Query Evaluation

UVic C SC 370

Dr. Daniel M. German
Department of Computer Science
University of Victoria

July 9, 2003 Version: 1.1.0

Overview

- What kind of info does the DBMS store in its catalog?
- How does the DBMS answer a query?
- What algorithms are used to perform a relational algebra operation?
- What are evaluation plans and how are they represented?
- Why do we want to get the **best** evaluation plan?

Assumptions

- We will use the following schema:

```
Sailors(sid: integer, sname: string, rating: integer, age: real)
Reserves(sid: integer, bid: integer, day: dates, rname: string)
```

- A page is 4k long
- The size of Reserves is 40 bytes long (100 tuples/page) and spawns 1000 pages
- The size of Sailors is 50 bytes long (80 tuples/page) and spawns 500 pages.

System Catalog (System Tables)

- Each database contains tables about the data contained in it.
  - Table: Name, attributes, indexes, integrity constraints
  - Index: Name, structure, search key
  - View: Name and definition
- And also about the DBMS itself:
  - Size of buffer pool, page size...

System Catalog...

- Also statistics:
  - **Cardinality**: Number of tuples in R: $NTuples(R)$
  - **Size**: Number of pages in R: $NPages(R)$
  - **Index Cardinality**: Number of distinct key values for index I: $NKeys(I)$
  - **Index Size**: Number of pages in index (for B-tree, number of leaf pages): $INPages(I)$
  - **Index Height**: The number of non-leaf levels in an index I: $IHeight(I)$
  - **Index Range**: Maximum and minimum values for the key of an index I: $ILow(I)$ and $IMax(I)$

Operator Evaluation

- Each relational operator has several alternative algorithms that implement it
- For many operators, none is universally better
- Several factors influence which algorithms performs best
  - Size of tables
  - Buffer pool
  - Buffer replacement policy

Three Common Techniques

- Indexing: For selection or join, use index
- Iteration: Examine each tuple
- Partitioning: Partition tuples on a sort key then work on smaller sets (sorting and hashing)

Access Paths

- An **access path** is a way of retrieving tuples from a table
- Two ways to do it:
  - Scan the file
  - Use an index, retrieve data (for some queries, we might not need to retrieve data)
- Every relational operator accepts one or more tables as input; the access paths have a big impact in their cost.

Access Paths...

- Consider a simple selection which is a **conjunction** of conditions of the form $attr$ **op** $value$ where the **op** is one of $<, \leq, =, \neq, \geq$
- This types of selections are called to be in **conjunctive normal form (CNF)** and each condition is a **conjunct**
- Why are queries in CNF useful?
- Intuitively, an index **matches** a selection condition if the index can be used to retrieve just the tuples that satisfy the condition.

Access Paths...

- A hash index **matches** a CNF selection if there is a term of the form $attribute = value$ for each attribute in the index's search key
- A tree index **matches** a CNF selection if there is a term of the form *attribute* **op** *value* for each attribute in a *prefix* of the index's search key:
  - If $\langle a \rangle$ and $\langle a, b \rangle$ are prefixes of key $\langle a, b, c \rangle$
  - For a tree index we can also evaluate comparisons different than equality , but not for a hash index
- For a table with an index we have 2 access paths (and potentially 3)

Selectivity of Access Paths

- The **selectivity** of an access path is the number of pages retrieved (index pages plus data pages) if we use this access path to retrieve the tuples
- The **most selective** access path is the one that retrieves the fewest pages
- The selectivity of an access path depends on its conjuncts
- Each conjunct acts as a filter
- The fraction of tuples that satisfy a given conjunct is called its **reduction factor**
- When there are several conjuncts, the fraction of tuples that satisfy all of them can be approximated by the product of their reduction factors (*when is this not true?*)

Example of Selectivity

- Assume we have a hash index H on Reserves with search key $\langle rname, bid, sid \rangle$
- We are given the CNF query: $rname =' Joe' \wedge bid = 5 \wedge sid = 3$
- The catalog contains the number of distinct keys for H: $NKeys(H)$, and the number of pages $NPages(Reserves)$, so we can approximate the reduction factor of this access plan:

$$\frac{NPages(Reserves)}{NKeys(H)}$$

Example of Selectivity...

- Assume now that we have an index on $\langle bid, sid\rangle$ and the CNF query is $bid = 5 \wedge sid = 3$
- If we know the number of different values for bid we can estimate the reduction factor of the first conjunct
- But usually, the DBMS does not, so it approximates to 0.1
- So we can **approximate** this query's selectivity as 0.01
- But the number of pages retrieved depends on whether the index is clustered or not

Example of Selectivity...

- What about a range condition like $day >' 8/9/2002'$?
- Assume a uniform distribution
- If we have a BTree on date the reduction factor is:

$$\frac{High(T) - value}{High(T) - Low(T)}$$

Algorithms for Selection

For a query $\sigma_{R.attr\ \mathbf{op}\ value(R)}$ we have the following alternatives:

- No index: scan
- Index, depends:
  - it is clustered or unclustered?
  - what is the **reduction factor** of the expression?
- Rule of thumb: for an unclustered index, if over **5%** of tuples are expected to match, then do scan

Projection

- Simple to implement, except when `DISTINCT` is used
- If no duplicates need to be eliminated:
  - Simple retrieve the tuples and eliminate unwanted columns
  - We might be able to do this with an index. *How?*
- If we need to drop duplicates, we need to sort the data
  - 1. Remove columns, sort, eliminate duplicates
  - 2. Remove columns **and** do first scan of sort, then keep doing sort, but in last pass of sort, eliminate duplicates

Projection with Indexes

- If we have an index with all the fields in the projection, then we only need to scan the leaf pages of the index
- If `DISTINCT` and all the attributes are a **prefix** to the key of a B-tree, then we don't have to scan the whole index:
  - Duplicates are adjacent!

Join

- It is a common and expensive operation
- For example: a join of $Reserves.sid = Sailors.sid$
- Suppose we have an index for Sailors on the sid column
- We can scan Reserves and find the matching Sailor.
- This algorithm is known as **index nested loops join**
- Assume we have a hash-based index using *Alternative 2* on sid of Sailors, and takes 1.2 I/Os on average to retrieve index entry.
  - There is one sailor per sid, hence one tuple to retrieve per reservation
  - Reserves is 1000 pages long (100 tuples per page)
  - Total cost: $2.2 * 10^5$ I/Os

Another Alternative: sort

- Sort both tables, then join
- Called **Sort-Merge Join**
- Assume we can sort them in 2 passes each
- Cost:
  - Sailors is 500 pages, Reserves is 1000
  - Cost of sorting: $4 * (500 + 1000) = 6000$
  - One more scan of sorted tables
  - Total: 7500 I/Os
- Cheaper, and data already sorted!

But sometimes index nested loops joins are desirable

- Suppose we only want the join for boat 101
- If the index is on bid, we don't have to read every boat.
- The decision of which join algorithm to use is based on the query as a whole, including selections and projections.

Group By and Set Operations

- Set operations require usually sorting the result, to eliminate duplicates
- Group-by is usually implemented with sorting also
  - Aggregates are implemented with in-memory counters
  - If there is a clustered index with the group-by attributes, it can be used to scan the table

Introduction to Query Optimization

- One of the **most** important tasks of a DBMS
- The same query can be expressed in many ways
- It makes it easy to write queries
- A given query can be evaluated in many ways, some cheaper than others (orders of magnitude difference)
- Good performance relies greatly in the quality of the **query optimizer**
- See figure 12.2

Optimizing Queries

- Queries can be seen as $\sigma, \pi, \bowtie$ algebra expression
- Optimizing an expression involves two basic steps:
  - **Enumerating** alternative plans for evaluating the expression
  - **Estimating the cost** of each plan
  - Choosing the plan with the **lowest** estimated cost

Query Evaluation Plans

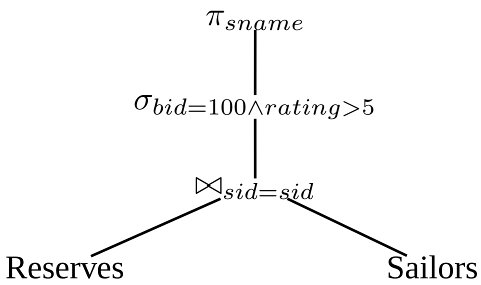
- A **query evaluation plan** (or simply **plan**) consists of an extended relational algebra tree, with additional annotations at each node indicating:
  - the **access methods** to use for each table
  - the **implementation** method

Query Evaluation Plan...

```
SELECT S.name FROM Reserves R, Sailors R WHERE
   R.sid = S.sid AND R.bid = 100 AND S.rating > 5
```

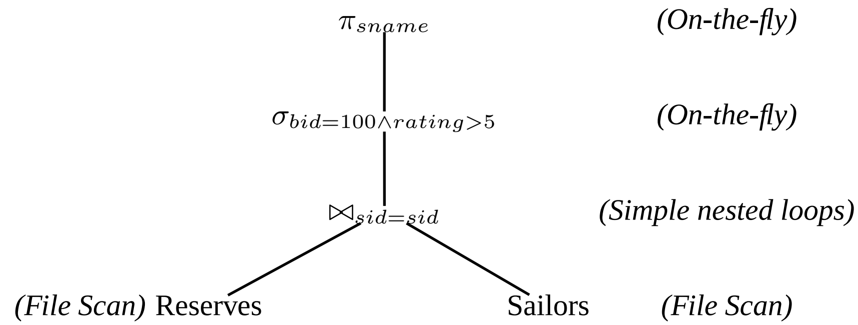
- This query can be expressed as:

  $\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$



Query Evaluation Plan...

- We also have to decide on the implementation of each operator

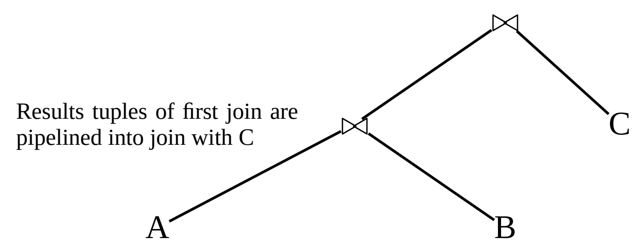


- This tree is a **query evaluation plan** for the SELECT
- Convention: the outer table is the left child of a join

Multi Operator Queries

- When the query involves several operators, sometimes the result of one is **pipelined** into the next
- In this case, no temporary relation is written to disk (**materialized**)
- The result is fed to the next operator as soon as it is available
- It is cheaper!
- When the input table to a unary operator is pipelined into it, we say it is applied **on-the-fly**

Pipelining



- Pipelining is a **control strategy**
- Results are produce one page at a time, used and then discarded

The Iterator Interface

- Once the **evaluation plan** is decided, it is executed by calling the operators in some order (possibly **interleaved**)
- Each operator has one or more inputs
- Passes result tuples to the next operator
- Materialization is usually done at the **input** stage of an operator
- When is it needed to **materialize**?
- Internally an operator has a uniform **iterator** interface:
  - **open**, **get_next**, **close**
  - It encapsulates materialization or on-the-fly processing
  - It also encapsulates use of indexes