ADMINISTRIVIA

Project #1 is due TODAY @ 11:59pm

Homework #3 is due Wednesday October 4th @ 11:59pm

Mid-term Exam is on Wednesday October 18th (in class)

Project #2 is due Wednesday October 25th @ 11:59am
LECTURE #08 CORRECTION

Nasty Deez Nutz In Yo Moth 2 months ago
why we do have to suffer with these bad lectures? professor pavlo sucks straight up.

REPLY 11

The14thChapter 9 hours ago
Yo i herd that andy pushed this old lady down the stairs. hes awful. databases are tight and all but he needs to stop with dez bad lectures. santa monica out!

REPLY 53

DaOldSchoolRapJiveTurkey94 2 weeks ago
Andy is awful. He speaks so fast that I get headaches. I wish somebody that was at CMU would stab him.

REPLY 139

View all 16 replies▼
William Cody Laeder  2 days ago

City/Farm don't use SIMD this hurts portability (Google ships farmhash in Chrome). They use a small buffer internally (normally 64, XXHash uses 256 for larger mode). If the hash internally tries to fill this buffer before it computes a digest (and XOR the old digest with the new 64bytes digest), and if it that buffer isn't full it does a unique
We are now going to talk about how the DBMS execute queries that retrieve data from the system's access methods.
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.
TODAY'S AGENDA

Processing Models
Access Methods
Expression Evaluation
Project #2
Processing Model

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
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.

Top-down plan processing.
Also called Volcano or Pipeline Model.
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
    AND B.value > 100

for t in A:
    emit(t)

for t in B:
    emit(t)

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

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

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

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

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

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

**Diagram:**
- `π A.id, B.value`
- `A.id=B.id`
- `σ value>100`

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

1. \( \text{out} = \{ \} \)
   
   For \( t \) in \( A \):
   
   \( \text{out}.\text{add}(t) \)

2. \( \text{out} = \{ \} \)
   
   For \( t \) in \( B \):
   
   \( \text{out}.\text{add}(t) \)

3. \( \text{out} = \{ \} \)
   
   For \( t \) in \( \text{child}.\text{Output()} \):
   
   If evalPred(t):
   
   \( \text{out}.\text{add}(t) \)

4. \( \text{out} = \{ \} \)
   
   For \( t_1 \) in \( \text{left}.\text{Output()} \):
   
   \( \text{buildHashTable}(t_1) \)
   
   For \( t_2 \) in \( \text{right}.\text{Output()} \):
   
   If probe\( (t_2) \):
   
   \( \text{out}.\text{add}(t_1 \bowtie t_2) \)

5. \( \text{out} = \{ \} \)
   
   For \( t \) in \( \text{child}.\text{Output()} \):
   
   \( \text{out}.\text{add}(\text{projection}(t)) \)

**SQL Query**

```
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.
→ More difficult to parallelize.

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
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

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

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

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

```
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!
```
SEQUENTIAL SCAN: OPTIMIZATIONS

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

Sequential Scan Optimizations:
- Prefetching
- Parallelization
- Zone Maps
- Buffer Pool Bypass
- Heap Clustering
ZONE MAPS

Pre-computed aggregates for the attribute values in a page.

DBMS can check the zone map first to decide whether it wants to access the page.

```
SELECT * FROM table
WHERE val > 600
```
BUFFER POOL BYPASS

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.
→ Memory is local to running query.
→ Works well if operator needs to read a large sequence of pages that are contiguous on disk.

Called "Light Scans" in Informix.
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.
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.
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

Later: Query Optimization
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"
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'
```
Set intersection can be done with bitmaps, hash tables, or Bloom filters.

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

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

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`)
- **Operation** `+`
- **Operation** `=`
- **Value** `1000`

**Diagram**:

```
Attribute(val)  +  Constant(1)
       |                |
       |  999          |
       |--------------|
       |  Parameter(0)|
       |-------------|
       |  1000       |
```
SELECT * FROM B
WHERE B.val = ? + 1

Table Schema
B→(int:id, int:val)

Query Parameters
(int:999)

Current Tuple
(123, 1000)

Attribute(val)

Constant(1)

Parameter(0)

1000

1

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:

\[
\text{val} = \text{Parameter(0)} + \text{Constant(1)}
\]

\[
\text{Parameter(0)} = 999
\]

\[
\text{Constant(1)} = 1
\]

Result:

\[
\text{val} = 1000 + 1 = 1001
\]

\[
\text{Attribute(val)} = \text{true}
\]
EVALUATING

Evaluating predicates in this manner is slow.

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

Consider **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.
PROJECT #2

You will build a **single-threaded** B+tree index.
→ Page Layout
→ Data Structure
→ Iterator.

We define the API for you. You need to provide the method implementations.

http://15445.courses.cs.cmu.edu/fall2017/project2/
THINGS TO NOTE

Do **not** change any file other than the six that you have to hand it.

We will provide an updated source tarball. You will need to copy over your files from Project #1.

Post your questions on Canvas or come to TA office hours.
→ We will **not** help you debug.
PLAGIARISM WARNING

Your project implementation must be your own work.
→ You may not copy source code from other groups or the web.
→ Do not publish your implementation on Github.

Plagiarism will not be tolerated. See CMU's Policy on Academic Integrity for additional information.
Next Class

More query execution
→ External Merge Sort
→ Join Algorithms