Lecture #12: Query Processing I

15-445/645 Database Systems (Fall 2020)
https://15445.courses.cs.cmu.edu/fall2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Query Plan

The DBMS converts a SQL statement into a query plan. Operators in the query plan are arranged in a tree. Data flows from the leaves of this tree towards the root. The output of the root node in the tree is the result of the query. Typically operators are binary (1–2 children). The same query plan can be executed in multiple ways. Most DBMSs will want to use an index scan as much as possible.

2 Processing Models

A DBMS *processing model* defines how the system executes a query plan. It specifies things like the direction in which the query plan is read in as well as what kind of data is passed between operators along the way. There are different models of processing models that have various trade-offs for different workloads.

These models can also be implemented to invoke the operators either from **top-to-bottom** or from **bottom-to-top**. Although the top-to-bottom approach is much more common, the bottom-to-top approach can allow for tighter control of caches/registers in pipelines.

Iterator Model

The *iterator model*, also known as the Volcano or Pipeline model, is the most common processing model and is used by almost every (row-based) DBMS. The iterator model allows for *pipelining* where the DBMS can process a tuple through as many operators as possible before having to retrieve the next tuple. The series of tasks performed for a given tuple in the query plan is called a *pipeline*.

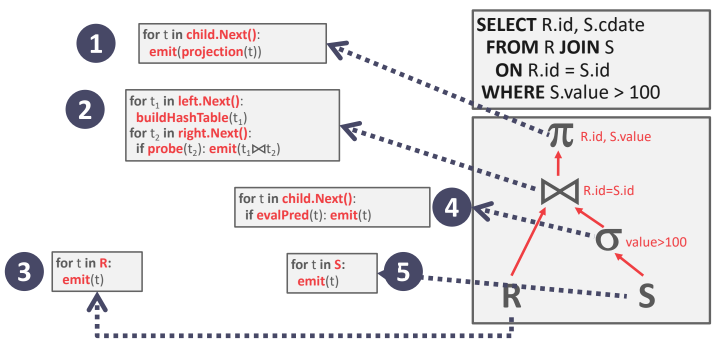
The iterator model works by implementing a `next` function for every operator in the database. Each node in the query plan calls `next` on its children until the leaf nodes are reached, which start emitting tuples up for processing. Each tuple is then processed up the plan as far as possible before the next tuple is retrieved. This is useful in disk-based systems because it allows us to fully use each tuple in memory before the next tuple or page is accessed. A sample diagram of the iterator model is shown in Figure 1.

Every query plan operator implements a `next` function as follows:

- On each call to `next`, 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. In this way, calling `next` on a parent calls `next` on its children. In response, the child node will return the next tuple that the parent must process.

Some operators will block until children emit all of their tuples (joins, subqueries, order by). These are known as *pipeline breakers*.

Output control works easily with this approach (`LIMIT`) because an operator can stop invoking next on its children operators once it has all the tuples that it requires.



```
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 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)
```

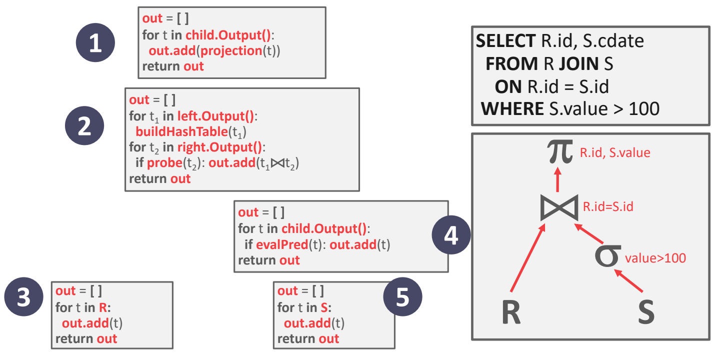
Figure 1: Iterator Model Example – Pseudo code of the different `next` functions for each of the operators. The `next` functions are essentially for loops iterating over the output of their child operator. For example, the root node calls next on its child, the join operator, which is an access method that loops over the relation R and emits a tuple up that is then operated on. After all tuples have been processed, a null pointer is sent that lets the parent nodes know to move on.

Materialization Model

The *materialization model* is a specialization of the iterator model where each operator processes its input all at once and then emits its output all at once. Instead of having a next function that returns a single tuple, each operator returns all of its tuples every time it is reached. To avoid scanning too many tuples, the DBMS can propagate down information about how many tuples are needed to subsequent operators. The operator "materializes" its output as a single result. The output can be either a whole tuple (NSM) or a subset of columns (DSM). A diagram of the materialization model is shown in Figure 2.

Every query plan operator implements an `output` function:

- The operator processes all the tuples from its children at once.
- The return result of this function is all the tuples that operator will ever emit. When the operator finishes executing, the DBMS never needs to return to it to retrieve more data.



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

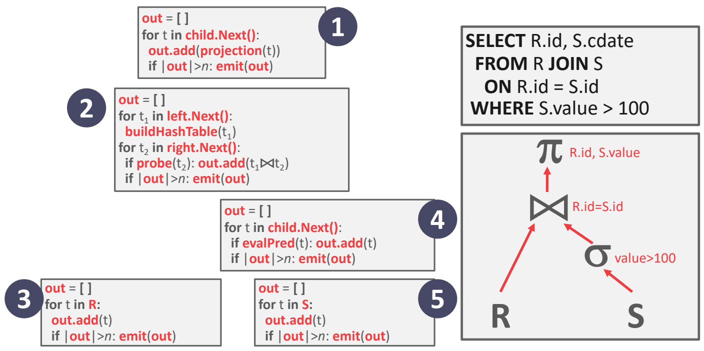
```
1
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out

2
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out

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

4
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out

5
out = [ ]
for t in S:
  out.add(t)
return out
```

Figure 2: Materialization Model Example – Starting at the root, the `child.Output()` function is called, which invokes the operators below, which returns all tuples back up.

This approach is better for OLTP workloads because queries typically only access a small number of tuples at a time. Thus, there are fewer function calls to retrieve tuples. The materialization model is not suited for OLAP queries with large intermediate results because the DBMS may have to spill those results to disk between operators.

Vectorization Model

Like the iterator model, the *vectorization model* has each operator implements a `next` function. However, each operator emits a *batch* (i.e., vector) of data instead of a single tuple. The operator's internal loop implementation is optimized for processing batches of data instead of a single item at a time. The size of the batch can vary based on hardware or query properties. See Figure 3 for an example of the vectorization model.



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

```
2
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
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

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

```
5
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
```

Figure 3: Vectorization Model Example – The vectorization model is very similar to the iterator model except at every operator, an output buffer is compared to the desired emission size. If the buffer is larger, then a tuple batch is sent up.

The vectorization model approach is ideal for OLAP queries that have to scan a large number of tuples because there are fewer invocations of the `next` function.

3 Access Methods

An *access method* is how the DBMS accesses the data stored in a table. In general, there are two approaches to access models; either data is read from an index or from a table with a sequential scan. The multi-index is an extension of the index scan that allows for multiple indexes to be accessed at once.

Sequential Scan

The sequential scan operator iterates over every page in the table and retrieves it from the buffer pool. As the scan iterates over all the tuples on each page, it evaluates the predicate to decide whether or not to emit the tuple to the next operator.

The DBMS maintains an internal cursor that tracks the last page/slot that it examined.

There are a number of optimizations available to help make sequential scans faster:

- **Prefetching:** Fetch the next few pages in advance so that the DBMS does not have to block when accessing each page.
- **Parallelization:** Execute the scan using multiple threads/processes in parallel.

- **Buffer Pool Bypass:** The scan operator stores pages that it fetches from disk in its local memory instead of the buffer pool in order to avoid sequential flooding.
- **Zone Map:** Pre-compute aggregations for each tuple attribute in a page. The DBMS can then decide whether it needs to access a page by checking its Zone Map first. The Zone Maps for each page are stored in separate pages and there are typically multiple entries in each Zone Map page. Thus, it is possible to reduce the total number of pages examined in a sequential scan. See figure 4 for an example of a Zone Map.
- **Late Materialization:** Each operator passes the minimal amount of information needed to the next operator (e.g., record id, offset to record in column). This is only useful in column-store systems (i.e., DSM).
- **Heap Clustering:** Tuples are stored in the heap pages using an order specified by a clustering index.

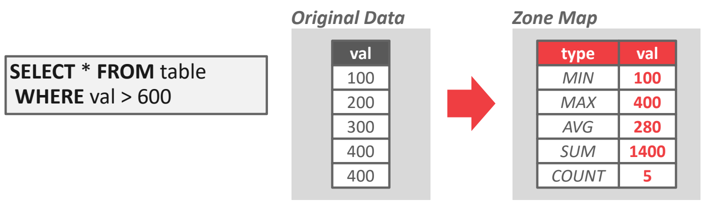


Figure 4: Zone Map Example – The zone map stores pre-computed aggregates for values in a page. In the example above, the select query realizes from the zone map that the max value in the original data is only 400. Then, instead of having to iterate through every tuple in the page, the query can avoid accessing the page at all since none of the values will be greater than 600.

Index Scan

The goal of an *index scan* is to identify an index in the table that will quickly allow the user to find the data he or she needs by avoiding useless operations.

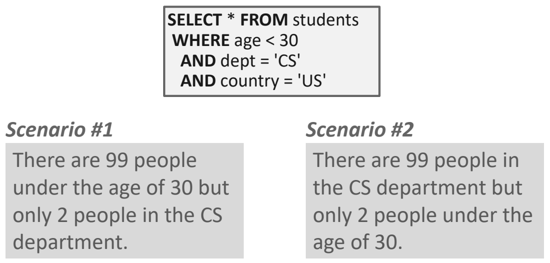

SELECT * FROM students
WHERE age < 30
AND dept = 'CS'
AND country = 'US'
Scenario #1
There are 99 people under the age of 30 but only 2 people in the CS department.
Scenario #2
There are 99 people in the CS department but only 2 people under the age of 30.

Figure 5: Index Scan Example – Consider a single table with 100 tuples and two indexes: `age` and `department`. In the first scenario, it is better to use the `department` index in the scan because it only has two tuples to match. Choosing the `age` index would not be much better than a simple sequential scan. In the second scenario, the `age` index would eliminate more unnecessary scans and is the optimal choice.

There are many factors involved in which index the DBMS decides to choose, such as:

- What attributes the index contains
- What attributes the query references
- The attribute's value domain
- Predicate composition
- Whether the index has unique or non-unique keys

A simple example of an index scan is shown in Figure 5.

More advanced DBMSs can support multi-index scans. When using multiple indexes for a query, the DBMS will compute sets of record IDs using each matching index, combine those sets based on the query's predicates, and retrieve the records and apply any predicates that may remain. The DBMS can use bitmaps, hash tables, or Bloom filters to compute record IDs through set intersection.

4 Modification Queries

Operators that modify the database (`INSERT`, `UPDATE`, `DELETE`) are responsible for checking constraints and updating indexes. For `UPDATE/DELETE`, child operators pass Record Ids for target tuples and must keep track of previously seen tuples.

There are two implementation choices on how to handle `INSERT` operators:

1. Materialize tuples inside of the operator.
2. Operator inserts any tuple passed in from child operators.

Halloween Problem

Halloween Problem is the anomaly where an update operation changes the physical location of a tuple, which causes a scan operator to visit the tuple multiple times.It can occur on clustered tables or index scans.

Originally discovered by IBM researchers while building **System R** in 1976.

5 Expression Evaluation

The DBMS represents a `WHERE` clause as an expression tree (see Figure 6 for an example). The nodes in the tree represent different expression types.

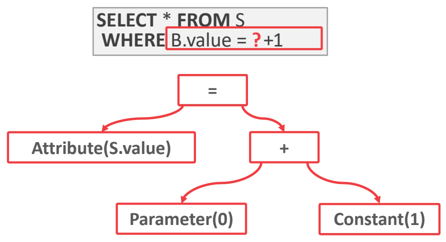


Figure 6: Expression Evaluation Example – A `WHERE` clause and a diagram of its corresponding expression.

Some examples of expression types that can be stored in tree nodes:

- Comparisons (=, <, >, !=)
- Conjunction (`AND`), Disjunction (`OR`)
- Arithmetic Operators (+, −, ∗, /, %)

- Constant and Parameter Values
- Tuple Attribute References

To evaluate an expression tree at runtime, the DBMS maintains a context handle that contains metadata for the execution, such as the current tuple, the parameters, and the table schema. The DBMS then walks the tree to evaluate its operators and produce a result.

Evaluating predicates in this manner is slow because the DBMS must traverse the entire tree and figure out what to do for each operator. A better approach is to just evaluate the expression directly.