# A graph based data structure for efficient implementation of main memory DBMS's

Philippe Pucheral, J.M. Thevenin

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 978

*Programme 4*

# A GRAPH BASED DATA STRUCTURE FOR EFFICIENT IMPLEMENTATION OF MAIN MEMORY DBMS's

**Philippe PUCHERAL**
**Jean-Marc THEVENIN**

**Février 1989**

*RR_0978*

# DBGRAPH:
# UNE STRUCTURE DE DONNEES POUR LES SGBD ORIENTES GRANDES MEMOIRES

Philippe **PUCHERAL***, Jean-Marc **THEVENIN****

*Laboratoire MASI, Université Paris 6
4, place Jussieu, 75005 Paris, France

**INRIA France
BP. 105, 78153 Le Chesnay, France
Network address theven@madonna.inria.fr

## Resumé

Compte tenu de la rapide croissance de la taille des mémoires centrales, il devient envisageable de conserver la partie active de la base de données en mémoire primaire. Afin de profiter de ce contexte, une nouvelle organisation physique des données est proposée. Cette organisation vise deux objectifs: être compacte et décomposable afin d'assurer que la base de données active tienne toujours en mémoire et accélerer l'exécution des opérations de l'algèbre relationnelle étendue en terme de temps CPU. Les tuples, index et résultats intermédiaires sont intégrés dans une structure de données unique appelée DBGraph. Les tuples et les valeurs d'attribut sont stockés séparément et constituent les noeuds du DBGraph. Les arêtes reliant tuples et valeurs sont matérialisées par des OIDs et constituent un graphe biparti. Ce graphe précompile les opérations de sélection, jointure et fermeture transitive. Une exécution ensembliste des opérations de l'algèbre relationnelle est générée par un parcours en *largeur d'abord* du DBGraph alors qu'une exécution en pipeline peut être obtenue par un parcours en *profondeur d'abord* du même DBGraph. Ces deux stratégies ont une complexité temporelle équivalente. Une évaluation du coût de stockage des DBGraphs démontre leur compacité. Enfin, des évaluations de performances montrent la supériorité, dans un contexte grande mémoire, de l'algorithme de fermeture transitive appliqué à un DBGraph sur celui utilisant des index de jointure, toujours considéré comme l'un des plus performants.

# A GRAPH BASED DATA STRUCTURE FOR EFFICIENT IMPLEMENTATION OF MAIN MEMORY DBMS's

## Abstract

Considering that in future DBMS's it will be possible to hold the active database in main memory, a new physical database organization is proposed. This organization aims two objectives: be compact and decomposable such as the active database always fits in main memory and speed up extended relational algebra in term of CPU time. Tuples, indices and temporary results are integrated in a unique data structure called DBGraph. Tuples and values are stored separately and constitute the vertices of the DBGraph. Edges between tuples and values are maintained using OID's in order to constitute a bipartite graph. This graph precompiles selection, join and transitive closure operations. Set oriented execution of relational algebra is generated using a breadth first traversal of this graph while pipeline execution is produced using a depth first traversal. The two strategies lead to the same temporal complexity. Storage cost evaluations demonstrate the compactness of DBGraph. Performance evaluations show that in a main memory context transitive closure on DBGraph outperforms transitive closure on join indices, still considered as one of the best algorithm.

Numéro de programme de recherche INRIA: 4

# A GRAPH BASED DATA STRUCTURE FOR EFFICIENT IMPLEMENTATION OF MAIN MEMORY DBMS's

Philippe PUCHERAL*, Jean-Marc THEVENIN**

*Laboratoire MASI, Université Paris 6
4, place Jussieu, 75005 Paris, France

**INRIA France
BP. 105, 78153 Le Chesnay, France
Network address theven@madonna.inria.fr

## Abstract

Considering that in future DBMS's it will be possible to hold the active database in main memory, a new physical database organization is proposed. This organization aims two objectives: be compact and decomposable such as the active database always fits in main memory and speed up extended relational algebra in term of CPU time. Tuples and indices are integrated in a unique data structure called DBGraph. Tuples and values are stored separately and constitute the vertices of the DBGraph. Edges between tuples and values are maintained using OID's in order to constitute a bipartite graph. This graph precompiles selection, join and transitive closure operations. Set oriented execution of relational algebra is generated using a breadth first traversal of this graph while pipeline execution is produced using a depth first traversal. The two strategies lead to the same temporal complexity. Storage cost evaluations demonstrate the compactness of DBGraph. Performance evaluations show that in a main memory context transitive closure on DBGraph outperforms transitive closure on join indices, still considered as one of the best algorithm.

## I INTRODUCTION

Upon the last years, technological progress raised up the idea that future DBMS's would have very large main memories. Therefore, the hypothesis that the complete database or the active database fit in main memory, becomes more realistic. The active database is defined as the subset of the database accessed by the current set of queries. This assumption has two main consequences on DBMS's design. First, transaction management must be redefined. Crash recovery when the primary copy of the database resides in main memory, constitutes the principal field of investigation in this area [DeWi84, Garc84, Eich87, Lehm87]. Secondly, the bottleneck of the system is no more the I/O but the CPU time required to perform an operation. Then physical database organization and query evaluation must be reconsidered to take advantage of this context [DeWi84, Amma85, Lehm86b, Naka87].

This paper focuses on physical database organization for efficient implementation of a Main Memory DBMS (MMDBS). The main objective is to speed up software execution of relational operations when the active database fits in main memory. In this context, data structures which precompile relational operations providing direct access to tuples in memory are well adapted.

Indeed, inverted data structures are no more limited by I/O. The second objective is to ensure compactness of data structures and efficient space utilization. It is a critical issue to ensure that the active database and all necessary indices fit in main memory.

Recent studies for MMDBS have been concentrated on efficient space utilization. Classical index structures have been reconsidered in this way. Although it is shown in [DeWi84] that AVL-Tree run faster that B-Tree in a main memory context, B-Tree will be prefered considering their compactness. The T-Tree index structure, merging the advantages of B-tree and AVL-Tree, was introduced in [Lehm86a]. A more radical approach [Amma85] uses as indices, arrays of tuple identifiers (TID's) pointing to tuples containing the indexed values. These arrays are simply sorted on the pointed values. Such a compaction allows to index each attribute of a relation. Some work was also dedicated to reduce the space occupancy of temporary results in order to save main memory for permanent data [Amma85, Bitt86, Lehm86b]. Finally, vertical partitionning [Hamm79] is well suited to fetch in memory only the relevant data for a given query.

Several kinds of indices can be added to a classical database organization to precompile relational operations. Inverted indices are widely used to perform selection. A generalized access path structure introduced in [Haer78] uses multi-relation indices which associate for each value one TID list per linked relation. Such indices are interesting for they can be used as well for selections as for precompiling joins. Specialized indices, called join indices, have been proposed in [Vald87] to prejoin relations. A join index materializes links between two relations with a table of two columns containing TID's linking tuples matching together. It was demonstrated that join indices lead to very efficient execution of join and transitive closure operations. These proposals satisfy the performance requirements of a MMDBS. Nevertheless, such indices suffer from the fact they are defined apart from the database architecture and then introduce extra storage and update costs. Furthermore, their use on temporary results is not obvious.

In order to get compact precompiling structures, some approaches integrate indices in the database storage architecture. These approaches store domain values apart from relations such that any domain value is stored only once [Miss82, Amma85]. In [Miss82], relations are not directly stored for they can be materialized through multi-relation domain indices. This organization favors join on basic relations but projection and operations on temporary results become critical. In [Amm85], relations and indices contain only pointers to domain values. Such approaches are well suited for MMDBS as they allow to store links between tuples and values in a very compact way with a reduced extra cost of update. Nevertheless query evaluation seems less efficient than with join indices since some links are not directly represented.

In this paper we propose an integrated database storage architecture to precompile extended relational algebra in a compact way. This architecture is based on a bipartite graph data structure called DBGraph. The whole database is stored as a DBGraph. This DBGraph contains a set of tuple vertices, a set of value vertices and edges connecting these two sets. Tuples are connected by one edge to each of their attribute values instead of containing directly these values. Domain values are also connected by one edge to each tuple referencing them for one of its attributes. Temporary results are linked to the tuples of basic relations from which they are extracted using temporary

edges. Thus they can be mapped onto the DBGraph in a uniform and compact way.

Selection, join on basic or temporary relations and transitive closure are speed up in a similar way by a simple traversal of this DBGraph. Selections are performed using edges between values and tuples. Links between tuples matching together for a Join R⊗S are established using edges linking tuples of R to values then edges linking values to tuples of S. Finally, the transitive closure of a relation is equivalent to the transitive closure of a sub-part of the DBGraph. As temporary results are linked to tuples of basic relations, they determine a set of entry vertices in the DBGraph which can be exploited by any operator. Depending on the graph traversal strategy applied on the DBGraph, a set oriented or a pipeline evaluation of a query can be produced. Pipeline evaluation suppress the need of temporary results. Finally, a DBGraph presents as good properties in term of storage cost as in term of performances. Their compactness is mainly due to the fact that any value is stored only once and that the same edges are used to precompile all operations. Furthermore, it is possible to partition the DBGraph in order to fetch in memory only the subpart of the DBGraph relevant for a query.

The main features of a DBGraph can be summarized as follows: (i) it is compact and decomposable, (ii) precompiles the queries of extended relational algebra, (iii) preserves its properties with temporary results and (iv) the queries can be evaluated in a set oriented or a pipeline way with a same level of performances.

The remainder of the paper is organized as follows. The DBGraph notion is formally introduced in section 2. Then two ways to perform extended relational queries on a DBGraph are studied in section 3. An efficient and compact implementation of a DBGraph is proposed in section 4. A storage cost evaluation compares this implementation of a DBGraph versus a Flat File organization in the same section. Finally, performances of a query execution are evaluated in section 5. Summary and future works are given in section 6.

## II MAPPING DATABASE INTO DBGRAPH

The DBGraph concept is primarily defined. Then, we show how selection and join operators can be expressed on a DBGraph. A query composed of a combination of these operators determines a subset of vertices of the DBGraph, called the set of pertinent vertices for this query. The query evaluation consists to apply traversal primitives on the DBGraph in order to reach these pertinent vertices.

### II.1 DBGraph definition

For the sake of clarity, we introduce notations for entities manipulated in the sequel of the paper. A database DB is defined as a set of relations denoted R and S (as only two relations will be useful in the following) and a set of value domains denoted Dj. The schema of a relation is an aggregation of attributes where r.k denotes the $k^{th}$ attribute of the relation R and $t_{r.k}$ denotes the value of this attribute for the tuple t. Each r.k takes its values on an attribute_domain denoted $d_{r.k}$. Several $d_{r.k}$

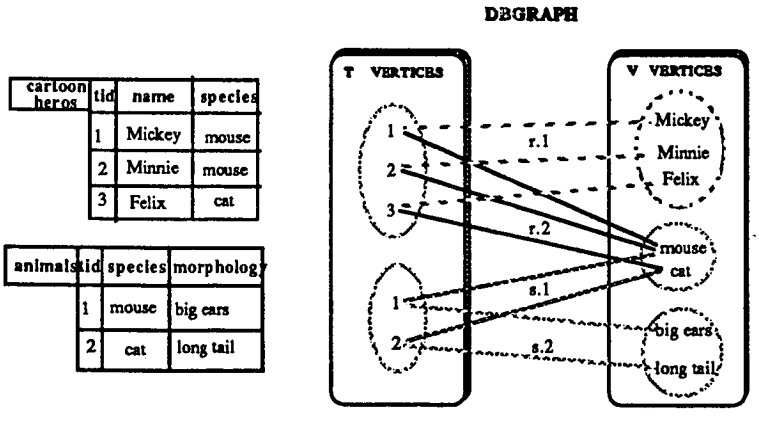
may share the same value domain Dj. Finally, we note T the set of all tuples of a database DB and V the set of all domain values of a database DB.

We can define an isomorphism between a database DB and a graph, called DBGraph. A DBGraph is a bipartite graph containing a set of tuple vertices, a set of value vertices and valued edges connecting these two sets. The set of tuple vertices holds all tuples of DB and the set of value vertices holds all domain values of DB. Each edge (t, v, r.k) of the DBGraph is an indirected valued edge connecting a tuple vertex t to a value vertex v. The valuation r.k of this edge specifies that the tuple t belongs to the relation R, that the value of the $k^{th}$ attribute of t is v and consequently that the domain of v is $d_{r.k}$. Then a tuple vertex is linked by an edge to each of its attribute value and a value vertex is linked by an edge to each tuple which references it for one of its attributes.The notion of DBGraph can be more formally defined as follows:

**Definition: DBGraph**

The DBGraph of a database DB is a valued bipartite graph DBG(X,A) where X=(T,V) is the set of vertices of DBG, A is the set of edges of DBG and the edge $(t, v, r.k) \in A$ iff $t \in T$, $v \in V$ and $t_{r.k} = v$.

**Properties:** A DBGraph is bipartite since T and V form a partition of X and there is not any edge connecting two vertices of T or two vertices of V. Then, each traversal of the DBGraph gives an alternance of tuples and values. Each couple of tuples, of the same or of different relations, having the same value for one of their attribute are connected by a path of length two. Finally, the relations form a partition of T and the value domains Dj form a partition of V.




For the sake of clarity, tuples are identified by a TID and the relation Cartoon_Heros (resp. Animals) is denoted by R (resp. S) to standardize valuations of edges.

**Figure 1:** instance of a DBGraph

## II.2 Querying a DBGraph

In the sequel, we consider all queries expressed on a database DB by a composition of the Select operator $\sigma_Q$ (where Q denotes the selection qualification), the Project operator $\pi_P$ (where P denotes the attribute list of projection) and the equi-Join operator $\otimes_M$ (where M denotes the equi-join predicate), defined in the usual way. Queries involving transitive closure operator are treated in a

separate section since transitive closure appears as a combination of the other relational operators. Aggregate functions and inequi-join operations are not discussed as DBGraph does not provide any optimisation of these functions. The select and join operators can be expressed on a DBGraph as follows.

- *select operator:*

  The select operator $\sigma_Q$ applied on a DBGraph, determines the subset $E_T$ of T containing all tuples of a given relation R which satisfies the qualification Q. To simplify, we assume Q to be a simple comparison predicate (r.k $\theta$ c) where c is a constant and $\theta$ is the comparator. The select operator is then expressed as follows:

  $E_T = \sigma_Q(T)$ *where* $E_T = \{t \in T / (v\ \theta\ c)$ *is true and* $(t, v, r.k) \in A\}$

  The extension of Q to conjunctions and/or disjunctions of predicates is handled performing unions and/or intersections of the $E_T$ sets corresponding to each predicate.

- *join operator:*

  Let Tr (resp. Ts) to be the subset of T vertices holding the tuples of relation R (resp. S). The join operator applied on Tr and Ts determines the set $E_{T,S}$ of couple of vertices corresponding to tuples matching together according to the join predicate (r.k=s.l). DBGraph properties implie that these tuple vertices are connected by a path of length two, as expressed below.

  $E_{r,s} = \otimes_M(Tr, Ts)$ *where* $E_{r,s} = \{(t1,t2) / t1 \in Tr, t2 \in Ts$
  *and* $(t1, v, r.k) \in A,\ (t2, v, s.l) \in A\}$

For a query QR, these two operators determine a subset of vertices of the DBGraph, called the set of pertinent vertices. This set can be reached in the DBGraph using two different strategies. The select and join operators defined earlier are set oriented. They induce consequently a traversal of a DBGraph from sets of vertices to sets of vertices in a similar manner as a *breadth first search* strategy [Sedg84]. On the other hand, the same set of pertinent vertices can be reached using a *depth first search* strategy [Gibb85]. This second strategy consists to evaluate complete paths of a graph one by one. Such traversal of the DBGraph is equivalent to a pipeline execution of a relational query and does not generate any temporary result.

## III STRATEGIES TO TRAVERSE A DBGRAPH

In this section, set oriented strategy and pipeline strategy are detailled and compared. For set oriented strategy, traversal primitives are first defined. Then selection, join and transitive closure algorithms are expressed on basic relations using these primitives. The introduction of temporary results in the evaluation of queries is discussed. Finally, pipeline evaluation of a query on a DBGraph is proposed as a good candidate strategy.

### III.1 Set oriented traversal primitives

Two primitives are defined to navigate in a DBGraph DBG(X,A) with X=(T,V).

- *succ_val primitive:*
  This primitive delivers the set $E_T$ of all tuples of a given relation R whose $k^{th}$ attribute value belongs to a given value set $E_V$. It is an application from V to T which associates a set of T vertices to a set of V vertices according to the valuations r.k of the corresponding edges.

  $E_T = succ\_val(E_V, r.k)$ *where* $E_T = \{ t \in T / v \in E_V$ *and* $(t, v, r.k) \in A\}$.

- *succ_tup primitive:*
  This primitive performs the projection of a given tuple t of a relation R on its $k^{th}$ attribute.

  $v = succ\_tup(t, r.k)$ *where* $v \in V$ *and* $(t, v, r.k) \in A$.

Combinations of these two primitives are allowed since their results and entry parameters are compatible together. As these primitives are set oriented, their combination determines a *breadth first traversal* of the DBGraph. We add the classical operations of union, intersection and cartesian product (denoted X) on $E_T$ sets, in order to construct intermediate and final results.

### III.2 Select and join algorithms

We have previously shown how select and join operators can be formally expressed on a DBGraph. We give now a transcription of these operators in term of traversal primitives. Queries involving select and join operators will thus be interpreted as a sequence of traversal primitives.

- *Select algorithm on DBGraph:*

```
Function σQ (Tr) : Er
        /* Tr is the subset of T vertices holding the tuples of the relation R */
        /* Q is a comparison predicate (r.k θ c) */
        begin
                Ev := SelectQ(V);
                Er := succ_val(Ev, r.k);
        end
```

The function $Select_Q(V)$ builts the set $E_V = \{v \in V / (v\ \theta\ c)=true\}$. This function can be optimized using indices on V as it will be detailled in section IV. In this case, selection on DBGraph yields similar performances as selection on inverted indices in main memory. The extension of Q to complex qualifications is obvious as discussed section II.2.

• *Join algorithm on DBGraph:*

```
Function ⊗1M(Tr,Ts) : Er,s
    /* Tr (resp. Ts) is the subset of T vertices holding the tuples of relation R (resp. S) */
    /* M is the join predicate (r.k=s.l) and dr.k, ds.l vary on the same domain Dj*/
    begin
        Er,s := ∅;
        for each v∈Dj do
            Er := succ_val(v, r.k);
            Es := succ_val(v, s.l);
            Etemp := Er X Es;
            Er,s := Er,s ∪Etemp;
        end for
    end
```

As domains are shared by several relations, Card(Dj) can be high in comparison to $Card(T_r)$ or $Card(T_s)$, where Card denotes the cardinal of a set. We propose a second join algorithm, based on a different traversal of the DBGraph, which run faster than the previous one in this particular case.

```
Function ⊗2M(Tr,Ts): Er,s
    /* We assume Card(Tr) < Card(Ts) < Card(Dj) */
    begin
        Er,s := ∅;
        for each t∈Tr do
            v := succ_tup(t, r.k);
            Es := succ_val(v, s.l);
            Etemp := t X Es;
            Er,s := Er,s ∪Etemp;
        end for
    end
```

In both algorithms, the union $E_{r,s} \cup E_{temp}$ can be advantagcously replaced by a concatenation since generation of double in $E_{r,s}$ is impossible. A DBGraph precompiles join operations in a similar way as join indices since comparisons between tuples are avoided during all the operation. The join is reduced to traverse links in main memory. Numerous comparisons with join indices will be introduced in the sequel of this paper. Indeed, the use of join indices is currently considered as one of the most convenient way to speed up join and transitive closure operations using direct links between tuples.

### III.3 Transitive closure on a DBGraph

Recently, the database community has manifested a strong interest in efficient evaluation of recursively defined relations [Hens84, Banc86]. The most popular approach to reach this goal is to define dedicated operators to compute the transitive closure (called TC) of a relation [Han86, Gard86]. TC algorithms have then becomed central algorithms in KBMS's.

Computing the TC of a relation R on attributes r.k and r.l, where $d_{r.k}$ and $d_{r.l}$ vary on the same domain Dj, consists to add in R all tuples deductible by a transitive application of joins (for clarity, R is assumed binary). In other words, if $t1_{r.l}=t2_{r.k}$, a new tuple $(t1_{r.k},t2_{r.l})$ must be deduced and added in R. For instance, if t1=(a,b) and t2=(b,c) then the tuple (a,c) is generated. When a

recursively defined relation is queried, only the relevant sub-set of its TC is built. For instance, if we consider the academic example of the Ancestor relation (recursively defined from the Parent basic relation), the retrieval of all John ancestors does not imply the generation of the whole Ancestor relation. The problem is solved by pushing up selection before processing recursion [Hens84, Gard86]. The corresponding TC algorithm can be computed by a loop of relational operators, as follow:

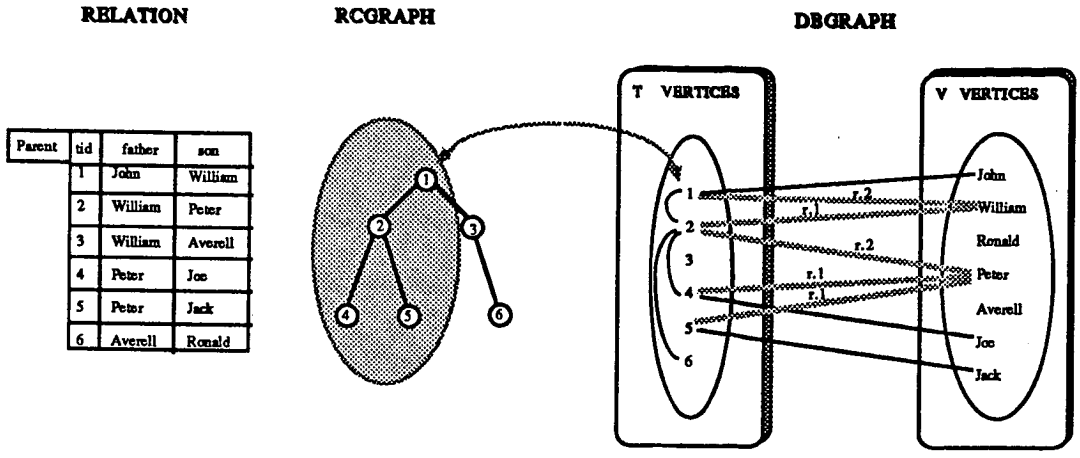
```
Function TC_Q(R):TCR
    /* compute the TC of R according to the initial query selection Q */
    /* and store the result in TCR */
    begin
        ΔR := π_P(σ_Q(R));                    /* σ_Q is applied to the basic relation */
        TCR := ΔR;                            /* before processing recursion */
        while ΔR ≠ Ø, do
            ΔR := π_P( ⊗_M(ΔR,R));            /* M= (Δr.l=r.k) and P= (Δr.k, r.l) */
            ΔR := ΔR - TCR;
            TCR := TCR ∪ ΔR;
        end while
    end
```

The difference ΔR := ΔR - TCR is mandatory to avoid TC program to loop forever in case of cyclic data (e.g sequence of R tuples like (a,b), (b,c), (c,a)). Numerous proposals have been made to optimize this algorithm in several ways: reduction of the number of loops [Ioan86], propagation of temporary results by join waves [Han86], speed up joins in the loop using join indices [Vald86].

A more natural and efficient way to elaborate the transitive closure of a relation consists to compute the TC of the sub-part of the DBGraph corresponding to this relation. To make the recursion easier to express, we introduce the notion of Relation_Closure_Graph (denoted RCGraph) which is directly derived from the DBGraph notion. The RCGraph of R on attributes r.k and r.l is a graph in which vertices represent tuples of R and two vertices t1 and t2 are connected by an edge (t1, t2) if the corresponding tuples match together according to the auto-join predicate (r.k=r.l). This RCGraph remain always virtual and traversals of a RCGraph are translated on a DBGraph using the classical *succ_val* and *succ_tup* primitives.

**Definition: Relation_Closure_Graph**
The RCGraph(Tr,Ar) of a relation R on attributes r.k and r.l is a virtual graph derived from the DBGraph(X,A) with X=(T,V) as follow: Tr is the subset of T vertices holding the tuples of relation R and (t1, t2) ∈ Ar iff t1, t2 ∈ Tr and (t1, v, r.k) ∈ A, (t2, v, r.l) ∈ A.

**Properties:** Two vertices ti and tj of the RCGraph(Tr,Ar) are transitively connected if there exists a path (ti<—>tk, tk<—>tl, ..., tn<—>tj) ∈ Ar. Thence, there is a direct mapping between the TC of R and the TC of its RCGraph.




The relation Parent is denoted by R to standardize valuations of edges. To simplify the figure, only the edges belonging to vertices of the encircled subpart of the RCGraph are represented in the DBGraph. The grey edges of the DBGraph materialize the virtual edges of this subpart of the RCGraph. These virtual edges are also represented between the T vertices for clarity.

**Figure 2:** Mapping between a RCGraph and a DBGraph

*Transitive closure algorithm on RCGraph:*

The query selection on the recursive relation determines a set $E_t$ of entry vertices in the RCGraph. According to this initial selection, the TC of the RCGraph consists, for each t of $E_t$, to find all vertices (called the descendants of t) reachable from t by a path. A visited vertex is marked to avoid going twice through the same path. This prevents infinite looping of the TC algorithm even in presence of cyclic data. The result of each RCGraph traversal is a spanning tree of root t over the descendants of t. A primitive is introduced to perform such a traversal on a RCGraph(Tr,Ar):

- *neighbor_tup primitive:*
  This primitive delivers the set $N_t$ of vertices directly connected (i.e. by only one edge) to the non marked vertices of an entry set $E_t$. Vertices of $N_t$ are then marked and not anymore considered for graph traversal.

  $N_t = neighbor\_tup(E_t)$ *where* $N_t = \{t' \in Tr \mid t \in E_t$ *and* $(t, t') \in Ar$ *and* $mark(t)=false\}$

  Note that parameter and result of *neighbor_tup* are homogeneous. Thus, a recursive application of this primitive is valid.

As the *neighbor_tup* primitive is set oriented, it defines a traversal of a RCGraph from sets of vertices to sets of vertices using a *breadth first search (BFS)* strategy. Such closure algorithm is well known in graph theory and [Sedg84] demonstrates that this algorithm stops and is of temporal complexity O(m) (where m=Card(Ar)). A RCGraph traversal for each element t of $E_t$ is accomplished using the following algorithm:

```
Function BFS(t):Desc
    /* Desc is the set of descendants of t */
    begin
        ΔDesc := neighbor_tup(t);
        Desc := ΔDesc;
        while ΔDesc ≠ ∅, do
            ΔDesc := neighbor_tup(ΔDesc);
            Desc := Desc ∪ ΔDesc;
        end while
    end
```

The difference operation between ΔDesc and Desc is unnecessary while cycles in data are avoided by marking all visited vertices and the union Desc∪ΔDesc can be replaced by a concatenation. Recall that the RCGraph notion remain always virtual. Then, the BSF function is translated on the original DBGraph in term of *succ_val* and *succ_tup* primitives.

This TC algorithm yields excellent performances in a main memory environment due to the fact that the DBGraph pre-compiles the whole TC operation. Section 5 shows that graph traversal outperforms all others methods when the sub-part of the DBGraph corresponding to the RCGraph fits entirely in main memory. Nevertheless, the behaviour of this algorithm is rapidly degraded if this hypothesis is not satisfied. The physical storage of the DBGraph has especially been studied to allow incremental loading of the pertinent sub-part (for a query) of a DBGraph. Note that DBGraph is a well suited support for a wide family of recursive processes. All processes expressed as traversal recursion [Rose86] can be speed up using a DBGraph structure.

### III.4 Dealing with temporary results

Join operations are generally speed up by the use of clustered or inverted indices on join attributes, join indices [Vald87] or multi-relations indices [Haer78]. Nevertheless, it is often difficult or even impossible to apply such indices on temporary relations. As we consider that join are frequently preceded by a selection on one (or the two) relation to be join, this limitation is severe. Solutions are proposed in [Vald87] to ensure the validity of join indices after selections. Selections produce a list of TID's corresponding to the relevant tuples. This list is then semi-joined with the join index in order to produce a new valid join index on temporary relations.

The DBGraph storage of relations allows a much more efficient resolution of this problem. As temporary results contain only links to tuples of basic relations from which they are extracted, they represent a simple extension of a DBGraph. Temporary results are represented as a set T' of temporary vertices connected with the corresponding T vertices via temporary edges. The degree (number of edges) of a temporary vertex is equal to the number of basic tuples involved in this temporary result. For instance, if temporary result is issued from selection (resp. join), each temporary vertex is connected with a unique vertex (resp. two vertices) of T, as illustrated figure 3.

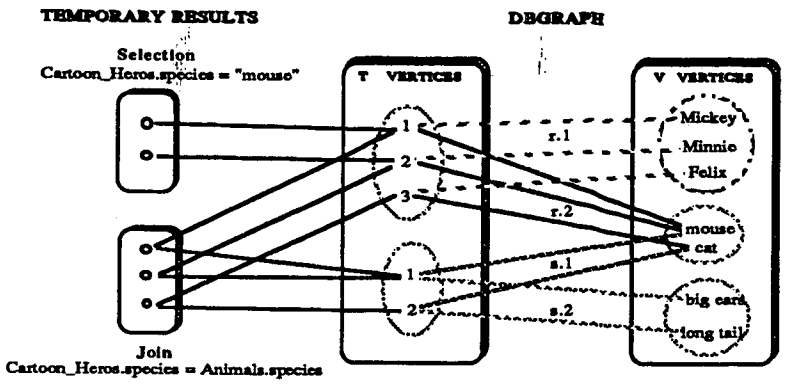



**Figure 3: temporary result from a selection and from a join**

This representation of temporary results presents three major advantages. First, temporary results are as compact as possible since they do not contain any value. It is on prime importance in MMDBS since it enforces the hypothesis that the active database fit in main memory. Secondly, project operations are posponed at the end of the query evaluation. The query optimizer is then discharged of the task to generate projections on non relevant attributes before join operations. This task has been proved to be time consuming in [Amma85]. Finally, temporary results preserve links to basic tuples. Then, a temporary result determines a set of entry vertices in the DBGraph which can be directly exploited by any operator. Consequently, join of temporary relations are speed up in a similar way as join of basic relations by a simple DBGraph traversal. Note however that a supplementary edge must be traversed for each element of the temporary result in order to reach the corresponding basic tuples.

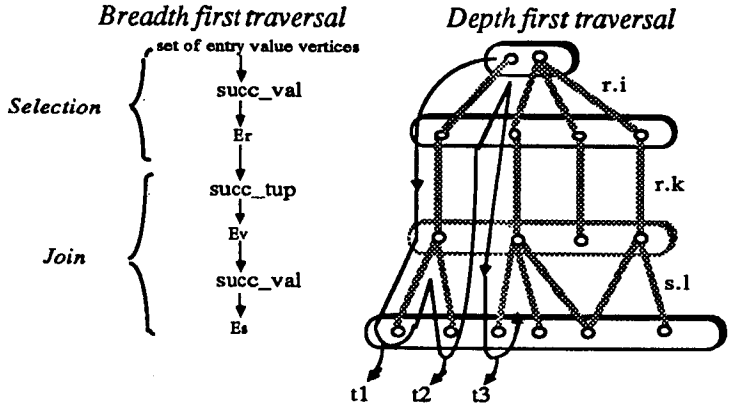
### III.5 Pipeline strategy versus set oriented strategy

A same query can be evaluated using a breadth-first traversal or a depth-first traversal of the DBGraph. The first strategy provides a set oriented execution of relational algebra when the second one produces *one tuple at a time* such as a pipeline strategy. Pipeline strategy has already been implemented in some relational systems. The pipeline strategy offers two major advantages. Temporary results have no more to be generated then main memory stands only for permanent data. In addition, tuples already produced can be displayed while query processing is still in progress. Nevertheless, pipeline execution is limited to the subpart of a query involving selection, join and projection operators since difference, intersection, aggregate and sort operators are purely set oriented. Furthermore, computing a join in a pipeline way without specialized structures can provide as low performances as those of a Nested Loop join algorithm.

The pipeline execution of a query on a DBGraph is informally introduced below. Let us consider a generic query QR on the form ($\sigma_Q$R1 $\wedge$ R1$\otimes$R2 $\wedge$ R2$\otimes$R3 $\wedge$... $\wedge$ Rn-1$\otimes$Rn). The number of joins in QR determines the length of a *pertinent path* connecting a tuple of R1 to a tuple of Rn and the corresponding join predicates determine the successive valuations of edges belonging to this path. A *pertinent path* covers all vertices needed to produce one tuple of the query result. The pipeline evaluation of the query QR on a DBGraph is expressed by a depth-first traversal as follows. The selection on R1 determines a set of entry value vertices in the DBGraph. Each value constitutes the

root of one or more paths in the DBGraph. These paths are explored one after the other and a result tuple is generated at each time that the extremity of a *pertinent path* is reached. Backtrack is applied at the extremity of each pertinent path or when a path fails. It is shown in [Gibb85] that the temporal complexity of a depth-first search is O(max(n,m)) (where n=Card(X) and m=Card(A)). Then DBGraph appears well suited for the pipeline evaluation of such queries since depth-first traversal yields similar performances as breadth-first traversal (O(m)). When selections in QR are applied on several relations, they can be handled in two ways. One solution is to apply first selections in a set oriented way. Selected vertices of the corresponding relations are marked. Then, non marked vertices are not considered during the *pertinent paths* exploration. A second solution consists to check the selection criteria for each tuple vertex reached during the *pertinent paths* exploration. Queries involving successive joins on the form (R1⊗R2 ∧ R1⊗R3... R1⊗Rn ) are more complex to evaluate. A simple depth first traversal is no more suited as several paths must be explored in parallel. Evaluation of such queries remain possible with good performances but the method is more complex to implement.

Breadth first traversal and depth first traversal are summarized figure 4 for a query expressed by a selection followed by a join.




A selection is first applied on attribute i of the relation R followed by a join r.k=s.l. A *pertinent path* for this query owns three edges with the successive valuations r.i, r.k, s.l. To clarify the figure, only a subset of candidate paths is represented. Four tuples of R satisfies the selection criteria and three tuples of R match with six tuples of S giving seven tuples in the result denoted by t1, t2, t3 ...

**Figure 4:** BFS and DFS strategies on a DBGraph

## IV IMPLEMENTATION OF A DBGRAPH

### IV.1 Physical implementation

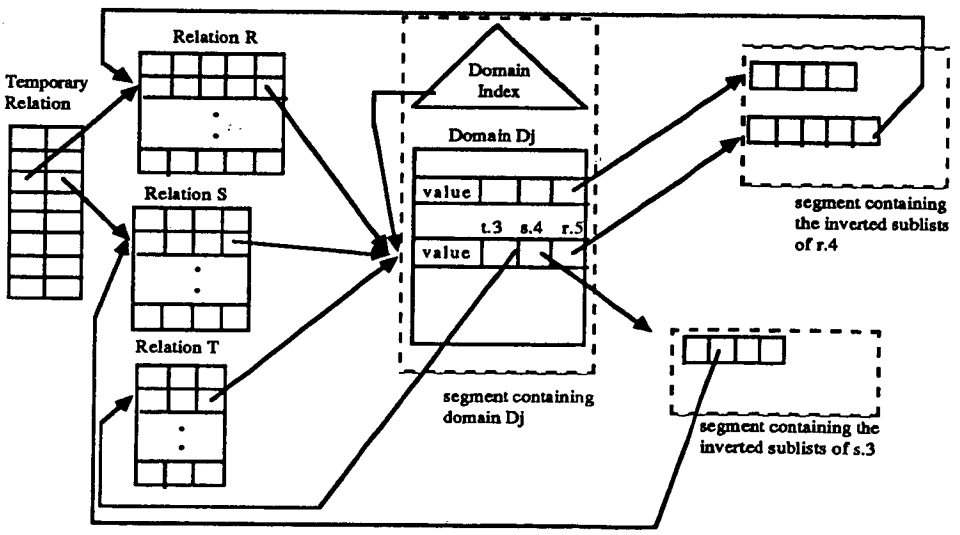
Numerous physical representations of graphs can be found in the litterature. We detail below a particular implementation of a DBGraph(X,A) where X=(T,V), well adapted to MMDBS.

Domain values are stored only once to preserve compact data structures. As the $D_j$ domains form a partition of V, all values varying on the same domain can be clustered in a separate segment. Thus, it is possible to load the values of one domain independantly of the others and to take advantage of

vertical partitionning. For the same reasons, tuples of one relation R are stored in one segment as the relations form a partition of T. Each object stored in a segment has a unique and invariant identifier called OID. Then, tuples and values can be referenced by OID's. Indices speed up selections on domain values. The index structure can be any one recommanded for MMDBS [Lehm86b]. Its particularity is to contain only OID's referencing the key values stored in the domain [Amma85]. In case of variable length keys, the gain in term of storage cost is important.

In the formal description of a DBGraph, edges of A are not oriented and may be traversed in both directions. In the implementation, an edge (t, v, r.k) is split in two arcs (oriented edges) materialized by OID's. The OID corresponding to an arc (v—>t) is stored in an inverted list attached to the value v. The valuation r.k of this arc is represented by the fact that inverted lists are subdivided in as much sublists as there is attributes r.k varying on the value domain. The valuation r.k of the arc (t—>v) is represented by the fact that the tuple t belongs to the segment of relation R and that the corresponding OID is stored in place of the $k^{th}$ attribute. Then, valuations of arcs are implicit and do not compromize the compactness of a DBGraph.

In summary, tuples of permanent relations are implemented via arrays of OID's and values are stored with collections of inverted sublists of OID's. The shortest way to store the inverted sublists is to put them one after the other behind the corresponding value, as in [Haer78]. We did not choose this solution for we want to be able to fetch in memory the inverted sublists corresponding to one attribute r.k independently of the others. Thus, in accordance with the vertical partitionning strategy adopted for domain values, all the inverted sublists corresponding to attribute r.k are stored in one segment. Domain values are followed by an array, called sublist array, containing OID's referencing the sublists. It contains one entry per attribute r.k varying on the domain and is indexed by r.k. Inverted lists corresponding to key attributes contain only one OID. In this case, many space is saved storing the OID directly in the sublist array. The physical implementation of a DBGraph is represented figure 5.




The temporary result is issued from a join between R and S.
The attribute t.3 is assumed to be a key attribute

**Figure 5**: physical implementation of a DBGraph

## 4.2 Storage cost evaluation

Flat file organization (FF) and DBGraph organization (DBG) are evaluated in term of storage cost. For the sake of simplicity we assume for DBG that domain index are stored as array of OID sorted on the referenced values, like in [Amm85]. In a similar way, FF indices are supposed to be stored as arrays of couples (value, TID) sorted on the values. In the sequel, $p$ denotes the size of an OID or a TID. We assume also that all domains of a DBGraph are indexed in order to speed up relational queries in all cases.

Given one domain containing $N$ values with an average length $l$ and $a$ attributes varying on this domain with an average cardinality $C$ for each relation, the storage cost of the corresponding subpart of the DBGraph yields:

$$
\begin{aligned}
Cost(DBG) &= N(l+ap) && \text{/* size of the domain segment containing } N \text{ values */}\\
&&& \text{/* with their corresponding sublist array */}\\
&+ aCp && \text{/* size of } a \text{ segments containing all inverted sublists */}\\
&&& \text{/* of one attribute */}\\
&+ aCp && \text{/* size of } a \text{ columns of OID in the relations */}\\
&+ Np && \text{/* size of the domain index */}\\
&= aC\left(2p + \frac{N}{aC}(l+p(a+1))\right)
\end{aligned}
$$

For a Flat File organization using $k$ indices, let an evaluation for the storage cost be as follow:

$$
\begin{aligned}
Cost(FF_k) &= aCl && \text{/* size of } a \text{ columns of values in the relations */}\\
&+ k(C(l+p)) && \text{/* size of } k \text{ indices */}\\
&= aC(l + k/a(l+p))
\end{aligned}
$$

In order to compare the storage cost of the two methods, we introduce the concept of domain selectivity that can be formalized as follows:

$$
S = \frac{\textit{number of domain values}}{\textit{number of attribute values varying on the domain}} = \frac{N}{aC}
$$

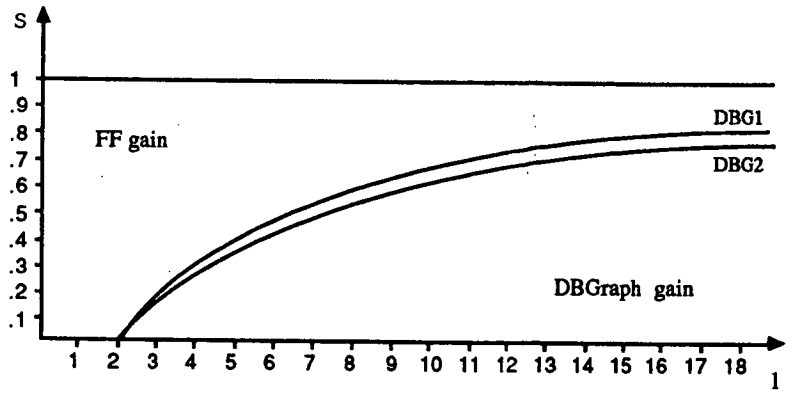
Then the difference between the two methods in term of storage cost depends on four parameters: $S$, $l$, $a$ and $k$. The cost equality is expressed by the following equation:

$$
\begin{aligned}
&Cost(FF_k) = Cost(DBG)\\
&aC(l + k/a(l+p)) = aC(2P + S(l+p(a+1)))\\
&S = \frac{l + k/a(l+p) - 2p}{l+p(a+1)}
\end{aligned}
$$

In case there is no index for the flat file organization ($k = 0$), we get:

$$
S = \frac{l-2p}{l+p(a+1)} = 1 - \frac{p(a+3)}{l-p(a+1)}
$$

Comparisons between FF without index and DBG are summarized figure 6 in a similar way as in [Miss82]. The plotted curves indicates the most efficient organization according to different values of $S$ and $l$ parameters. Average length of values $l$ is expressed in units of $a$ on the abscissa.




l is expressed in unit of p.
Values of the dependant variable: k=0.
Curve DBG1 corresponds to a=1, and curve DBG2 corresponds to a=2.

**Figure 6**: DBGraph versus Flat File organization without index

Each curve devides the plan in a gain region for FF and a gain region for DBGraph. They show that DBGraph is generally more space consuming than flat file organization without index. This extra-cost diminishes when domain selectivity decreases ($S$ becomes low) or $l$ increases. Thus, DBGraph should not introduce an extra-cost for domain containing enumerated types. For domain containing short key attributes like numbers, the extra-cost is compensated by two facts: all joins, generally performed on key attributes, will be speed-up by the DBGraph and $S$ can be low as several relations share the same set of keys. Furthermore, the evaluation do not take into account that inverted sublists for key attributes are directly stored in the sublist array. It is also important to point out that losts due to high domain selectivities or short values on several attributes can be balanced by large gains obtained on a few enumerated type domains. It can be show that for relation schemas introduced in the Wisconsin benchmark [Bitt83], a DBGraph organization is well-suited. Defining a new attribute on a domain implies to add a new entry on the sublist array of each value of this domain. To preserve efficient storage, the new attribute range of values should have a wide intersection with the values already defined in the domain, which decreases the domain selectivity. This phenomena is shown by a comparison of the three curves.

Figure 7 gives a comparison of DBGraph versus flat file organization using one join index or $k$ selection indices (with $k$=0, 1 or 2) for a domain holding two attributes. To simplify the evaluation of join indices size, we consider join on a key attribute such as at most one tuple of the second relation matches with each tuple of the first relation. It is a favorable case for join indices. The

space occupancy of a flat file organization including join indices is evaluated according to the following formula:

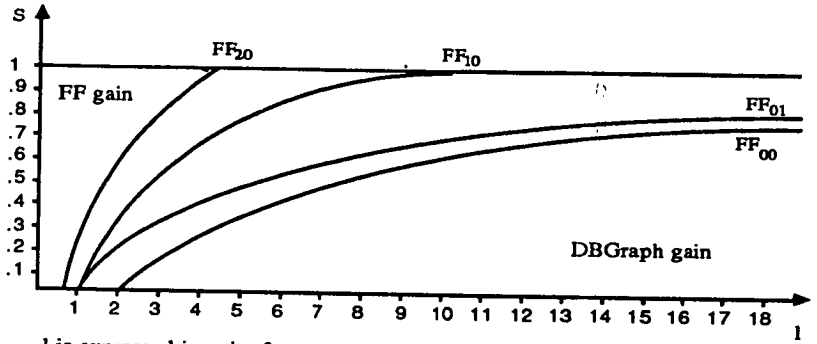
$$Cost(FF_{kj}) = Cost(FF_k) \quad /* \text{ size of FF organization using k selection indices } */$$
$$+ j(C\,2p) \quad /* \text{ size of j join indices containing C rows of two TID's } */$$

Then the equivalence between DBGraph and flat files with join and selection indices in term of storage cost is given by the following formula:

$$Cost(FF_{kj}) = Cost(DBG)$$

$$aC(l + k/a(l+p) + 2jp/a) = aC(2p + S(l + p(a+1)))$$

$$S = \frac{l + k/a(l+p) + 2jp/a - 2p}{l + p(a+1)} \qquad \text{with a=2, k=0 and j=1, we get} \quad S = \frac{l-p}{l+3p}$$




l is expressed in unit of p.
Values of the dependant variable: a=2,
Curve $FF_{20}$ corresponds to k=2, j=0; curve $FF_{10}$ corresponds to k=1, j=0;
curve $FF_{01}$ corresponds to k=0, J=1 and curve $FF_{00}$ corresponds to k=0, j=0 and then is equivalent to curve DBG2

**Figure 7**: DBGraph versus Flat file organization with indices

It appears clearly that DBGraph becomes rapidly the best candidate organization as soon as indices are added in the flat file schema. As the number of indices determines the performances of software queries evaluation in a main memory context, the superiority of a DBGraph organization is obvious. The extra-cost involved by selection index is strongly dependent of value lenght *l*. Comparing curves $FF_{01}$ and $FF_{10}$ show that selection indices are much more space consuming than join indices. This is mainly due to the fact that join indices contains only TID's instead of values. For the same reasons, the gap between $FF_{01}$ and $FF_{00}$ curves decreases progressively as *l* increases since TID size can then be neglected in regard to value size.

## IV.3 Updating DBGraph and concurrency control handling

Updating a tuple in a DBGraph consists to delete links with old values and to create links with new values for each updated attribute. In most cases values are shared by several attributes. Then insertion (resp. suppression) of one link just incure one update in the corresponding inverted sublist. When values are not shared, three pages must be updated in order to update the domain

index, insert (resp. suppress) the value in the domain and create (resp. delete) the corresponding inverted sublist. This scattering of data, due to the partitionning of the DBGraph, could introduce undesirable I/O. Mecanisms of differed updates on disk, generally used in MMDBS [Garc84] [DeWi84], are well adapted to avoid this drawbrack.
Insertion and suppression of links in a DBGraph have the same complexity and are quite simple. This property is interesting if we make the comparison with join indices. At each insertion of a new tuple in a relation, join index maintenance requires to select all tuples of the other relation matching with this tuple. It is argued that this overhead incured by join indices can be shared largely with referencial integrity checking in case of join on foreign keys. Nevertheless, there is still an overhead while the DBGraph precompiles the referencial integrity checking as well as selection and join operations.

As all operations are performed on the DBGraph, there could be heavy contention on that structure. We envision to suppress lock maintenance on domain indices using a multi-level transaction protocol [Weik84]. This proposal is based on the fact that logical operations on indices (such as insert a new value) commute whereas operations on their physical structure (such as node splitting in a B-Tree) do not commute. The multi-level transaction protocol allow to temporarily lock indices during the update in order to ensure the logical operation atomicity then to release these locks at the logical operation end. Therefore contention on indices is solved but phantom problems [Eswa76] appear. Considering that each entity addressable with an OID can be locked and that values are stored separately from the domain indices, phantoms can be avoided in DBGraphs locking domain values or inverted sublists during the all transaction. Then, only the minimal subpart of the domain index remain locked after the update. It could be a response to the critical problem of locking indices [Baye77].

## V PERFORMANCE EVALUATIONS

For the sake of conciseness, TC algorithm is the only one evaluated in details since transitive closure is expressed as a combination of the other operators. Selection algorithm on DBGraph yields the same level of performances as classical methods using inverted indices. Its efficiency is determinated by the index structure chosen to perform the $Select_Q(V)$ operation, which is independant of the DBGraph structure themselve. Thus, selection algorithm is not further detailed. The first join algorithm on DBGraph $\otimes 1_M(R,S)$ is linear in Card(Dj) as the second one $\otimes 2_M(R,S)$ is linear in Card(R), where R is the smallest relation to be joined. The DBGraph structure pre-compiles join operations in a similar way as join indices whose effectiveness has no more to be proved [Vald87]. The major difference is that join index provides direct links between tuples matching together as DBGraph provides only transitive links. Then join on permanent relations runs faster with join indices but the gap is minor considering the main memory context. In compensation DBGraph is more efficient to perform join on temporary results, as temporary result determines a set of entry vertices in the DBGraph which can be directly exploited by the join operator. On the contrary, the use of join indices on temporary results requires costly

transformations as detailed section III.4. The Project operator constitutes the main drawback of the DBGraph structure. It requires one access to each value belonging to the attribute list of projection. This drawback is balanced by the fact that projections are posponed at the end of query evaluation and then performed only once.

According to the main memory environment, evaluations of TC algorithms are expressed in CPU time. Relation_Closure_Graph traversal is compared to join loops on join index which is considered as one of the most efficient TC algorithm [Vald86]. Execution time of the original selection and time to compose the final result will be neglected as they are constant and independant of the two methods. We use in the evaluation parameters from [Banc86], well suited to model different types of graphs:

*di = in degree of a node, giving the average number of edges entering a node,*
*do = out degree of a node, giving the average number of edges going out a node,*
*h = height of the graph, giving the longest path in the graph.*

Thus, a genealogical tree in which any parent has three childrens and any child has two parents will be defined by *di*=2, *do*=3, *h*=number of generations. The expansion factor, modeling the average number of vertices accessed at each iteration of a breadth first traversal, is defined by $E = di / do$. For the join loops method, $E$ is an approximation of join selectivity at each iteration. Other parameters used in the evaluation are the following :

$t$ = *Card(R) where R is the input relation of the TC algorithm,*
$\sigma$ = *selectivity factor of the initial selection applied to R,*
$t*do$ = *estimation of the cardinal of the auto-join index of relation R,*
$a$ = *access time for one word in main memory,*
$c$ = *time needed for comparing two OID's.*

## V.1 Cost of TC on join index

The TC algorithm on join index is derived from the basic TC algorithm detailled section III.3. Loops of joins are directly applied on join index instead of being applied on relations. We assume the join index to be sorted according to the column upon which the selection applies. It is the most favorable case as each successor of a vertex can then be found in $\log_2 n$ index access (n being the join index cardinal) instead of n/2 access if the index where not sorted. To always guarantee this favorable case, it is proposed in [Vald87] to maintain two copy of the join index on disk , each one sorted on a different column. At each iteration of the internal loop of the TC algorithm, the cardinal of $\Delta R$ is multiplied by the expansion factor $E$ during the join $\Delta R := \otimes_M(\Delta R, R)$. Before processing recursion, $\Delta R$ is initialized using $\sigma * t$ elements resulting from the initial selection applied on R. At iteration i-1, $\Delta R$ is thus composed of $\sigma * t * E^{i-1}$ elements where $E^{i-1}$ denotes E*E*...*E (i-1) times.

Then, the $i^{th}$ iteration of TC algorithm on join index may be splitted in four steps :

(i) Performing the join $\Delta R:=\otimes_M(\Delta R,R)$ requires $\sigma t*E^{i-1}$ searches on the join index. As a search requires $\log_2 t*do$ reads and comparisons (the join index is sorted),the resulting cost is:

$$Join\_Cost = [\sigma*t*E^{i-1} * log2(t*do)]\ (a+c).$$

(ii) The result of step (i) contains $\sigma*t*E^{i-1}*E$ elements. Writing this intermediate result requires:

$$Write\_Cost = \sigma*t*E^{i}*a.$$

(iii) This result must be sorted before processing the difference and union operations. It yields:

$$Sort\_Cost = [\sigma*t*E^{i}*log_2(\ \sigma*t*E^{i})]\ (2a+c).$$

(iv) Union between the intermediate result $\Delta R$ and the recursive relation TCR being built (TCR is assumed to be sorted as the union of two sorted relations gives a sorted relation). At each iteration, TCR is augmented of $\sigma*t*E^{i}$ elements. Thus, TCR contains $\Sigma_{k=1 \text{ to } i-1}\ (\sigma*t*E^{k})$ elements at iteration (i-1). Difference and union are performed with a simple scan of $\Delta R$ and TCR, which yields :

$$Union\_Diff\_Cost = 2*[\ \sigma*t*E^{i} + \Sigma_{k=1\ to\ i-1}\ \sigma*t*E^{k}\ ]\ (2a+c).$$

Finally, the global cost of the TC algorithm using Join Index (JI) for all iterations is :

$$JI\_Cost = \Sigma_{i\ =\ 1\ to\ h}\ (\ Join\_Cost + Write\_Cost + Sort\_Cost + Union\_Diff\_Cost\ ).$$

## V.2 Cost of TC on Closure_Graph

The evaluation of the graph traversal consists in applying the BFS function to each element of the input set $\sigma*t$ of vertices. The *neighbor_tup* primitive will be called h times during a BFS execution. At each application of the form $N_t$=*neighbor_tup*($N_t$), the size of $N_t$ is multiplied by the expansion factor E. Thence, at the $i^{th}$ iteration, the *neighbor_tup* primitive is applied on a set containing $E^{i-1}$ vertices (materialized by OID's) and returns a set containing $E^{i}$ vertices. For each of the $E^{i-1}$ vertices, the *neighbor_tup* primitive performs three accesses to get the tuple (vertex) from its OID (due to the implementation of OID's), two more accesses to verify that the vertex is not yet marked and to mark it, one access to read the concerned attribute identifier (OID corresponding to an arc t—>v ) of that tuple, then three accesses to get the attribute value in the domain, and finally two accesses to get the corresponding inverted sublist. Thus, the retrieval of all successors of a given vertex using the *neighbor_tup* primitive requires 11 memory accesses. In summary, an execution of the BSF function for the h iterations requires:

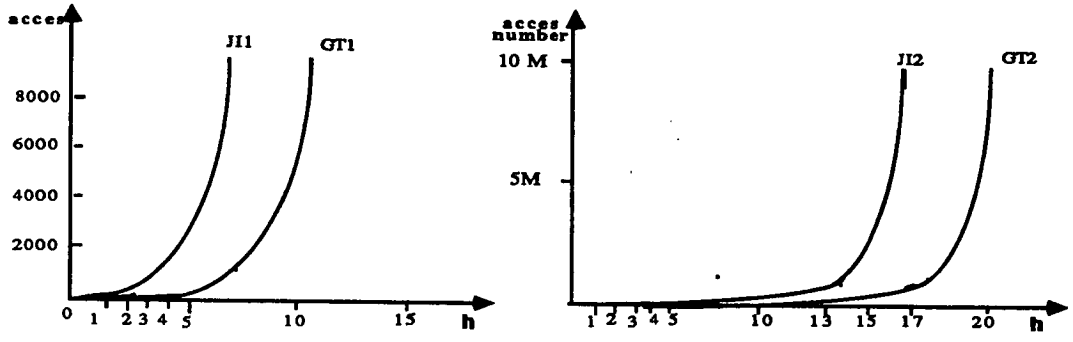
$$BSF\_COST = \Sigma_{i=1\ to\ h}(11*E^{i-1})$$

As the BSF function is applied to each element of the input set $\sigma*t$, the total cost of the graph traversal approach appears to be:

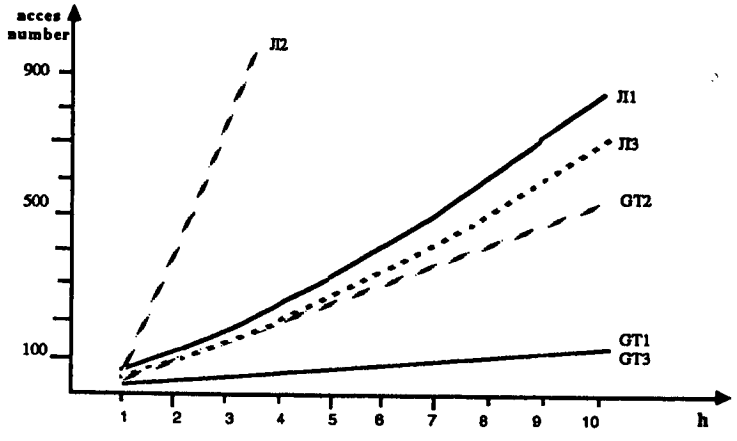
$$GT\_COST = 11*\ \sigma*t*\Sigma_{i=1\ to\ h}\ E^{i-1}$$

## V.3 Comparison of costs

Figure 8 displays graphical comparisons for typical values of parameters E, σ, t, h. The obtained results are given in memory access numbers (i.e., c is set equals to 2a). For the sake of clarity, the curves are represented as continuous although h can only be given natural number values.




Values of the dependant variables : E=2, σt=1, t=10.000
JI and GT curves are plotted on two different scales to compare both curves for all values of h.




Values of the dependant variables : E=1.
Bold curves (JI1 and GT1) : correspond to σt=1, t=10.000.
Hached curves (JI2 and GT2) : correspond to σt=4, t=10.000.
Dotted curve (JI3) : correspond to σt=1, t=1000. The GT curve is invariant in function of t.

**Figure 8**: Performance variations of JI and GT algorithms

Analysis of these curves yields several remarks. First, algorithms JI and GT behave exponentially in function of h when E is greater than 1. This natural result derives from the fact that the number of accessed vertices at each iteration increases exponentially. Note however that GT algorithm is more "resistant" than JI algorithm as its exponential factor is smaller. Secondly, the GT algorithm is linear when E = 1 while the JI algorithm is linear only if E=1 and σt=1. When σt is greater than 1, the logarithmic cost of sorting intermediate results must be added to the JI algorithm evaluation. Finally, JI algorithm does not have constant performances in function of t. Indeed, the size of the join index,

given by t*do, determines the cost of each index search.

In summary, it appears that the GT algorithm outperforms the JI algorithm. This is mainly due to the fact that the DBGraph structure pre-compiles the whole TC operation. The strongest advantages of the GT algorithm appearing through the evaluation are the simplicity of detecting cyclic data and the direct access without scans and comparisons to the successors or predecessors of a vertex. Then, TC operator ceases to be the dramatically time consuming operator of KBMS's.

## VI SUMMARY AND FUTURE WORK

Considering that in future DBMS's it will be possible to hold the active database in main memory, a new physical database organization has been proposed. Tuples and indices are integrated in a unique data structure called DBGraph. This data structure aims two objectives: be compact and decomposable such as the active database always fits in main memory and speed up extended relational algebra in term of CPU time.

In a DBGraph, tuples and values are stored separately. Links between tuples and values are maintained using OID's in order to constitute a bipartite graph. This graph precompiles selection, join and transitive closure operations. Two strategies where described to traverse this graph in order to answer a relational query. Set oriented execution of relational algebra is generated using a breadth first search strategy while pipeline execution is produced using a depth first search strategy. As the two strategies lead to the same temporal complexity, the second one is recommended for it minimizes the storage cost of temporary results even if it is more complex to implement.

Storage cost evaluations have demonstrated the compactness of a DBGraph. Compared to a flat file organization without any index, a DBGraph is more costly for short values with high domain selectivity but more compact in other cases. When adding indices in flat file organization, DBGraph appears to be the best candidate even for short values. Finally, it is easy to partition the DBGraph in small segments in order to take advantage of vertical partitionning. This enforces the probability that active database fits in main memory.

Performance evaluations show that transitive closure on DBGraph outperforms transitive closure on join indices and more generally that DBGraph is well suited to support recursive processing. For the sake of conciseness, other operators where not evaluated in detail since the transitive closure operator is expressed as a composition of other relational operators. Performance gains using DBGraph should be similar to those obtained using indices for selection and join indices for join on basic relations. Furthermore DBGraph speeds up operations on temporary results in a similar way as on basic relations. Finally DBGraph are much easier to update than join indices.

As all operations are performed on the DBGraph, there could be heavy contention on that structure. A solution has been introduced to avoid this drawback. It could be a response to the critical problem of locking indices. Future works should be done in this area.

**Aknowledgements**

The authors wish to thank Karima Bennis and Sylvie Renard for fruitful discussions and for their active participation in prototyping the DBGraph data structure.

## REFERENCES

[Amma85] Ammann A. C., Hanrahan M. B., and Krishnamurthy R., "Design of a Memory Resident DBMS", Proc. of IEEE COMPCON 1985.

[Banc86] Bancilhon F. : "An Amateur's Introduction to Recursive Query Processing Strategies", ACM SIGMOD Proc., May 1986, Austin, Texas.

[Baye77] Bayer R., Schkolnick M., "Concurrency of Operations on B-Trees", Acta Informatica, 9, 1977.

[Bern87] Bernstein P.A., Hadlacos, Goodman N., "Concurrency Control and Recovery in Database Systems", Addison-Wesley Ed., 1987.

[Bitt83] Bitton D., DeWitt D. J., Turbyfill C., "Benchmarking Database Systems: a Systematic Approach", Proc. of the 9th int. Conf. on VLDB, Florence, Nov. 1983.

[Bitt86] Bitton D., Turbyfill C., "Performance Evaluation of Main Memory Database Systems", Cornell University, TR 86-731.

[Dewi84] DeWitt D., Katz R., Olken F., Shapiro L., Stonebraker M., Wodd D., "Implementation Techniques for Main Memory Database Systems", Proc. of SIGMOD, Boston, June 1984.

[Eich87] Eich M. H., "MARS: The Design of a Main Memory Database Machine", 5th IWDM Proc., Karuizawa, Oct. 1987.

[Eswa76] Eswaran K. P., Gray J. N., Lorie R., Traiger L. L., "The Notion of Consistency and Predicate Locks in a Database System", Proc. of ACM, V19, No 11, Nov 1976.

[Garc84] Garcia Molina H., Lipton R.J., Valdes J. "A Massive Memory Machine", Proc. IEEE COMPCON 1984.

[Gard86] Gardarin G., de Maindreville C. : "Evaluation of Database Recursive Programs as Recurrent Function Series", ACM SIGMOD Proc., Austin, May 1986.

[Gibbo85] Gibbons A., "Algorithmic graph theory", book, Cambridge University Press, 1985.

[Haerd78] Haerder T., "Implementing a Generalised Access Path Structure for a Relational Database", Proc. of ACM TODS, Vol. 3, No 3, Sept 1978.

[Hamm79] Hammer M., Niamir B.,"A Heuristic approach to attribute partitioning", Proc. of ACM SIGMOD, 1979.

[Han85] Han J., Lu H., "Some Performance Results on Recursive Query Processing in Relational Database Systems" , Proc. Data Engineering Conf., Los Angeles, February 1986.

[Hens84] Henschen L.J., Naqvi S.A., "On compiling queries in recursive first-order databases", JACM, Vol. 31, N° 1, Jan. 1984.

[Ioan86] Ioannidis Y. E., "On the Computation of the Transitive Closure of Relational Operators", Proc. of 12th int. Conf. on VLDB, Kyoto, August 1986.

[Lehm86a] Lehman T. J., Carey M.J., "A Study of Index Structure for Main Memory Database Management Systems", Proc. of 12th int. Conf. on VLDB, Kyoto, August 1986.

[Lehm86b] Lehman T. J., Carey M.J., "Query Processing in Main Memory Database Management Systems", ACM SIGMOD Proc., Austin, May 1986.

[Lehm87] Lehman T. J., Carey M.J., "A Recovery Algorithm for a High-Performance Memory-Resident Database System", Proc. of ACM SIGMOD, San Francisco, May 1987.

[Miss82] Missikov M., "A Domain Based Internal Schema for Relational Database Machines", ACM SIGMOD Proc., New-York, June 1982.

[Naka87] Nakano R., Kiyama M., "MACH: Much Faster Associative Machine", 5th IWDM Proc., Karuizawa, Oct. 1987.

[Rose86] Rosenthal A. , Heiler S., Dayal U., Manola F., "Traversal Recursion : A Practical Approach to Supporting Recursive Applications", ACM SIGMOD Proc., Austin, May 1986.

[Sedg84] Sedgewick R. : " Algorithms" , Book, Addison-Wesley Pub., 1984.

[Seli79] Selinger P. G., Astrahan M. M., Chamberlin D. D., Lorie P. A., Price T. G., "Access Path Selection in a Relational Database Management System", Proc. of ACM SIGMOD, Boston, 1979.

[Vald86] Valduriez P., Boral H., "Evaluation of Recursive Queries Using Join Indices",Proc. 1st International Conference on Expert Database Systems, Charleston, 1986.

[Vald87] Valduriez P., "Join Indices", Proc. of ACM TODS, Vol. 12, No 2, June 87.

[Weik84] Weikum G., Schek H., "Architectural Issues of Transaction Management in Multi-Layered Systems", Proc. of the 10th int. Conf. on VLDB, Singapore, Aug. 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique