ECE 464: Databases

Topic: Query Evaluation

This topic is based on Chapter 12 of the textbook, "Overview of Query Evaluation"

- http://www.csbio.unc.edu/mcmillan/Media/Comp521F10Lecture16.pdf
- http://users.cms.caltech.edu/~donnie/cs121/CS121Lec12.pdf
- http://pages.cs.wisc.edu/~paris/cs838-s16/lecture-notes/lecture10.pdf

Query evaluation refers to the activities of a *relational DBMS* in response to a query

Various strategies of executing the query are considered and evaluated, and one is chosen

SQL queries are translated into an extended form of *relational algebra*

Query evaluation plans are represented as trees of relational operators along with labels that identify the algorithm to use at each node (the leaves represent *instances* of *tables*)

Thus, relational operators are the building blocks for evaluating queries, and the implementations of these operators must be carefully optimized

Generally, algorithms for operators can be combined in various ways to execute a query

The process of finding a good evaluation plan is called **query optimization**

This topic starts with a discussion of how a DBMS describes the data that it manages, including tables and *indexes*

Also important is the **system catalog**, corresponding to a database is a collection of special tables storing *metadata* about the tables and indexes (more on this soon)

This chapter provides an overview of these topics, enough to move on to a future topic that discusses how to design and tune a database (Chapter 20)

Certain aspects of query evaluation (external sorting, implementations of operators, and a typical query optimizer) are discussed in more detail in Chapters 13 through 15, which we will skip

A number of example queries examined in this topic use the following schema:

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

These examples will also rely on the following assumptions:
- Each tuple of Reserves is 40 bytes long
- A page can hold about 100 Reserves tuples

- We have 1000 pages of such tuples
- Each tuple of Sailors is 50 bytes long
- A page can hold 80 Sailors tuples
- We have 500 pages of such tuples

Each *file* in a relational database stores either the tuples (*data records*) in a table or the entries (*data entries*) in an index

The collection of files corresponding to users' tables and their indexes represents the **data** in the database

Additionally, a relational DBMS maintains information about the tables and indexes it contains

This descriptive information is stored in a collection of special tables called *catalog tables*

The collection of catalog tables is called the **system catalog** (a.k.a. the **data dictionary**, or just the **catalog**)

The system catalog also stores some system-wide information, such as the size of the buffer pool, the page size, etc.

Additionally, the following information is stored about *tables*, *indexes*, and *views*:

- For each table:
    - The table name, file name (or some identifier), and file structure (e.g., heap file) of the file storing the table
    - The attribute name and type of each attribute
    - The index name of each index on the table
    - The integrity constraints on the table
- For each index:
    - The index name and structure (e.g., B+ tree) of the index
    - The search key attributes
- For each view:
    - The name of the view
    - The definition of the view

Statistical information about tables and indexes are also stored and updated periodically (but not every time a table is modified); such information might include:

- The number of tuples in each table
- The number of pages for each table
- The number of distinct key values for each index
- The number of pages for each index (only considering leaf pages for B+ trees)
- The number of non-leaf levels for each B+ tree index
- The minimum and maximum key values for each index

The catalog tables also contain information about *users*, such as account information and authorization information

An elegant aspect of relations DBMSs is that the system catalog is itself a collection of tables

For example, we might store information about attributes of tables in a catalog table called Attribute_Cat that can be defined as follows:

```
Attribute_Cat(attr_name: string, rel_name: string, type: string,
position: integer)
```

Fig 12.1 shows an example of an *instance* of the Attribute_Cat relation

Note that the catalog tables describe all the tables in the database, including the catalog tables themselves

At the implementation level, when the DBMS needs to find the schema of a catalog table, the code that retrieves this information must be handled specially

The catalog tables can be queried by users with the proper access

Real DBMSs vary in their catalog schema design, so we are just seeing an example

Several alternative algorithms are available for implementing each *relational operator*, and generally no single algorithm is universally superior

A few simple techniques that are used to develop algorithms for operators include:

- *Indexing*: If a selection or join condition is specified, use an index to examine just the tuples that satisfy the condition
- *Iteration*: Examine all tuples in an input table; if only a few fields are needed, and there is an index whose key contains all these fields, we can scan the index instead
- *Partitioning*: We can often decompose an operation into a less expensive collection of operations on partitions of a table

All of these techniques will be discussed in more detail

An **access path** is a way of retrieving tuples from a table

One possible access path involves scanning an entire file and keeping only tuples that match a selection condition

Another possibility involves using an index (still possibly requiring a selection condition)

Consider a simple selection expressed in *conjunctive normal form* (CNF), with each condition, or *conjunct*, of the form *attr* op *value*, and op is a comparison operator

CNF means that the clauses are strung together with AND (^)

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 (which could be a *composite search key*)

A B+ 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

For example, <a>, <a, b>, and <a, b, c> are all prefixes of key <a, b, c> (but not <a, c>)

If an index matches some subset of the conjuncts in a selection condition (in CNF), the conjuncts that the index matches are the *primary conjuncts*

The following examples will illustrate the notion of *access paths*

If we have a hash index H on the search key <rname, bid, sid>, we can retrieve Reserves tuples that satisfy the condition: `rname = 'Joe' ^ bid = 5 ^ sid = 3`

If the index were a B+ tree, we could also retrieve Reserves tuples that satisfy the condition: `rname = 'Joe' ^ bid = 5`; i.e., the index matches this condition

Let's say we have an index (either type) on the search key <bid, sid> and the selection condition `rname = 'Joe' ^ bid = 5 ^ sid = 3`

We can use the index to retrieve the tuples that satisfy `bid = 5 ^ sid = 3` (these are the primary conjuncts) and apply the additional condition to each retrieved tuple

Let's say we have one index on the search key <bid, sid> and also a B+ tree index on day, and the search condition is: `day < 9/9/2002 ^ bid = 5 ^ sid = 3`

We have a choice; we can use either index to retrieve Reserves tuples that satisfy the primary conjuncts for that index, and check the other conjuncts for each retrieved tuples

The **selectivity** of an access path is the number of pages retrieved (index pages plus table pages) if we use the access path to retrieve all desired tuples

The most selective access path is the one that retrieves the fewest pages

The fraction of tuples in a table that satisfy a given conjunct is called the *reduction factor*

When there are several primary conjuncts, we can estimate the fraction of tuples that satisfy all of them by multiplying the reduction factors of the conjuncts

This approximation treats the conjuncts as independent filters; that will not always be true in actuality

If the index is clustered, the fraction of tuples that satisfy a condition is also the fraction of pages that will be retrieved from the data file

Next we will discuss the evaluation algorithms for the main relational operators (discussed in more detail in Ch. 14, which we will skip)

The **selection** operation is a retrieval of tuples from a table, and its implementation is essentially covered by the discussion of access paths

If an index matches the complete selection condition, we can use it to retrieve the tuples

If an index matches certain conjuncts, we can use it to retrieve matching tuples and apply any remaining conditions to further restrict the result set

Assume that we have a selection on the Reserves table of the form `rname < 'C%'`

The book does not explain the notation, but I infer that it specifies the retrieval of all reservations with names that start with A or B, based on the rest of the analysis

They estimate that roughly 10% of the Reserves tuples are in the result (based on the assumption that names are uniformly distributed, and the sizes specified earlier)

Based on the assumptions mentioned at the start of the topic, this amounts to 10,000 retrieved tuples, or 100 pages

If we are using a clustered B+ tree index on the *rname* field of Reserves, we can retrieve the qualifying tuples with 100 I/Os (plus a few I/Os to traverse the tree)

If the index is unclustered, however, we could have up to 10,000 I/Os in the worst case

Here, as a rule of thumb, they say that it is probably cheaper to scan the entire table (instead of using an unclustered index) if over 5% of the tuples will be retrieved

My note: Earlier in the book, they seemed to suggest 10% as an appropriate cutoff

The **projection** operation requires us to drop certain fields of the input

The expensive aspect of the operation is to ensure that no duplicates appear in the result

If the DISTINCT keyword is not included in the SELECT clause, this step is skipped; we simply iterate through the table or an index whose key contains all the necessary fields

If we must remove duplicates, we typically must first obtain the fields in question from all tuples, then do a sort, and then eliminate duplicates

The text mentions *external sorting*, which they say here typically requires two or three passes; this is discussed in detail in Chapter 13, which we will skip

Joins are expensive operations and they are very common

Systems typically support several algorithms to carry out joins

Consider a join of Reserves and Sailors, with the join condition `Reserves.sid = Sailors.sid`

Suppose that the table Sailors has an index on the *sid* column

We can scan Reserves and, for each tuple, use the index to probe Sailors for matching tuples; this approach is called **index nested loops join**

Suppose we have a hash-based index using alternative (2) on the *sid* attribute of Sailors and that it takes about 1.2 I/Os on average to retrieve the appropriate page of the index

Since *sid* is a key for Sailors, there is at most one matching tuple; and *sid* in Reserves is a foreign key, so exactly one tuple in Sailors will match each tuple in Reserves

To scan Reserves, we need to retrieve 1000 pages

There are 100,000 tuples in Reserves, and for each, retrieving the index page containing the rid (record id) of the matching Sailors tuple costs 1.2 I/Os on average

The Sailors page containing the qualifying tuple also has to be retrieved

So this leads to 100,000 * (1 + 1.2) I/Os to retrieve matching Sailors tuples

The total cost is thus 221,000 I/Os

Now let's assume that there is no index that matches the join condition on either table, so we cannot use index nested loops (I add: or we can just ignore the index)

In that case, we can sort both tables on the join column, and then scan them to find matches; this is called **sort-merge join**

Assume we can sort Reserves and Sailors in two passes each (using an external sort), and each pass involves reading and writing every page

Thus, the sorting leads to 2 * 2 * 1000 = 4000 I/Os for Reserves and 2 * 2 * 500 = 2000 I/Os for Sailors

The second phase of the sort-merge join requires an additional scan of both tables (I add: read one page from each table at a time keep track of two pointers or indexes; they don't explain it)

Thus the total cost is 4000 + 2000 + 1000 + 500 = 7500 I/Os

Note that the cost of sort-merge join, which does not require a pre-existing index, is considerably lower than the cost of index nested loops join

Also, the result is sorted on the join column or columns

The book mentions that other join algorithms that also do not rely on existing indexes are often cheaper than index nested loops; they are discussed in Ch. 14 (which we will skip)

However, there are cases for which the index nested loops join is useful; the algorithm has the nice property that it is *incremental*

For the example, the cost of the join is incremental in the number of Reserves tuples that we process

If some additional selection in the query allows us to consider only a small subset of Reserves tuples, we may be able to avoid computing the join in its entirety

For example, if we want the result of the join for boat 101, and there are very few such reservations, we can, for each such Reserves tuple, probe Sailors, and we are done

If we use sort-merge join, on the other hand, we have to san the entire Sailors table at least once

Observe that the choice to use an index nested loops join is based on considering the query as a whole, including the extra selection on Reserves

This leads us to our next topic, **query optimization**

We will not be considering group-by clauses, aggregate operators (e.g., COUNT), intersection, union, etc; the book discusses these in Ch. 14 (which we will skip)

Good performance of an RDBMS relies greatly on the quality of *the query optimizer*

The optimizer generates alternative plans and chooses the one with the least estimated cost

The cost difference between the best and worst plans can be several orders of magnitude

A query is essentially treated as a $\sigma - \Pi - \bowtie$ (selection – projection – join) algebra expression, with the remaining operations (if any) carried out on the result

I add: The notation does not imply anything about the order of the operations

A **query evaluation plan** (or simply *plan*) consists of an extended relational algebra tree

Additional annotations at each node indicate the access methods to use for each table and the implementation method to use for each relational operator

Consider the following SQL query:

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

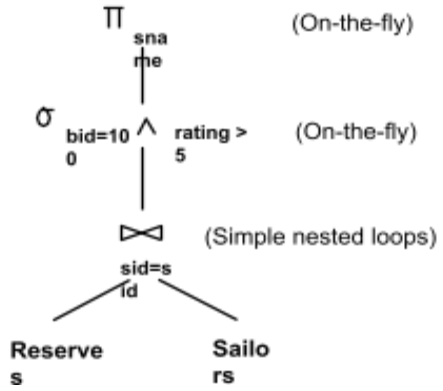
This query can be expressed in relational algebra as follows:

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

This expression partially specifies how to evaluate the query; we first compute the natural join of Reserves and Sailors, then perform the selections, and then project the *sname* field

Let's consider a simple nested loops join (*without an index*) with Reserves as the outer table; we can apply selections and projections to each tuple in the result as it is computed

This query plan is shown in tree form below (taken from Fig. 12.4):



This uses the convention that the outer table of a join is the left child of the operator

The "on-the-fly" annotations indicate that the result of the join is never stored in its entirety; the output of the join is *pipelined* into the selections and projects that follow

If the output of an operator is, alternatively, saved in a temporary table, we say that the tuples are *materialized*

Consider a more general example, involving a join of the form $(A \bowtie B) \bowtie C$

Both joins can be evaluated in a pipelined fashion using some version of a nested loops join; the book describes the process, which is more-or-less like a triple nested loop

The actual code that implements relational operators typically supports an *iterator* interface, thus hiding implementation details, and naturally supporting pipelining

Let's analyze more fully the proposed solution for the example, *ignoring the cost of writing out the final result* (which would be common to all algorithms)

The cost of the join is 1000 + 1000 * 500 = 501,000 page I/Os

I add: This clearly assumes that for each page of Reserves, you compare to each page of Sailors (as opposed to looping through Sailors for each tuple of Reserves)

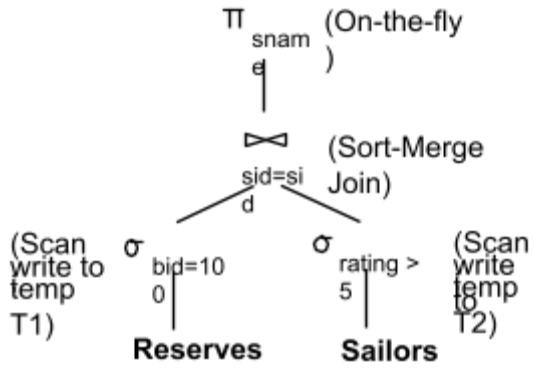
The selections and projections are done on the fly and do not incur additional I/Os, so this represents the total cost

Since joins are expensive, a good heuristic tries to reduce the sizes of the tables to be joined as much as possible

One approach is to apply selections early, if possible

In the example, *bid* = 100 involves only the attributes of Reserves, and the selection *rating* > 5 involves only the attributes of Sailors; they can be applied before the join

This query plan is shown in tree form below (taken from Fig. 12.6):



Let's assume that five buffer pages are available (which is a small number compared to what is likely in practice)

The cost of applying *bid* = 100 to Reserves is the cost of scanning Reserves (1000 pages) plus the cost of writing the temporary table, say T1

Let's assume there are 100 boats, and that reservations are uniformly spread across all boats; then we can estimate that the number of pages in T1 is 10

The cost of applying *rating* > 5 to Sailors is the cost of scanning Sailors (500) plus the cost of writing a temporary table, say T2

If we assume that ratings are uniformly distributed over the range 1 to 10, we can approximate the size of T2 as 250 pages

Let's also assume we do a straight-forward *sort-merge join* of T1 and T2

The book states that with five buffer pages available, we can sort T1 (which has 10 pages) in two passes (each pass requires reading and writing); the cost of this sort is 2 * 2 * 10 = 40 page I/Os

The book also states that we need four passes to sort T2 which has 250 pages; the cost is 2 * 4 * 250 = 2000 page I/Os

To merge the sorted versions, we need to scan both tables; the cost of this step is 10 + 250 = 260

The final projection is done on-the-fly, and we are ignoring the cost of writing the final result

The total cost is thus the cost of the selection (1000 + 10 + 500 + 250) and the cost of the join (40 + 2000 + 260), for a total of 4060 page I/Os

Sort-merge join is one of several join methods; suppose instead we use a *block nested loops join*

Using T1 as the outer table, for every three-page block of T1, we scan all of T2; thus we scan T2 four times (presumably, for the final scan, only one page of T1 would be used)

I add: They don't explain why we are using three-page blocks, but if we are assuming five buffer pages, I suppose we need one page for pieces of T2 and one for output

The cost of the join is therefore the cost of scanning T1 (10) plus the cost of scanning T2 four times (4 * 250 = 1000)

The cost of selection has not changed (1000 + 10 + 500 + 250), so the total cost is now 10 + 1000 + 1760 = 2770 page I/Os

A further refinement is to push the projection before the join, just like we did with the selections

As we scan Reserves and Sailors to do the selections, we can eliminate unwanted columns, reducing the sizes of temporary tables (in this case T1 and T2)

The reduction in the size of T1 is substantial, because only the *sid* field needs to be retained from Reserves

The *sid* and *sname* fields of Sailors are retained for T2

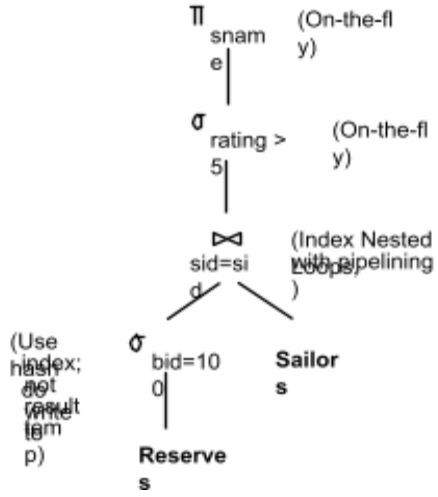
It is likely that T1 will now fit within three buffer pages, and we can perform a block nested loops join with a single scan of T2

The total cost would drop to about 2000 I/Os

If indexes are available on the Reserves and Sailors tables, better query evaluation plans may be available

Suppose we have a clustered hash index on the *bid* field of Reserves, and an unclustered hash index on the *sid* field of Sailors (two indexes in total, one for each table)

Then we can use the following query evaluation plan (taken from Fig. 12.7)



The selection *bid* = 100 is performed on Reserves using the hash index to retrieve only matching tuples

Assume, as before, that 100 boats are available and reservations are uniform

Then the estimated number of selected tuples is 100,000 / 100 = 1000

Since the index is clustered, these 1000 tuples will be adjacent, requiring 10 page I/Os

For each selected tuple, we retrieve the matching Sailors tuple (there will only be one, since *sid* is a primary key) using the other hash index

The selected Reserves tuples are not materialized and the join is pipelined

For each tuple in the result of the join, we perform the selection (*rating* > 5) and projection of *sname* on the fly

The selection of Reserves tuples costs 10 I/Os

There are 1000 Reserves tuples selected, and for each, the cost of finding the matching Sailors tuples requires 1.2 I/Os on average (expected for a hash index)

All remaining selections and projections are performed on the fly, so the total cost is 1210 I/Os

It is possible to refine this plan even further if the index on the *sid* field of Sailors is clustered (we won't do a full analysis here, but we will briefly talk about it conceptually)

Suppose we materialize the results of performing the selection *bid* = 100; this table contains 10 pages (in addition to the selection, we have used 20 page I/Os)

With five buffer pages, we can sort the temporary table (presumably according to *sid*) in two passes, requiring 2 * 2 * 10 = 40 I/Os

The sort can be even faster if the projection of *sid* is pushed before the sort, because the temporary table would be smaller (it could probably be sorted in one pass)

I add: The book does not explain the rest well; we can now loop in sorted order through the *sid* values of sailors who reserved the boat with *bid*=100 (some might occurring multiple times)

Since the hash index on the *sid* field of Sailors is clustered, we would not be retrieving pages from Sailors once per record; in fact, each page from Sailors would be retrieved at most once

At times, pushing selections and projections before a join can be even more striking

Consider this query (with one extra conjunct added to the condition involving *day*):

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND R.bid = 100 AND S.rating > 5 AND R.day =
'8/9/2002'
```

Assume further that *bid* and *day* form a candidate key for Reserves

First 10 page I/Os will be used to select Reserves tuples with *bid* = 100 (presumably still assuming a clustered index); then selection of the specified day will be applied on the fly

Assuming that *bid* and *day* form a key for Reserves, only one tuple will be selected, and a single retrieval from Sailors (using the hash) will take place

The total cost would be about 11 page I/Os

In contrast, if we modify the original plan to perform the additional selection on *day*, the cost remains at 501,000 page I/Os

The final section of the chapter briefly discusses how a query optimizer decides which alternative plans to consider (considering all possible plans is not feasible)

The section then briefly talks about how estimates of the cost of each plan are computed

We will see some of these topics again when we cover physical database design (Ch. 20)