An IO-based Cost Model for the Carquinyoli Genetic Optimizer *

Victor Muntés-Mulero\textsuperscript{1}, Josep Aguilar-Saborit\textsuperscript{1}, Calisto Zuzarte\textsuperscript{2}, and Josep-L. Larriba-Pey\textsuperscript{1}

\textsuperscript{1} DAMA-UPC, Computer Architecture Dept., Universitat Politècnica de Catalunya, Campus Nord UPC, C/Jordi Girona Mòdul D6 Despatx 117 08034 Barcelona, Spain, vmuntes, jaguilar, larri@ac.upc.edu
WWW home page: http://research.ac.upc.edu/DAC/DAMA-UPC
\textsuperscript{2} IBM Canada Ltd, IBM Toronto Lab., 8200 Warden Ave., Markham, Ontario, Canada L6G1C7 calisto@ca.ibm.com

Abstract. The Carquinyoli Genetic Optimizer (CGO) is a genetic query optimizer based on the idea of genetic programming. As any query optimizer, CGO needs a cost function to evaluate the fitness of a query execution plan (QEP). When comparing the costs of different plans, the optimizer uses a criteria to choose the best QEP by means of a cost function. This report presents the cost model used in CGO.

1 A brief introduction to CGO

The Carquinyoli Genetic Optimizer (CGO) is a sound and complete genetic optimizer based on the principles of genetic programming. Inspired in natural selection, CGO randomly creates a first initial population of query execution plans. Once the population is created, the evolution begins by applying different genetic operations iteratively. The behavior of a genetic programming system is out of the scope of this report. More information about CGO can be found in [6].

The structure of a query execution plan (QEP) in CGO corresponds to the typical structure usually assumed in the literature. A QEP is represented by a directed data flow graph where leaf nodes represent accesses to relations that produce a data flow and, the intermediate nodes, process and combine the data from their input nodes using physical implementations of the relational operations of PROJECT, JOIN, etc. The root node returns the results. The physical implementations of the relation operations used in the QEP are called plan operations and are described in more detail later.

CGO assumes that there is a previous module that parses the query and transforms declarative statements into a graph. This graph contains a vertex for each referenced relation and edges joining a pair of vertices when a join condition between attributes of these relations appear in the query statement.

\* Research supported by the IBM Toronto Lab Center for Advanced Studies and UPC Barcelona
1.1 Plan Operations

CGO includes the basic operation implementations [3], which are typically used in most commercial DBMSs.

- Scan Operations: CGO takes into account the two basic scan operations: sequential or table scan and index scan.
  - **Sequential scan**: this implementation will traverse the relation sequentially obtaining all the tuples. It can later apply some constraints discarding some of the tuples.
  - **Index scan**: when an index is created on an attribute of a table, we can access that table using this structure. Of course, the optimizer can always choose not to use the index even if it is available.

- Unary operations: this type of operations might be implemented with different objectives. CGO considers two different unary operations, namely Sort operations and Temp operations. The former is responsible of sorting and it is used every time we need to sort data during the query execution process. The latter is used to materialize an intermediate temporary result which, in some cases, is necessary to improve performance.

- Binary or join operations: CGO supposes that the DBMS engine has the three basic join implementations: Hybrid Hash Join, Nested Loop Join and Merge-Scan Join. The Hybrid Hash Join implementation considers the use of bit filters [1] to reduce the number of internal I/O accesses.

1.2 CGO internal structures

CGO represents relations or tables in a database using an internal structure called rel, the main fields of which are:

```c
typedef struct rel {
    unsigned int id;
    struct rel_info *rInfo;
    double card;
    double size;
    double size_pages;
    double tupleSize;
    lAttrib lAttr;
    ...
} rel;
```

Each relation has a list of attributes (represented by lAttrib) that contains the attributes or fields in that relation. Now, we show the simplified structure used to represent an attribute:
typedef struct attribute_s {
    struct attribute_info *aInfo;
    double unique_ratio;
    double selectivity;
} attribute;

The structure attribute_info contains the fixed information of the attribute, i.e. the name, the size, the type, etc. Also, each attribute has an associated unique ratio and selectivity. The unique ratio of an attribute \( a \) of a relation \( R \), \( u(a_R) \), ranges from \( \frac{1}{|R|} \) to 1 and represents the ratio of different values for \( a \) in \( R \) divided by the cardinality of \( R \), expressed as \( |R| \). Namely, \( u(a_R) = 1 \) means that \( a_R \) has not repeated values, while \( u(a_R) = \frac{1}{|R|} \) implies that \( a_R \) contains a unique values repeated \( |R| \) times. The selectivity represents the probability for a tuple to remain in the data flow without being discarded. We call selectivity of an attribute \( a_R \), \( s(a_R) \), the resulting selectivity of applying all the restrictions and join conditions affecting that attribute coming from relation \( R \). This value is very important, since in order to know the effects of joining attribute \( a_R \) with attribute \( b_S \) we must know, for instance, the percentage of values that where discarded from \( a_R \). As an example, suppose that we apply a restriction on \( a_R \) that discards the 40% of the values in \( a_R \). This implies that the restriction has a selectivity of 0.6. Suppose that \( b_S \) is a foreign key that points to \( a_R \). When we join \( a_R \) and \( b_S \), 40% of the values in \( b_S \) will not find a matching counterpart in \( a_R \) and, therefore, will be discarded\(^3\). For this reason, in order to be able to calculate the operation cardinalities, it is important to keep the selectivities that affect the different attributes when they ascend through the QEP structure.

2 Cost Model

The cost model used in CGO is a simple cost model based on that proposed in [7]. In order to simplify the cost functions, the cost model of CGO only takes into account the I/O accesses incurred by the evaluated QEP, ignoring the CPU cost associated to the operations.

2.1 General Assumptions

There are several general assumptions and restrictions that we must know before presenting the cost model:

- CGO assumes that data are not skewed and that attributes are orthogonal among them. Adding the capability of managing skewed data to CGO would imply the need for implement a more complex structure of catalogues, with specific information about the distribution of data. Since the catalogues do not belong to the optimizer domain, the creation of a complex catalogue schema is out of the scope of our work.

\(^3\) Note that we can assure that the 40% of the tuples containing \( b_S \) do not find a counterpart in the set of tuples containing \( a_R \) because we assumed that data are not skewed.
Currently, CGO does only allow equijoin conditions in the queries. This limitation does not respond to any restriction imposed by the genetic optimizer, but to our aim to simplify the whole system. It is important to note that, although possible, non-equijoin conditions are not used as frequently as equijoin conditions. For instance, the TPC-H benchmark, bases all the join conditions of its queries in equijoin conditions. Additionally, non-equijoin conditions, even assuming non-skewed data, require extra information from the catalogues, like the minimum and the maximum values of an attribute, the density of the values, etc.

CGO considers that two joining keys will always have a semantic relation defined through primary and foreign keys. Again, joining two semantically independent keys would require extra catalog information.

2.2 Basic definitions

In this subsection we introduce all the formalisms used to build the mathematical functions of the optimizer.

Relations. The information obtained for the user is always extracted from a set of relations in a database. We refer to the size of the relation in pages using $|R|$. In order to express the size in bytes of a record or tuple of a relation $R$ we use the expression $||R||$.

Query Execution Plan. Given a tree-shaped QEP called $T$, we define $\theta_T$ as the set of operations in the tree and $|\theta_T|$ as the cardinality of this set.

We call $T_o$ a QEP where the operation $o$ is the root operation of the binary tree structure.

Any set of operations $\theta$ is divided into three subsets $S$, $U$ or $J$. We call $S$ the set containing the scan operations. We call $U$ the subset containing the unary operations. Finally, we call $J$ the subset of binary or join operations. It is important to note that an operation $o$ can be included in only one of the subsets $S$, $U$ or $J$, since they are disjunct, $S \cap U = \emptyset$, $S \cap J = \emptyset$ and $U \cap J = \emptyset$.

Given an operation $o$, the number of child operations may vary from 0 to 2. For unary operations ($U$), like Sort or Temp operations, we call the unique child operation $o_c$. For each binary operation ($J$), we have two children operations $o_l$ (left child) and $o_r$ (right child).

Therefore, each operation $o$ in a QEP receives one or two input data flows and generates an output data flow. CGO explains the data flow characteristics by representing them as one or two input relations and an output relation. Despite the pipelined nature of the execution, we can refer to the number of tuples coming from a child operation as the cardinality of a data flow, for instance. Therefore, we say that a unary operation $o_1 \in U$ has an input relation, $R_{input}(o_1)$, and a binary operation $o_2 \in J$ has two input relations, $R_{left}(o_2)$ and $R_{right}(o_2)$. In the case of a scan operation $s \in S$ the input relation corresponds to a base relation in the database. The output relation of an operation $o$ is called $R_{output}(o)$. 
Nevertheless, specially with the operations in $\mathcal{J}$, the name of the input relations usually varies depending on the implementation used for the operation. Specifically, when we have a Nested Loop join operation $o$ we typically call the left child outer operation, $o_{outer}$, and the left subplan rooted by the $o_{outer}$ subplan, $T_{outer}$. Analogously, we use $o_{inner}$ and $T_{inner}$ for the right child operation of a Nested Loop join.

In the case of a Hash join, the right data flow is used to build the hash table and it is frequently referred as the build relation. The left child operation provides tuples which look for a counterpart in the hash table. This data flow is also called probe relation.

Any join operation $o$ has at least one join condition $j$ that fixes a criteria to decide whether two tuples, one from each input relation, must qualify and be projected to the upper operation. Typically, a join condition includes two attributes which, in turn, are called join attributes. The join attribute coming from the left side is called $a_{o,j}^l$ while the attribute coming from the right side is called $a_{o,j}^r$. Depending on the join implementation we will also see $a_{o,j}^o_{inner}$ or $a_{o,j}^o_{build}$ instead of $a_{o,j}^r$, and $a_{o,j}^o_{outer}$ or $a_{o,j}^o_{probe}$ instead of $a_{o,j}^l$.

We specify the implementation used by an operation using a superscript on the operation name. Thus, a scan operation $o$ can be expressed as $o_s$ or $o_i$ depending on the operation implementation (sequential and index respectively). Unary operations can be expressed as $o^t$ or $o^{sort}$ depending on whether they are Temp or Sort operations. Analogously we can express a join operation using $o^{\mathcal{J}}$, $o^{m_j}$ or $o^{h_j}$ depending on the join implementation.

Finally, any operation $o$ in $\mathcal{S}$ has an associated set $C(o)$. $C(o)$ is the set of constraints affecting the relation accessed by the scan operation $o$ and $s_c$ is the selectivity associated to this constraint, where $c$ is a constraint in $C(o)$.

**General Cost Formula.** Given a QEP $T$, each operation $o$ in $\theta_T$ has an associated cost $c_{op}(o)$ that depends on the number of tuples processed by the operation and the specific method used to perform that operation, which may be different for each possible implementation.

In general, the recursive formula used to calculate the cost of a QEP $T$ rooted by an operation $o$ would be as follows:

$$c(T_o) = \begin{cases} c_{op}(o) & \text{if } o \in \mathcal{S} \\ c_{op}(o) + c(T_{o_1}) & \text{if } o \in \mathcal{U} \\ c_{op}(o) + c(T_{o_1}) + c(T_{o_2}) & \text{if } o \in \mathcal{J} \end{cases}$$

### 3 Cardinality and selectivity

In the optimization problem we are not only concerned about the cost, but we have to calculate the cardinalities of the intermediate relations and the effect of the restrictions on the values in the different attributes. For this reason, it is very important to accurately calculate the output cardinality of every operation. The cardinality of an operation is independent from the operation implementation.

In section 1.2, we remarked the importance of keeping the selectivity that affects each attribute projected through the QEP structure. We show the relationship between this selectivity and the operations to obtain the cardinality.
3.1 Selectivity computation

We define the selectivity of a scan operation \( o \) as:

\[
s(o) = \prod_{c \in C(o)} s_c
\]

where \( C(o) \) is the set of constraints affecting the relation accessed by the scan operation \( o \).

A unary operation, in our case, does not modify the selectivity applied on the attributes since both, a Sort operation and a Temp operation, do not discard any tuples.

In the case of binary operations we first present the simple case where an operation has a single join condition and, later, the more complex case with multiple join conditions.

**Single Join Condition** In those cases where we only have a single join condition \( j \), the selectivity of an operation \( o \) is calculated as:

\[
s(o) = s(a_{o,j}^r)
\]

where \( a_{o,j}^r \) is the join attribute coming from the right input relation.

**Multiple Join Condition** CGO allows for cyclic query graphs and, in such occasions, the number of join conditions in some join operations is larger than one. Given an operation \( o \) with \( N \) join conditions \( j_1, ..., j_N \), its selectivity is calculated by:

\[
s(o) = \prod_{i=1}^{N} s(a_{o,j_i}^r)
\]

3.2 Cardinality computation

Once the selectivity is computed we must calculate the output cardinality.

In the case of a scan operation \( o \), the cardinality is always calculated as

\[
|R_{output}(o)| = s(o) \cdot |R| \quad o \in \mathcal{S}
\]

where \( R \) is the base relation scanned from the database.

For join operations, we first calculate the number of qualifying tuples as the product between the cardinality coming from the left relation and the selectivity of the operation, and then we divide it into the unique ratio of the join attribute coming from the right relation \( a_r \), which is equivalent to multiply by the number of repeated values per key in \( a_r \):

\[
|R_{output}(o)| = \frac{s(o) \cdot |R_l|}{u(a_r)} \quad o \in \mathcal{S}
\]

For the same reasons explained above, the output cardinality of a unary operation is not modified with respect to the input cardinality, \( |R_{output}(o)| = |R_{input}(o)| \).
4 Cost functions

In this section, we present the functions used by CGO in order to calculate the costs of the operations in a QEP.

Before entering in details, we must first define some global system parameters that we will eventually need for the formulae:

- $S_{\text{page}}$: size of a page in bytes.
- $M$: represents the memory of the system in pages reserved for each operation.

In the next subsections we analyze the formulae used per each operation type.

4.1 Scan operations

Sequential Scan (SS). As it was stated before, a sequential scan must go through all the tuples of the relation. Let us suppose a sequential scan operation $o^s$, $o^s \in S$, that scans a relation $R$, we define the cost of this operation as:

$$c_{\text{op}}(o^s) = ||R|| \quad o^s \in S$$

Additionally, our optimizer can consider a blocking technique in order to reduce the number of accesses to disk for sequential scans (in the DB2 UDB optimizer, this is known as sequential prefetch [4]). When this feature is activated the number of accesses to disk is divided by a constant $C_{\text{SP}}$ that represents the size of the block. In this case the cost would be computed as follows:

$$c_{\text{op}}(o^s) = \frac{||R||}{C_{\text{SP}}} \quad o^s \in S$$

Index Scan (IS). If we have an index created for an attribute of the relation to scan, we can choose to access through the index. We model our index like a B-tree where each node has $b$ elements. We assume that the leaves in the B-Tree are linked through pointers that allow sequential scans using the index structure.

For index scans CGO takes into account some of the different index properties: the possibility of pushing down predicates, different methods for accessing leaf pages, prefetching techniques, whether the index is clustered or not, etc.

While the cost of a Sequential Scan can be calculated easily (we do not need information from the parent node in the execution plan and we know that it will return all the tuples from the scanned relation), the cost of an Index Scan is more difficult to calculate because it is not always possible to figure out, without knowing the context where the scan is used in the QEP, the number of times the operation will be invoked by its parent node. For this reason, in some occasions, in the presence of an index scan, the operation above it in the QEP has the responsibility for computing the cost of the lower index scan.

Since the cost of an index scan depends on its situation in the QEP, we differentiate the behavior of an index scan using the following characteristics:
- **STANDALONE vs ONDEMAND**: an index operation can be considered *stand alone* or *on demand* depending on its relation with the upper operation. It is considered *stand alone* in those cases where the upper operation ignores the existence of an index scan. For instance, a Sort operation does not use the index structure since it must access all the tuples coming from the input relation in order to sort them. It does not have any need for an specific value. Another example would be a Hash join operation. The Hash join operation builds a hash table using all the joining values coming from the inner relation and, thus, it ignores whether an index scan is used or not in the lower operation that provides the inner tuples. The same happens in the outer side. An index scan is considered to be *on demand* when the upper operation has knowledge of the existence of an index scan and makes an explicit use of the advantages provided by that index. For instance, a Nested Loop join operation would clearly benefit from having an index scan on the joining attribute in the inner relation. The index scan would make it possible, for any tuple coming from the outer relation, to directly access all the tuples with the same value in the inner table, through the index meta-structure. Therefore, the number of accesses to the index in the latter situation depends exclusively on the upper operation and, consequently, on the cost of the operation.

- **SEQ vs NSEQ**: the access to the data through an index can be *sequential* or *not sequential*. By sequential we mean that we traverse the index leaves sequentially. On the contrary, not sequential means random accesses. In the first case we must take into account locality, in the second case the probability of exploiting some locality is clearly reduced.

- **ORDERED vs UNORDERED**: in those cases where an index scan is used *on demand* the required values can be ordered or unordered. Again, this may have an important impact on the cost due to the benefits provided by locality.

Next, we detail some possible scenarios typically present in real QEPs and the classification for the index scan that CGO uses in each situation.

- **An index scan providing tuples to a Hash join operation** (or under any other join operation implementation if the index attribute is not the joining key). In this situation we must differentiate between two possibilities: (i) if there exist a restriction on the index attribute, which would be the typical case, we consider the index scan *stand alone* and not sequential, since we only access those values satisfying the restriction; (ii) if there is not such restriction, which is also possible in those cases where we are not interested in any other attribute in the relation but the joining key, we consider the index scan *stand alone* and sequential.

- **A Merge-Scan join operation receiving tuples from an index scan** on the joining attribute would be considered an index scan *on demand*, sequential an ordered.

- **For a Nested Loop join**, the classification of the index scan would depend on the order of the tuples coming from the outer relation. If those tuples are ordered using the join condition it would be considered an index scan *on demand*, not sequential an ordered. If such an order does not exist then it would be considered *on demand*, not sequential and unordered.

Before analyzing each case separately, we must define some properties of the set of requested values, for the case of an index working *on demand*, i.e. when an upper join
condition explicitly benefits from an index scan. Specifically, we are interested in the selectivity and the unique ratio of the attribute coming from the opposite side in the join operation. In other words, let us suppose that a Nested Loop join operation \( \sigma_{A_i}, o \in J \), has two join attributes \( a_{o_{nl}}^i, j \) and \( a_{o_{nl}}^i, j \) and exists an index \( I \) created on \( a_{o_{r}}^j, j \). The Nested Loop join, as we will see later, can access the index using the values provided by \( a_{o_{nl}}^i, j \). However, the knowledge of some characteristics such as the selectivity of \( a_{o_{nl}}^i, j \) or its unique ratio, are necessary to calculate some probabilities related to the access to the index. For example, if the selectivity on \( a_{o_{nl}}^i, j \) is very low, then the probability of reutilizing the structures of \( I \) loaded into memory by previous accesses to the index is also very low.

Therefore, from the point of view of an index scan, we call \( V_{req} \) the ordered set of values requested to the index. \( V_{req} \) has an associated selectivity \( s(V_{req}) \) and a unique ratio \( u(V_{req}) \). From the point of view of the upper join operation \( V_{req} \equiv a_{o_{nl}}^i, j \) if the index scan is located on the right side of the operation.

We divide the calculation of the cost of an index scan into three phases. First, the cost to access the index structure. Second, the number of extra accesses associated to the accesses to the data in the relation. Finally, we calculate the total cost depending on the type of access to the index.

**Index Structure Accesses Cost.** Since our cost model only considers I/O accesses, the cost of the accesses to the index meta-structure must only be considered when the index does not fit in memory.

We call \( r_{NM}(I) \) the ratio of index \( I \) that does not fit in memory. This value can be easily calculated using the size of the index in pages and \( M \) which, as explained above, represents the memory in the system in pages reserved for each operation.

In those cases where the index is not accessed sequentially, the number of accesses per tuple depends on the number of levels in the B-tree structure minus one, \( l - 1 \), since CGO assumes that the root of all the indexes in the system is fixed in memory in order to reduce the number of I/O accesses.

Specifically, we use the following formula to calculate the cost per tuple access of an index scan operation \( o^i \) that uses an index \( I \) on an attribute \( a_{ind} \):

\[
c_{pt}^{(o^i)} = r_{NM}(I) \cdot (l - 1) \cdot s(a_{ind}) \quad o^i \in \mathcal{S}
\]

where \( s(a_{ind}) \) is the selectivity of the constraints on the attribute of the index. In those cases where there is not such a constraint, \( s(a_{ind}) = 1 \).

Additionally, if the index is classified as on demand and the accesses are ordered, two accesses on the same value do not double the number of accesses to disk, since the index structure necessary to access that value has already been loaded into memory. Also, if \( s(V_{req}) \) is close to 1, it is possible that two consecutive values, although being different, need to access the same nodes in the B-tree and, therefore, do not need to incur in extra I/O.

For the first case we use the following formula:

\[
c_{pt}^{(o^i)} = r_{NM}(I) \cdot (l - 1) \cdot s(a_{ind}) \cdot u(V_{req}) \quad o^i \in \mathcal{S}
\]
In order to take into account the second special situation, we should divide the results of (1) by \( s(V_{req}) \cdot b \).

However, if the leaves in the index are traversed sequentially, the index must not access all the intermediate nodes in the structure, but only preserve a pointer to the last accessed index leaf.

The cost of the operation in this case would depend on the type of access. If the index scan is *stand alone* we calculate the cost per tuple as follows:

\[
c_{pt}(o^j) = \frac{r_{NM}(I) \cdot u(a_{ind})}{b} \quad o^j \in S
\]

If the index scan is *on demand* we consider that it will always be ordered, since otherwise it would not correspond to any possible situation. In this case we calculate the cost per tuple using:

\[
c_{pt}(s) = \frac{r_{NM}(I) \cdot s(a_{ind}) \cdot u(V_{req})}{s(V_{req}) \cdot b} \quad o^j \in S
\]

**Fetch operation cost.** If necessary, after accessing the index structure, the index scan operation must access the data in the relation to obtain the requested information. Specifically, if we ask for other attributes in the scanned relation, different from the index attribute, we will need at least an extra I/O access.

It is necessary to calculate the number of I/O accesses to data per each access to an index. We must differentiate between clustered and non-clustered indexes, since the former may benefit from the spacial locality and, thus, may reduce the cost of the operation.

Given an index scan operation \( o^j \), if the index is clustered we calculate the number of data accesses per tuple \( (d_{pt}) \) as follows:

\[
d_{pt}(o^j) = \frac{||R_{tup}||}{u(a_{ind}) \cdot S_{page}} \quad o^j \in S
\]

since we know that tuples containing the same value in the indexed attributed are most probably located in the same physical page. \( R \) is the base relation read by the scan operation.

However, if the index is not clustered we cannot make this assumption. In this case, CGO calculates the average number of tuples per page containing repeated values in the indexed attribute. Therefore, the formula used is:

\[
d_{pt}(o^j) = \frac{u(a_{ind}) \cdot ||R_{tup}|| \cdot |R|}{S_{page} \cdot u(a_{ind})} = \frac{||R_{tup}|| \cdot |R|}{S_{page}} \quad o^j \in S
\]

If the average number of tuples with the same value in the indexed attribute is lower than 1, CGO considers that we have to execute as many extra accesses as repeated values per unique key.

Finally, if the requests on the index are ordered and repeated values are requested, it is probable that the data that has already been loaded into memory for the first request
is still present for the next request. We must calculate whether the requested tuples fit in the buffer pool. If they fit, then we must divide \( da_{pt}(o') \) by \( u(V_{req}) \).

Analogously, if the index is not \textit{on demand} and we perform a sequential access through the leaves is probably because there is a constraint on the indexed attribute. If this is the case, we must multiply \( da_{pt}(o') \) by the selectivity of the restriction in the scan operation.

**Index scan operation cost.** Once we have calculated these two intermediate values we are in a position to calculate the cost of the index scan operation.

If an index scan \( o' \) is \textit{stand alone} we calculate the cost function of the operation per tuple as:

\[
\text{c}_{op}(o') = |R| \cdot \left( c_{pt}(o') + da_{pt}(o') \right) \quad o' \in S
\]

On the other hand, if it is classified as an index scan \textit{on demand}, we return the cost by tuple and it will be the responsibility of the upper operation to multiply the cost of the index scan by the number of times that the index scan operation is invoked. Thus the cost formula per tuple would be:

\[
\text{c}_{op}(o') = c_{pt}(o') + da_{pt}(o') \quad o' \in S
\]

### 4.2 Unary operations

CGO considers two clearly different kinds of unary operations: Sort and Temp operations. In this subsection we detail the formulae used to cost these operations.

**Sort operations.** Given a Sort operation \( o^{sort} \), the number of I/O accesses incurred by the operation depends on the amount of data that fits in memory. We calculate the cost of this operation as follows:

\[
\text{c}_{op}(o^{sort}) = 2 \cdot (|R_{input}(o^{sort})| \cdot |R - M|) \quad o^{sort} \in U
\]

We multiply by 2 since all the data spilled to disk must be written first and read later from the physical device.

**Temp operations.** The formula presented in section 2.2 defines the general method to obtain the cost associated to an execution plan. However, there is an exception for this general formula. A Temp operation materializes an intermediate result in order to avoid the recalculation costs. This implies that, the first time we execute the Temp operation, the cost will be much higher than the cost of later executions.

For this special situation we define the cost of a Temp operation \( o' \), \( c_{op}(o') \), as the cost of reading the materialized results, \( c_{op}(o') = ||R_{output}(o')|| \). And we define an extra cost function \( c_{temp}(o') \) that refers to the cost of creating the materialized result. We formalize this cost function:

\[
\text{c}_{temp}(o') = ||R_{output}(o')|| + c(T_{o'}) \quad o' \in U
\]

Equation (2) defines the cost of materializing an intermediate result as the cost of calculating the lower subplan \( T_{o'} \) plus the cost of saving the resulting relation to disk.
4.3 Binary operations

In section 1.1 we enumerated the three join implementations considered in CGO. In this section we study the cost formulae used to calculate the cost of each join operation.

Nested Loop Join (NL). In the classic Nested Loop join the input relations are typically called outer and inner relations. The basic algorithm gets a tuple from the outer relation and then traverses the whole inner relation. In general, the computational cost of this algorithm is $O(|R_o| \cdot |R_i|)$.

As we stated before, the cost associated to a Nested Loop join operation is vastly affected by the implementation of the lower operations. For example, an index scan used to read the tuples coming from the inner relation of a Nested Loop join may improve drastically the cost of the join operation. CGO also assumes that, except in the presence of indexes, the inner relation is materialized. If this was not the case, we should recalculate the results of the inner plan for every tuple in $R_o$. Therefore, we must study the formulae depending on different scenarios: a Nested Loop join without feeded by single sequential scans or other join operations, a Nested Loop join with index scans and a Nested Loops join with an lower Temp operation on the inner relation.

We first present the cost formula corresponding to a classical Nested Loop join operation $o^{nl}$, when neither index scans on the joining attribute nor Temp operations are used in the inner relation:

$$c(T_{o^{nl}}) = c(T_{o^{nl}_{outer}}) + |R_{outer}(o^{nl})| \cdot c(T_{o^{nl}_{inner}})$$

Although this formula depicts the main idea behind a Nested Loop operation, we must say that a Nested Loop is usually not used for equijoins if we do not have an index on the inner side, or at least a materialization of the results coming from the inner side. In fact, CGO forces a Temp operation if this situation occurs in a QEP and the cost of calculating the subplan is larger than the cost of reading the materialized results.

Therefore, the following formula depicts the case when the data qualifying from the inner side is materialized, i.e. a Temp operation is used.

$$c(T_{o^{nl}}) = c(T_{o^{nl}_{outer}}) + |R_{outer}(o^{nl})| \cdot c(T_{o^{nl}_{inner}}) + c_{temp}(o^{nl}_{inner})$$

In other words, we only repeat the read of the materialized relation as many as $|R_o|$ times, but we do not have the need to recalculate the subplan.

However, CGO also considers the existence of a buffer pool in order to reutilize data and reduce the number of I/O accesses. If the amount of materialized data is small and it fits in memory, then we calculate the cost of a Nested Loop join operation as follows:

$$c(T_{o^{nl}}) = c(T_{o^{nl}_{outer}}) + c(T_{o^{nl}_{inner}}) + c_{temp}(o^{nl}_{inner})$$

In the presence of indexes on the join attribute in the inner relation of the join, we also use equation (3), although $c(T_{o^{nl}_{inner}})$ does not reflect the cost of returning all the tuples, but the average cost of returning a single tuple, as shown in subsection 4.1.
Merge-Scan Join (MS). Similarly to the case of the Nested Loop join, the cost of a Merge-Scan join operation is affected by the implementation used in lower operations. A Merge-Scan Join scans two sorted relations and then merges them. Without further improvements, this join implementation can only handle equijoins.

In the absence of indexes, the formula used to calculate the cost in a Merge-Scan join operation $o_{ms}^{m}$ is:

$$c(T_{o_{ms}^{m}}) = c(T_{o_{ms}^{l}}) + c(T_{o_{ms}^{r}}) \quad o_{ms}^{m} \in J$$

When one of the two lower operations in a Merge-Scan join is an index scan with an index on the join attribute, analogously to the Nested Loop join case, we must take special considerations in order to calculate the cost. We use the following equations:

$$c(T_{o_{ms}^{m}}) = \begin{cases} c(T_{l}) + |R_{l}(o_{ms}^{m})| \cdot c(T_{o_{ms}^{r}}) & \text{if } o_{l}^{m} \in J \\ c(T_{r}) + |R_{r}(o_{ms}^{m})| \cdot c(T_{o_{ms}^{l}}) & \text{if } o_{l}^{m} \in J \end{cases}$$

where $o_{ms}^{m} \in J$.

When two indexes are available, one on the left child and the other one on the right child of the Merge-Scan join operation, this operation, in CGO, only benefits from one of them. Namely, one of the input relations is used to get the values to be joined and the other relation is accessed through the index to get the potential partners. For this reason we calculate both the cost using the left index and the cost using the right index, and choose the lower costed case.

Hash Join (HJ). The last implementation we will consider in this model is the Hash join. In the case of a Hash join operation, the implementation of the lower operations is not taken into account in order to calculate the cost of the operation. However, since the Hash join needs to build a hash table in memory, it is possible that the amount of memory available in the system for this operation is not enough to fit the data structure. This implies that, following the typical methodology proposed for the Hybrid Hash Join in [3], the join operation will incur in extra I/O.

CGO calculates the cost of a Hash join operation $o_{hj}^{hj}$ as follows:

$$c_{op}(o_{hj}^{hj}) = 2 \cdot \left( ||R_{build}(o_{hj}^{hj})|| - M + (1 - \frac{M}{||R_{build}(o_{hj}^{hj})||}) \cdot ||R_{probe}(o_{hj}^{hj})|| \right) \quad o_{hj}^{hj} \in J$$

Therefore, the cost of an execution plan with a Hash join operation $o_{hj}^{hj}$ in its root is:

$$c(T_{o_{hj}^{hj}}) = c_{op}(o_{hj}^{hj}) + c(T_{o_{hj}^{l}}) + c(T_{o_{hj}^{r}}) \quad o_{hj}^{hj} \in J$$

In addition, CGO considers the use of bit filters [1], also called Bloom filters, in order to reduce the amount of internal I/O accesses in the Hash join operations. The use of Bloom filters in Hash join operations is a common practice in most commercial DBMS.

Bit filters do not always provide an improvement on the cost of a Hash join operation. In some cases, the bit filters needed to obtain the desired I/O reduction, are too
space-consuming and jeopardize the proper execution of the join operation. For this reason, our optimizer estimates the main parameters of the bit filters and following a set of rules decides whether they must be used in the join operation or not.

We estimate the size of a bit filter in operation $o^{h_j}$, for join condition $j$, using the following formula:

$$bf_{size} = \frac{2}{1 - (1 - BFE^{\frac{1}{2}} \cdot u(a_{inner}^{h_j} \cdot |R_{inner}(o^{h_j})|)}$$

where $BFE$ is a constant that fixes the predicted false positive cases. In our experiments we typically use $BFE = 0.05$ which means that from all the tuples accepted by the bit filter only 5% of them are false positives. This value is calculated in bits and it is properly transformed to its size in pages $bf_{pages}$ before using it in the Hash join cost formulae.

The selectivity of the bit filter is calculated as:

$$bf_{select} = s(a_{outer}^{h_j}) + (1 - s(a_{outer}^{h_j})) \cdot (1 - e^{-\frac{|R_{inner}(o^{h_j})| \cdot u(a_{inner})}{bf_{size}}})$$

Detailed formula derivations can be found in [5] and an example of a practical use of the formula can be found in [2].

The final cost function for the Hash join operation is the following:

$$c_{op}(o^{h_j}) = 2 \cdot (||R_{build}(o^{h_j})|| - M + (1 - \frac{M}{||R_{build}(o^{h_j})||}) \cdot ||R_{probe}(o^{h_j})|| \cdot bf_{select})$$

5 Summary

We have presented the cost model used for CGO. It is based on the I/O activity of different implementations of operations in the relational model.

Our cost model basically covers the following operations: index and sequential scans, Nested Loop joins, Merge-Scan joins, Hash join, Sort operations and Temp operations to materialize intermediate results.

References

