Query Execution
Part 1
ADMINISTRIVIA

Homework 3 due last night
→ Solutions will be available today

Midterm exam Wednesday!

Project 2 is available
→ First checkpoint due Friday, March 3rd (15% of P2 grade)
→ Overall due Wednesday, March 22nd (85% of P2 grade)
PROJECTS

Do write your own tests.
Do practice defensive programming.
Do use a profiler to find performance problems.

Do not use Gradescope for debugging.
Do not directly email TAs for help. Post to Piazza.
LAST TIME: JOIN ALGORITHMS

Nested Loop Join
→ Naïve
→ Block
→ Index

Sort-Merge Join

Hash Join
→ Simple
→ GRACE (Externally Partitioned)
→ Hybrid
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

```
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 (e.g., \texttt{LIMIT}) 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

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

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

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

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

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

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, but...

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

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

2. \[ \text{out} = [] \]
   \[ \text{for } t_1 \text{ in } \text{left.Next}(): \]
   \[ \text{buildHashTable}() \]
   \[ \text{for } t_2 \text{ in } \text{right.Next}(): \]
   \[ \text{if } \text{probe}(t_2): \text{out.add}(t_1 \odot t_2) \]
   \[ \text{if } |\text{out}| > n : \text{emit(out)} \]

3. \[ \text{out} = [] \]
   \[ \text{for } t \text{ in } R: \]
   \[ \text{out.add}(t) \]
   \[ \text{if } |\text{out}| > n : \text{emit(out)} \]

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

5. \[ \text{out} = [] \]
   \[ \text{for } t \text{ in } S: \]
   \[ \text{out.add}(t) \]
   \[ \text{if } |\text{out}| > n : \text{emit(out)} \]

SQL:

\[
\text{SELECT } R.\text{id}, S.\text{cdate} \\
\text{FROM } R \text{ JOIN } S \\
\text{ON } R.\text{id} = S.\text{id} \\
\text{WHERE } S.\text{value} > 100
\]
VECTORIZATION MODEL

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

Allows for operators to more easily 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 the way that the DBMS accesses the data stored in a table.

→ Not defined in relational algebra.

Three basic approaches:

→ Sequential Scan
→ Index Scan (many variants)
→ Multi-Index Scan

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

```python
for page in table.pages:
    for t in page.tuples:
        if evalPred(t):
            # Do Something!
```
This is almost always the worst thing that the DBMS can do to execute a query, but it may be the **only** choice available.

**Sequential Scan Optimizations:**
- Prefetching
- Buffer Pool Bypass
- Parallelization
- Heap Clustering
- Late Materialization
DATA SKIPPING

Approach #1: Approximate Queries (Lossy)
→ Execute queries on a sampled subset of the entire table to produce approximate results.
→ Examples: BlinkDB, Redshift, ComputeDB, XDB, Oracle, Snowflake, Google BigQuery, DataBricks

Approach #2: Zone Maps (Loseless)
→ Pre-compute columnar aggregations per page that allow the DBMS to check whether queries need to access it.
→ Trade-off between page size vs. filter efficacy.
→ Examples: Oracle, Vertica, SingleStore, Netezza, Snowflake, Google BigQuery
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>
</tbody>
</table>

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>

SELECT * FROM table WHERE val > 600
INDEX SCAN

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 have 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.

Examples:
→ DB2 Multi-Index Scan
→ PostgreSQL Bitmap Scan
→ MySQL Index Merge
MULTI-INDEX SCAN

With an index on `age` and an index on `dept`:
→ We can retrieve the Record IDs satisfying `age < 30` using the first,
→ Then retrieve the Record IDs satisfying `dept = 'CS'` using the second,
→ Take their intersection
→ Retrieve records and check `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.

```sql
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 modifying the target table and its indexes.

→ Constraint checks can either happen immediately inside of operator or deferred until later in query/transaction.

The output of these operators can either be Record Ids or tuple data (i.e., RETURNING).
MODIFICATION QUERIES

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

CREATE INDEX idx_salary ON people (salary);

UPDATE people
SET salary = salary + 100
WHERE salary < 1100
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.

Solution: Track modified record ids per query.
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

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```
PREPARE xxx AS
SELECT * FROM S
WHERE S.val = $1 + 9

EXECUTE xxx(991)

**Execution Context**

Current Tuple
(123, 1000)

Query Parameters
(int:991)

Table Schema
S→ (int:id, int:val)

**Expression Evaluation**

\[
\text{Attribute}(S.val) = \text{true} + \text{Parameter}($1) + \text{Constant}(9)
\]

1000 + 991 + 9 = 1990
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 this predicate:
**WHERE S.val=1**

A better approach is to just evaluate the expression directly.
→ Think JIT compilation
CONCLUSION

The same query plan can be executed in multiple different ways.

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

Expression trees are flexible but slow. JIT compilation can (sometimes) speed them up.
NEXT CLASS

Parallel Query Execution