Query Processing
ADMINISTRIVIA

Project #2 – Checkpoint #1 is due Monday October 9\textsuperscript{th} @ 11:59pm

Mid-term Exam is on Wednesday October 17\textsuperscript{th} (in class)
UPCOMING DATABASE EVENTS

SQream DB Tech Talk
→ Thursday Oct 4th @ 12:00pm
→ CIC 4th Floor
The operators are arranged in a tree. Data flows from the leaves toward the root.

The output of the root node is the result of the query.

```
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
```
TODAY'S AGENDA

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

Three approaches:
→ Iterator Model
→ Materialization Model
→ Vectorized / Batch 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.

Top-down plan processing.
Also called **Volcano** or **Pipeline** Model.
ITERATOR MODEL

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

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

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

for t in A:
    emit(t)

for t in B:
    emit(t)

SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
```
ITERATOR MODEL

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

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

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

   for t in A:
     emit(t)

   for t in B:
     emit(t)

SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
ITERATOR MODEL

SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100

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

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

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

for t in A:
    emit(t)

for t in B:
    emit(t)
### SELECT A.id, B.value
### FROM A, B
### WHERE A.id = B.id AND B.value > 100
ITERATOR MODEL

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

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

Output control works easily with this approach. → Limit
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.

Bottom-up plan processing.
MATERIALIZATION MODEL

```
out = { }
for t in child.Output():
    out.add(projection(t))
```

```
out = { }
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1⨝t2)
```

```
out = { }
for t in child.Output():
    if evalPred(t): out.add(t)
```

```
out = { }
for t in A:
    out.add(t)
out = { }
for t in B:
    out.add(t)
```

```
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
```
MATERIALIZATION MODEL

\[ \text{SELECT } A.id, B.value \\
\text{FROM } A, B \\
\text{WHERE } A.id = B.id \text{ AND } B.value > 100 \]
MATERIALIZATION MODEL

```
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
```
**MATERIALIZATION MODEL**

- **1.** `out = {}` for `t` in `A`:
  - `out.add(t)`

- **2.** `out = {}` for `t` in `B`:
  - `out.add(t)`

- **3.** `out = {}` for `t` in `child.Output()`:
  - `if evalPred(t): out.add(t)`

- **4.** `out = {}` for `t_1` in `left.Output()`:
  - `buildHashTable(t_1)`
  - `for t_2 in right.Output()`:
    - `if probe(t_2): out.add(t_1⨝t_2)`

- **5.** `out = {}` for `t` in `child.Output()`:
  - `out.add(projection(t))`

**SQL Example:**

```sql
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
```
MATERIALIZATION MODEL

Better for OLTP workloads because queries typically only access a small number of tuples at a time.

→ Lower execution / coordination overhead.

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

Like Iterator Model, each operator implements a `next` function.

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.
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100

VECTORIZATION MODEL

1
out = {}
for t in child.Output():
    out.add(projection(t))
    if |out| > n: emit(out)

2
out = {}
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1⨝t2)
    if |out| > n: emit(out)

3
out = {}
for t in child.Output():
    out.add(t)
    if |out| > n: emit(out)

out = {}
for t in A:
    out.add(t)
    if |out| > n: emit(out)

out = {}
for t in B:
    out.add(t)
    if |out| > n: emit(out)
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
    AND B.value > 100
VECTORIZATION MODEL

Ideal for OLAP queries
→ Greatly reduces the number of invocations per operator.
→ Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.
PROCESSING MODELS SUMMARY

**Iterator / Volcano**
- Direction: Top-Down
- Emits: Single Tuple
- Target: General Purpose

**Vectorized**
- Direction: Top-Down
- Emits: Tuple Batch
- Target: OLAP

**Materialization**
- Direction: Bottom-Up
- Emits: Entire Tuple Set
- Target: OLTP
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

```sql
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.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.

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

```
SELECT * FROM table
WHERE val > 600
```
LATE MATERIALIZATION

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

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

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

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

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

\[
\text{SELECT} \ \text{AVG}(C) \\
\text{FROM} \ \text{foo} \ \text{JOIN} \ \text{bar} \\
\text{ON} \ \text{foo}.b = \text{bar}.b \\
\text{WHERE} \ a > 100
\]
LATE MATERIALIZATION

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

```
SELECT AVG(C)
FROM foo JOIN bar
ON foo.b = bar.b
WHERE a > 100
```
HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a clustering index.

If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.
Tuples are sorted in the heap's pages using the order specified by a clustering index.

If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.
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
INDEX SCAN

Suppose that we a single table with 100 tuples and two indexes:
→ Index #1: age
→ Index #2: dept

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

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 terms.

Postgres calls this **Bitmap Scan**
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.

SELECT * FROM students
WHERE age < 30
AND dept = 'CS'
AND country = 'US'
Set intersection can be done with bitmaps, hash tables, or Bloom filters.

```
SELECT * FROM students
WHERE age < 30
AND dept = 'CS'
AND country = 'US'
```
Set intersection can be done with bitmaps, hash tables, or Bloom filters.

```
SELECT * FROM students
WHERE age < 30
AND dept = 'CS'
AND country = 'US'
```
Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.
The DBMS represents a \textbf{WHERE} clause as an \textbf{expression tree}.

The nodes in the tree represent different expression types:

→ Comparisons ($=$, $<$, $>$, $!=$)
→ Conjunction (\textbf{AND}), Disjunction (\textbf{OR})
→ Arithmetic Operators ($+$, $-$, $\times$, $/$, $\%$)
→ Constant Values
→ Tuple Attribute References
**EXPRESSION EVALUATION**

**Execution Context**

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

**SQL Query**:

```sql
SELECT * FROM B
WHERE B.val = ? + 1
```

**Expression Tree**:

- `Attribute(val)`
- `Parameter(0)`
- `Constant(1)`
- `+`
- `=`
**EXPRESSION EVALUATION**

```
SELECT * FROM B
WHERE B.val = ? + 1
```

**Execution Context**

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

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

- **Table Schema**
  - B→(int:id, int:val)

**Expression Tree**

```
Attribute(val) = +
```

- **Parameter(0)**
- **Constant(1)**
- **1000**
EXECUTION CONTEXT

SELECT * FROM B
WHERE B.val = ? + 1

Current Tuple
(123, 1000)
Query Parameters
(int:999)
Table Schema
B→(int:id, int:val)

Expression Evaluation

Attribute(val)

Parameter(0)
Constant(1)

1000
999

= +
**EXPRESSION EVALUATION**

**Execution Context**

```
SELECT * FROM B
WHERE B.val = ? + 1
```

- **Current Tuple**: (123, 1000)
- **Query Parameters**: (int:999)
- **Table Schema**: `B→(int:id, int:val)`

**Expression Diagram**

```
= 
Attribute(val) +
1000 999
Parameter(0) Constant(1)
```

**Attribute** (`val`)

**Constant** (1)

**Parameter** (0)
**EXPRESSION EVALUATION**

**Execution Context**

**SELECT** * FROM B
WHERE B.val = ? + 1

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

```
= 
true

Attribute(val)
```

```
+ 
1000
```

```
Parameter(0)
```

```
Constant(1)
```

```
999
```

```
1
```