Project #2 is due Sunday, Oct 17th @ 11:59pm

Mid-Term Exam is Wednesday, Oct 13th
→ During regular class time @ 3:05-4:25pm
→ Open book / open notes
→ Will include all material covered before mid-term
→ See Piazza post for more details
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**
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
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100

for t in R:
emit(t)

for t in S:
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 R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
for t in child.Next():
    emit(projection(t))

for t_1 in left.Next():
    buildHashTable(t_1)
for t_2 in right.Next():
    if probe(t_2): emit(t_1⨝t_2)

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

for t in R:
    emit(t)

for t in S:
    emit(t)

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

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 R:
     emit(t)

   for t in S:
     emit(t)

   for t in S:
     emit(t)
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 R:
    emit(t)

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

```
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

---

1. For `t` in `child.Next()`:
   - `emit(projection(t))`

2. For `t_1` in `left.Next()`:
   - `buildHashTable(t_1)`
   - For `t_2` in `right.Next()`:
     - If `probe(t_2)`:
       - `emit(t_1 \bowtie t_2)`

---

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 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)

3. 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)

3. 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

π R.id, S.value
R.id=S.id
σ value>100
R S

for t in R:
emit(t)

for t in S:
emit(t)
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
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.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 \times t2)

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

for t in R:
    emit(t)

for t in S:
    emit(t)
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
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.
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., 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

```
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)
return out
```

```
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

```
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(t1uxtapositiont2)
return out
```

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

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

```
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

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

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

3. \[
\text{out} = [ ] \\
\text{for } t \text{ in child.Output():} \\
\text{if evalPred}(t): \text{out.add}(t) \\
\text{return out}
\]

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

\[
\begin{align*}
\pi & \quad R.id, S.value \\
\bowtie & \quad \text{R.id=}
\end{align*}
\]

\[
\begin{align*}
\sigma & \quad \text{value>100}
\end{align*}
\]
All Tuples

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 $\bowtie$ t2)
   return out

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

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

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

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

3. \[
\text{out} = [ ] \\
\text{for } t \text{ in child.Output():} \\
\quad \text{if } \text{evalPred}(t) \text{ return out} \\
\text{out.add}(t) \\
\text{return out}
\]

4. \[
\text{out} = [ ] \\
\text{for } t \text{ in child.Output():} \\
\quad \text{out.add}(t) \\
\text{return out}
\]

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

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.
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.
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
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

**VECTORIZATION MODEL**

1. 
```python
out = []
for t in child.Next():
    out.add(projection(t))
if |out| > n: emit(out)
```

2. 
```python
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)
```

3. 
```python
out = []
for t in R:
    out.add(t)
if |out| > n: emit(out)
```

4. 
```python
out = []
for t in child.Next():
    if evalPred(t):
        out.add(t)
    if |out| > n: emit(out)
```

5. 
```python
out = []
for t in S:
    out.add(t)
    if |out| > n: emit(out)
```
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.
ACCESS METHODS

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
→ Multi-Index / "Bitmap" Scan

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```
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
→ Multi-Index / "Bitmap" Scan

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```
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
→ Buffer Pool Bypass
→ Parallelization
→ Heap Clustering
→ Zone Maps
→ Late Materialization
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>
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.

**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>

**SQL Query**

```
SELECT * FROM table
WHERE val > 600
```
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.
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
```
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
```
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
```
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
```
DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

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

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

```sql
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**.
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'**.

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

\[
\text{SELECT } * \text{ FROM students }
\text{WHERE age < 30 AND dept = 'CS' AND country = 'US'}
\]
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.

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

```
SELECT * FROM students
WHERE age < 30
AND dept = 'CS'
AND country = 'US'
```
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 + 100)
    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 < 1100
UPDATE QUERY PROBLEM

for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    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 < 1100
CREATE INDEX idx_salary ON people (salary);

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

for t in child.Next():
  removeFromIndex(idx_salary, t.salary, t)
  updateTuple(t.salary = t.salary + 100)
  insertIntoIndex(idx_salary, t.salary, t)

for t in people:
  emit(t)
for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    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 < 1100

Index(people.salary)
**UPDATE QUERY PROBLEM**

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

```sql
UPDATE people
SET salary = salary + 100
WHERE salary < 1100
```

```python
for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

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

Index(people.salary)

(999, Andy)
UPDATE QUERY PROBLEM

CREATE INDEX idx_salary
    ON people (salary);

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

Index(people.salary)

for t in child.Next():
    (999, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

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

```python
for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

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

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

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

[Index(people.salary)](999,Andy)
UPDATE QUERY PROBLEM

CREATE INDEX idx_salary ON people (salary);

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

Index(people.salary)

for t in child.Next():
    (999, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

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

CREATE INDEX idx_salary
ON people (salary);

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

for t in child.Next():
    (1099, Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    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 < 1100

for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

for t in people:
    emit(t)

Index(people.salary)
```
UPDATE QUERY PROBLEM

CREATE INDEX idx_salary
ON people (salary);

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

Index(people.salary)

for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

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

CREATE INDEX idx_salary ON people (salary);

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

Index(people.salary)

for t in child.Next():    (1099,Andy)
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    insertIntoIndex(idx_salary, t.salary, t)

for t in people:    emit(t)
for t in child.Next():
    removeFromIndex(idx_salary, t.salary, t)
    updateTuple(t.salary = t.salary + 100)
    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 < 1100
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
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
```
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 S
WHERE S.value = ? + 1
EXPRESSISON EVALUATION

```
SELECT * FROM S
WHERE S.value = ? + 1
```
**EXECUTION CONTEXT**

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

**Current Tuple**
(123, 1000)

**Query Parameters**
(int:999)

**Table Schema**
S→(int:id, int:value)

**Expression Evaluation**

- Attribute(S.value)
- Parameter(0)
- Constant(1)
- +
- =

```
Constant(1) = Parameter(0) + Attribute(S.value)
```
SELECT * FROM S WHERE S.value = ? + 1

**Execution Context**

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

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

**Execution Context**

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

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

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

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

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

**Execution Context**

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

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

- **Attribute**: `S.value`
- **Parameter**: `0`
- **Constant**: `1`
- **Expression**: `Attribute(S.value) = Parameter(0) + Constant(1)`
```sql
SELECT * FROM S
WHERE S.value = ? + 1
```

**Execution Context**

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

**Expression Evaluation Diagram**

```
Attribute(S.value) = +
```

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

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

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

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

**Table Schema**: `S → (int:id, int:value)`

**Expression Evaluation**

```
Attribute(S.value) = +

Parameter(0) + Constant(1)
```

**Result**: 1000
**EXECUTION CONTEXT**

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

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

**Expression Evaluation**

```
1000
```

```
= 
```

```
Attribute(S.value) + Parameter(0)
```

```
Constant(1)
```

```
999
```
**Expression Evaluation**

**Execution Context**

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

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

Diagram:
- Attribute(S.value)
- Parameter(0)
- Constant(1)
- 1000
- 999
- =
- +

- Expression: (999 + 1)
- Result: 1000
**Execution Context**

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

Current Tuple: (123, 1000)

Query Parameters: (int:999)

Table Schema: S→(int:id, int:value)

Expression Evaluation:

- **Attribute(S.value)**
  - 1000

- **Parameter(0)**
  - 999

- **Constant(1)**
  - 1

- **=**
  - 1000

- **+**
  - 1000
**EXECUTION CONTEXT**

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

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

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

**Table Schema:** S→(int:id, int:value)

**Expression Evaluation Diagram:**

- ` Attribute(S.value)`
  - `1000`
- ` +`
  - `1000`
- ` =`
  - `true`
- ` Parameter(0)`
  - `999`
- ` Constant(1)`
  - `1`
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 1=1**
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 1=1**

A better approach is to just evaluate the expression directly.

→ Think JIT compilation
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: \textbf{WHERE 1=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.
NEXT CLASS

Parallel Query Execution