Procesamiento de queries

CI-0127 Bases de Datos, Universidad de Costa Rica

Sivana Hamer

Importante

Este documento recopila contenidos de diversos de sitios web especializados, académicos y documentos compartidos por universidades. Toda la información es utilizada con fines estrictamente académicos. Para más información, puede ver las referencias.

1 Processing a query

DBMS are high-level query languages that are declarative in nature. High-level query languages indicate *what* results are expected but not *how* to get them. Meanwhile, in lower-level navigation languages (e.g., DML or the hierarchical DL/1), the programmer indicates *how* the query is executed. Thus, the query needs to be processed from a high-level query into an executable query plan.

The query execution process has the following steps shown in Fig. 1. An example of the inputs and outputs of each step can be seen in Fig. 2. In the following paragraphs, each step is described:

Requests a query. First, a query is requested from the application. The request is then received by the database to be processed and executed.

Validates the query. The query is validated syntactically and semantically. First, the query uses a *scanner* to identify the keywords, attributes name and relationships names. Then, the syntax is validated with a *parser*. Finally, the attributes and relationship names are validated by checking if the names being queried are related to the particular database queried. The result of this step is the generation of *query tree* or *query graph* that represents the query.

Selects the query. Using the representation of the query, the database must devise a strategy or plan to recover the information. Choosing the strategy is called *query optimization*. Not always the best strategy is selected, as it could be time consuming to calculate, thus it is more accurate to say that a good strategy is selected. There are two strategies for optimization, either heuristics (Section 3) or estimation (Section 5).

Generates the code. From the good execution plan, code is generated so it can be executed by a *code generator*.

Executes query code. The code is executed in a *runtime database processor* in either compiled or interpreted mode. The result is the tuples that satisfy the query or the error message.

2 Translating SQL queries to relational algebra

A SQL query from a RDMS is translated to the equivalent extended algebra expression, represented as a query tree. We can decompose every query into *query blocks* that contains a single **SELECT-FROM-WHERE-GROUP BY-HAVING** expression.

- A *nested subquery block* occurs when the result of an inner query is used by an outer block.

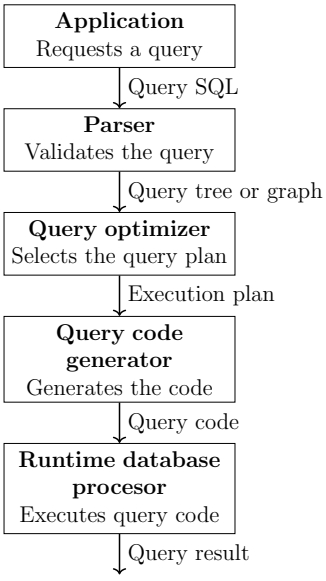

Figure 1: High-level query processing

- A *correlated nested subquery* occurs when a tuple variable from an outer block is used in the **WHERE** clause of the inner block. DBMSs use many techniques to unnest these queries.

Example: Get the name of the students with a higher admission grade than the student with id B66666.

Is separated into two query blocks:

- **Inner block.** Get the admission grade of the student with id B66666.

$$\pi_{AdmissionGrade}(\sigma_{id='B66666'}(STUDENT))$$

- **Outer block.** Get the name of the students with a higher admission grade than c.

$$\pi_{Fname,Lname}(\sigma_{AdmissionGrade>c}(STUDENT))$$

3 Heuristic optimization

A *heuristic* is a rule that works well in most cases, but is not guaranteed to work. The heuristics will improve the performance of the representation of a query (query tree or a query graph). As query graphs have only one representation per query, in practice query optimizers use query trees as operations have order.

The query is decomposed into *query blocks* (Section 2). Then, the *canonical tree* is generated, which is the initial query tree. This tree is optimized using relational algebra equivalence rules while using heuristics that preserve the equivalence of the tree. After finishing the transformation a *final query tree* is generated.

The outline of the algorithm to produce a more efficient tree to execute (in most cases) is described in the following steps.

1. We will break up any σ operations with boolean operations into a sequence of σ operations (Cascade of σ). Separating the σ allows for a higher degree of liberty to move operations in the tree.

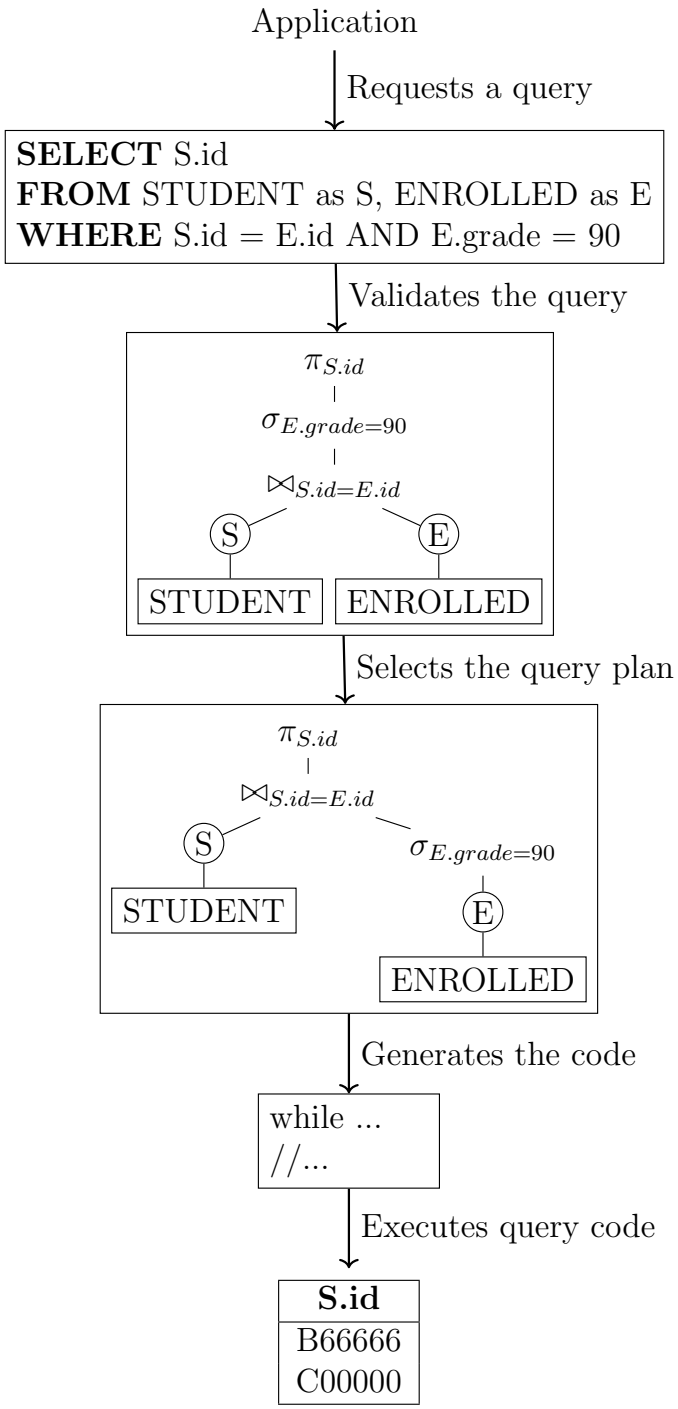

Figure 2: Example of the inputs and outputs of each step high-level query processing

```
SELECT Fname, Lname
FROM STUDENT
WHERE AdmissionGrade > (SELECT AdmissionGrade
                        FROM STUDENT
                        WHERE id = 'B66666');
```

```
SELECT AdmissionGrade
FROM STUDENT
WHERE id = 'B66666'
```

```
SELECT Fname, Lname
FROM STUDENT
WHERE AdmissionGrade > c
```

2. Move down σ operations as far down as it can go to be closer with the attributes of the relationship (Commutativity of σ, Commuting σ with π, Commutativity of σ with $\bowtie$ and $\times$, and Commutativity of σ with set operations).
   (a) If the condition involves only one table, it means it is a *select condition* and can be moved all the way to the leaf node of the table.
   (b) If it involves two tables, that means that it represents a *join condition* and can be moved above the location where the two tables are combined.
3. Rearrange the leaf nodes based on the following criteria (Commutativity $\bowtie$ and $\times$, and associativity of $\bowtie$, $\times$, $\cap$ and $\cup$).
   (a) Execute first the most restrictive σ, thus move the relationship before. The most restrictive σ is the one that produces the least amount of tuples, with the smallest absolute size, or the smallest selectivity.
   (b) Make sure that the order of the nodes does not cause $\times$. This may not be done if the previous relations had a select to a key field.
4. If there is $\times$ with a σ operation, combine it if it represents a $\bowtie$ condition (Converting a $(\sigma, \times)$ sequence into $\bowtie$).
5. Break down and move π as far down as possible. Create new π if needed. Only leave the π needed by each operation. (Cascade of π, Commuting σ with π, Commutativity of σ with $\bowtie$ and $\times$, and Commutativity of π with $\cup$).
6. Identify trees that can be represented by a single algorithm.

Example: Get the student IDs of those who have taken the 'CI-0127' course born after 1990

```
SELECT S.id
FROM STUDENT as S, ENROLLED as E, COURSE as C
WHERE C.sigla = 'CI-0127' AND S.id = E.id AND C.sigla = E.sigla AND S.Byear > 1990
```

$$\pi_{S.id}(\sigma_{C.sigla='CI-0127'ANDS.id=E.idANDC.sigla=E.siglaANDS.Byear>1990}(S \times E \times C))$$

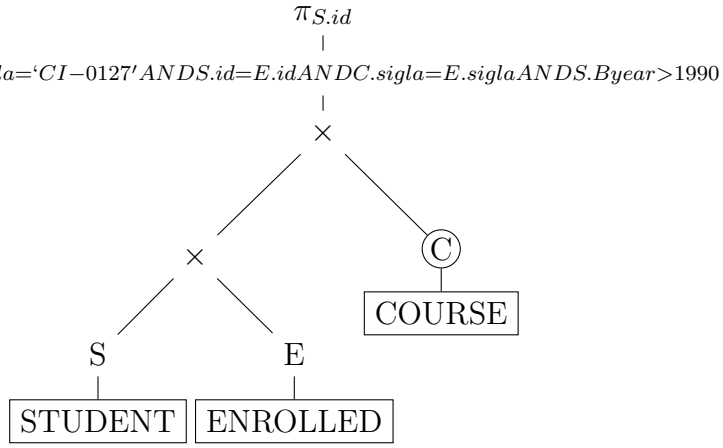

Figure 3: Canonical tree

4 Query execution plan

After creating the optimized execution tree (Section 3), for each operation an implementation strategy is chosen. There are two approaches for executing the query:

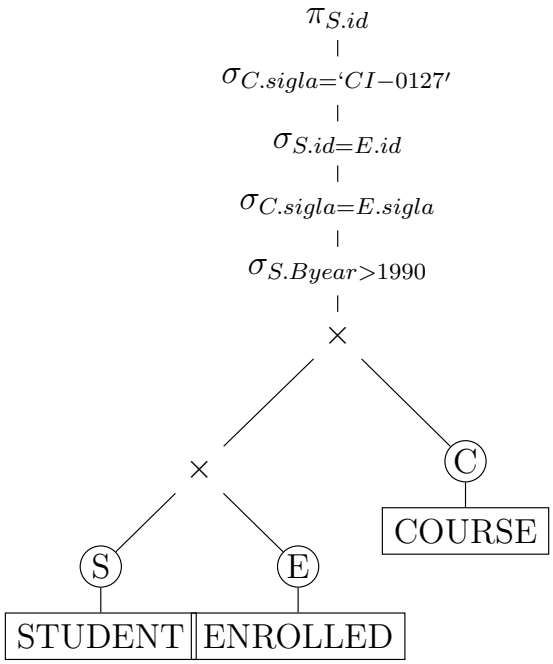

Figure 4: Query tree after breaking up σ operations

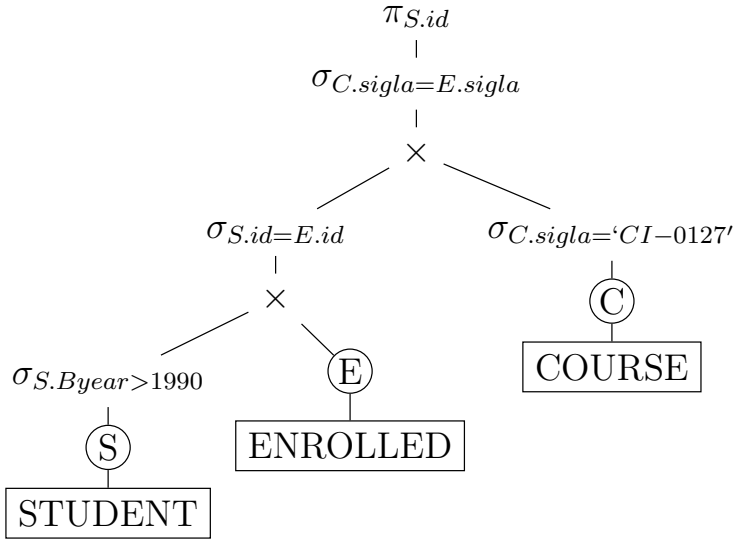

Figure 5: Query tree after moving down σ operations

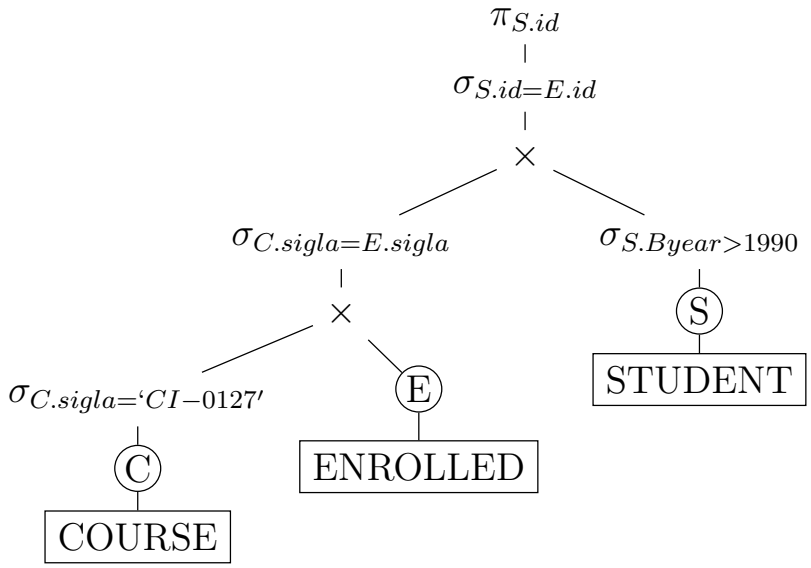

Figure 6: Query tree after selecting the most restrictive σ

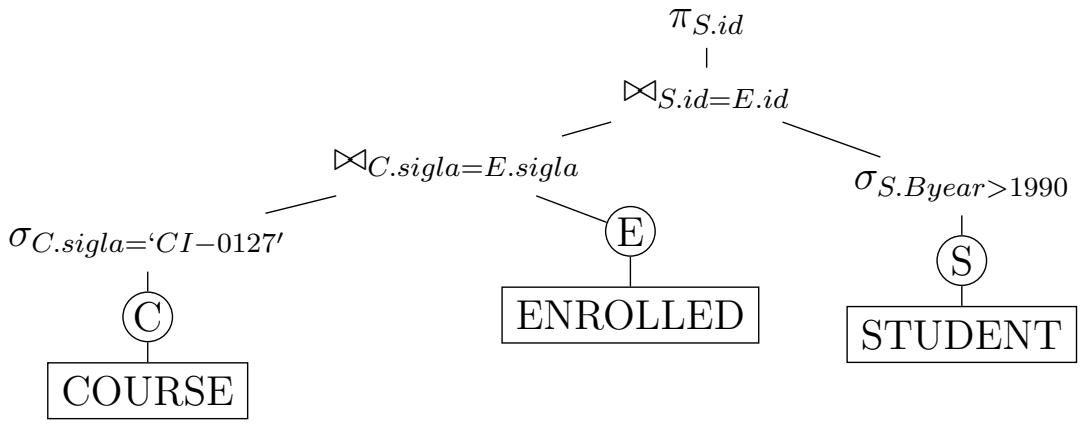

Figure 7: Query tree after converting to $\bowtie$

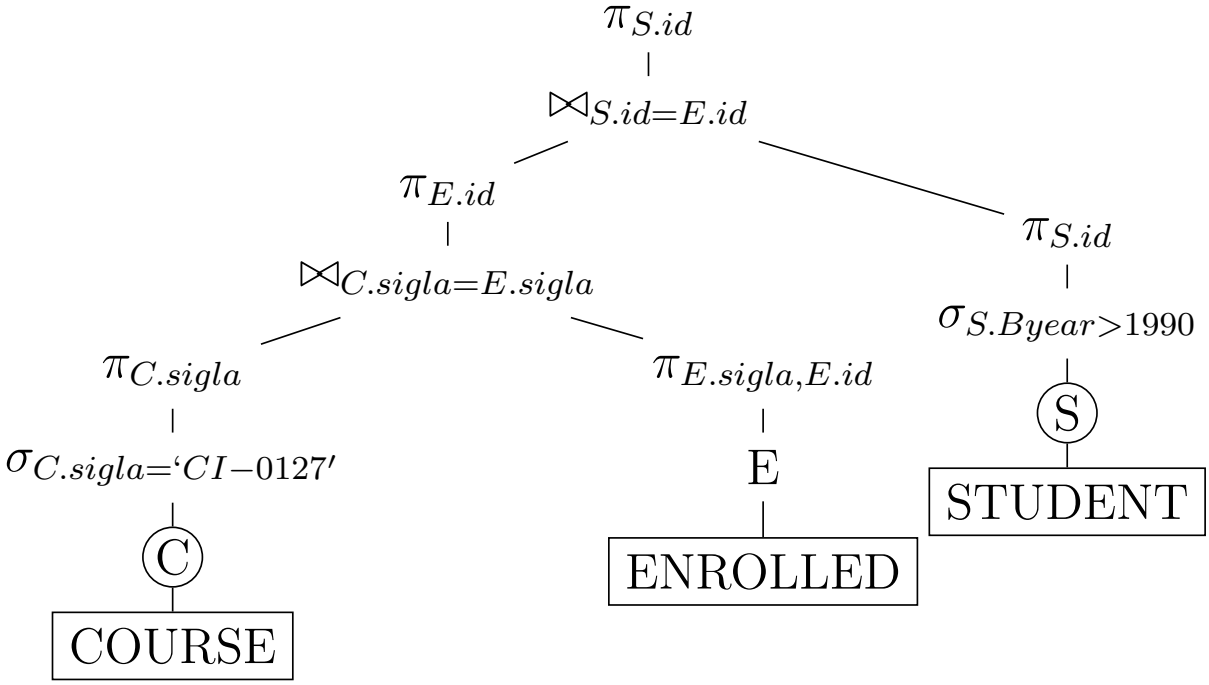

Figure 8: Query tree after moving down π

Materialized evaluation. The result of an operation is stored as a temporary relation. Thus, it is physically materialized.

Pipelined evaluation. The resulting tuples are forwarded directly to the next operation through a buffer. This has an advantage in cost for writing and reading from disk.

5 Cost-based query optimization

Another way to optimize involves *estimating the cost* of different executing strategies and choosing the plan with the *lowest cost estimate*. Not all strategies are considered as too much time will be spent estimating. This approach is more suited for compiled queries, optimizing at compile time, rather than interpreted queries that less time-consuming optimization works best. Traditional optimization techniques are used to search the *solution space* to find a solution that minimizes our cost functions (estimates).

The cost depends on several underlying metrics:

- **Access cost to secondary storage (a.k.a disk I/O cost).** Cost of reading and writing data blocks between main memory and secondary storage. This measures the number of block transfers. This dependinds the type of access structures and other factors such as the allocation of memory.
- **Disk storage cost.** Cost of storing in disk any intermediate files generated by an execution strategy for the query.
- **Computation cost (a.k.a CPU cost).** Cost of using the CPU such as to search, order and computing fields. This measures the CPU used, but it is tough to estimate.
- **Memory usage cost.** Cost of main memory buffers used during query execution.
- **Communication cost (a.k.a network cost).** Cost of shipping the query and the results from the database to the application that requests the query. This measures the number of messages sent.

For large databases there is an emphasis on minimizing disk I/O cost, meanwhile for smaller databases the emphasis is on minimizing CPU cost. For distributed databases the network cost is also important to consider. We tend to only use one single factor, the disk I/O cost, as it is difficult to assign suitable weights to each cost component.

6 Information cost functions

To estimate the cost of different execution strategies, we need to keep track of information for the cost functions. This is stored in the database catalog. The following information can be tracked:

- r_R: Number of tuples for R.
- b_R: Number of blocks occupied in storage for R.
- bfr_R: Number of tuples per block for R.
- $NDV(A, R)$: Number of distinct values of A in R.
- sl_A: Selectivity of attribute A for a condition c. $sl_A = \frac{s_A}{r_R}$
- s_A: Selection cardinality of A. $s_A = sl_A * r_R = \frac{r_R}{NDV(A,R)}$
- x_A: Number of index levels for A.
- b_{I1A}: Number of first level blocks of the index of A.

- n_B: Available buffer space in memory.

Some of the information changes rarely (e.g., x_R), while other information changes frequently (e.g, r_R). Thus, the optimizer will use similar but not exact up-to-date numbers for estimating the cost.

A *histogram* is a data structure (e.g., table) maintained by the DMBs that saves the data distribution. We can use them to help with estimating, as without it we would have to assume aspects about the data such as uniformity or independence that are probably not true. We can store the total tuples per attribute in a *bucket*, that describes the range of possible values. Some variations are used:

- An *equi-width* histogram divides the ranges of values into equal subranges.
- An *equi-height* or *equi-depth* histogram divides the values into buckets of similar size.

7 Sorting algorithms

While implementing the query, a common algorithm used is sorting as tuples in a data have no order. We use sorting for **ORDER BY**, sort-merge algorithms (e.g., $\bowtie$, $\cap$, $\cup$), and duplicate elimination (e.g, π).

If all the data fits in the DBMS main memory, we can use standard sorting algorithms (e.g., quicksort). While, *external sorting algorithms* are algorithms suitable for large files that can't fit entirely in main memory. Typically, an *external merge sort* strategy is used for external sorting. The data is split into seperate *runs*, and then combines them into merged sorted runs. The DBMS cache has a buffer space, with each buffer being able to store one disk block. The algorithm has two phases, explained in the following subsections.

Sorting phase

The *runs* (portions or pieces) that are available in the buffer are sorted using an internal algorithm. Then, the results are written back as temporary sorted runs.

Merging phase

The runs are merged with *merge pases*. The *degree of merging* (d_M) are the sorted subfiles that can be merged in each step with at least one buffer block used to save the result. $d_M = min(n_B - 1, n_R)$.

Cost

	Operators	Formula	Example
SOR	n_B = available buffer space b = number of file blocks n_R = number of initial runs	$n_R = \lceil b/n_B \rceil$	$b = 1024$ and $n_B = 5$ disk blocks $n_R = \lceil 1024/5 \rceil = 205$ runs with blocks of 5 except for the last run with 4 blocks.
MER	d_M = degree of merging m_P = merge pases	$m_P = \lceil log_{d_M}(n_R) \rceil$	$n_B = 5$, then $d_M = 4$ at a time $m_P = \lceil log_4(205) \rceil = 4$ pases

The performance of the algorithm can be measured in disk block accesses (either reads or writes) required. The following formula estimates the cost:

$$(2 * b) + (2 * b * (log_{d_M}(n_R))) = 2 * b * (log_{d_M}(n_R) + 1)$$

- $(2 * b)$ represents the number of blocks sorted requiring a read and write from and to memory.

- $(2 * b * (log_{d_M}(n_R)))$ represents the number of block accesses while merging. As each block is merged $(log_{d_M}(n_R)$ passes.
- The minimum requirements are $n_B = 3$, $d_M = 2$ and $n_R = \lceil b/3 \rceil$. The worst-case performance of the algorithm is $(2 \cdot b) + (2 * b * (log_2(\lceil b/3 \rceil)))$.

8 Algorithms SELECT operation

The SELECT operation has several algorithms, depending on the type of selection, to search a record in a disk file that satisfies a condition. *Files scans* search file records to find the tuples that satisfy a condition. *Index scans* search indexes to find the tuples that satisfy a condition.

- The optimizer should select the method that *retrieves the fewest records*.
- The *selectivity (sl)* is a ratio (results is between 0 and 1) that is the number of tuples that satisfy a condition to the number of tuples in the relation. 0 represents no records satisfying the condition, while a 1 represents all records.
- *Simple conditions* have only one condition. The methods used to implement the condition are $S1$ through $S7b$.
- *Conjunctive conditions* have multiple conditions separated by **AND**s. We first try methods $S8$, $S9$ or $S10$ for every condition. We can also use $S7a$ or $S7b$. If there are still remaining conditions, we will use $S1$. The form is the following.

$$\sigma_{c_1 AND \cdots AND c_n}(R)$$

- *Disjunctive conditions* have multiple conditions separated by **OR**s. If there is an access path for all the conditions we can use it, but if not we have to use $S1$. We can use any method from $S1$ to $S7$ for every simple condition. We have to remove duplicates. The form is the following.

$$\sigma_{c_1 OR \cdots OR c_n}(R)$$

In the following subsections, each SELECT operation algorithm with their cost is explained.

S1 - Linear search (brute force)

For every file record we test if the condition c is satisfied. Each block is loaded into the main memory to read. We can always use linear search, regardless of how the file is stored. The other algorithms are not always applicable, but are faster than linear search.

The cost for the worst case (e.g., record end key-attribute, non-key attribute selection or no record) we have to check all the b_R.

Cost: b_R

S2 - Binary search

If we are selecting with an equality comparison on an *ordered* file, we can binary search.

For binary search, we can find a key attribute in $log_2(b_R)$ time. Furthermore, if it is a non-key attribute then we might have to retrieve $\lceil (\frac{s_A}{bfr_R}) \rceil - 1$. Given that s_A records may satisfy the condition, with $\frac{s_A}{bfr_R}$ file blocks containing the selected records minus the first record retrieved.

Cost: $log_2(b_R) + \lceil (\frac{s_A}{bfr_R}) \rceil - 1$

S3a - Primary index

If we are selecting with an equality comparison on a *key attribute* with a primary index, we can use the index to find the unique record.

For every level (x_R) we have to check one disk block to get the results. Plus, one block to access the data from the file.

Cost: $x_R + 1$

S3b - Hash key

If we are selecting with an equality comparison on a *key attribute* with a hash key, we can use the key to find the unique record.

Only one disk block must be accessed to get the result in a normal hash.

Cost: 1

S4 - Multiple primary index

If we are selecting with a comparison condition $\{<, \leq, >, \geq\}$ on a *key attribute* with a primary index, we can find the records using the corresponding equality condition. Then, we can find the corresponding preceding or following records in the ordered file.

For every level (x_R) we have to check one disk block to get the results. Then, half of the file records (b_R) will satisfy the condition c that will be accessed.

Cost: $x_R + \frac{b_R}{2}$

S5 - Multiple clustering index

If we are selecting with an equality comparison on a *non-key attribute* with a clustering index, we can use the index to find all the records.

For every level (x_R) we have to check one disk block to get the results. Given that s_A records may satisfy the condition, with $\frac{s_A}{bfr_R}$ file blocks containing the selected records.

Cost: $x_R + \lceil(\frac{s_A}{bfr_R})\rceil$

S6 - B^+-tree index

If we are selecting a with a secondary B^+-tree index on an *equality comparison*, we can use the tree to find key or non-key values. We can also use a range of values for a range query.

For every level of the tree (x_R) we must check one disk block of the results, plus the reference to the disk block pointer. If the index is on a non-key attribute, s records will satisfy the condition c. If it is a range, it is $x_R + \frac{b_{I1A}}{2} + \frac{r}{2}$.

Cost: $x_R + 1 + s$

S7a - Bitmap index

If we are selecting with a condition based on *values of an attribute*, we can use the corresponding bitmaps with OR or AND operators to get the records.

S7b - Functional key

If the condition involves an *expression* with a functional index, we can use the index to retrieve the records.

S8 - Conjunctive individual index

If an attribute with a single simple condition satisfies the requirements for $S2$, $S3a$, $S3b$, $S4$, $S5$ or $S6$, apply the method and for the remaining simple conditions apply the remaining conjunctive select condition.

S9 - Conjunctive composite index

If we are selecting multiple attributes involved in an equality condition with a composite index or hash (a data structure with both fields combined), we use the additional data structure.

S10 - Conjunctive intersection record pointers

If we are selecting secondary indexes on more than one of the simple conditions with indexes of the *record pointers*, we can get the set of record points and intersect the pointers to gather the records.

The cost estimates for block transfers of selection algorithms $S1$ to $S6$ are shown in Table **??**. For the other algorithms, we can estimate the cost using these methods, depending on the one used.

9 Algorithms JOIN operation

The JOIN operation has several algorithms, but it is very time consuming. The following algorithms will provide an overview for EQUIJOIN (or NATURAL JOIN). *Two-way joins* are joins between two files, while if there are more files they are *multi-way joins*. We will only focus on two-way joins.

- The *join selectivity* (js) represents the number of tuples generated after the join. The following formula represents the js of joining the tables R and S with the resulting relationship saved in $T = R \bowtie_c S$.
$$js = \frac{r_T}{r_R * r_S} = \frac{1}{max(NDV(A,R), NDV(B,S))}$$
- A $js = 1$ is the same as applying $\times$.
- The *join cardinality* (jc) is the size of the resulting file after the join operation. The following formula represents the jc.
$$jc = js * r_R * r_S$$
- The results of writing the resulting cost file to disk is represented by the following formula.
$$\frac{js * r_R * r_S}{bfr_R S} = \frac{jc}{bfr_R S}$$
- The *join selection factor* is the fraction of records joined of the relationships.

In the following subsections, each JOIN operation algorithm with their cost is explained.

J1 - Nested-loop join (nested-block join)

For every record r in R, get all records s in S. Then, test $r[A] = s[B]$ for the attributes A of R and B of S. R is the *outer relation* in the *outer loop*, while S is the *inner relation* in the *inner loop*. We then gather the combined records that satisfy the condition.

First, we need to load the times of the outer loop b_R. Then, we need to load every block to memory at least once if they do not fit in memory thus we will perform $b_R * b_S$ accesses. With*more than three buffers* we can perform the operation more efficiently using buffers. It is advantageous to use the relation with fewer blocks as the outer-loop relation due to the effect of the outer-loop relation.
Cost: $b_R + (\lceil \frac{b_R}{n_B - 2} \rceil * b_S)$

Algorithm:

foreach block B_R in R:
  foreach block B_S in s:
    foreach tuple r in B_r:
      foreach tuple s in B_s:
        if $(s == r)$:
          save$(\ s, r)$

Example:

For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $b_S = 13$, $b_E = 2000$, and $n_B = 3$.

If we use $ENROLLED$ as the outer relation, then:

$$b_E + (b_E * b_S) = 2000 + (2000 * 13) = 28000$$

If we use $STUDENT$ as the outer relation, then:

$$b_S + (b_S * b_E) = 13 + (13 * 2000) = 26013$$

If we had $n_B = 10$ with $STUDENT$ as the outer relation, then:

$$b_S + (\lceil \frac{b_S}{n_B - 2} \rceil * b_E) = 13 + (\lceil \frac{13}{10 - 2} \rceil * 2000) = 13 + (\lceil \frac{13}{8} \rceil * 2000) = 13 + (2 * 2000) = 4013$$

J2 - Indexed based nested-loop join

If an index or hash key exists for A or B, then the attribute without the structure will be used as the outer loop while the inner structure with the access structure will be the inner loop. We then evaluate the records, gathering those that satisfy the condition.

If there is an index x_B for attribute B of relation S, we can retrieve every record in R and use the index for S. Using the smaller relation with the highest join factor should be more efficient. The cost depends on the type of index.

Cost:

- **Secondary index:** s_B is the selection for the join attribute of B in S.

$$b_R + *(r_R * (x_B + 1 + s_B))$$

- **Clustering index:** s_B is the selection cardinality of B.

$$b_R + *(r_R * (x_B + \frac{s_B}{bfr_B}))$$

- **Primary index:**

$$b_R + *(r_R * (x_B + 1))$$

- **Hask key:** h is the number of blocks estimated to retrieve a record. In linear and static hashing it is estimated as 1, while for extended hashing it is 2.

$$b_R + *(r_R * h))$$

Algorithm:

foreach tuple r in R:
    foreach tuple s in **Index** $(r_i = s_j)$:
        if $(s == r)$:
            save (s, r)

Example:

For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $STUDENT$ has a primary index with $X_{S.id} = 1$. Furthermore, $ENROLLED$ has a secondary index with $X_{E.id} = 2$ and $S_{E.id} = 80$. Also, $b_S = 13$, $b_E = 2000$, $r_E = 10000$, $r_S = 125$, and $n_B = 3$.

If we use $ENROLLED$ as the outer relation, then:

$$b_E + (r_E * (X_{S.id} + 1)) = 2000 + *(10000 * (1 + 1)) = 220000$$

If we use $STUDENT$ as the outer relation, then:

$$b_S + (r_S * (x_{E.id} + 1 + s_{E.id})) = 13 + (125 * (2 + 1 + 8)) = 10388$$

J3 - Sort-merge join

If both R and S are physically sorted by the attributes, we can sort both of the attributes A and B and iterate by joining the smallest of the two values.

If we need to sort each table, we must include this in the cost. If not, the only cost is the merging as we only need to scan once every table to compare the results.

Cost:

- **Sort:** This might not be needed. We use as an example the external merge sort strategy.

$$(2 * b) + (2 * b * (log_{d_M}(n_R))) = 2 * b * (log_{d_M}(n_R) + 1)$$

- **Merge:**

$$b_R + b_S$$

```
//Optional sorts
cursor_R =R.sort(A).cursor()
cursor_S =S.sort(B).cursor()
while (cursor_R AND cursor_S): //We stop if there are no more values
    if (cursor_R > cursor_S):
        cursor_S.next()
    elif (cursor_R < cursor_S):
        cursor_R.next()
    else:
        save(cursor_R, cursor_S)
        cursor_S.next()
```

- **Total:**

$$Sort + Merge$$

Algorithm:

Example:

For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $b_S = 13$, $b_E = 2000$, $r_E = 10000$, $r_S = 125$, and $n_B = 3$.

If the relations $STUDENT$ and $ENROLLED$ are sorted at the join attribute, then:

$$b_E + b_S = 2000 + 13 = 2013$$

If not then we must sort $STUDENT$ and $ENROLLED$ at the join attribute, thus with external merge sort:

$$dM_S = min(n_B - 1, b_S) = min(2, 13) = 2$$

$$Sort_S = 2 * b_S * (\lceil log_{dM_S}(\lceil \frac{b_S}{n_B} \rceil + 1) \rceil = 2 * 13 * (\lceil log_2(\lceil \frac{13}{3} \rceil)) \rceil + 1) = 26 * (\lceil log_2(5) \rceil + 1) = 104$$

$$dM_E = min(n_B - 1, b_E) = min(2, 2000) = 2$$

$$Sort_E = 2{*}b_E{*}(\lceil log_{dM_E}(\lceil \frac{b_E}{n_B} \rceil) \rceil{+}1) = 2{*}2000{*}(\lceil log_2(\lceil \frac{2000}{3} \rceil)) \rceil{+}1) = 4000{*}(\lceil log_2(667) \rceil{+}1) = 44000$$

$$b_E + b_S + Sort_E + Sort_S = 2000 + 13 + 104 + 44000 = 46117$$

J4 - Partition-hash join (hash join)

We first start with the *partitioning phase* by saving the partition of each file using a hash function h. We partition both relations. The hash key is the attribute used to join the relations. The records with the same $h(Z)$ value are saved in the same bucket. In the second *probing phase* we scan the other relation to and use the hash function for every join attribute to find an entry in the table with the matching tuple.

To partition both relations, we will require $2(b_R + b_S)$ accessed. We are required to read the blocks and then write them back hashed. Then, in the probing phase we must read $b_R + b_S$ blocks once more.

Cost: $3 * (b_R + b_S)$

Algorithm:

build hash_table HT_R for R
foreach tuple s in S
    if ($h(s)$ in HT_R):
        save (s, r)

Example:

For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $b_S = 13$, $b_E = 2000$ and $n_B = 3$. Thus:

$$3 * (b_R + b_S) = 3 * (2000 + 13) = 3 * 2013 = 6039$$

References

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems.* Pearson, 2016, vol. 7, chapter 19.

[2] A. Crotty and M. Li. Lecture #09 to lecture #14. [Online]. Available: https://15445.courses.cs.cmu.edu/fall2021/schedule.html

