Query Execution + Retrospective

CREATING THE NEXT®

Administrivia

- Assignment 4 and Sheet 4 due on Dec 3.
- Project presentations on Dec 1 and Dec 6.
- Report due on Dec 6.

Today's Agenda

Query Execution (Part 1)

1.1 Recap

1.2 Processing Models

1.3 Access Methods

1.4 Expression Evaluation

1.5 Retrospective

Recap

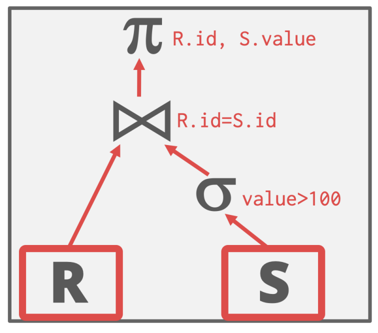
Join Algorithms: Summary

Join Algorithm	IO Cost	Example
Simple Nested Loop Join	M + (m x N)	1.3 hours
Block Nested Loop Join	M + (M x N)	50 seconds
Index Nested Loop Join	M + (M x C)	Variable
Sort-Merge Join	M + N + (sort cost)	0.75 seconds
Hash Join	3 x (M + N)	0.45 seconds

Query Plan

- 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.

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



Processing Models

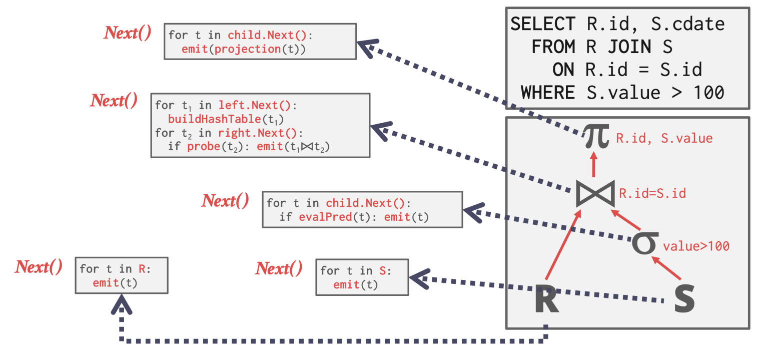
Processing Model

- 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
```

Next()

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

Next()

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

Next()

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

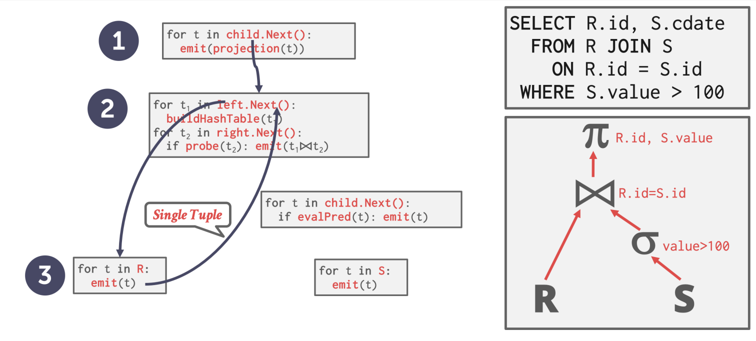
Next()

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

Next()

```
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)
```

Single Tuple

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

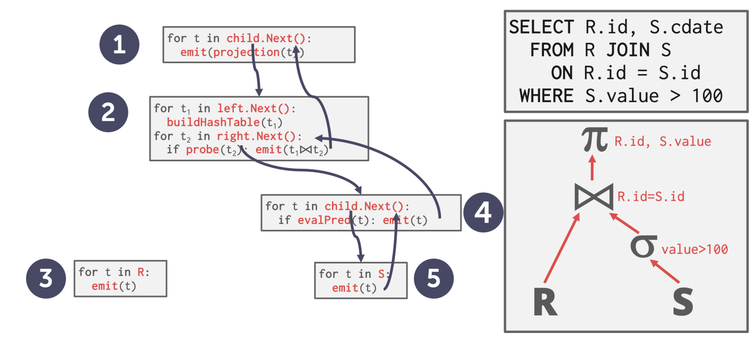
3

```
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
```

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
```

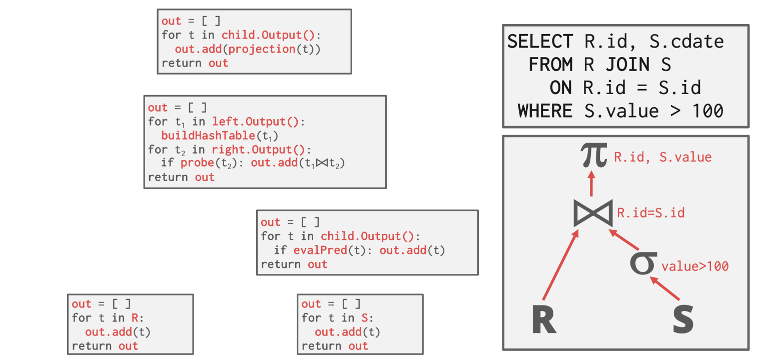
Iterator Model

- This is used in almost every DBMS. Allows for tuple **pipelining**.
- Some operators have to block until their children emit all of their tuples.
- These operators are known as **pipeline breakers**
  - Joins, Subqueries, Order By
- Output control (*e.g.*, `LIMIT`) works easily with this approach.
- **Examples:** SQLite, MySQL, PostgreSQL

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 (*e.g.*, `LIMIT`).
  - 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
```

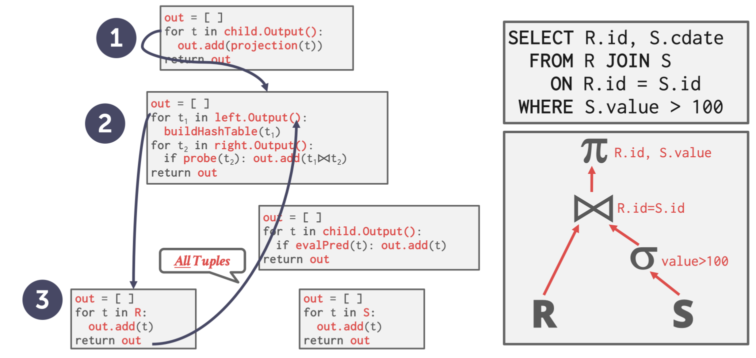
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

```
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(t1⋈t2)
return out
```

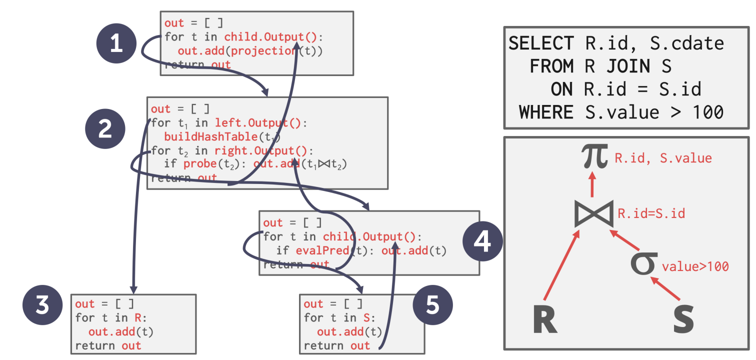
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

```
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(t1⋈t2)
return out
```

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

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

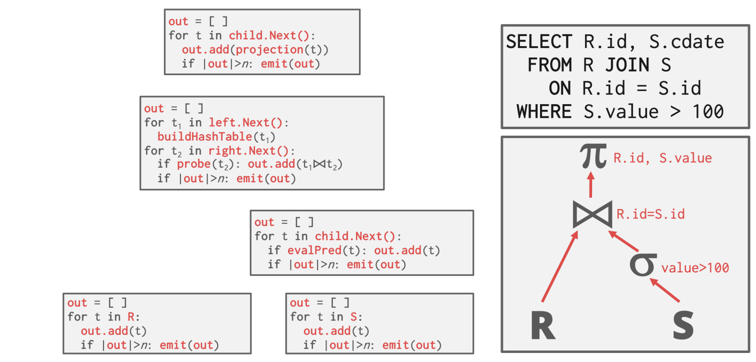
Materialization Model

- 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.
- **Examples:** MonetDB, VoltDB

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.
  - Useful in in-memory DBMSs (due to fewer function calls)
  - Useful in disk-centric DBMSs (due to fewer IO operations)

Vectorization Model



```
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)
```

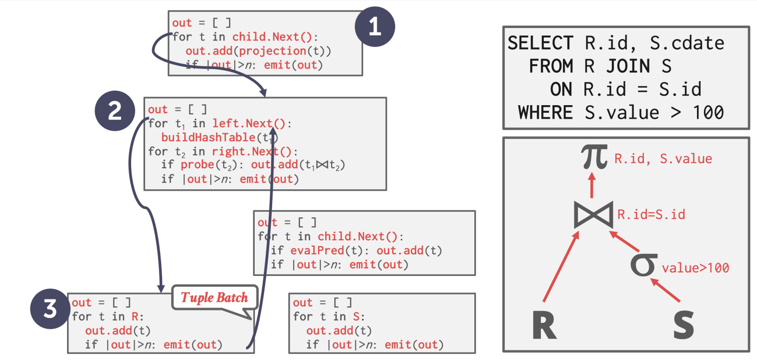
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

```
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)
```

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

Vectorization Model



```
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)
```

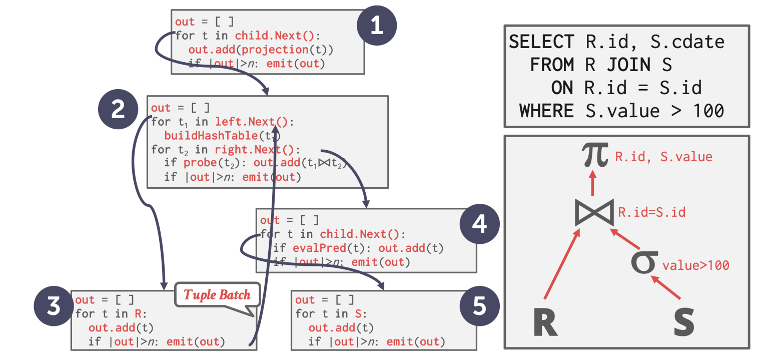
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

```
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)
```

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

Vectorization Model



```
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 child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

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

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

Vectorization Model

- 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.
- **Examples:** Vectorwise, Snowflake, SQL Server, Oracle, Amazon RedShift

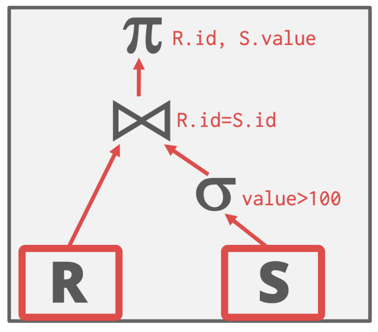
Plan Processing Direction

- **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

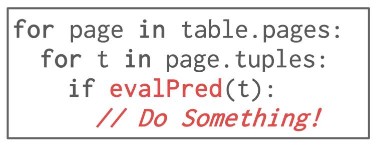
Access Methods

- An **access method** is a way that the DBMS can access the data stored in a table.
  - Located at the bottom of the query plan
  - 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.
  - Uses a buffer for materialization and vectorization processing models
- 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!
```

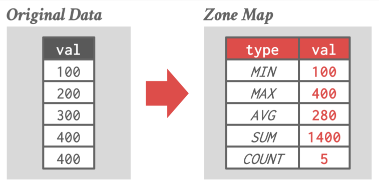
Sequential Scan: Optimizations

- This is almost always the worst thing that the DBMS can do to execute a query.
- Sequential Scan Optimizations:
  - Prefetching
  - Buffer Pool Bypass
  - Parallelization
  - 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.

```
SELECT *
  FROM R
  WHERE val > 600
```



Original Data

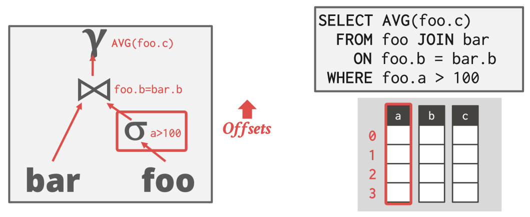
val
100
200
300
400
400

Zone Map

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5

Late Materialization

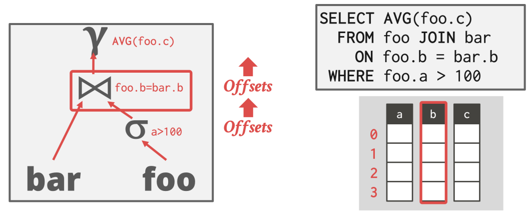
- DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



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

Late Materialization

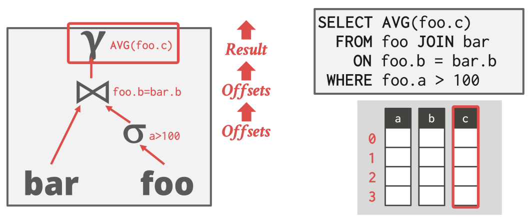
- DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



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

Late Materialization

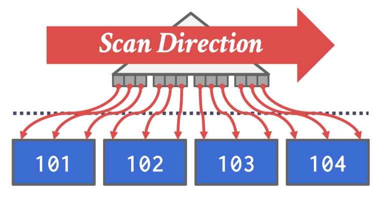
- DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



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

Heap Clustering

- Tuples are sorted in the heap's pages based on the order specified by the 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 **query optimizer** 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'
```

  - 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.

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**.

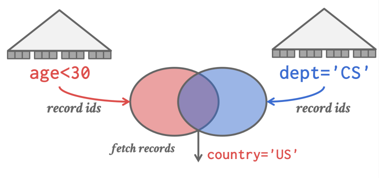
Multi-Index Scan

- With an index on 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'
```

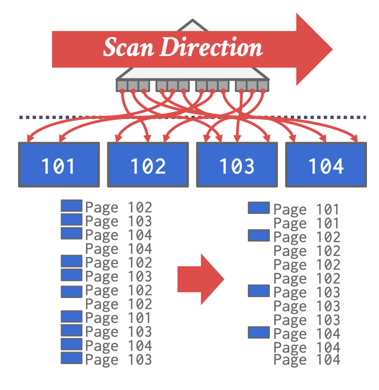
Multi-Index Scan

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



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.



Page 102
Page 103
Page 104
Page 104
Page 102
Page 103
Page 102
Page 102
Page 101
Page 103
Page 104
Page 103

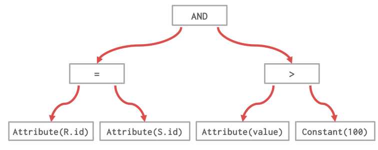
Page 101
Page 101
Page 102
Page 102
Page 102
Page 102
Page 103
Page 103
Page 103
Page 104
Page 104
Page 104

Expression Evaluation

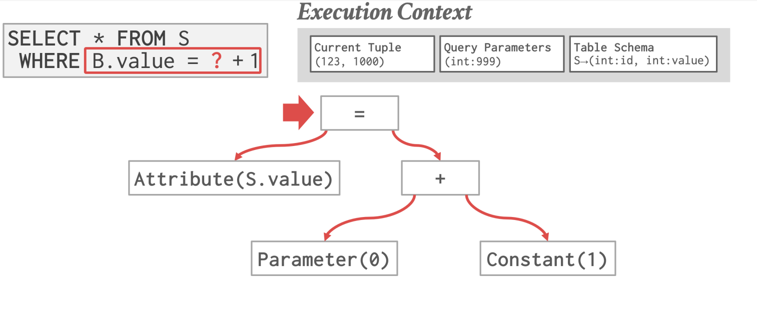
Expression Evaluation

- 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, S
  WHERE R.id = S.id
                   AND S.value > 100
```



Expression Evaluation



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

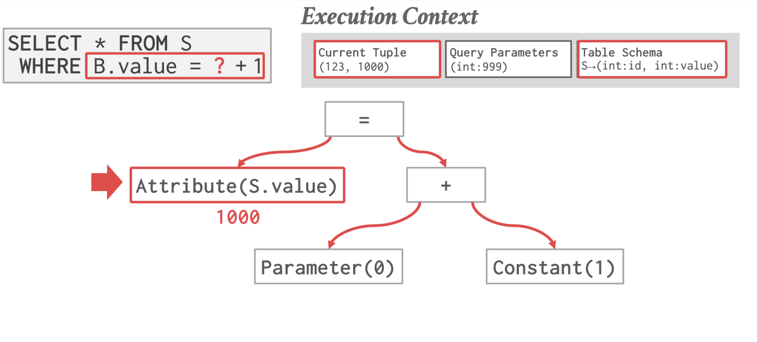
Execution Context

Current Tuple
(123, 1000)

Query Parameters
(int:999)

Table Schema
S→(int:id, int:value)

Expression Evaluation

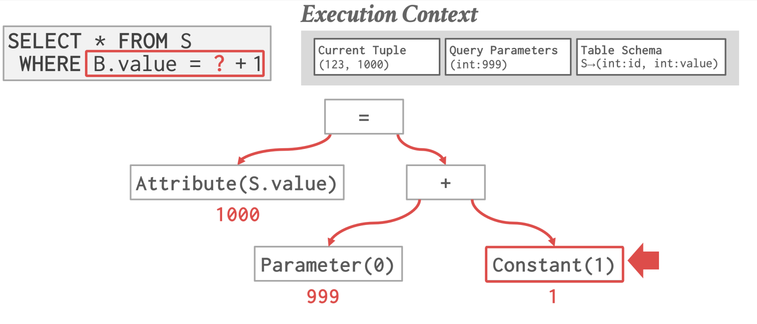


```
SELECT * FROM S
 WHERE B.value = ? + 1
```

Execution Context

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

Expression Evaluation



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

Execution Context

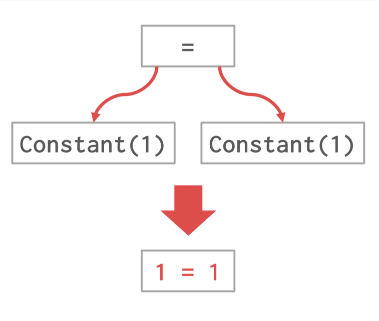
Current Tuple
(123, 1000)

Query Parameters
(int:999)

Table Schema
S→(int:id, int:value)

Expression Evaluation

- 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 Just-In-Time (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.

Retrospective

What did we learn

- You are tired of systems programming
- You are exhausted
- Let’s take a step back and think about what happened

Lessons learned

- Systems programming is hard
- Become a better programmer through the study of database systems internals
- Going forth, you should have a good understanding how systems work

Big Ideas

- Database systems are awesome – but are not magic.
- Elegant abstractions are magic.
- Declarativity enables usability and performance.
- Building systems software is more than hacking
- There are recurring motifs in systems programming.
- CS has an intellectual history and you can contribute.

What Next?

- We have barely scratched the surface. Follow-on course: CS 8803 (DBMS Implementation - Part II)
  - Query Optimization
  - Concurrency Control
  - Logging and Recovery Methods
  - Query Compilation + Vectorization
- Stay in touch
  - Tell me when this course helps you out with future courses (or jobs!)
  - Ask me cool DBMS questions

Parting Thoughts

- You have surmounted several challenges in this course.
- You make it all worthwhile.
- Please share your feedback via CIOS.