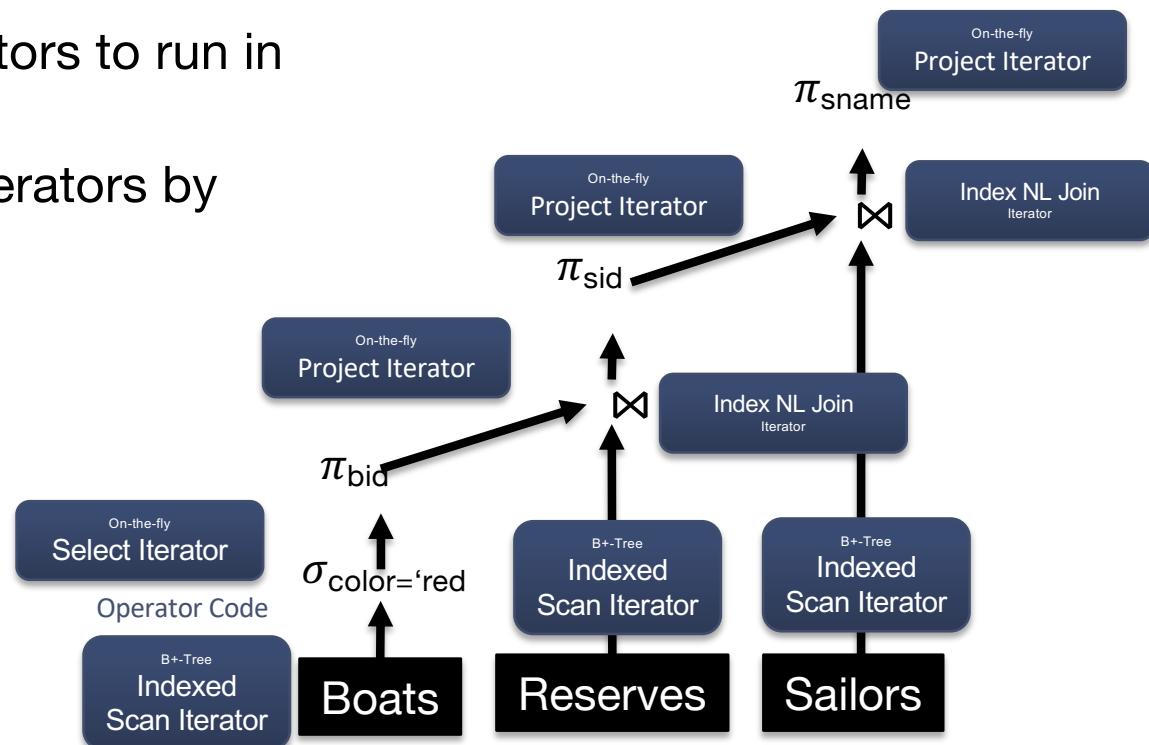


# Query Executor Instantiates Operators


$$\pi_{\text{sname}}(\pi_{\text{sid}}(\pi_{\text{bid}}(\sigma_{\text{color}=\text{'red'}}(\text{Boats})) \bowtie \text{Res}) \bowtie \text{Sailors})$$

- Query optimizer selects operators to run in sequence (i.e., the query plan)
- Query executor runs these operators by creating instances thereof
- Each operator instance:
  - Implements **iterator interface**
  - Efficiently executes operator logic forwarding tuples to next operator

$\pi, \sigma, \bowtie$



# Iterator Interface



The relational operators implemented as subclasses of the class Iterator:

```
abstract class iterator {
    void setup(List<Iterator> inputs); // Configure the input (children) args
    void init(args);                  // Invoked before calling next: sets up state
    tuple next();                    // Invoked repeatedly: return another tuple
    void close();                   // Invoked when finished
}
```

- **Pull-based computation model**
  - e.g., Console calls init on root operator of query plan, and then next
    - If tuple is not ready, this next request propagates down the query plan recursively
  - init/next can result in either *streaming* ("on-the-fly") or *blocking* ("batch") algorithm for that operator:
    - streaming: small, constant amount of work per call
    - blocking: does not produce output until it consumes its entire input, i.e., all rows from children!
    - Q: examples?
- Any iterator can be input to any other, since they all implement the same interface (composability)
- **State:** iterators may maintain substantial private "internal" state
  - e.g., hash tables, running counts, large sorted files ...

5  
"streaming"  
"

## Example: Select (on-the-fly)



- A streaming operator: small amount of work per tuple produced

- `init(predicate):`

```
child.init()  
pred = predicate; // local variable storing state  
current = NULL; // local cursor (current tuple)
```

- `next():`

```
while (current != EOF && !pred(current))  
    current = child.next(); // give us another tuple  
} // exit if pred is satisfied or EOF  
return current; // return current tuple or EOF
```

- `close():`

```
child.close()
```

Does not cost I/Os

5

## Example: Heap Scan

- Leaf of the query plan

- **init(relation):**

```
heap = open heap file for this relation;  
cur_page = heap.first_page();  
cur_slot = cur_page.first_slot();
```

// file handle  
// first page  
// first slot on that page

- **next():**

```
if (cur_page == NULL) return EOF;  
current = [cur_page, cur_slot];  
cur_slot = cur_slot.advance();  
if (cur_slot == NULL) {  
    cur_page = cur_page.advance();  
    if (cur_page != NULL)  
        cur_slot = cur_page.first_slot();  
}  
return current;
```

current record id

// we will return this recordId  
// advance the slot for subseq. calls  
// advance to next page, first slot

- **close():**

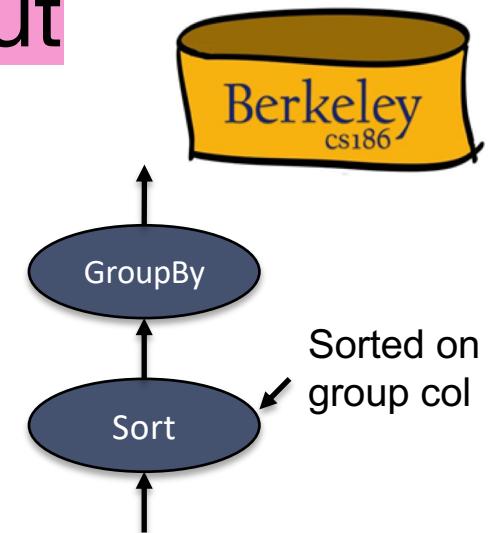
```
heap.close()
```

// close file



# Example: Group By on Sorted input

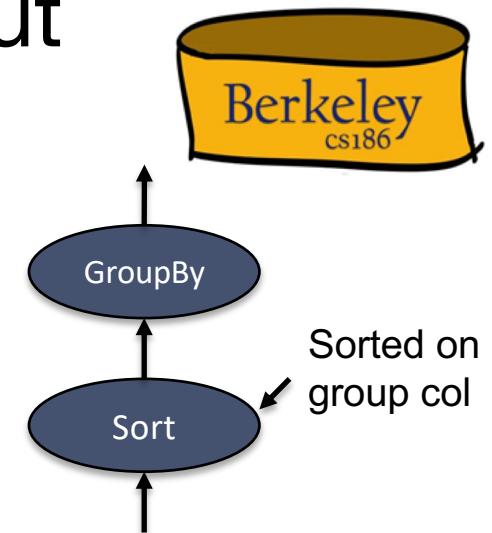
agg_type	state	init	merge(x)	final
COUNT	count	0	count ++	count
SUM	sum	0	sum += x	sum
AVG	?	?	?	?
MIN	?	?	?	?



- Say input is sorted, and we want to do a group by
  - Sort not necessary, can also do group by with hashing
- Keep “group” in iterator state, add merge(tuple) function
  - Initialize group state
  - Operate one tuple at a time in next, and merge tuple with existing group state
    - Create new group if needed
  - Return result tuple when done with group

# Example: Group By on Sorted input

agg_type	state	init	merge(x)	final
COUNT	count	0	count ++	count
SUM	sum	0	sum += x	sum
AVG	[count, sum]	[0, 0]	[count++, sum+=x]	sum / count
MIN	min	+infinity	min > x ? x : min	min



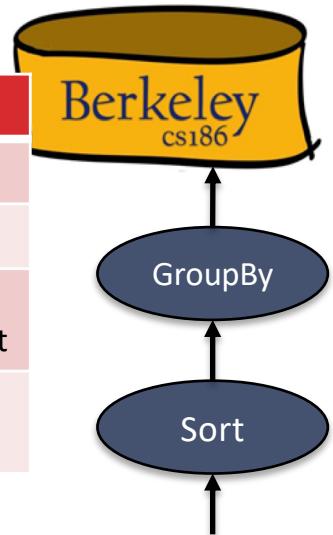
- Say input is sorted, and we want to do a group by
  - Sort not necessary, can also do group by with hashing
- Keep “group” in iterator state, add merge(tuple) function
  - Initialize group state
  - Operate one tuple at a time in next, and merge tuple with existing group state
    - Create new group if needed
  - Return result tuple when done with group

# Example: Group By on Sorted input

- `init(group_keys, aggs):`  
 `child.init();`  
 `cur_group = NULL; // no group so far`
- `next():`  
 `result = NULL;`  
 `do {`  
 `tup = child.next();`  
 `if (group(tup) != cur_group) {`  
 `if (cur_group != NULL)`  
 `result = [cur_group, final() of all aggs]`  
 `cur_group = group(tup);`  
 `call init() on group state`  
 `}`  
 `call merge(tup) to merge tup into state`  
 `} while (!result);`  
 `return result;`
- `close():`  
 `child.close()`

agg_type	state	init	merge(x)	final
COUNT	count	0	count ++	count
SUM	sum	0	sum += x	sum
AVG	[count, sum]	[0, 0]	[count++, sum+=x]	sum / count
MIN	min	+infinity	min > x ? x : min	min

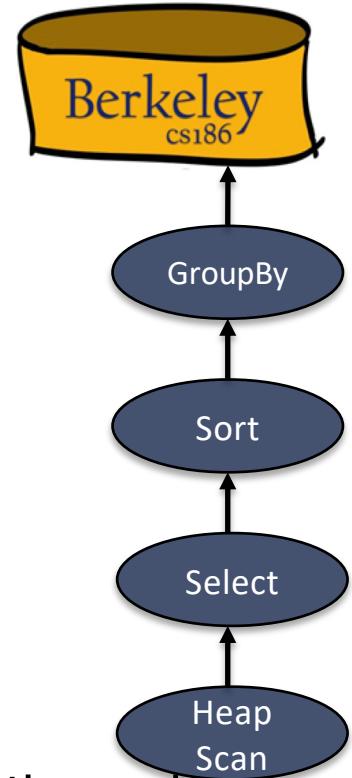
`// New group!`  
`// Have we seen a group previously?`  
`// Form result for that current group`  
`// Initialize new group state`  
  
`// Exit if current group result is formed`



Neat: only maintains one tuple of partial results in memory at any time!

# A Full (Single Table) Query Plan

- A Query Plan is Single-threaded!
- Exercise: trace the calls that lead to flow of tuples:
  - Call init() on the root GroupBy
    - How does init() recurse down the chain and return?
  - Call next() on root
    - How does next() recurse down the chain and return a tuple?
  - Note how the blocking operator (sort) interacts with the other, streaming operators
    - Select and GroupBy are essentially streaming operators
- We don't store each operator output on disk; tuples stream through the plan's call stack
  - Some operators like Sort use disk internally – but not exposed outside the operator



# Summary



- We just finished our tour of the upper layer of the DBMS
- We have now seen how SQL queries can be represented using relational algebra trees
- We have also seen how relational algebra operators are implemented
  - init / next / close
  - Streaming vs blocking
- We saw earlier the different algos for implementing various relational algebra operators
  - Sort, hash, join, etc
- We didn't cover how SQL text are translated into RA trees
  - Take CS164!