# Cost-based Selection of Path Expression Processing Algorithms in Object-Oriented Databases

**Georges Gardarin[a,b], Jean-Robert Gruser[b], Zhao-Hui Tang[a]**

[a]CNRS-PRiSM Laboratory
University of Versailles-St-Quentin
78035 Versailles, France
*firstname.lastname@prism.uvsq.fr*

[b]Projet Rodin
INRIA, Rocquencourt
78153 Le Chesnay Cedex, France
*firstname.lastname@inria.fr*

## Abstract

*An object query can include a path expression to traverse a number of related collections. The order of collection traversals given by the path expression may not be the most efficient to process the query. This generates a critical problem for an object query optimizer to select the best execution plan. This paper studies the different algorithms to process path expressions with predicates, including depth first navigation, forward and reverse joins. Using a cost model, it then compares their performances in different cases, according to memory size, selectivity of predicates, fan out between collections, etc.. It also presents a heuristic-based algorithm to find profitable n-ary operators for traversing collections, thus reducing the search space of query plans to process a query with a qualified path expression. An implementation based on the O2 system demonstrates the validity of the results.*

## 1. Introduction

One of the advantages of object-oriented DBMSs (OODBMSs) is that they support class relationships, thus permitting objects to refer to each other directly through pointers. Objects can have attributes which are references to other objects.

The concept of object navigation is an important aspect of object databases. In query languages, required navigations among objects are specified using path expressions. At each level of a path, a predicate can be used to restrict the navigation, and a variable can be used to refer to the selected objects in query results or further in the query qualification. We call such path expressions *qualified path expressions*; they unify the navigational and declarative aspects of object queries.

Up to now, a lot of research has been done in object query languages [Cat93, GV92, KKS92], which emphasizes the expressive power of path expressions in user queries. There is also research work done on physical access methods to support path expressions as path indexes [KKD89, Ber91, KM90]. [KGM91] has proposed an algorithm for efficiently assembling complex objects. [SC89] have developed a comparison of traditional value based joins and pointer based joins. System designers often think that in object databases, traditional joins are no longer necessary since objects point at each other and object navigations can replace joins. For example, the ObjectStore system does not implement value-based join algorithms. However, it is not obvious that following pointers is always very efficient, especially when the memory size is small as it may require to swap out data pages. Very few papers give precise comparisons among different ways of executing path expressions.

In this paper, we propose and compare different algorithms to search objects satisfying a qualified path expression. The basic algorithms for evaluating path expressions include depth-first fetch, forward join and reverse join. We then show that the mix of traditional joins together with pointer based navigations can give very efficient strategies. To compare the various approaches and include them in a cost-based optimizer, we develop a cost model for estimating different collection traversal algorithms. We also compare the performances of the proposed algorithms through an implementation based on the O2 system.



**Proceedings of the 22nd VLDB Conference**
**Mumbai(Bombay), India, 1996**

We show the different performances of these algorithms with different memory buffer sizes.

In this paper, we also propose a simple heuristic based on a cost model to select the best path traversal algorithms. Since a path expression can be executed using a mix of pointer based and traditional value based joins, the search space of the query optimizer becomes very large. In object system, the size of search space is increased due to the richness of different access methods. Implementing a complex search strategy is beyond the capabilities of most current OODBMS optimizers. Thus, to make our results simply applicable in current systems, we propose a simple heuristic to select an approximately optimal path expression processing strategy.

The remainder of this paper is organized as follows. Section 2 introduces qualified path expressions and presents three different algorithms for evaluating them. Section 3 gives heuristic rule for the query optimizer to generate the search space for executing a complex path expression and to select the optimal combination of algorithms. Section 4 precisely determines the costs of basic operators to evaluate path expressions. Section 5 presents the implementation results of different algorithms evaluating the same path expression with different memory configurations. Section 6 concludes the paper.

## 2. Path Traversal Algorithms

In this section, we describe three basic algorithms that can be used to evaluate a path expression. These algorithms correspond to various types of traversal of an object network as illustrated in Figure 1. They are called Depth-First-Fetch (DFF), Breadth-First-Fetch (BFF), and Reverse-Breadth-First-Fetch (RBFF). BFF is based on Forward Join (FJ) while RBFF is based on Reverse Join (RJ). During query optimization, a path can be cut into several subpaths and each of these subpaths can be traversed using a different algorithm.

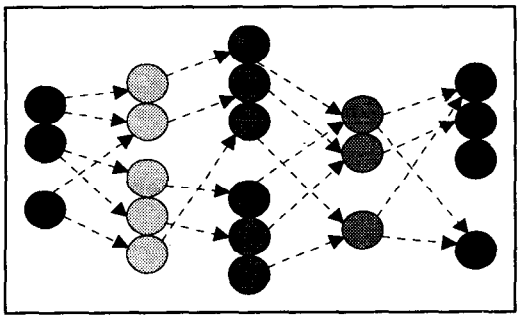

*Figure 1 — An object network.*

### 2.1 Qualified Path Expressions

Let {C1, C2, ..., Cn} be a set of related collections. Each collection Ci contains a set of homogeneous objects of type Ti. From the source C1 to the target Cn, collections are linked through attributes : Ai is an attribute of Ci objects whose values are collections of 1 or more objects in class Ci+1. A *qualified path expression* is an expression of the form :

X1[P1].A1.X2[P2].A2.X3[P3]...An-1.Xn[Pn]

where Xi is a variable representing an object of collection Ci, Ai a relationship role attribute, and Pi is a predicate which qualifies Ci objects. Pi is optional and can be empty, which means true.

The semantics of a qualified path expression is a set of tuples S { [X1, X2, ... Xn] }, where each tuple corresponds to a path of qualifying objects, i.e., an instance X1, X2, ... Xn contains OIDs of linked objects satisfying the associated predicate in the path. S is called a *supporting table* for the qualified path expression. Note that, when evaluating a qualified path expression, it is not always necessary to instantiate the supporting table of OID tuples (i.e., the semantics) as required attribute values can be directly extracted from objects. Thus, in the supporting tables, OIDs are often replaced by attribute values required for assembling the query result, or omitted if no longer used. To improve the performance of path traversal, these supporting tables may be hashed on the value of OID.

We illustrate qualified path expressions using three collections {Companies, Persons, Vehicles} linked by relationships Employs and Owns, as represented in Figure 2. The double arrow in the figure represents 1-to-many relationship between two collections. Qualified path expressions can be used as complex predicates in some variations of object SQL. Various syntaxes have been proposed for that, as in ESQL [GV92], SQL3 [Mel93], and OQL [Cat93]. Below, we suggest a direct form in which qualified path expressions are used as complex predicates.

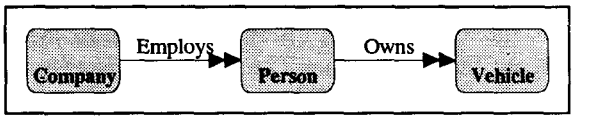


*Figure 2 — Examples of linked collections.*

The following query retrieves the name of the companies, the name of the employed persons, the number of the owned vehicle, for all companies in Paris whose employees younger than 30 have cars of power greater than 10 :

SELECT C.Name, P.Name, V.Number

```
FROM Companies C, Persons P, Vehicles V
WHERE C[City="Paris"].Employs.P[Age<30].Owns.V[Pow> 10]
```

It uses the qualified path expression C[City = "Paris"].Employees.P[Age < 30].Cars.V[Pow > 10] in the WHERE clause. The corresponding query is expressed in OQL as follows :

```
SELECT C.Name, P.Name, V.Number
FROM C IN Companies, P IN C.Employs, V IN P.Owns
WHERE C.City="Paris" AND P.Age<30 AND V.Pow>10
```

Directly supporting qualified path expressions in the query language will give less procedurality to the language. Linking attributes can be seen as stored predicates. Further, it permits to simply express queries with branching paths, which is known to be difficult in OQL (e.g., expressing C.Employs.P and C.Promotes.P may require two variables and one predicate in OQL). Proposed object query language standards (e.g., OQL and SQL3) are rather procedural with regards to qualified path expressions; they simply express them as nested SELECT, which could bias certain query optimizers. We show in the following sections that the optimal order of collection traversals is complex to determine.

## 2.2 DFF Traversal

Depth-First-Fetch (DFF) is the natural algorithm for evaluating a path expression. It follows the path from the root to the target collection, using a depth first graph traversal algorithm. The corresponding operator is an n-ary operator denoted DFF. It processes the path expression by navigating through object references following the order of the collections in the path expression. If there are multiple references from an object to the objects in the next collection, the navigation follows the depth first order. It assembles objects accessed through OIDs, which are efficiently decoded in most object database systems.

The advantage of DFF is that it is an nary operator that does not generate intermediate results. The result objects are assembled one at a time, which allows the system to return an answer before having processed the whole path. The CPU time is restricted to the time required to test the predicates and assemble the results. When objects are clustered according to the traversed relationships, the I/O time is significantly reduced. In general, this operator is very efficient when the memory size is large enough to avoid swapping of objects at level K in the tree when processing objects at level K+I, otherwise it could be very costly. The detailed performance analysis is given in section 4.

DFF is different from a set of pipelined binary joins which is a popular technique in relational DBMSs. Neither DFF nor pipelined multijoin needs to memorize temporary results. But the latter is a set-oriented operation and can not terminate the join process before all the tuples in a relation are evaluated. The granularity of DFF is finer since it processes objects one-at-a-time rather than by sets of objects. It also never processes objects of level K not pointed by at least one object of level K-1, which is an advantage of pointer-based joins over value-based joins.

## 2.3 BFF Traversal and Forward Join

Breadth-First-Fetch (BFF) traversal processes the tree of objects using a Forward Join (FJ) algorithm which is based on pointer chasing between two collections. Successive binary joins of collections are performed from the source collection to the target, following the path in a forward order.

To process a qualified path expression of the form X1[P1].A1.X2[P2]...An-1.Xn[Pn] from collection C1 to collection Cn, (n-1) successive joins of type Si+1 = FJ(Si,Ci.Ai,Pi+1) are performed, where Si designates the supporting table of objects satisfying the subpath expression X1.[P1].A1...Ai-1.Xi[Pi]. The join criteria is simply the traversal of Ci.Ai pointers. A new supporting table must be generated at each step to record the OID mappings among the different collections. The algorithm gains in ordering the supporting tables according to the OIDs of the objects of the next collection to be traversed. Thus, each page of a traversed collection is only loaded once. Ordering can be simply achieved using an order preserving hash function.

The advantage of the BFF algorithm is to avoid nested loop OID comparisons like in traditional nested loop joins; thus the CPU cost is not high, although some time is spent to maintain the hashed supporting tables. Note that the FJ operator can be considered as a special case of the DFF one, where only two collections are involved in the path expression. However, the BFF algorithm requires the construction of supporting tables, which is both costly in memory size and CPU.

## 2.4 RBFF Traversal and Reverse Join

As BFF, Reverse-Breadth-First-Fetch (RBFF) performs a sequence of binary joins between two neighbor collections to traverse the path, but it proceeds in the reverse order of the path. Thus, each join is called a Reverse Join (RJ). The join criterion is the member-

ship of the second collection object identifier to the first collection pointer attribute values. To process a qualified path expression of type X1[P1].A1.X2[P2]...An-1.Xn[Pn] from collection C1 to collection Cn, (n-1) successive joins of type Si-1 = RJ(Si,Ci-1.Ai-1,Pi-1) are performed, where Si designates the supporting table of objects satisfying the subpath expression Xi[Pi].Ai...An.Xn[Pn] and RJ is the OID membership join algorithm in reverse order. As there is no direct link from the Ci collection to the Ci-1 one, a value based join must be used to check the OID membership condition.

RBFF is efficient when the predicate in the last collection is selective as well as the intermediate joins. Thus, the supporting tables at each step remain relatively small and can generally be entirely loaded in memory. However, RJ doesn't benefit of the clustering of traversed collections Ci according to the Pi selection predicate. It performs value-based comparisons of OIDs, which is in general inefficient in CPU.

## 2.5 Support tables

A support table can be regarded as a collection of tuples of qualified object identifiers and attributes. Two support tables can be joined together if there exists a common supported collection between them. Figure 3 shows some support tables generated during path traversals. In (a), collection A is forward joined with collection B to get an OID mapping between A objects and C objects; the support table Tb is then forward joined with the C collection to get a mapping between A and D (b). (c) is an example of reverse joins between collection D and E, and (d) shows a join between two support tables.

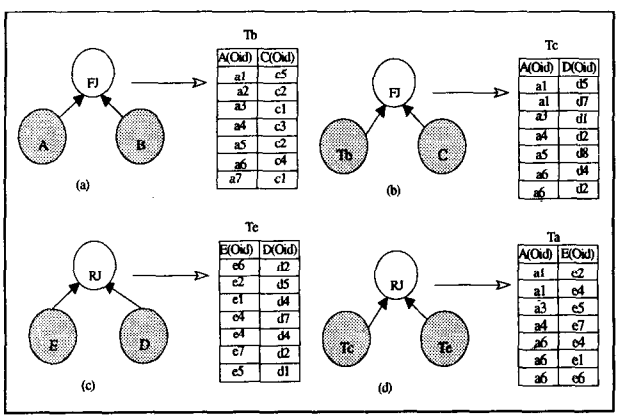


*Figure 3— Join between support tables.*

# 3. Mixed Strategies to Evaluat Complex Path Expressions

In this section, we examine strategies mixing the three algorithms given above to evaluate path expressions. We particularly evaluate the size of the search space using these three algorithms for traversing a path of length (n-1).

## 3.1 Query Plans as Processing Trees

A processing tree (PT) is a graphical representation of an execution plan [KBZ86]. A PT is a labeled n-ary tree where the leaf nodes represent collections of objects, the non-leaf nodes represent operators (e.g., Selection, FJ, RJ, DFF), and the edges represent temporary collections (e.g., support tables T1, T2). We extend the traditional PT in the sense that certain intermediate nodes may have more than two children such as DFF operator, which starts from the objects in the left most collection and navigates to the right most collection.

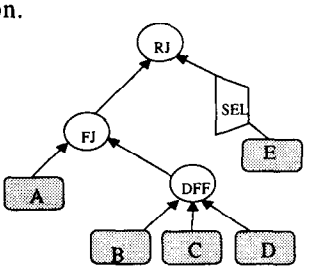


*Figure 4 — A processing tree.*

Figure 4 gives an example of a PT. We consider three kinds of operators: unary such as select and index_select; binary such as FJ and RJ; and n-ary such as DFF. Each PT node has annotations indicating the detail of the algorithms, the projection results and the qualifying predicates of the input collections. The execution of a PT follows bottom-up order. A join node (FJ or RJ) captures the join between an outer node (i.e., its left operand) and an inner node (i.e., its right operand). Given a PT rooted at node N, the cost of the PT is computed recursively using the formula :

$$Cost(PT) = Cost(N) + \sum_i Cost(Child_i)$$

where $Child_i$ is the i-th child node of node N in the PT. In the following, to evaluate costs, we use classical notations : ||Ci|| denotes the cardinality of collection i, |Ci| denotes the number of disk pages of collection i, $Sel_i$ denotes the selectivity of the collection i with predicate Pi, and $fan_{i,j}$ denotes the fan out of the considered relationship from collection i to collection j.

## 3.2 Processing Tree Transformations

As shown in Section 2, a qualified path expression can be executed using different combinations of operators. Each execution plan corresponds to one specific PT. The costs of different PTs differ a lot, even though they are equivalent in semantics. The objective of a query optimizer is to avoid the worse cases and to pick up one of the most efficient PTs. Different execution plans can be generated by the optimizer from an initial one based on a set of transformation rules. For example, Figure 5 illustrates several transformed PTs of the PT given in Figure 4.

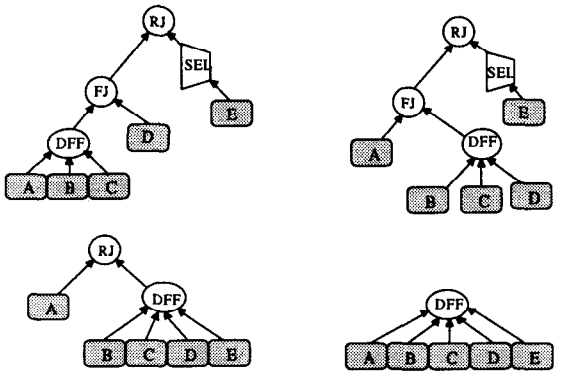


*Figure 5— Transformed processing trees.*

The number of execution plans explored by the optimizer depend on the applied search strategy. As in relational databases, where the problem of large search space is mainly caused by join series, in OODBMS the search space of a query is exponential according to the length of path expressions. In modern query optimizer architectures [FV94, FG94], different components are driven by different search strategies; thus, it would be useful to have a special combination of strategies for optimizing path expressions. For example, we could first apply a greedy strategy to select the algorithm for traversing certain neighbor collections; then the path would be cut into subpaths, which would greatly reduce the search space.

## 3.3 Generating the Search Space

Relational database systems rely on the join operator to assemble tuples of different tables for answering queries. Although the order of joins does not affect the final result, it does determine to a large extent the response time of the query. In object databases queries, series of joins are replaced by path expressions. The number of equivalent plans becomes even larger, since path expressions can be executed not only by series of binary joins as usual in relational databases, but also by assembling efficiently the objects [KGM91]. [TL91] gives an estimation of the size of the search space when only binary joins are considered. In this section, we present a method to measure the search space where nary joins like DFF exist.

Suppose we have a path A.B.C.D...Z as in Figure 6. Links with double arrows represent multiple object references, while single arrows represent monovalued object references. Let n be the number of collections traversed and x be the number of different binary join algorithms. Let us consider a given processing tree. For each binary join, x different join algorithms can be selected, which yields $x^{n-1}$ possible execution plans. Thus, without considering a multi-join algorithm, the size of the Search Space (*SS*) is given by :

$$SS(n) = \frac{(2n-2)!}{n!(n-1)!} * x^{n-1}$$

where $x^{n-1}$ gives the combinations of binary join algorithms among n collections and $\frac{(2n-2)!}{n!(n-1)!}$ gives the different orders to execute these joins. For example, if the path length equals 2 and there are only two different join algorithms, FJ and RJ, then the size of the search space for processing the path expression is SS(3)=8. When multi-joins are supported, the size of the search space is increased. Series of joins can be replaced by one DFF operator.

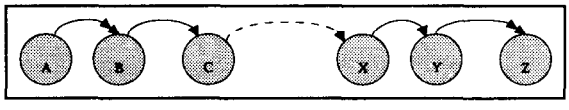


*Figure 6 — A sample path.*

Figure 7(a) gives all the possible join combinations for a path expression with length 5. Each circle represents a join operator, FJ, RJ or DFF. The number in each circle indicates the starting and ending joined collections in the path. The bottom level of the pyramid gives the binary joins; the level above gives the DFF operators involving three collections; the top level circle represents a DFF operator applied to all the six collections. This pyramid has n-1 levels. Looking closer to the pyramid, smaller pyramids inside can be isolated, like the two marked with solid and dotted lines. The size of each pyramid varies, but the summit of each small pyramids represents a DFF operator equivalent to the sequence of binary join operators at the bottom level. For example, a join between the collections 0 and 1 followed by a join between the collections 1 and 2 is equivalent to a DFF operator on collections 0, 1, and 2.

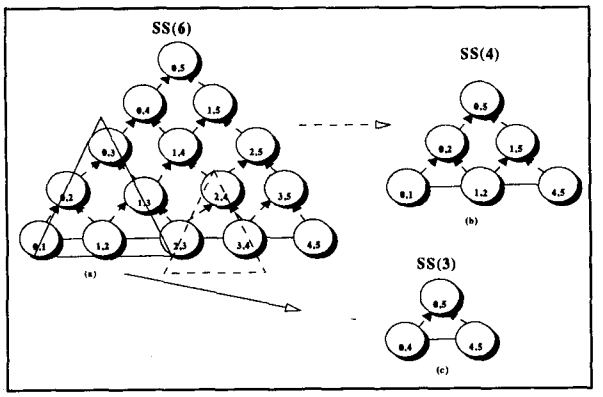


*Figure 7 — Extending search space with DFF.*

Let us assume now that the size of the search space of a path with 6 collections is *SS(6)*. It contains a pure binary join sub-space of size $\frac{(2*6-2)!}{6!(6-1)!}*2^5$. Further, if part of the traversing is done using DFF — for example using DFF to traverse collection 2,3,4, — then *SS(6)* includes the case b in Figure 7, whose search space size is *SS(4)*. There are totally 4 different DFFs on three collections (012, 123, 234, 345), each of them transferring pyramid (a) to different pyramids of 3 levels with a search space of size *SS(4)*. We can also execute a DFF on 4 collections, as in the small pyramid marked out with solid line in Figure 7.a. Collections 0,1,2,3 become an atomic element since they are traversed together by one DFF operator. This transfers pyramid (a) to the case of (c) with three collections ((0123), 4, 5) whose search space is *SS(3)*. There are totally 3 different DFFs on four collections (0123, 1234, 2345), each of them transfers pyramid (a) to different pyramids of two levels. For the same reason, there are 2 different DFFs on five collections(01234, 12345), each of them transferring the original path to a path with 2 atomic collections whose search space is *SS(2)*. Finally, there exists one DFF which traverses the whole path in a forward order based on OID navigations, which transfers the original path query to a simple query on one atomic collection, whose search space is *SS(1)*. But in the above analysis, there is a case counted twice, which applies DFF both on collection 012 and collection 345, whose search space is *SS(2)*.

Finally, the precise search space of a path with 6 collections is given by :

$$SS(6) = \frac{10!}{6!*5!}*2^5 + 4*SS(4) + 3*SS(3) + 2*SS(2) + SS(1) - SS(2)$$

If we consider only 2 different binary joins (FJ and RJ) and one nary join (DFF), we obtain SS(1) = 1, SS(2) = 2, and the total search space for SS(6) = 1544. More generally, the size of the search space for different numbers of linked collections is given in Table 1.

| Number of collections | Search Space | Number of collections | Search Space |
|---|---|---|---|
| 1 | 1 | 5 | 256 |
| 2 | 2 | 6 | 1544 |
| 3 | 9 | 7 | 9910 |
| 4 | 45 | 8 | 65462 |

*Table 1— Size of search space.*

Note that the results in the above table are calculated using only 3 different algorithms, when there are no reverse links in the paths. When the path graph becomes more complicated (e.g., with reverse links), or when there are more join algorithms (e.g., different hash join algorithms), the search space is much larger, and it is even more necessary to apply certain heuristics for selecting a PT. In the next section, we present such a heuristic to pick up profitable n-ary operators before considering binary operators.

## 3.4 Heuristics to Reduce the Search Space

From Table 1, we can see that the search space for optimizing a path expression is exponential to the path length. If the query optimizer can immediately find the profitable nary operators to apply on a number of collections, the search space will be largely reduced since those collections linked by the nary operator can be considered as one single collection.

For this purpose, we propose a data structure called *Access Matrix*. It is a two dimensional array. Each node inside the matrix $Node_{i,j}$ represents the path information from collection i to collection j. Thus it is a n*n matrix with $n^2$ nodes, where n is number of collections involved in the path expression. The information associated with each node includes the following :

- Weight : the path length from collection i to j;
- Cluster : whether the collections are clustered according to the path;
- PathIndex : whether there is a path index between collection i and j;
- DFF : whether the nary DFF operator is profitable from i to j;

The attributes of a node can also be extended, for example, to add the statistics of a collection, the type of index, etc. The DFF attribute has three states{-1,0,1}. The state 1 means the nary operator DFF from collection i to collection j is profitable compared to a set of binary join operators. The state 0 means the contrary. The state -1 means that it is impossible to perform DFF operators between collection i and j, because there is no link starting from collection i to

collection j. This case only appears in the node $Node_{i,j}$ where i>j (in the case that there is no inverse link from collection i to j).

As stated in section 2, DFF is an efficient operator since it is based on the navigation following the pre-computed links and no intermediate result has to be generated. But DFF generates many random disk accesses if collections are not clustered according to the links. When memory buffer size is small, the response time of the DFF operator is increasing rapidly due to memory disk swaps. The performance analysis and measurements presented in the following sections confirm this fact.

In summary, DFF appears superior to BFF or RBFF when the memory size is large enough to avoid reading twice a page, due to the fact that intermediate structures are not required as with forward join or reverse join. DFF is also superior when the collections are clustered according to the path, as only one cluster is read to traverse all pointers from a given object. In case of a path index, special algorithms have to be considered as the supporting table is directly available. Thus, we propose the following heuristic to select DFF when there is no path index from i to j.

**Heuristic Rule for DFF :** Select DFF from Ci to Cj iff one of the following condition holds :

- The collections are clustered according to the path.
- The available memory buffer size is superior to M, the expected number of pages to access in Ci+1, ... Cj using the DFF algorithm.

The last factor (M) can be evaluated using different estimators. We propose a formula for estimating M in the next section.

In our approach, there are two steps to optimize a query with a path expression. In the first step, the optimizer parses the *Access Matrix*. The objective is to find some profitable nary operators along the path. Once these profitable nary operators are found, collections traversed by the same operator are considered as one atomic collection; thus the path length is reduced. In the second step, the query optimizer applies classical optimization techniques such as join permutation and algorithm selection (FJ or RJ) based on cost estimation using an efficient search strategy. As we have already shown, the search space is exponential in the path length. Thus, it is important that the second step be processed within a largely reduced search space.

## 3.5 Estimating Page Block Hits for DFF

To apply the given heuristic, the optimizer has to evaluate the expected number of pages to traverse with the DFF algorithm. This can be done using the Yao formula [Yao77]. It estimates the number of block hits for selecting k records from a collection which contains n records and is grouped into m blocks as being :

$$Yao(n,m,k) = m * (1 - \prod_{i=1}^{k} \frac{n - \frac{n}{m} - i + 1}{n - i + 1})$$

Let $X_k$ be the number of distinct objects to select in collection Ck. Then, the number of pages to access in collection Ck is given by yao($\|Ck\|$, $|Ck|$, $X_k$). Thus, the number of pages to access in collections Ci+1 to Cj is given by the formula :

$$M = \sum_{k=i+1}^{j} yao(\|Ck\|, |Ck|, X_k)$$

The value of Xi can be estimated using the database statistics. An approximation will be given in the next section when computing the DFF cost.

# 4. Cost Evaluation of Operators

In the following we present a cost model for the different path traversal algorithms studied in section 2. This is useful to complete the heuristic described in section 3.

## 4.1 Cost Model Parameters

There are several different components of the cost : the CPU cost is the cost of processing CPU instructions; the IO cost is the cost of read and write operations between memory and disk. We assume that :

- m is the available memory size for processing a query,
- p is the page size,
- *move(O)* denotes the time to copy an object O in memory, which is a pure CPU cost,
- *comp* denotes the time to compare two value in the memory, which is a pure CPU cost,
- *hash* denotes the CPU cost to find the memory address of an OID.
- *fan(C1,C2)* denotes the average number of references from a C1 object to C2 objects.

## 4.2 Cost of Temporary Result

Each operator generates an output result. As mentioned above, results may be support tables, i.e., tables of tuples of OIDs. Assuming a support table of size *card* in number of tuples and denoting

*move(proj)* the CPU cost required to project the attributes with size *proj* and write the projection result in memory, we obtain :

$$OutPut\_CPU\_Cost(card, proj) = move(proj) * card$$
$$OutPut\_IO\_Cost(card, proj) = \frac{card * proj}{p}$$

## 4.3 Cost of the Forward Join Operator

The IO cost of the Forward Join consists of two parts : charging objects from the C1 and C2 collections to memory and writing the result file to disk. $Sel_{C1}$ is the selectivity of a predicate on C1 if it exists, otherwise it is equal to 1. When the available memory buffer size m satisfies the formula $m \geq yao(\|C2\|, |C2|, X2)$, there is no need of reading several time the same page of C2 as seen in the previous section. Thus, the IO_Cost is equal to the total page number of C1 plus the number of page block hits on C2.

When the available memory size is smaller, some pages need to be loaded multiple times and for each reference from a C1 object to C2 object. For collection C1, each object is charged only once for forward join operator. Once all the objects in the same data page have been processed, this page can be freed from the memory buffer. At any instance in the memory buffer, the algorithm needs only one C1 data page and (m-1) C2 data pages. Suppose the C2 objects referred by C1 are uniformly distributed in all the data pages of C2. We estimate each time for dereferencing an OID of a C2 object, there are (m-1)/$|C_2|$ possibilities that the object is in memory. The number of OIDs to dereference is given by the formula $\|C_1\| * Sel_1 * fan(C_1, C_2)$ as each selected object in C1 has an average of fan(C1,C2) pointers to C2 objects. Thus, the additional number of I/Os in the case of not enough available memory to hold C2 is : $(1 - \frac{m-1}{|C_2|}) * \|C_1\| * Sel_1 * fan(C_1, C_2)$.

Collecting all the components yields the IO cost of the forward join :

$$FJ\_IO\_Cost(C_1, C_2) = OutPut\_IO\_Cost(\|C_1\| * Sel_1 * Sel_2 * fan(C_1, C_2), proj)$$
$$+ \quad |C_1| + yao(\|C_2\|, |C_2|, X_2)$$
$$+ \quad \left\{ (1 - \frac{m-1}{|C_2|}) * \|C_1\| * Sel_1 * fan(C_1, C_2) \quad if \quad m < yao(\|C_2\|, |C_2|, X_2) \right\}$$

The CPU cost also consists of two parts : the CPU cost for finding the memory address of each C2 object referred by C1 objects and the CPU cost for projecting the result. It yields :

$$FJ\_CPU\_Cost(C_1, C_2) = \|C_1\| * fan(C_1, C_2) * hash$$
$$+ \quad OutPut\_CPU\_Cost(\|C1\| * Sel_1 * fan(C_1, C_2) * Sel_2, proj)$$

## 4.4 Cost of the Reverse Join Operator

Reverse join is a traditional value-based set-oriented join. Although reverse join does not benefit from the navigational aspects provided by object systems, it can still be very efficient for processing a path expression, especially when a selection has been done on the second collection resulting in a list of qualifying OIDs saved in a support table. Then, one of the inputs of the RJ is a table of OIDs directly in memory.

The cost of value-based join has been analyzed using different join algorithms in relational systems. The reader can refer to [Sha86] for nicely revisited formulas. For processing path expressions, the reverse join is based on comparing attributes of OID type. One interesting problem is to estimate the size of temporary results after a reverse join. In the following, we present an estimation method for that size. Assuming classical formulas for nested loop joins, we derive the CPU and I/O costs.

The average number of references of a $C_1$ object to $C_2$ objects is $fan(C_1,C2)$. Thus, the total number of references from $C_1$ objects to $C_2$ objects equals to $fan(C_1,C_2)*\|C_1\|$. For each object in collection $C_2$, the average number of references from $C_1$ objects is :

$$fan'(C1, C2) = \frac{fan(C_1, C_2) * \|C_1\|}{\|C_2\|}$$

Let $T_2$ be the support table for collection C2, containing OIDs of C2 objects. As C2 has been filtered using predicate P2 of selectivity $Sel_2$ there are $\|C_2\|*Sel_2$ entries in $T_2$. So there should be $fan'(C_1,C_2)*\|C_2\|*Sel_2$ logical links from $C_1$ objects to the relevant $C_2$ objects. For an object in collection $C_1$, the probability of not being involved in these logical links is :

$$P(0) = (1 - \frac{1}{\|C1\|})^{\|C_2\| * Sel_2 * fan'(C_1, C_2)}$$

Finally, the number of qualifying $C_1$ objects after reverse join is equals to the following :

$$N = (1 - P(0)) * \|C_1\|$$

The previous formula gives the size in number of objects of the results. Assuming a nested loop join, we can derive the CPU and I/O costs of RJ. They are given by the following formulas :

$$RJ\_CPU\_Cost(C_1,T_2) = \|C_1\| * fan(C_1,C_2) * \|C_2\| * Sel_2 * Comp + OutPut\_CPU\_Cost(N,proj)$$

$$RJ\_IO\_Cost = OutPut\_IO\_Cost(N,proj) + \begin{cases} |C_1| * \frac{|C_2| * Sel_2}{m-1} + |C_1| & if\ m < |C_2| * Sel_2 \\ |C_1| + |C_2| & if\ m >= |C_2| * Sel_2 \end{cases}$$

Note that the cost formulas neglect the time spent to generate the hashed support table $T_2$ on C2. In many cases, the support table $T_2$ generated on collection C2 contains only OIDs. As the size of an OID is about 4-8 bytes, a data page may store several hundreds of OIDs. Thus often $T_2$ can be entirely kept in memory. Only CPU time has to be added to the first formula, for example $\|C_2\| * Sel_2 * Ass$, where $Ass$ is the time to insert an OID in the support table.

## 4.5 Cost of the DFF Operator

DFF is an efficient operator for traversing the path when the memory size is big, since it profits the navigation aspect of object systems. It consumes few CPU and does not generate any temporary results for evaluating a path. The CPU cost for DFF without taking into account the output of results and the checking of the selection predicates is simply the time of traversing the graph, decoding the OIDs and evaluating the predicates if any, which yields :

$$DFF\_CPU\_Cost = hash * \|C1\| * (1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} (fan(Cj, Cj+1) * Sel_i))$$

The IO cost can vary depending of the available memory size. When the memory is large, we have :

$$DFF\_IO\_Cost = |C1| + \sum_{i=2}^{n} yao(\|Ci\|, |Ci|, X_i)$$

The Xi's are also used in the heuristic rule of section 3. A more precise value can be derived from [GGT95] :

$$X_i = (1 - (1 - \frac{1}{\|C_i\|})^{X_{i-1} * Sel_{i-1} * fan_{i-1,i}}) * \|C_i\|$$

$X_i$ is the number of objects in collection i to select when evaluating the subpath X1[P1].A1.X2[P2]...Ai-1.Xi[Pi]. $(1 - \frac{1}{\|C_i\|})^{X_{i-1} * Sel_{i-1} * fan_{i-1,i}}$ gives the probability of an object in collection i not to be involved in the evaluation of the path expression. When $\|C_i\| >> X_{i-1} * Sel_{i-1} * fan_{i-1,i}$, which means that only a small proportion of objects in collection $C_i$ is referenced by the objects from collection $C_{i-1}$, $X_i$ can be approximated by $X_{i-1} * Sel_{i-1} * fan_{i-1,i}$; when $\|C_i\| < X_{i-1} * Sel_{i-1} * fan_{i-1,i}$, $X_i$ is close to $\|C_i\|$.

When the memory is small, the IO cost of DFF can be very high since objects which are already loaded into memory may be replaced and then loaded again during the navigation. In the worst case, the number of IOs can be close to the value of $\|C_1\| * (1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} fan(Cj, Cj+1) * Sel_i)$. Thus, it seems better to avoid DFF when the memory size is relatively smaller.

# 5. Performance Study

This section presents the performance evaluations of different collection traversal algorithms in various cases.

## 5.1 The Experiment Platform

The experimentation platform is based on the O2 system and the OO7 Benchmark on a SUN Sparc 20 with 96 M bytes of memory. To test the different algorithms, we generate a scalable tree of OO7 objects. This tree is composed of 5 levels of Assemblies with a maximum fan out equal to 5. Since we set the root collection to 256 ComplexAssemblies, the ending collection stores 160,000 BaseAssemblies. The size of the test base is about 20M bytes. Objects belonging to the same collection are grouped together; links between objects of two neighbor collections are randomly generated. We have implemented three collection traversal algorithms described in section 2 with O2C. For BFF, at each step, the OID values inside the support table are hashed. This avoids random access to the objects of the next collection in the following step of BFF. RJ is implemented using the hash join algorithm.

## 5.2 FJ versus RJ on Two Collections

In the first step, we compare two binary join algorithms (FJ and RJ) on two large collections when the memory buffer size varies. Figure 8 shows the response time of processing a path expression *a.b[pred]* with a predicate on the second collection.

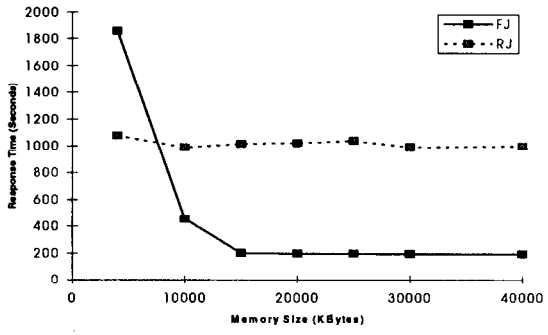


*Figure 8 — Forward Join and Reverse Join.*

The fan out between these two collections is set to 5. In this case, the condition of the heuristic rule de-

scribed in Section 3 returns 11 Mbytes, which means when the memory buffer size is less than this limit, the FJ operator is not profitable. We can see that this value is quite close to the breakpoint of the FJ in Figure 8, slightly left shifted. Compared to FJ, RJ is less sensitive to the memory size. The response time of the RJ does not change unless the memory becomes very limited .

## 5.3 BFF versus DFF

We now analyze the executions of a DFF compared to the execution of a sequence of FJ (BFF) on a path schema involving 5 collections (path length equals 4). Figures 9(a) and 9(b) gives the response time for different fan out settings in function of the memory size.

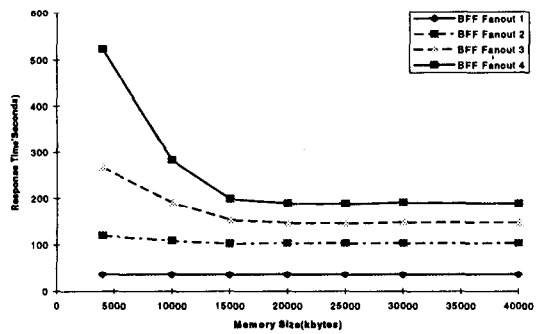


*Figure 9(a)—Fan out variation for the BFF.*

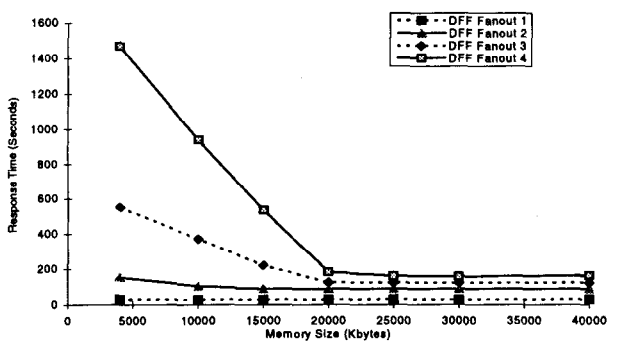


*Figure 9(b) — Fan out variation for the DFF.*

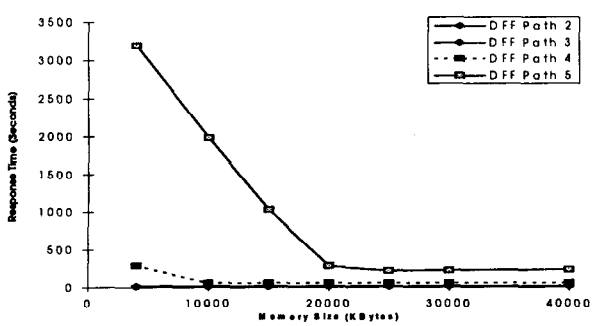


*Figure-9(c) Path length variation for the DFF*

Figures 9(c) and 9(d) show the response time of the BFF and the DFF when varying the path length. We observe for the two algorithms that their respective breakpoint arises earlier and that the inclined angle of curves drastically increases when the path length increases. Figures 9(c) and 9.(d) confirms that in general, when path length becomes longer and memory size becomes relatively smaller, BFF outperforms DFF.

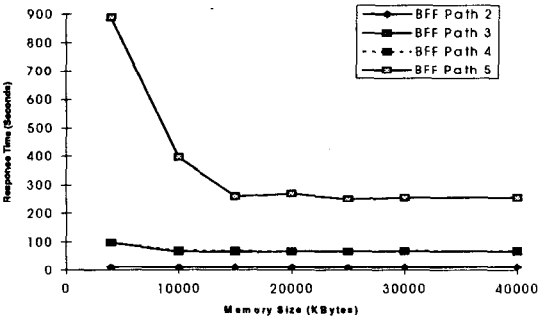


*Figure 9(d) — Path length variation for the BFF.*

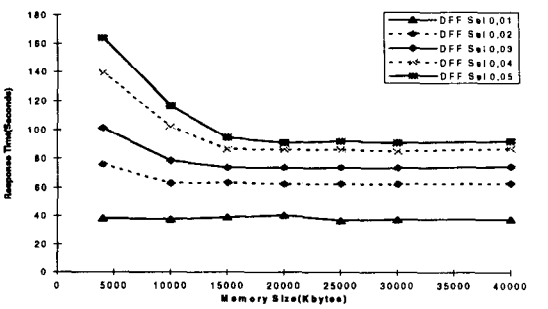


*Figure 9 (e) — DFF : Selectivity of the first predicate*

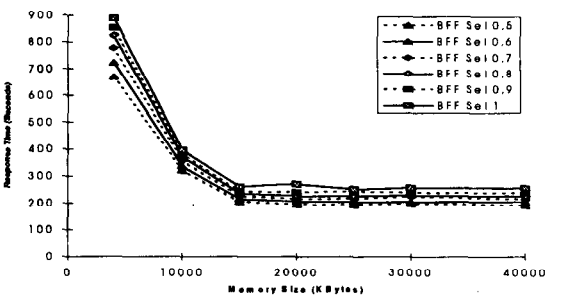


*Figure 9(f)— BFF : Selectivity of the first predicate.*

Figure 9(e) and 9(f) show the response time of the BFF and the DFF when the selectivity on the first predicate changes. We can see that when the selectivity of the first collection increases, the break point of DFF is shifted to the right. The reason is that when there are more qualified objects from the starting collection, there are many more objects belonging to the following collections involved by the path expression, thus yielding the increasing of the value M.

From all the experiments reported in Figure 9(a) to 9(f), we conclude that for DFF the fan out and path length factors have more impact on the performance than the predicate selectivity, which only has a linear effect on a limited part of memory. Thus the poor performance of the DFF is mainly due to memory-disk faults yielding multiple accesses to certain pages.

## 5.4 BFF, DFF and RBFF

RBFF is implemented using a serial of hash join. The experiment of Figure 10 compares three different collection traversal algorithms for processing a path expression with length 4.

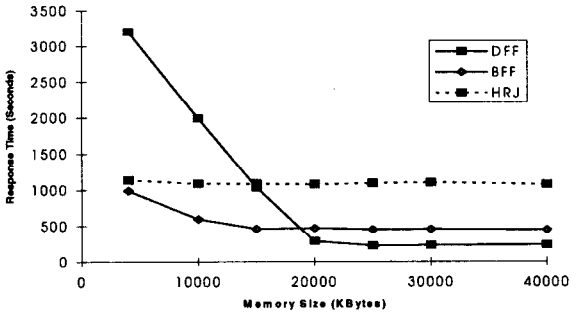


*Figure 10 — BFF, DFF and HRJ.*

In Figure 10 we can see that when the memory buffer size is large, DFF outperforms BFF (200 seconds) and HRJ (1000 seconds) mainly due to the fact that it requires neither to generate any intermediate results nor to charge any unreferenced objects. There is no predicate restriction on any of the five collections. Applying our heuristics, M is close to 19Mbytes. The curve of DFF confirms that when the memory size is below this number, DFF starts to increase drastically. The response time of DFF can be several times more than BFF and HRJ in the memory zone less than 10 M bytes. Using our heuristic rule, the query optimizer can avoid applying DFF with small memory size (compared to collection size). BFF and HRJ are relatively stable as the memory buffer size varies. Compared to HRJ, BFF is more sensitive to memory buffer size since it is a pointer-based join. We notice that when the memory is less than 10 M bytes, the response time of BFF starts to increase.

## 5.5 Mixed Strategies

From above experiments, we have shown the behavior of different path traversal algorithms in relation to factors such as fan out, selectivity, path length, memory size. The experimental results show that none of these algorithms dominates the others in any case. DFF is an algorithm that can either be very efficient or very inefficient depending on the memory size. When the memory is small, a mix of these three strategies could have better performance. Figure 11(a) displays two processing trees with mixed path traversal algorithms. Figure 11(b) shows the response time for traversing a path expression with length 4 and fan out 5. There are five execution plans in Figure 11 (b). The two above mixed plans plus DFF, BFF and HRJ.

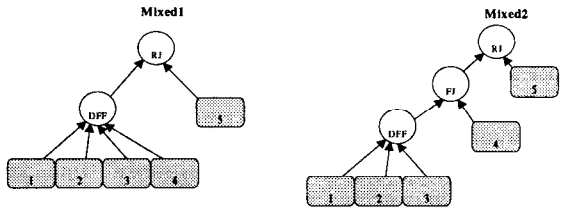


*Figure 11(a) — Mixed strategies*

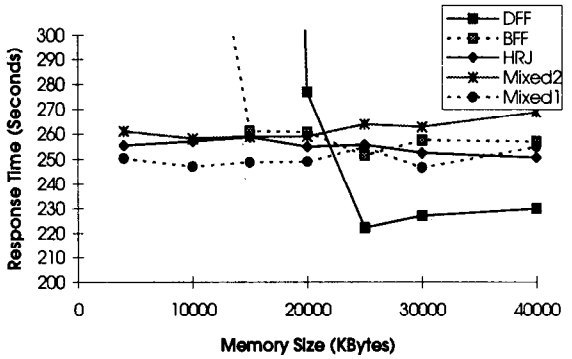


*Figure 11(b) — BFF,DFF HRJ and Mixed.*

The heuristics gives M an estimation of 21 M bytes. Thus, when the memory buffer is bigger than this value, DFF stays the best algorithm. But when the memory buffer size is smaller, the Mixed1 tree becomes the most efficient. Mixed1 becomes the most efficient. HRJ is quite stable when memory buffer size varies. Mixed2 behaves almost the same as HRJ since the it uses two hash joins for traversing the two biggest collections C4 and C5. For very large databases, the memory buffer size is usually much smaller than the collection average size. Therefore, it is crucial for the optimizer to find the good mixed algorithm for executing the query.

## 6. Conclusion

This paper studies one of the open problems in object-oriented database systems, namely path expression processing. It defines *qualified path expression*, which merges selection predicates with path traversals. It introduces a cost model for path traversal and compares the performances of different collection traversal algorithms both analytically and experimentally. The results show that each algorithm has a best

range of application. This paper also suggests a data structure named *Access Matrix* to explore profitable nary operators and apply the heuristic rule to reduce the search space of the query optimizer.

More specifically, the performance study shows that in different cases the costs of different traversal algorithms vary a lot. Each of these algorithms has its advantages and disadvantages. Depth-first-fetch algorithm is profitable when the memory size is large and becomes very expensive when memory size is reduced. Binary traversal algorithms are not too sensitive to the memory buffer size compared to nary collection traversal algorithms. Traditional join algorithms can still be very efficient compared to pointer chasing, particularly when the selectivity of the predicate applied on the last collection of a path is small. Today several leading commercial OODBMS products mainly rely on naive navigation for processing a query with a path expression. Traditional binary join algorithms based on OIDs (RBFF) are not even implemented. Current systems should be improved by implementing various path traversal algorithms. Of course, their query optimizer should master the condition when to apply these different algorithms. The results of the implementation prove that the heuristics we propose for finding profitable nary operators are correct. When the memory buffer size is smaller than the condition we defined in the heuristic rule, the cost of navigation operator is raising up rapidly.

## References


[Ber91] E. Bertino. An indexing technique for object-oriented databases. In Proceedings of Int. Conf. Data Engineering, April 1991.

[BMG93] J.A. Blakeley, W.J.McKenna, and G. Graefe, Experiences building the open OODB optimizer. In proceedings of ACM Sigmod, 1993.

[Cat93] R.G.G. Cattell. The Object Database Standard : ODMG-93. Morgan Kaufmann, 1993.

[FG94] B. Finance and G. Gardarin. Rule-based query optimizer with adaptable search strategies. In Data and Knowledge Engineering, 13(2), 1994.

[FV94] D. Florescu and P. Valduriez, Rule-based query processing in the IDEA system, In Proceedings of Int. Symp. on Advanced Database Technologies and Their Integration, Nara, Japan, October 1994.

[GGT95] G. Gardarin, J.R. Gruser and Z.H. Tang, A cost model for clustered object -oriented databases. In Proceedings of 21st International Conference on Very Large Databases, Zurich, 1995.

[GV92] G. Gardarin and P. Valduriez. ESQL2 : An object-oriented SQL with F-Logic semantics, In Int. Conf. on Data Engineering, Phoenix, 1992.

[KBZ86] R. Krishnamurty, H. Boral, C. Zaniolo. Optimization of nonrecursive queries, In Proceedings of 12th International Conference on Very Large Databases, 1986.

[KKS92] M. Kifer, W. Kim and Y. Sagiv. Querying object-oriented databases, In Proceedings of ACM-SIGMOD International Conference., 1990.

[KKD89] K.C. Kim, W. Kim and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, Object-oriented concepts, Databases, and Applications, 371-392. AddisonWesley, 1989.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In Proceedings of the ACM-SIGMOD International Conference., Atlantic City, 1990.

[KGM91] T. Keller, G. Graefe and D. Maier. Efficient Assembly of Complex Objects. In Proceedings of ACM-SIGMOD International Conference on Management of Data, 148-157., 1991.

[Mel93] J. Melton, editor. IOS/ANSI Working Draft Database SQL (SQL3). X3H2-93-091 ISO DBL YOK-003, 1993.

[SC89] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In Proceddings of SIGMOD Int. Conf. on Management of Data, New Jersey, May 1990.

[Sha86] L. Shapiro, "Join Processing in Database Systems with Large Main Memories", ACM Transactions on Database Systems, Vol 11 n°3, p239-264, September 1986.

[TL91] K.L. Tan and H.Lu, A note on the strategy of multiway join query optimization problems, In Proceedings of SIGMOD Int. Conf. On Management of Data, 1991.

[Yao77] S.B. Yao. Approximating the number of accesses in database organizations. In Comm. of the ACM, 20(4) :260, April 1977.