Query Execution – Part I
**ADMINISTRIVIA**

**Homework #3** is due Sun Oct 18\(^{th}\) @ 11:59pm

**Mid-Term Exam** is Wed Oct 21\(^{st}\)
→ Morning Session: 9:00am ET
→ Afternoon Session: 3:20pm ET

**Project #2** is due Sun Oct 25\(^{th}\) @ 11:59pm
PROJECTS

Write your own tests.
Practice **defensive programming**.
Profile your code to find performance problems.

Do **not** use Gradescope for debugging.
Do **not** directly email TAs for help.
The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

The output of the root node is the result of the query.
TODAY'S AGENDA

- Processing Models
- Access Methods
- Modification Queries
- Expression Evaluation
A DBMS's processing model defines how the system executes a query plan. → Different trade-offs for different workloads.

Approach #1: Iterator Model
Approach #2: Materialization Model
Approach #3: Vectorized / Batch Model
ITERATOR MODEL

Each query plan operator implements a **Next** function.
- On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples.
- The operator implements a loop that calls next on its children to retrieve their tuples and then process them.

Also called **Volcano** or **Pipeline** Model.
**ITERATOR MODEL**

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

- **Next() for R**
  ```
  for t in R:
    emit(t)
  ```

- **Next() for S**
  ```
  for t in S:
    emit(t)
  ```

- **Next() for child**
  ```
  for t in child.Next():
    emit(projection(t))
  ```

- **Next() for left**
  ```
  for t1 in left.Next():
    buildHashTable(t1)
    for t2 in right.Next():
      if probe(t2): emit(t1⨝t2)
  ```

- **Next() for right**
  ```
  for t2 in right.Next():
    if probe(t2):
      emit(t1⨝t2)
  ```

- **Next() for child**
  ```
  for t in child.Next():
    emit(t)
  ```

- **Next() for evalPred**
  ```
  for t in child.Next():
    if evalPred(t): emit(t)
  ```

- **Next() for probe**
  ```
  for t in child.Next():
    if evalPred(t): emit(t)
  ```
**SELECT** R.id, S.cdate
**FROM** R **JOIN** S
**ON** R.id = S.id
**WHERE** S.value > 100

---

**ITERATOR MODEL**

1. for t in child.Next():
   emit(projection(t))

2. for t₁ in left.Next():
   buildHashTable(t₁)
for t₂ in right.Next():
   if probe(t₂): emit(t₁⨝t₂)

   **Single Tuple**

3. for t in R:
   emit(t)
for t in S:
   emit(t)
ITERATOR MODEL

1. for t in child.Next():
   emit(projection(t))

2. for t1 in left.Next():
   buildHashTable(t1)
   for t2 in right.Next():
     if probe(t2): emit(t1 ⨝ t2)

3. for t in R:
   emit(t)

4. for t in child.Next():
   if evalPred(t): emit(t)

5. for t in S:
   emit(t)

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

This is used in almost every DBMS. Allows for tuple pipelining.

Some operators must block until their children emit all their tuples.
→ Joins, Subqueries, Order By

Output control works easily with this approach.
MATERIALIZATION MODEL

Each operator processes its input all at once and then emits its output all at once.
→ The operator "materializes" its output as a single result.
→ The DBMS can push down hints into to avoid scanning too many tuples.
→ Can send either a materialized row or a single column.

The output can be either whole tuples (NSM) or subsets of columns (DSM)
MATERIALIZATION MODEL

```
out = [ ]
for t in child.Output():
    out.add(projection(t))
return out
```

```
out = [ ]
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1 ∙ t2)
return out
```

```
out = [ ]
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in R:
    out.add(t)
out.add(t)
return out
```

```
out = [ ]
for t in S:
    out.add(t)
out.add(t)
return out
```

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

1. \( \text{out} = [ ] \)
   for \( t \) in \text{child.Output()}:
   \( \text{out}.\text{add}(\text{projection}(t)) \)
   return \( \text{out} \)

2. \( \text{out} = [ ] \)
   for \( t_1 \) in \text{left.Output()}:
   \( \text{buildHashTable}(t_1) \)
   for \( t_2 \) in \text{right.Output()}:
   if \( \text{probe}(t_2) \): \( \text{out}.\text{add}(t_1 \bowtie t_2) \)
   return \( \text{out} \)

3. \( \text{All Tuples} \)

\( \text{out} = [ ] \)
for \( t \) in \text{R}:
\( \text{out}.\text{add}(t) \)
return \( \text{out} \)

\( \text{out} = [ ] \)
for \( t \) in \text{S}:
\( \text{out}.\text{add}(t) \)
return \( \text{out} \)

SELECT \( R.\text{id}, S.\text{cdate} \)
FROM \( R \) JOIN \( S \)
ON \( R.\text{id} = S.\text{id} \)
WHERE \( S.\text{value} > 100 \)
**MATERIALIZATION MODEL**

1. `out = []
   for t in child.Output():
     out.add(projection(t))
   return out`

2. `out = []
   for t1 in left.Output():
     buildHashTable(t1)
   for t2 in right.Output():
     if probe(t2):
       out.add(t1⨝t2)
   return out`

3. `out = []
   for t in R:
     out.add(t)
   return out`

4. `out = []
   for t in child.Output():
     if evalPred(t):
       out.add(t)
   return out`

5. `out = []
   for t in S:
     out.add(t)
   return out`

---

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

Better for OLTP workloads because queries only access a small number of tuples at a time.
→ Lower execution / coordination overhead.
→ Fewer function calls.

Not good for OLAP queries with large intermediate results.
VECTORIZATION MODEL

Like the Iterator Model where each operator implements a `Next` function in this model. Each operator emits a `batch` of tuples instead of a single tuple.

→ The operator's internal loop processes multiple tuples at a time.
→ The size of the batch can vary based on hardware or query properties.
VECTORIZATION MODEL

**Example Code:**

```python
out = [ ]
for t in child.Next():
    out.add(projection(t))
    if |out|>n: emit(out)

out = [ ]
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): out.add(t1⨝t2)
    if |out|>n: emit(out)

out = [ ]
for t in child.Next():
    if evalPred(t): out.add(t)
    if |out|>n: emit(out)

out = [ ]
for t in R:
    out.add(t)
    if |out|>n: emit(out)

out = [ ]
for t in S:
    out.add(t)
    if |out|>n: emit(out)
```

**SQL Query:**

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

out = [ ]
for t in child.Next():
    out.add(projection(t))
if |out|>n: emit(out)

out = [ ]
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): out.add(t1⨝t2)
if |out|>n: emit(out)

out = [ ]
for t in R:
    out.add(t)
if |out|>n: emit(out)

out = [ ]
for t in S:
    out.add(t)
if |out|>n: emit(out)

out = [ ]
for t in S:
    out.add(t)
if |out|>n: emit(out)
VEC TOR I Z AT ION MO DE L

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.
Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.
PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom
→ Start with the root and "pull" data up from its children.
→ Tuples are always passed with function calls.

Approach #2: Bottom-to-Top
→ Start with leaf nodes and push data to their parents.
→ Allows for tighter control of caches/registers in pipelines.
An **access method** is a way that the DBMS can access the data stored in a table.
→ Not defined in relational algebra.

Three basic approaches:
→ Sequential Scan
→ Index Scan
→ Multi-Index / "Bitmap" Scan
SEQUENTIAL SCAN

For each page in the table:
→ Retrieve it from the buffer pool.
→ Iterate over each tuple and check whether to include it.

The DBMS maintains an internal cursor that tracks the last page / slot it examined.

```
for page in table.pages:
    for t in page.tuples:
        if evalPred(t):
            // Do Something!
```
SEQUENTIAL SCAN: OPTIMIZATIONS

This is almost always the worst thing that the DBMS can do to execute a query.

Sequential Scan Optimizations:
→ Prefetching
→ Buffer Pool Bypass
→ Parallelization
→ Heap Clustering
→ Zone Maps
→ Late Materialization
ZONE MAPS

Pre-computed aggregates for the attribute values in a page. DBMS checks the zone map first to decide whether it wants to access the page.

Original Data

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>400</td>
</tr>
</tbody>
</table>

SELECT * FROM table WHERE val > 600

Zone Map

<table>
<thead>
<tr>
<th>type</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>100</td>
</tr>
<tr>
<td>MAX</td>
<td>400</td>
</tr>
<tr>
<td>AVG</td>
<td>280</td>
</tr>
<tr>
<td>SUM</td>
<td>1400</td>
</tr>
<tr>
<td>COUNT</td>
<td>5</td>
</tr>
</tbody>
</table>
DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

```
SELECT AVG(foo.c) FROM foo JOIN bar ON foo.b = bar.b WHERE foo.a > 100
```
LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

```
SELECT AVG(foo.c)
FROM foo JOIN bar
ON foo.b = bar.b
WHERE foo.a > 100
```
LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

\[ \text{SELECT} \ \text{AVG}(\text{foo.c}) \ \text{FROM} \ \text{foo JOIN} \ \text{bar ON} \ \text{foo.b} = \text{bar.b} \ \text{WHERE} \ \text{foo.a} > 100 \]
The DBMS picks an index to find the tuples that the query needs.

Which index to use depends on:
→ What attributes the index contains
→ What attributes the query references
→ The attribute's value domains
→ Predicate composition
→ Whether the index has unique or non-unique keys
Suppose that we a single table with 100 tuples and two indexes:
→ Index #1: age
→ Index #2: dept

**Scenario #1**
There are 99 people under the age of 30 but only 2 people in the CS department.

**Scenario #2**
There are 99 people in the CS department but only 2 people under the age of 30.

```
SELECT * FROM students
WHERE age < 30
  AND dept = 'CS'
  AND country = 'US'
```
MULTI-INDEX SCAN

If there are multiple indexes that the DBMS can use for a query:
→ Compute sets of record ids using each matching index.
→ Combine these sets based on the query's predicates (union vs. intersect).
→ Retrieve the records and apply any remaining predicates.

Postgres calls this **Bitmap Scan**.
**MULTI-INDEX SCAN**

With an index on \texttt{age} and an index on \texttt{dept},

→ We can retrieve the record ids satisfying \texttt{age<30} using the first,

→ Then retrieve the record ids satisfying \texttt{dept='CS'} using the second,

→ Take their intersection

→ Retrieve records and check \texttt{country='US'}. 

```
SELECT * FROM students
WHERE age < 30
    AND dept = 'CS'
    AND country = 'US'
```
MULTI-INDEX SCAN

Set intersection can be done with bitmaps, hash tables, or Bloom filters.

```
SELECT * FROM students
WHERE age < 30
AND dept = 'CS'
AND country = 'US'
```
MODIFICATION QUERIES

Operators that modify the database (INSERT, UPDATE, DELETE) are responsible for checking constraints and updating indexes.

UPDATE/DELETE:
→ Child operators pass Record Ids for target tuples.
→ Must keep track of previously seen tuples.

INSERT:
→ Choice #1: Materialize tuples inside of the operator.
→ Choice #2: Operator inserts any tuple passed in from child operators.
**UPDATE QUERY PROBLEM**

```
for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
updateTuple(t.salary = t.salary + 1000)
insertIntoIndex(idx_salary, t.salary, t)

for t in people:
    emit(t)
```

```
CREATE INDEX idx_salary
    ON people (salary);

UPDATE people
    SET salary = salary + 100
WHERE salary < 1000
```

Index(people.salary)
UPDATE QUERY PROBLEM

for t in child.Next():
   removeFromIndex(idx_salary, t.salary, t)
   updateTuple(t.salary = t.salary + 1000)
   insertIntoIndex(idx_salary, t.salary, t)

for t in people:
   emit(t)

CREATE INDEX idx_salary 
ON people (salary);

UPDATE people
   SET salary = salary + 100
WHERE salary < 1000

Index(people.salary)

(999, Andy)
UPDATE QUERY PROBLEM

```python
for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 1000)
    insertIntoIndex(idx_salary, t.salary, t)

for t in people:
    emit(t)
```

CREATE INDEX idx_salary
ON people (salary);

UPDATE people
SET salary = salary + 100
WHERE salary < 1000

Index(people.salary)
UPDATE QUERY PROBLEM

CREATE INDEX idx_salary
ON people (salary);

UPDATE people
SET salary = salary + 100
WHERE salary < 1000

for t in child.Next():
    (999, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 1000)
    insertIntoIndex(idx_salary, t.salary, t)

for t in people:
    emit(t)

Index(people.salary)
CREATE INDEX idx_salary ON people (salary);

UPDATE people
SET salary = salary + 100
WHERE salary < 1000

for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 1000)
    insertIntoIndex(idx_salary, t.salary, t)

for t in people:
    emit(t)
UPDATE QUERY PROBLEM

```sql
CREATE INDEX idx_salary ON people (salary);

UPDATE people
SET salary = salary + 100
WHERE salary < 1000

for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 1000)
    insertIntoIndex(idx_salary, t.salary, t)

for t in people:
    emit(t)
```

Index(people.salary)

(1099, Andy)
UPDATE QUERY PROBLEM

CREATE INDEX idx_salary ON people (salary);

UPDATE people
SET salary = salary + 100
WHERE salary < 1000

for t in people:
    emit(t)

Index(people.salary)

for t in child.Next():
    (1099, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 1000)
    insertIntoIndex(idx_salary, t.salary, t)
UPDATE QUERY PROBLEM

CREATE INDEX idx_salary
    ON people (salary);

UPDATE people
    SET salary = salary + 100
WHERE salary < 1000

Index(people.salary)
HALLOWEEN PROBLEM

An anomaly where an update operation changes the physical location of a tuple, which causes a scan operator to visit the tuple multiple times.

→ Can occur on clustered tables or index scans.

First discovered by IBM researchers while working on System R on Halloween day in 1976.
The DBMS represents a **WHERE** clause as an **expression tree**.

The nodes in the tree represent different expression types:
- Comparisons (=, <, >, !=)
- Conjunction (AND), Disjunction (OR)
- Arithmetic Operators (+, -, *, /, %)
- Constant Values
- Tuple Attribute References
**EXECUTION CONTEXT**

**SELECT** * FROM S
WHERE B.value = ? + 1

**Execution Context**

- **Current Tuple**: (123, 1000)
- **Query Parameters**: (int:999)
- **Table Schema**: S→(int:id, int:value)

**Expression Evaluation**

- **Attribute(S.value)**
- **Parameter(0)**
- **Constant(1)**
- **+**
- **=**
### Expression Evaluation

**Execution Context**

- **SELECT** `*` **FROM** `S`
- **WHERE** `B.value = ? + 1`

**Current Tuple**
- (123, 1000)

**Query Parameters**
- (int:999)

**Table Schema**
- `S\to (int:id, int:value)`

**Expression Evaluation Diagram**

- `Attribute(S.value)`
- `Constant(1)`
- `Parameter(0)`
- `1000`
- `+`
- `true`
- `1000`
- `1000`
- `999`
Evaluating predicates in this manner is slow.

→ The DBMS traverses the tree and for each node that it visits it must figure out what the operator needs to do.

Consider the predicate "WHERE 1=1"

A better approach is to just evaluate the expression directly.

→ Think JIT compilation
CONCLUSION

The same query plan be executed in multiple ways.

(Most) DBMSs will want to use an index scan as much as possible.

Expression trees are flexible but slow.
NEXT CLASS

Parallel Query Execution