Efficient Main Memory Data Management Using the DBGraph Storage Model [1]

Philippe Pucheral [2], Jean-Marc Thévenin, Patrick Valduriez

INRIA - Rocquencourt BP. 105, 78153 Le Chesnay, France

Abstract

The requirements for a main memory data storage model are both compactness and efficient processing for all database operations. The DBGraph storage model, proposed in this paper, achieves these goals. By representing the entire database in a unique graph-based data structure, called DBGraph, it fully exploits the direct-access capability of main memory systems. For example, Selection, Join and Transitive closure operations over base or temporary relations are performed by a DBGraph traversal without tuple comparison and move. Furthermore, it is decomposable so that only the useful subset of the database can be loaded from disk without format conversion. Complex database queries can be processed by either set-oriented or pipelined mode depending on the way the graph is traversed. Analysis shows good storage occupancy and excellent performance for both update and retrieval operations.

1. INTRODUCTION

The rapidly decreasing cost of RAM makes main memory database systems (MMDBS) a cost-effective solution to high-performance data management [Eich89]. Disk-based database systems have their performance limited by the I/O bottleneck [Cope86]. In a MMDBS, the useful subset of the database, called the *active database*, may be entirely contained in main memory, thereby eliminating the I/O bottleneck. Furthermore, by allowing transactions to commit in safe (battery-backed-up) RAM [Cope89], the complexity and overhead of recovery management are significantly reduced. However, designing a MMDBS requires addressing two main issues: efficient space utilization and efficient processing of all database operations. For practical reasons, these issues have been typically addressed separately.

Efficient space utilization is necessary to hold the active database entirely in main memory. Towards this goal, compact data structures such as T-trees [Lehm86a] or array structures [Amma85] have been proposed to organize permanent data efficiently in main memory. To some extent, some work has also considered the minimization of space occupancy for temporary data [Bitt86, Lehm86b].

Efficient data processing requires exploiting the direct (pointer-based) data access capability of main memory [DeWi84]. Indices are typically logical pointer-based data structures optimizing certain database operations. For example, inverted indices [Card75], join indices [Vald87] and transitive relationship indices [Agra89] optimize respectively select, join and transitive closure operations. These indices "precompile" the operation into a dynamic data structure. Because they have been designed for disk-based systems, they are separate from the base data. In a main memory context, they may well introduce a significant storage and update overhead. Futhermore, temporary data are managed differently from base data, thereby making difficult indexing of temporary data.

In this paper, we address these issues together and propose an integrated main memory storage model. Unlike integrated disk-based storage models [Cope85, Vald86a] or models for dedicated hardware [Miss82, Miss83] which optimize some operation at the expense of some others, it provides efficient support for all database operations with good space occupancy. This model is based on a graph structure, called *DBGraph*, to represent the entire database. The DBGraph storage model (DBG) is intended to support various higher-level models and associated languages. However, for simplicity, we will illustrate its use with relational algebra extended with the transitive closure operator. The latter operator is of utmost importance for supporting queries on recursively defined relations [Vald86b]. An earlier vertion of the DBGraph storage model, proposed in [Puch89a], focused on transitive closure operation. In this paper, the model

1 This Work has been partially supported by the STRETCH Esprit project.

2 Philippe Pucheral is with SAGEM 93700 Pontoise, France.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

has been refined and we concentrate on all database operations.

A DBGraph is a bipartite graph composed of a set of tuple-vertices, a set of value-vertices and a set of edges connecting these two sets. Compactness is obtained by storing each attribute value only once and using the same edges to precompile all database operations. A DBGraph can be naturally partitioned into subgraphs that can be clustered on disk. Therefore, loading a subpart of the DBGraph into main memory can be done efficiently without format conversion. Temporary tuples can be mapped onto the DBGraph using temporary edges so that either permanent or temporary data can be treated the same way. Arguments for managing all data uniformly are provided in [Cope90]. Non-recursive retrieval, such as select and join, can be efficiently processed by a single traversal of the graph. Update operations can also be processed with minimal overhead. Furthermore, complex queries can be efficiently processed in either set-oriented or pipelined mode depending on the graph traversal (breadth-first versus depth-first), the latter eliminating the need to store intermediate results.

The paper is organized as follows. Section 2 provides a formal definition of DBG in terms of graph structure and primitive operations. In Section 3, we define retrieval and update operator algorithms based on these primitive operations. In Section 4, we discuss set-oriented versus pipelined query processing on a DBGraph. Section 5 argues for a specific implementation of the DBGraph. Based on such implementation, the storage occupancy and performance of Join and update operations are analyzed. Section 6 gives our conclusions.

2. DBGRAPH STORAGE MODEL

In this section, we give a formal description of the DBGraph storage model (DBG) including its primitive operations. This provides a sound basis to express any kind of retrieval and update operation independent of any conceptual data model or lower-level implementation choice.

2.1. DBGraph definition

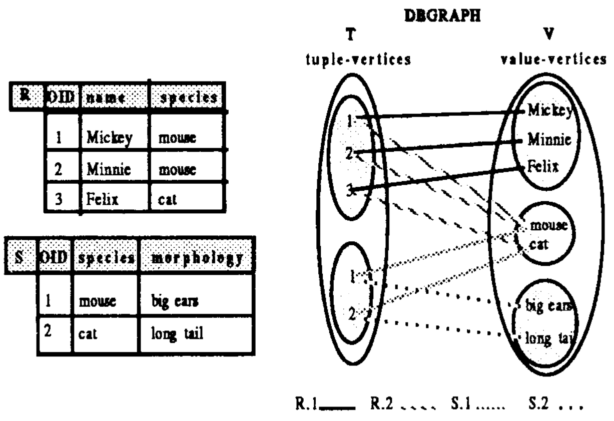
We first introduce a few notations. We consider a database DB composed of a set of relations. In most of our examples we will use only two relations named R and S. These relations are defined over a number of domains, each domain j being denoted by Dj. A relation schema is an aggregation of attributes, each of a given domain of values. We denote by R.k the kth attribute of relation R and $t_{R.k}$ the value of attribute k for tuple t. Finally, we denote by T the set of all the tuples of a database DB and V the set of all the domain values of DB.

We can define an isomorphism between a database DB and a graph, called DBGraph, as follows (see Figure 1 for an example). A DBGraph is a bipartite graph containing a set of *tuple-vertices* holding all the tuples of T, a set of *value-vertices* holding all the domain values of V, and *valued-edges* connecting these two sets. Each edge (t, v, R.k) of the DBGraph is an indirected valued edge connecting a tuple-vertex t with a value-vertex v. The valuation R.k indicates that t belongs to relation R and that v is the value of its kth attribute. Thus, a tuple-vertex is linked by one edge to each of its attribute values. Conversely a value-vertex is linked by an edge to each tuple that references it for one of its attributes. The DBGraph concept can be formally defined as follows:

Definition : DBGraph

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

A DBGraph is bipartite since T and V constitute a partition of X and there is no edge connecting two vertices of T or two vertices of V. Thus, a DBGraph traversal involves an alternance of tuples and values. Each couple of tuples having the same value for one of their attributes are connected by a path of length two. Finally, tuple vertices (resp.value vertices) may be grouped on a relation basis (resp.domain basis) since the relations form a partition of T (resp. V).



R	OID	name	species
	1	Mickey	mouse
	2	Minnie	mouse
	3	Felix	cat

S	OID	species	morphology
	1	mouse	big ears
	2	cat	long tail

Figure 1: analogy between a database and its DBGraph

2.2. Primitive Operations

We now define primitive operations to traverse and update a DBGraph. Complex database operations can be expressed in a simple and uniform fashion by composition of these operations, independent of the physical DBGraph implementation. This provides a high-level description of all algorithms which should result in higher modularity.

- the *succ_val* operation delivers the subset ΔR of tuple vertices corresponding to all the tuples of relation R whose kth attribute value is equal to a given value v. It is an application from V to T that determines the subset of T vertices connected to the vertex v by edges valued by R.k.

$\Delta R = succ\_val(v, R.k)$
where $\Delta R = \{t \in T / v \in V$ and $(t, v, R.k) \in A\}$

- the *succ_tup* operation is an application from T to V that determines the V vertex connected to a given T vertex by an edge valued by R.k.

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

These two operations may be combined since their result and input arguments are compatible. Thus, any DBGraph traversal can be expressed by a combination of them. Similarly, any database update can be expressed by a combination of the following operations.

- the *insert_tup* operation performs all the DBGraph updates caused by the addition of a new tuple $t(v_1, \ldots, v_n)$ in a given relation R. Tuple t is inserted in T and each of its attribute values is inserted in V if and only if this value does not already exist. Thus, the uniqueness of values in V is ensured. Finally, the edges connecting this tuple and its attribute values are added to A.

$G'(X',A') = insert\_tup\ (\ t(v_1, \ldots, v_n), R, G(X,A)\ )$
where $X' = (T', V')$ with $T' = T \cup t$,
$V' = V \cup \{v_1, \ldots, v_n\}$,
and $A' = A \cup \{(t,v_1,R.1), \ldots, (t,v_n,R.n)\}$

- the *delete_tup* operation performs all the DBGraph updates caused by the deletion of a tuple $t(v_1, \ldots, v_n)$. Tuple t is deleted from T. Each of its attribute values is deleted from V if and only if no other tuple references the same value. The edges connecting this tuple to its attribute values are removed.

$G'(X',A') = delete\_tup\ (\ t(v_1, \ldots, v_n), R, G(X,A)\ )$
where $X' = (T', V')$ with $T' = T - t$,
$V' = V - \{v / \forall (t',v,S.k) \in A, t'=t\}$,
and $A' = A - \{(t,v_1,R.1), \ldots, (t,v_n,R.n)\}$

- the *modify_tup* operation performs all the DBGraph updates caused by the modification of one attribute of a tuple t. The old attribute value is deleted from V if it is no longer referenced and the new one is inserted in V if it does not already exist. Finally, tuple t is disconnected from its old attribute value and connected to the new one.

$G'(X',A') = modify\_tup\ (\ t(\ldots, v_i = v'_i, \ldots), R, G(X,A)\ )$
where $X' = (T', V')$ with $T' = T$,
$V' = (V \cup v'_i)$
$- \{v_i \text{ iff } \forall (t',v_i,S.k) \in A, t'=t \text{ and } S.k=R.i\}$,
and $A' = (A - (t,v_i,R.i)) \cup (t,v'_i,R.i)$

3. DATABASE OPERATIONS

DBG is intended to support set-oriented database languages based on different data models. In this section, we use relational algebra extended with transitive closure as a paradigm to validate DBG. We show how relational operators can be composed easily using the DBGraph primitive operations.

3.1. Select

The select operator, denoted by σ_Q, applied to relation R, determines the subset R_σ of T vertices corresponding to all the tuples of relation R which satisfy the qualification Q. For simplicity, we assume that Q is a simple comparison predicate $(R.k\,\theta\,c)$ where c is a constant and θ is a comparator. Thus, R_σ contains all the tuple vertices connected to a value satisfying Q, by an edge valued by R.k. The select operator is then expressed as:

$R_\sigma = \sigma_Q(R)$ where $R_\sigma = \{t \in T / (v\ \theta\ c)$ is true and $\exists\ (t, v, R.k) \in A\}$

The execution of the select operation is similar to that using inverted indices. The set of values satisfying the selection criteria is first determined. Then the matching tuples are obtained by applying the succ_val primitive to this set of values.

```
Function $\sigma_Q$ (R) : $R_\sigma$
  begin
    $R_\sigma := \emptyset$;
    $\Delta V := Select_Q(V)$;
    for each $v \in \Delta V$ do
      $\Delta R := succ\_val(v, R.k)$;
      $R_\sigma := R_\sigma \cup \Delta R$
    end for
  end
```

$Select_Q(V)$ builts the set $\Delta V = \{v \in V / (v\ \theta\ c) \text{ is true}\}$. This function can be optimized using indices on V (see details in Section 5). The generalization of Q to handle conjunctions or disjunctions of predicates requires unions or intersections of the R_σ sets corresponding to each predicate.

3.2. Join

The join operator, denoted by $\otimes_M$, applied to R and S determines the set $RS_\otimes$ of couples of T vertices corresponding to the matching tuples. We consider a join predicate M of the form (R.k = S.l) where R.k and S.l take values on the same domain Dj. The DBGraph definition insists that the matching tuples are connected by a path of length two, as shown below.

$RS_\otimes = \otimes_M(R, S)$
where $RS_\otimes = \{(t_1, t_2) / t_1 \in T, t_2 \in T,$
$\exists\ (t_1, v, R.k) \in A,\ \exists\ (t_2, v, S.l) \in A\}$

An obvious way to perform the join is to first scan Dj. Then, for each value of this domain, two successive applications of the succ_val primitive deliver the subsets ΔR and ΔS where ΔR (resp. ΔS) contains the tuples of R (resp. S) connected to this value by an edge valued by the join attribute. Finally, the Cartesian product of ΔR and ΔS gives the join result since each tuple of ΔR match with each tuple of ΔS.

```
Function $Join1_M(R,S) : RS_\otimes$
  begin
    $RS_\otimes := \emptyset$;
    for each $v \in Dj$ do
      $R := succ\_val(v, R.k)$;
      $\Delta S := succ\_val(v, S.l)$;
      $RS_\otimes := RS_\otimes \cup (\Delta R \times \Delta S)$
    end for
  end
```

As domains are shared among relations, Card(Dj) can be high compared to Card(R) or Card(S) (Card denotes the cardinality of a set). Another join method, based on a different traversal of the DBGraph, outperforms the previous one in this case. It consists of first scanning the smallest operand relation. Then, the join attribute value of each tuple of this relation is retrieved through the succ_tup primitive. Finally, the application of the succ_val primitive gives access to the subset of tuples of the other operand relation having the same value of join attribute.

```
Function $Join2_M(R,S): RS_\otimes$
  /* We assume $Card(R) < Card(S) < Card(Dj)$ */
  begin
    $RS_\otimes := \emptyset$;
    for each $t \in R$ do
      $v := succ\_tup(t, R.k)$;
      $\Delta S := succ\_val(v, S.l)$;
      $RS_\otimes := RS_\otimes \cup (t \times \Delta S)$
    end for
  end
```

In both algorithms, the union $RS_\otimes \cup (\Delta R \times \Delta S)$ (resp $RS_\otimes \cup (t \times \Delta S)$) can be profitably replaced by a concatenation because the generation of duplicates in $RS_\otimes$ is impossible. Union, intersection and difference operations can all be supported by a customized version of the join operator, as suggested in [Brat84, Thev89].

3.3. Other retrieval and update operators

The project operator $\pi_P(R)$ consists of applying the primitive operation v:=succ_tup(t, R.k) for each tuple $t \in R$, to successively retrieve the value of each attribute k specified in the project list P. Because of its simplicity, the project algorithm need not be discussed.

Updating a relation involves the insert_tup, delete_tup and modify_tup primitives. Each of these primitives performs the DBGraph updates induced respectively by an insertion, a deletion or a modification of a tuple. Thus, a set-oriented update operation is simply a loop with call to these primitive operations. Integrity constraint checking for update operations will be discussed in Section 5.

3.4. TRANSITIVE CLOSURE

Transitive closure is a basic operator for efficiently computing queries against recursively defined relations [Banc86, Vald86b]. Building the transitive closure of relation R on attributes R.k and R.l consists in computing the least fixpoint of the following equation:

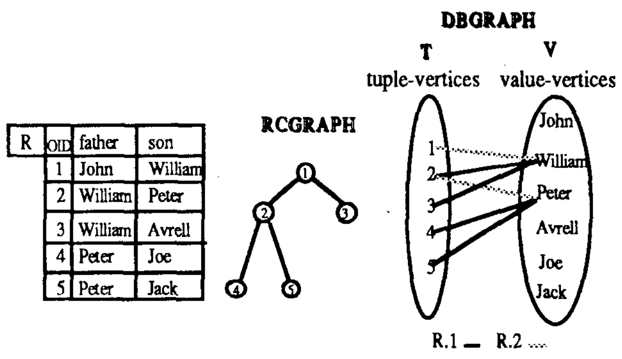
$$R^+ = R \cup \pi_P(\otimes_M(R^+, R)) \text{ with } p = (R^+.k, R.l) \text{ and } M = (R^+.l = R.k)$$

An efficient algorithm, called Semi-Naïve [Banc85], consists of a loop of relational operators. The algorithm incorporating selection can be expressed as follows :

```
Function $TC_Q(R): R^+$
  /* compute the TC of R according to the initial
  query selection Q */
  begin
    /* $\sigma_Q$ is applied to the basic relation */
    $\Delta R := \sigma_Q(R)$;
    $R^+ := \Delta R$;      /* before processing recursion */
    while $\Delta R \neq \emptyset$ do
      $\Delta R := \otimes_M(\Delta R, R)$;
      $\Delta R := \Delta R - R^+$;
      $R^+ := R^+ \cup \Delta R$;
    end while
  end
```

The difference $\Delta R := \Delta R - R^+$ guarantees the algorithm's termination in the case of cyclic data.

The DBGraph can be exploited to perform transitive closure more naturally and more efficiently. The idea is to perform a recursive traversal on the sub-part of DBGraph corresponding to relation R. To make recursion easier to express, we introduce the notion of Relation-Closure-Graph (RCGraph). The RCGraph of R on the attributes R.k and R.l is a graph in which all vertices represent tuples of R (see Figure 2). Two vertices t_1 and t_2 are connected by an edge if the corresponding tuples match according to the join predicate (R.k=R.l). In fact, each edge connecting two tuples in the RCGraph corresponds to a path of length two in the DBGraph. Two vertices t_i and t_j are transitively connected if there exists a path $(t_i \longleftrightarrow t_k, t_k \longleftrightarrow t_l, \ldots, t_n \longleftrightarrow t_j)$ in the RCGraph. Thus, there is a direct mapping between the transitive closure of R and the transitive closure of its RCGraph.



For simplicity, only the edges involved in the RCGraph are shown in the DBGraph.

Figure 2: Mapping between RCGraph and DBGraph

A select on R determines a set ΔR of entry vertices in the RCGraph. Then, the transitive closure of the RCGraph consists of, for each t of ΔR, finding all the vertices (called the descendants of t) reachable from t by a path. Similar to many graph traversal algorithms, a visited vertex is marked to avoid passing twice through the same path. Such marking eliminates the difference operation of the previous algorithm. The result of each RCGraph traversal is a spanning tree of root t over the descendants of t. Actually, the RCGraph is virtual and all RCGraph traversals are translated into DBGraph traversals.

For performance reasons, we slightly modify the succ_val operation to incorporate marking at traversal time. The resulting operation, called *succ_val*(v,R.k)*, delivers and marks all non marked tuple vertices connected to the value v by one edge valued by R.k. The use of the succ_tup and succ_val* primitives defines a traversal of the DBGraph following a *breadth-first search (BFS)* strategy [Sedg84], as follows :

```
Function BFS(t): D
  /* D is the set of descendants of t */
  begin
    $\Delta D := t;$
    while $\Delta D \neq \emptyset$ do
      $\Delta D' := \emptyset;$
      for each $t' \in \Delta D$ do
        $v' := succ\_tup(t', R.k);$
        $\Delta D' := \Delta D' \cup succ\_val^*(v', R.l);$
      end for
      $\Delta D := \Delta D';$
      $D := D \cup \Delta D;$
    end while
  end
```

The transitive closure operation is performed by simply traversing links. Furthermore testing the termination condition is greatly simplified by marking. The unions $D:=D \cup \Delta D$ and $\Delta D':=\Delta D' \cup succ\_val^*(v', R.l)$ can be efficiently implemented by a concatenation since marking the tuples avoid duplicate generation. Using weighted graphs [Gard88], this algorithm can be extended to support shortest path problems, path enumeration and more generally all computations that can be expressed as traversal recursion [Rose86].

4. QUERY PROCESSING

For clarity of exposition, the database operations introduced in Section 3 have been described in a set-oriented way, independent of their integration in a query execution plan. A combination of these operators induces a *breadth-first search* traversal of the DBGraph. Each operation produces a temporary result which must be materialized and consumed by the next operation. However, a pipelined execution of a query can be obtained by a *depth-first search* traversal of the DBGraph. A depth-first search strategy has two major advantages. First temporary results need not be generated, thereby saving space for permanent data. Second, tuples already produced can be displayed while the query processing is still in progress. The pipelined evaluation mode is restricted to the subpart of a query involving selection, join and projection operators since difference, intersection, aggregate and sort operators are purely set-oriented.

4.1. Set-oriented processing

Set-oriented processing requires the management of temporary results. In most data storage models, it is often impossible to speed up operations on temporary results using indices. As join operations are frequently preceded by selections, this limitation is severe. However it is possible to maintain the join indices validity after selection [Vald87]. Typically, a selection delivers a list of OID's referencing the relevant tuples that is then semi-joined with the join index in order to produce a new valid join index on the temporary relations.

In the DBGraph model, temporary results are materialized by temporary vertices connected through temporary links with the tuples of the basic relations from which they are extracted, as shown in Figure 3. The degree (number of edges) of a temporary vertex is equal to the number of basic tuples involved in the temporary result. The main advantage is that temporary results preserve the links to basic tuples. Consequently, operations on temporary relations are speed up by a DBGraph traversal in a way similar to operations on basic relations. For example, consider the query $\otimes_M(\sigma_Q(R),S)$ with Q=(R.2="mouse") and M=(R.2=S.1), illustrated in Figure 3. Scanning the temporary result $\sigma_Q(R)$ determines a set of entry vertices in the DBGraph that can be directly exploited by the join operator, using the Join2 algorithm introduced in Section 3.2.

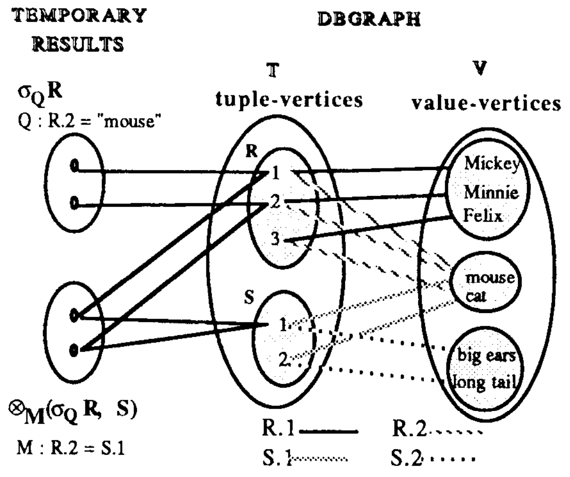


Figure 3: DBGraph extended to temporary results

This temporary result representation has two other advantages. First, it is quite compact since it does not contain any value. Thus, it enforces the hypothesis that the active database fit in main memory. Second, the project operations are posponed until the end of the query evaluation. This simplifies query optimization [Amma85].

4.2. Pipelined processing

Let us consider a generic query QR of the form ($\sigma_Q R_1 \wedge R_1 \otimes R_2 \wedge R_2 \otimes R_3 \wedge \ldots \wedge R_{n-1} \otimes R_n$). The number of joins in QR determines the length of a *pertinent-path* connecting a tuple of R1 to a tuple of Rn where the edges valuations are determined by the join predicates. A pertinent-path covers all the vertices from which a result tuple is produced. The selection on R1 in QR determines a set of entry value-vertices in the DBGraph. Each of these values is the root of one or more paths in the DBGraph. These paths are explored one after the other and a result tuple is produced each time the extremity of a pertinent-path is reached. Backtracking is applied at the extremity of each pertinent path or when a path fails. This execution mode can be directly translated in the program shown in Figure 4.

```
$\Delta V = Select_Q(V)$
  $\Delta R_1 = succ\_val(v, R_1.i)$
  for each $t_1 \in \Delta R_1$ do
    /* join $R_1 \otimes R_2$
    $v_1 = succ\_tup(t_1, R_1.j)$
    $\Delta R_2 = succ\_val(v_1, R_2.k)$
    for each $t_2 \in \Delta R_2$ do
      /* join $R_2 \otimes R_3$
      $v_2 = succ\_tup(t_2, R_2.l)$
      $\Delta R_3 = succ\_val(v_2, R_3.m)$
      •
      •
      for each $t_n \in \Delta R_n$ do
        /* generation of a result tuple
        $produce\_tup(t_1, t_2, \ldots, t_n)$
```

Figure 4: Execution plan for QR

In the general case, the relation-connection-graph of a query [Ullm80] may contain chains, loops, cycles and forks, where loops stand for selections and the other arcs stand for joins (see Figure 5).

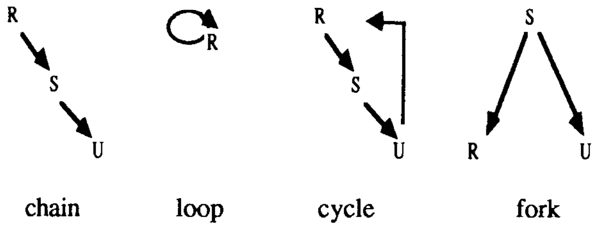


Figure 5 : relation connection graph structure

The query used above involves a chain with one loop on the first relation of this chain. Loops on other relations can be handled in two ways. One solution is to apply these selections first to mark the selected tuple vertices and then to explore the pertinent-paths without considering unmarked vertices. The second solution consists in checking the selection criteria for each tuple vertex reached during the pertinent-paths exploration. A cycle is handled as a chain whose last operation is an inter-attribute selection between attributes of two relations already joined. A fork is a special case of a chain where two joins are handled in the same loop. The corresponding sequences of instructions are given in Figure 6. They may constitute part of more complex execution plans [Thev89].

```
   CYCLE
for each $t_1 \in R$ do
  $v_1 = succ\_tup(t_1, R.i)$
  $\Delta S = succ\_val(v_1, S.j)$
  for each $t_2 \in \Delta S$ do
    $v_2 = succ\_tup(t_2, S.k)$
    $\Delta U = succ\_val(v_2, U.l)$
    for each $t_3 \in \Delta U$ do
      $v_3 = succ\_tup(t_3, U.m)$
      $v_4 = succ\_tup(t_1, R.n)$
      if $v_3 = v_4$ then
        $produce\_tup(t_1, t_2, t_3)$

   FORK
for each $t_1 \in S$ do
  $v_1 = succ\_tup(t_1, S.i)$
  $\Delta R = succ\_val(v_1, R.j)$
  $v_2 = succ\_tup(t_1, S.k)$
  $\Delta U = succ\_val(v_2, U.l)$
  for each $t_2 \in \Delta R$ do
    for each $t_3 \in \Delta U$ do
      $produce\_tup(t_1, t_2, t_3)$
```

Figure 6: cycle and fork execution plans

The temporal complexity of a depth-first search is O(max(Card(X),Card(A))) while that of a breadth-first search is O(Card(A)) [Gibb85]. Thus pipelined and set-oriented strategies have similar complexity on a DBGraph.

5. IMPLEMENTATION AND PERFORMANCE

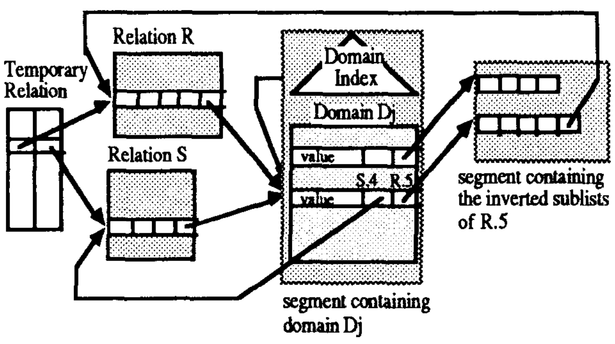
5.1. DBGraph Implementation

There are many ways to implement a graph. We detail below a particular DBGraph implementation which achieves both data compactness and efficient processing of database operations in an MMDBS context. This DBGraph implementation is shown in Figure 7.

Domain values are stored only once to preserve compactness. Since the domains form a partition of the set of values V, all the values varying over the same domain can be clustered in a separate segment. Taking advantage of vertical partitionning, the values of one domain can be loaded independent of the others. Similarly, since the relations form a partition of the set of tuples T, the tuples of one relation can be stored in one segment. Each object stored in a segment has a unique and invariant identifier (OID). Thus, tuples and values can be referenced by OID's.

In the formal definition 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 (t—>v) and (v—>t) by means of OID's. A tuple is implemented as an array of OID's referencing each of its attributes values. The valuation R.k of an arc (t—>v) is implicit since tuple t belongs to the segment containing relation R and the OID corresponding to this arc is stored in place of the kth attribute of tuple t. An arc (v—>t) is implemented by an OID 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 divided in as many sublists as there are attributes sharing this value. Thus, all arc valuations are determined by the relations and domains schema. They remain implicit at the instance level and do not compromise the DBGraph compactness.

The simplest way to store the inverted sublists is to have them one after the other behind their corresponding value. However this solution precludes fetching in memory the inverted sublists corresponding to one attribute R.k independent of the others. Instead, following the vertical partitionning strategy adopted for domain values, all the inverted sublists corresponding to one attribute R.k are grouped in one segment. With each domain value is associated a sublist array containing the OID's referencing all the sublists attached to this value. It contains one entry per attribute R.k varying on that domain and is indexed by R.k. Inverted lists corresponding to key attributes contain only one OID. In this case, much space is saved by storing the OID directly in the sublist array.


Relation R
Temporary Relation
Domain Index
Domain Dj
Relation S
value
S.4
R.5
value
segment containing the inverted sublists of R.5
segment containing domain Dj

The temporary relation results from a join between R and S. The attribute S.4 is assumed to be a key attribute.

Figure 7: A possible DBGraph implementation

Indices may be added on domain values to speed up selections on all attributes varying on the same domain. The index structure can be any one recommanded for MMDBS [Lehm86b]. We choose indices containing only OID's referencing the key values in order to reduce the storage cost for variable length keys [Amma85].

When processing a query in a set-oriented way, a temporary link is always traversed along the same direction. Thus, it can be implemented by an arc instead of an edge to reduce the cost of building the temporary result. Each temporary vertex t' resulting from a selection on a base relation R is connected to a single tuple t of R. It can be implemented by either keeping the OID of t (this allows to reach t from t') or marking t in a bit string of length Card(R) (this allows to reach t' from t). The only way to implement a temporary vertex resulting from a join is as a tuple of OID's representing arcs from the temporary vertex to tuples of T. Thus, the temporary links issued from selection can be implemented in both directions while those issued from join are implemented in a single way. Consequently, the join operations have to be ordered properly to avoid joins of join results. For instance, a sequence of joins (R⊗S)⊗(U⊗V) will be reordered in (((R⊗S)⊗U)⊗V) before execution [Puch89b].

5.2. Storage cost evaluation

In this section, DBG is compared with the classical Flat File organization (FF) in terms of storage cost. FF is chosen for it is well known and provides good storage performance. For both organizations, we consider the storage cost of *n* attributes varying on the same domain Dj and coming from one or several relations.

The DBG storage cost evaluation incorporates the values stored in the domain Dj, their attached sublist array and inverted sublists, the domain index and the OID's stored in place of the *n* attributes in the relations. For simplicity, we assume that the domain index is stored as an array of OID's sorted on the referenced domain values [Amm85].

The FF storage cost evaluation incorporates the values of the n attributes stored in the relations, k selection indices with $k \geq 0$ and j join indices with $j \geq 0$. The selection indices are supposed to be stored as arrays of couples (value, OID) sorted on the values, where the OID references the inverted sublist attached to the value. To simplify evaluation of join index size, we consider joins on key attributes. In this frequent case, each tuple of one relation matches with at most one tuple of the other relation.

5.2.1. Evaluation parameters

We introduce the following parameters :

l	average length of the Dj values
a	size of an OID (address)
$Card(R)$, $Card(S)$	cardinality of relation R (resp. S);
S_R, S_S	selectivity factor of one attribute of R (resp. S); $S_R = Card(\pi_i(R))/Card(R)$, where π_i denotes project on attribute i;
S_{avg}	average attribute selectivity factor for the attributes varying over domain Dj;
L	overlaping factor expressing the intersection between the values taken by the n attributes varying over domain Dj ($1/n \leq L \leq 1$); $L=1/n$ when each domain value is shared by the n attributes and $L=1$ when the n attributes do not share any value.

when needed, we use $\Sigma Card(Rel) = \sum_{i=1}^{n} Card(Rel_i)$ to simplify the formulas;

The cardinality of domain Dj can be approximated by $L\ \Sigma(S_{avg}\ Card(Rel))$. In the same way, the average cardinality of a selection index can be expressed by $\Sigma(S_{avg}\ Card(Rel))/n$ and the one of a join index by $\Sigma(Card(Rel))/n$, where $\Sigma Card(Rel)/n$ is the average cardinality of a relation.

5.3.2. DBGraph versus Flat File storage model

The storage cost incurred by DBG yields:

$Cost\ (DBG) =$

/* size of the domain values and their sublist array

$L\ \Sigma(S_{avg}\ Card(Rel))\ (l + n\ a)$

/* size of all inverted sublists

$+\ a\ \Sigma Card(Rel)$

/* size of the OID columns in the relations

$+\ a\ \Sigma Card(Rel)$

/* size of the domain index

$+\ L\ \Sigma(S_{avg}\ Card(Rel))\ a$

$= \Sigma Card(Rel)\ (\ 2a +\ L\ S_{avg}\ (l + a\ (n + 1)\ )\ )$

For FF using k selection indices and j join indices, the storage cost is :

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

/* size of the attribute values and their sublist array

$l\ \Sigma Card(Rel)$

/* size of k selection indices

$$+\ k\ (S_{avg}\ \frac{\Sigma Card(Rel)}{n}\ (l + a) + a\ \frac{\Sigma Card(Rel)}{n}\ )$$

/* size of j join indices

$$+\ j\ \frac{\Sigma Card(Rel)}{n}\ 2a$$

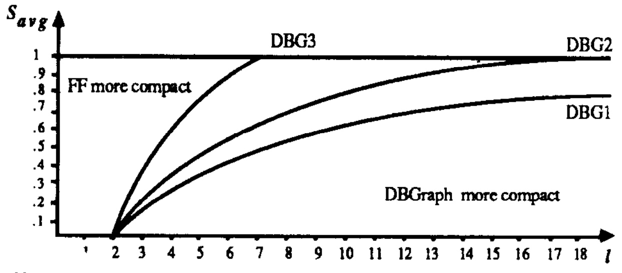
$$= \Sigma Card(Rel)\ (\ l + \frac{k}{n}\ (S_{avg}\ (l + a) + a) + \frac{j}{n}\ 2a)$$

The cost equality between the two storage models is expressed by the following equation:

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

$$S_{avg} = \frac{n\ l + a\ (k + 2j\ -2n)}{n\ L\ (\ l + a\ (n+1)\ )\ -\ k\ (l + a)}$$

Comparisons between FF without index and DBG are shown in Figure 8. The plotted curves indicate the most compact organization according to different values of the S_{avg} and l parameters, where l is expressed in units of a.

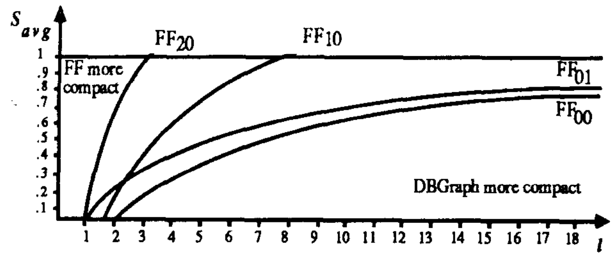


Values of the fixed parameters : n=2, L=1 (DBG1)
n=2, L=0.75 (DBG2)
n=2, L=0.5 (DBG3)

Figure 8: Compactness of DBG versus FF without index

Each curve divides the plan into a gain area for FF and a gain area for DBG. Curve DBG1 shows that, when $L=1$, DBG is in general more space consuming than FF without index. This is because there is no shared values between the attributes varying on the same domain. The lower L is, the more compact DBG is. Low S_{avg} and high l (e.g., enumerated type attribute) favor DBG while high S_{avg} and low l (e.g., short key attribute) favor FF. DBG's higher storage cost on several attributes must be compared with the large gains obtained on a few enumerated type domains. For example, DBG and FF yield almost the same cost to store the relations of the Wisconsin benchmark [Bitt83].

Figure 9 compares DBG versus FF using j join indices (with j=0 or 1) and k selection indices (with k=0, 1 or 2). The parameter L is set to 1, which is the worst case for DBG.



Value of the fixed parameters : n=2, L=1.
FF$_{20}$ corresponds to k=2, j=0 and FF$_{10}$ corresponds to k=1, j=0.
FF$_{01}$ corresponds to k=0, J=1 and FF$_{00}$ corresponds to k=0, j=0.
FF$_{00}$ is then equivalent to curve DBG1 of the figure 8.

Figure 9: DBGraph versus Flat file organization with indices

DBG becomes rapidly best as indices are added to FF. Since the number of indices determines the performance of query execution in a main memory context, the superiority of DBG is obvious. Note that DBG provides the same indexation level as one selection index per attribute and one join index per possible join in FF. The overhead involved by a selection index is strongly dependent of the value lenght l. 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 OID's instead of values.

5.3. Join evaluation

We compare algorithm Join2 (described in Section 3.2) with two algorithms : the well-known join algorithm using inverted indices (II) and the join index based algorithm (JI) [Vald87]. Both algorithms use an efficient data structure to speed up join processing. They are compared in the case of permanent and temporary operand relations. Algorithm Join1 is not considered since it does not handle temporary relations well. Detailed comparisons between Join1 and Join2 can be found in [Puch89b].

5.3.1. Evaluation parameters

We use the following parameters in addition to those introduced in Section 5.2.1. The semi-join selectivity factor J_{RS} of relation R, is defined by : $J_{RS} = Card(Semi\text{-}join(R,S))/Card(R)$. The average cardinality of the inverted lists attached to the join attribute values of relation S is defined by $1/S_S$. The cardinality of the R⊗S result is thus : $Card(R \otimes S) = J_{RS} Card(R)/S_S$. The join algorithm execution times are evaluated in terms of memory access unit (denoted by u). The other system parameters are :

- d: time for decoding an OID, assumed to be $3u$,
- o: time for comparing two OID, assumed to be $2u$,
- v: time for comparing two values, assumed to be $2ul$ where l is the average size of a value
- w: time for writing a word in memory, assumed to be u.

5.3.2. Join of two base relations

Join2 scans the smallest relation (assumed to be R) and for each tuple checks whether there is an inverted list associated with the join attribute of the second relation (S). If there is one, a Cartesian product between the current tuple of R and the inverted list is performed, yielding :

```
time(Join2) =
        /* scan the smallest relation
    Card(R)
            /* call succ_tup to get the join attribute value
        ((u + d)
            /* check the existence of an inverted sublist for
            the join attribute of S
        + o)
        /* number of R tuples matching with an S tuple
    + J_RS Card(R)
            /* access to the inverted sublist for the S join
            attribute
        (d
            /* Cartesian product of an inverted list and an OID
        + (u + (u + 2w)/S_S))
        = Card(R) (6 + J_RS (4 + 3/S_S)) u
```

The join algorithm using an inverted index scans the smallest relation (assumed to be R) and for each tuple searches the join attribute value in the index of the second relation (S). Then a Cartesian product is performed between the current tuple of R and the inverted list found in the index of S. This gives:

```
time(II) =
        /* scan the smallest relation
    Card(R)
            /* access the join attribute value
        (u
            /* searches the join attribute value in the index of S
        + v log2(S_S Card(S)))
        /* number of R tuples matching with an S tuple
    + J_RS Card(R)
            /* access to the inverted sublist for the join
            attribute of S
        (d
            /* Cartesian product of an inverted list and an OID
        + (u + (u + 2w)/S_S)
= Card(R) (1 + 2l log2(S_S Card(S)) + J_RS (4 + 3/S_S)) u
```

The join index algorithm needs only to copy the join index, which gives :

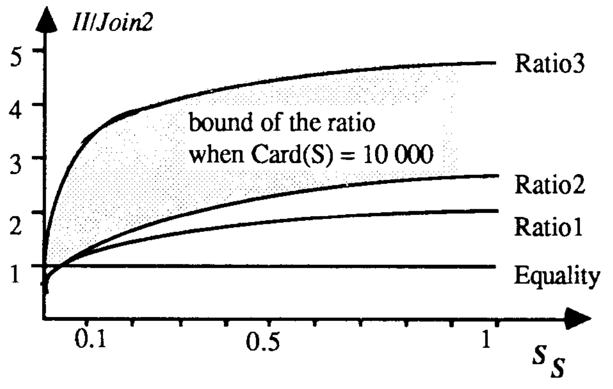
```
time(JI) =
        /* join index cardinality (in the general case)
    J_RS Card(R)/S_S
            /* copy an OID couple
        2 (u + w)

    = Card(R) (4J_RS /S_S) u
```

To reduce the significant number of parameters in the evaluation, the algorithms comparaisons are expressed in ratio form :

$$\frac{time(II)}{time(Join2)} = \frac{1 + 2l \log_2 (S_S Card(S)) + J_{RS} (4 + 3/S_S)}{6 + J_{RS} (4 + 3/S_S)}$$

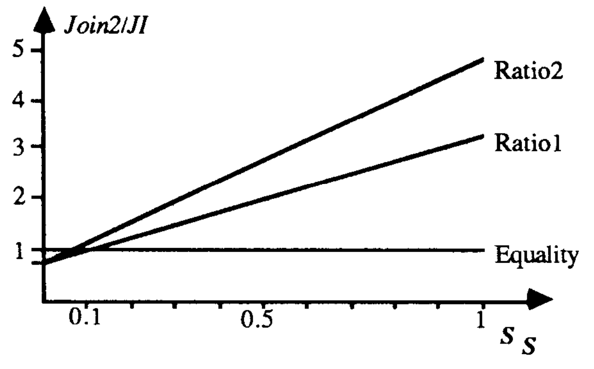
$$\frac{time(Join2)}{time(JI)} = \frac{6 + J_{RS} (4 + 3/S_S)}{4 J_{RS}/S_S}$$

The first ratio depends on the parameters J_{RS}, S_S and $Card(S)$ and the second one depends only of J_{RS} and S_S. They are plotted in Figures 10 and 11.



Value of the fixed parameter: l=1.
Ratio1 corresponds to $Card(S)$=1000 and J_{RS} =1.
Ratio2 corresponds to $Card(S)$=10000 and J_{RS} =1.
Ratio3 corresponds to $Card(S)$=10000 and J_{RS} ->0.

Figure 10: II versus Join2



Ratio1 corresponds to J_{RS} =1.
Ratio2 corresponds to J_{RS} =0.5.

Figure 11: Join2 versus JI

The form of the curves in Figure 10 denotes the importance of the logarithmic index search time in algorithm II and demonstrates the superiority of Join2. Moreover, the time for comparing two values has been minimized by setting l to 1. The slope of the curves Ratio1 and Ratio2 in Figure 11 depends on J_{RS}. The superiority of JI over Join2 is quite natural since JI constitutes a minimal bound for a join between two permanent relations (a simple copy of the result need be done). Low values of J_{RS} favor IJ because Join2 accesses many irelevant tuples. Note that this drawback is avoided by Join1. Curve Ratio1, corresponding to J_{RS} =1, defines the lowest bound for the ratio $time(Join2)/time(JI)$. In summary, the curves analysis shows that : $time(JI) < time(Join2) < time(II)$ for all joins involving two permanent relations. High values for J_{RS} minimizes these differences by increasing the time to produce the join result common to all algorithms.

5.3.3. Join involving temporary relations

Algorithms Join2 and JI are now compared for the join $\Delta R \otimes S$, where ΔR is a temporary relation (based on R) and S is a base relation. Relation ΔR is materialized by a list of OIDs referencing tuples of R. Algorithms Join2 and JI, renamed $Join2_{TP}$ and JI_{TP}, have to be slightly modified to work on temporary results. Algorithm II has the same behaviour as Join2 when dealing with temporary operands, thus the difference between the two algorithms should remain constant for both permanent and temporary relations. Therefore we concentrate on the comparison between Join2 and JI.

Algorithm $Join2_{TP}$ scans ΔR and decodes each OID to access the corresponding tuple of R. The join of this tuple of R with relation S is performed like in Join2, which gives :

```
$time(Join2_{TP})$ =
        /* scan the temporary relation $\Delta R$
    $Card(\Delta R)$
            /* read and decode an OID $\in \Delta R$
        $((u + d)$
            /* apply succ_tup to get the join attribute value
        $+ (u + d)$
            /* check the existence of an inverted sublist for
            the join attribute of S
        $+ o)$
        /* number of $\Delta R$ tuples matching with an S tuple
    $+ J_{RS}\ Card(\Delta R)$
            /* access to the inverted sublist for the join
            attribute of S
        $(d$
            /* Cartesian product of an inverted list and an OID
        $+ (u + (u + 2w)/S_S))$

        $= Card(\Delta R)\ (10 + J_{RS} (4 + 3/S_S))\ u$
```

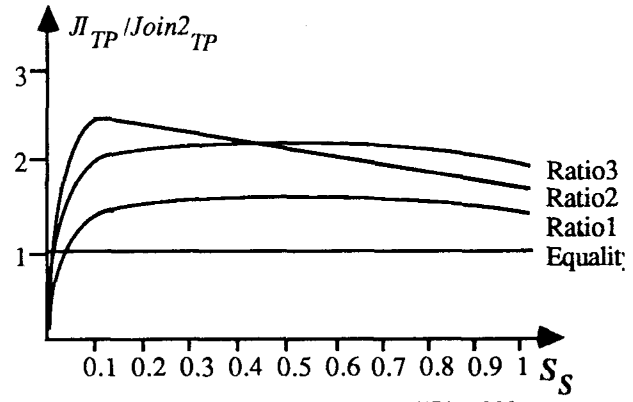
Algorithm JI_{TP} performs a semi-join between the join-index and ΔR in order to select in the join index the couples of OID's such that the R-OID belongs to ΔR. This operation can be optimized using two versions of the join index, each one sorted on the OID's of one relation [Vald87]. Thus, checking whether the OID belongs to ΔR can be done by a dichotomic search, yielding :

```
time(JI_TP) =
        /* scan of the temporary relation ΔR
    Card(ΔR)
        /* read an OID ∈ ΔR
    (u
        /* dichotomic search of this OID in the join-index
      + o  log2(J_RS Card(R)/S_S))
      /* copy the couples of the join-index matching for
       the semi-join
    + 2(u + w) J_RS Card(ΔR)/S_S
```

$= Card(\Delta R)\ (1 + 2 \log_2(J_{RS}\ Card(R)/S_S) + 4 J_{RS}/S_S)\ u$

The ratio between $time(JI_{TP})$ and $time(Join2_{TP})$, shown in Figure 12, is given by the formula:

$$\frac{time(JI_{TP})}{time(Join2_{TP})} = \frac{1 + 2\ \log_2(J_{RS}\ Card(R)/S_S) + 4J_{RS}/S_S}{10 + J_{RS}\ (4 + 3/S_S)}$$



Ratio1 corresponds to J_{RS} =1, $Card(R)$=1000.
Ratio2 corresponds to J_{RS}=0.1, $Card(R)$=10000.
Ratio3 corresponds to J_{RS} =1, $Card(R)$=10000.

Figure 12: $\mathrm{JI_{TP}}$ versus $\mathrm{Join2_{TP}}$

For J_{RS} varying from 0.1 to 1, the ratio $time(JI_{TP})/time(Join2_{TP})$ is between 1.5 and 2.5. The superiority of Join2 over Join_Index is mainly due to the logarithmic search time in the join index. The difference between the two algorithms increases with Card(R) (see Ratio3 compared to Ratio1), since Card(R) determines the size of the join-index. This logarithmic search time is predominant when J_{RS} is low. However, when J_{RS} is high, the time to produce the result common to both algorithms becomes significant. This explains the logarithmic shape of the curves. The curves shape is amplified by S_S (see Ratio2 compared to Ratio3) because the effects of S_S and J_{RS} are combined through the factor J_{RS}/S_S. In summary, Join2 outperforms JI for joins involving at least one temporary relation, which is the general case.

5.4. Update performance

Updating the attribute R.k of a tuple t in a DBGraph consists of deleting an existing link with value v_1 and creating a new link with value v_2. This incures an OID update in tuple t and the update of the inverted sublists associated with values v_1 and v_2. If value v_2 (resp. v_1) is not shared, it is also necessary to insert v_2 (resp. suppress v_1) in the domain, update the domain index and create (resp. delete) the corresponding inverted sublist attached to that value. The resulting update cost is roughly equivalent to the cost of updating an indexed attribute in FF. Creating (resp deleting) a tuple t consists of creating (resp. deleting) links with each of its attribute values.

Generally integrity constraints must be checked for update operations. A large class of integrity constraints can be efficiently checked on the fly during the DBGraph domain updates without incuring any additional cost. This powerful mechanism is illustrated below on two widely used integrity constraints.

- *unique key constraints:* at tuple insertion time, if an OID already exists in the inverted sublist associated with the domain value corresponding to the key attribute, then the constraint is violated and the insertion fails.
- *referential constraints:* if an attribute S.j references a key attribute R.k, each tuple t inserted in S must satisfy the predicate $(\exists t' \in R / t_{S.j} = t'_{R.k})$. Conversely, each tuple t deleted from R must satisfy the predicate $(\forall t' \in S / t'_{R.k} \neq t_{S.j})$. Inserting a tuple t in S requires to update the inverted sublist associated with the domain value corresponding to attribute t.j. If the inverted sublist of attribute R.k attached to the same domain value is empty, then the referential constraint is violated and the insertion fails. Similarly, when deleting a tuple t from R, the inverted sublist of attribute S.j attached to the value of attribute R.k must be empty in order to satisfy the referential constraint.

6. CONCLUSIONS

In this paper, we proposed the DBGraph storage model (DBG) for efficient main memory data management. This model achieves both compactness and efficient processing for all database operations. Although it is intented for various higher level data models, we stayed in the relational realm for simplicity.

The definition of DBG was given independent of any implementation detail in terms of graph structure and primitive operations on that structure. This facilitated the description of complex operation algorithms in an abstract form. By storing tuples and values separately and linking them through OID's, a DBGraph precompiles select, join and transitive closure over the entire database. We proposed an efficient implementation of the DBGraph in terms of objects (tuples, values, indices) clustered into segments.

DBG exhibits four important properties. First, a DBGraph can be partitioned so that the active database can be entirely maintained in main memory. Taking advantage of the DBGraph's vertical partitioning, the proposed implementation allows clustering of partitions into disk segments and efficient loading in main memory without format conversion. Second, all database operations can be performed by a DBGraph traversal without tuple comparison and move. Third, the temporary tuples are integrated in the DBGraph and thus can be processed as efficiently as permanent tuples. Fourth, pipelined (resp. set-oriented) query processing can be obtained by depth-first (resp. breadth-first) traversal of the DBGraph. Although more difficult to implement, pipelined processing should be preferred since it minimizes space occupancy for temporary results.

Our analysis showed that DBG provides good storage occupancy. Compared to the flat file organization (FF) without index (the simplest compact way of storing data), DBG is less efficient for short attribute values with high domain selectivity (i.e., many different values). As indices are added to FF, DBG rapidly becomes best in all cases.

Our performance analysis has concentrated mainly on join for it is the most critical performance-related operation in relational systems. Join on a DBGraph takes advantage of pointer-based data access as in [Shek90]. We first compared the DBG algorithm Join2 versus the join algorithm using inverted indices (II) and the join algorithm using join indices (JI) when the operand relations are both permanent. The results indicate that Join2 always outperforms II but, for good join selectivity, JI outperforms Join2. However, in the more likely case that one relation is temporary (e.g., after a select), Join2 generally outperforms JI. The update performance of DBG is roughly equivalent to that of FF with inverted indices. In addition, DBG has the ability of checking important integrity constraints (unique key, referential) with no overhead during updates.

Performance advantages of DBG for other operations (select, transitive closure, etc.) are reported in [Puch89b]. Its excellent performance for graph traversal operations such as transitive closure should be compared to that of more specialized data structures [Agra89] which also represent relationships between tuples directly. Implementation of DBG is on-going on top of a UNIX-based object manager. It will enable us to validate the analytical results with performance measurements. Furthermore, it will give us a basis to extend DBG for complex object support.

ACKNOWLEDGEMENTS

The authors wish to thank Karima Bennis, Michel Freville and Sylvie Renard for fruitful discussions and for their active participation in prototyping the DBGraph storage model.

REFERENCES

[Agra89] Agrawal R., Bargia A., and Jagadish H., "Efficient Management of Transitive Closure Relationships in Large Data and Knowledge Bases", ACM SIGMOD Int. Conf., Portland, Oregon, May 1989.

[Amma85] Ammann A., Hanrahan M., and Krishnamurthy R., "Design of a Memory Resident DBMS", IEEE COMPCON, San Fransisco, California, February 1985.

[Banc86] Bancilhon F., "An Amateur's Introduction to Recursive Query Processing Strategies", ACM SIGMOD Int. Conf., Washington, D.C., May 1986.

[Bitt83] Bitton D., DeWitt D., Turbyfill C., "Benchmarking Database Systems : a Systematic Approach", Int. Conf. on VLDB, Florence, Italy, November 1983.

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

[Brat84] Bratbergsengen K., "Hashing Methods and Relational Algebra Operations", Int. Conf. on VLDB, Singapore, August 1984.

[Card75] Cardenas A., "Analysis and Performance of Inverted Data Base Structures", CACM, Vol. 18, No. 5, May 1975.

[Cope85] Copeland G., Khoshafian S., "The Decomposition Storage Model", ACM SIGMOD Int. Conf., Austin, Texas, May 1985.

[Cope86] Copeland G., Khoshafian S., Smith M., Valduriez P., "Buffering Schemes for Permanent Data", Int. Conf. on Data Engineering, Los Angeles, California, February 1986.

[Cope89] Copeland G., Keller T., Krishnamurthy R., Smith M., "The Case for Safe RAM", Int. Conf. on VLDB, Amsterdam, the Netherlands, August 1989.

[Cope90] Copeland G., Franklin M., Weikum G., "Uniform Object Management", Int. Conf. on EDBT, Venice, Italy, March 1990.

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

[Eich89] Eich M.H., "Main Memory Database Research Directions", Int. Workshop on Database Machines, Deauville, France, June 1989.

[Gard88] Gardarin G., Pucheral P., "A Graph Operator to Process Efficiently Linear Recursive Rules in a Main Memory Oriented DBMS", 3rd Database Symposium, Recife-Pernambuco, Brazil, Mars 1988.

[Gibb85] Gibbons A., "Algorithmic Graph Theory", Cambridge University Press, 1985.

[Lehm86a] Lehman T., Carey M., "A Study of Index Structures for Main Memory Database Management Systems", Int. Conf. on VLDB, Kyoto, Japan, August 1986.

[Lehm86b] Lehman T., Carey M., "Query Processing in Main Memory Database Management Systems", ACM SIGMOD Int. Conf., Washington, D.C., May 1986.

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

[Miss83] Missikov M., Scholl M., "Relational Queries in a Domain Based DBMS", ACM SIGMOD Int. Conf., San Jose, May 1983.

[Puch89a] Pucheral P., Thévenin J.-M., "A Graph Based Data Structure for Efficient implementation of Main Memory DBMS's", Int. Workshop on Database Machines, Deauville, France, June 1989.

[Puch89b] Pucheral P., "Extensibilité et Performance d'un SGBD Basé sur un Gérant d'Objets", PhD Dissertation, University of Paris 6, December 1989.

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

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

[Shek90] Shekita E., Carey M., "A Performance Evaluation of Pointer-Based Joins", ACM SIGMOD Int. Conf., Atlantic City, New Jersey, May 1990.

[Thev89] Thévenin J.-M., "Architecture d'un Système de Gestion de Bases de Données Grande Mémoire", PhD Dissertation, University of Paris 6, December 1989.

[Ullm82] Ullman J., "Principle of Database Systems", Computer Science Press, 1982.

[Vald86a] Valduriez P. , Khoshafian S., Copeland G., "Implementation Techniques of Complex Objects", Int. Conf. on VLDB, Kyoto, Japan, August 1986.

[Vald86b] Valduriez P., Boral H., "Evaluation of Recursive Queries Using Join Indices", Int. Conf. on Expert Database Systems, Charleston, April 1986.

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