Database Graph Views : A Practical Model to Manage Persistent Graphs[1]

Alejandro Gutiérrez †‡ Philippe Pucheral † Hermann Steffen ‡ Jean-Marc Thévenin ††

(†) University of Versailles
PRiSM Laboratory
78000 Versailles, France
{gutier, pucheral}@prism.uvsq.fr

(‡) Universidad de la República
Computer Science Department
11300 Montevideo, Uruguay
steffen@fing.edu.uy

(††) University of Toulouse 1
IRIT / SIG Laboratory
31042 Toulouse, France
thevenin@cix.cict.fr

Abstract

Advanced technical applications like routing systems or electrical network management systems introduce the need for complex manipulations of large size graphs. Efficiently supporting this requirement is now regarded as a key feature of future database systems. This paper proposes an abstraction mechanism, called *Database Graph View*, to define and manipulate various kinds of graphs stored in either relational, object oriented or file systems. A database graph view provides a functional definition of a graph which allows its manipulation independently of its physical organization. Derivation operators are proposed to define new graph views upon existing ones. These operators permit the composition, in a single graph view, of graphs having different node and edge types and different implementations. The graph view mechanism comes with an execution model where both set-oriented and pipelined execution of graph operators can be expressed. The graph view operators form a library which can be integrated in database systems or applications managing persistent data with no repercussion on the data organization.

1 Introduction

The efficient manipulation of graphs becomes an important challenge for advanced database systems addressing technical applications. Geographical information systems, routing systems, military systems and more generally applications managing networks have to perform complex computations (e.g. shortest path, maximum capacity path, bill of materials) on voluminous graphs [Cruz88]. Nodes and edges of the graphs are generally represented by complex objects stored in databases or files under various forms. Nodes and edges of different types can be linked in the same graph (e.g. power stations, connectors, switches) making difficult the traversal of the graph by a recursive algorithm [Impr93]. In some applications, the same graph may have to be seen at different levels of detail. In such a case, only the most detailed level is stored in the database and higher levels have to be computed based on the detailed one (e.g. the connection between two power stations may be derived by aggregating basic electrical lines supporting the connection). Finally, some connections between nodes and edges may result from complex calculus that must be computed at traversal time (e.g. two power stations are connected if all the elementary electrical lines materializing the connection are in operational state).

Considerable research efforts focused on the integration of transitive closure operators in database systems to support complex queries on recursively defined data [VB86, AJ87, Lu87, PTV90, IRW93]. Generalized transitive closure operators integrating computations on node and edge labels have also been defined to cope with practical graph traversal problems [Agr87, ADJ88, CN89, DAJ91]. Most of these works have been conducted in the relational realm thereby laying down a relational representation of the graphs (e.g. {(origin, destination, labelEdge)}). Recent proposals define languages to query graphs where nodes and edges are represented by typed objects. In the GraphLog [CM90] and Gram [AS92] languages, regular expressions are used to specify a set of relevant paths in graphs where nodes and edges may have different types. This constitutes a first step to satisfy the requirements stated above. However, these proposals are more concerned with query expression than with execution of graph operators on specific graph implementations.

This paper focuses on database mechanisms to handle graphs issued from different paradigms and data models. The database graph view model is introduced to allow the definition of graph operators without

[1] This work is partially funded by the Esprit project IMPRESS n° 6355

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

precluding a specific graph representation in the database. A database graph view (or graph view for short) is a functional definition of a graph specifying the way this graph should be traversed, independently of its implementation. Each graph view defines a specific underlying graph over the database objects. Thus, several underlying graphs can be defined on the same set of objects by means of several graph views. An important feature of the model is that graph views can be defined on existing databases without reconsidering the data organization. In addition, either general purpose graph operators (e.g. generalized transitive closure) or application-dependent graph operators can exploit the graph view model.

Derivation operators are proposed to define new graph views by composing already existing ones. Unary derivation operators can be used to select some nodes and edges in a graph view and to project the labels of the resulting nodes and edges on relevant values. Binary derivation operators can be used to build a graph view resulting from the union, the intersection or the difference of two graph views. Precise rules are given to properly define node and edge identifiers in order to guarantee a clear semantics for the binary operators when applied on graph views having different node and edge types. We show that graph operators like generalized transitive closure operators can be applied on derived graph views either in a pipelined mode or in a set-oriented mode. In the former case, derivation rules are evaluated each time the successors of a node have to be accessed while in the later case, the derived graph view is materialized prior to the graph operator execution in order to speed up the recursive process. The best strategy depends on several parameters among them the size of the underlying graph and the number of traversed edges during the processing of the graph operator. We started the implementation of the graph view model in the IMPRESS Esprit project. We currently validate it in three different contexts, namely a Geographical Information System (GIS), an electrical network maintenance system in Spain and a milk dispatching system in Uruguay.

This paper is organized as follows: Section 2 introduces the database graph view model. This includes the definition of a graph view and the semantics of the derivation operators that make possible the definition of complex graph views derived from already existing ones. Section 3 focuses on query processing on graph views. Different operational implementations of graph views are first discussed. Then some hints are given for a query optimizer to exploit both pipelined and set-oriented execution modes these implementations can support. Section 4 gives our conclusions.

2 Database Graph View

2.1 Application Sample

The database graph view model aims at supporting the requirements of technical applications managing data organized as graphs. Throughout this paper, we use a routing system application to illustrate these requirements and to motivate the solutions we propose. This routing system uses the database presented in Figure 1 that could be stored in a GIS. Two initial graphs are explicitly defined in this database, namely a Road graph and a Railway graph. While the railway segments are directly connected to train stations, materializing the connections between road segments and cities requires a geometrical calculus (i.e. r.origin $\subset$ c.shape, where r is a road segment, c is a city and $\subset$ is a geometric operator checking the inclusion of a point into a region). In addition, we can define the CityTrain graph as an abstraction of the Railway graph. CityTrain is concerned only with the train connection between cities thereby aggregating in a single node all the stations located in the same city and in a single edge all the connections between these stations. Clearly, there is a need for an abstraction mechanism to map the nodes and edges of the Road, Railway and CityTrain underlying graphs and the objects physically stored in the database.

```
class City {                    class Station {
CityName name;                  StationName station_name;
int population;                 CityName city_name; }
Region shape; }

class RoadSegment {             class RailwaySegment {
Point origin, destination;      StationId origin, destination;
RoadType roadtype;              Time duration;
Length length; }                ... }

name Cities : set (City);
name RoadSegments : set (RoadSegment);
name RailwaySegments : set (RailwaySegment);
```

Figure 1: Routing database represented in an object oriented form

A typical requirement of the routing system over this database could be to query the graph of strategic roads, defined as the graph composed of cities and roads such that the roads connect cities which are not served by the train. Defining this graph upon the Routing database is a tedious task for the application designer. This introduces the need for ad-hoc operators to properly derive new underlying graphs from existing graphs having different node and edge types.

2.2 Graph View Formal Definition

We can identify two main elements that participate in the scenario presented above. First, a set of objects which compose the database defined using a particular data model and second, a set of underlying graphs defined over this database. We first give a formal definition of a graph view to fix a non ambiguous semantics for the underlying graphs that can be represented.

Conceptually, let Ω be the set of objects of the database. Let $S = \{C_1, C_2, ..., C_n\}$ be the *database scheme*. Each C_i is a collection of objects composed of attributes of the form $A_j : t_j$ where A_j is the attribute

```
DefineGraphView  Road (CityID, RoadSegmentID, (name : CityName, population : int),
                                              (roadtype : RoadType, distance : Length))
BuildNodeSet is
        Select C.CityID, (C.name, C.population) from C in Cities
BuildEdgeSet is
        Select   R.RoadSegmentID, C1.CityID, C2.CityID, (R.roadtype, R.length)
        From     C1 in Cities, C2 in Cities, R in RoadSegments
        Where    R.origin ⊂ C1.shape                    /* here, ⊂ is a geometric operator */
        and      R.destination ⊂ C2.shape
```

Figure 2: Operational definition of the Road graph view

name and t_j is the corresponding attribute domain or type. Let D be the universal domain defined as the union of all attribute domain values of the database.

A *database graph view* of Ω is a labelled multigraph G = (N, E, LN, LE, Incidence, LabelNode, LabelEdge) where,

- N is the set of nodes (node identifiers) of the database graph view taken from Ω.
- E is the set of edges (edge identifiers) of the database graph view taken from Ω.
- LN is a set of node labels taken from D.
- LE is a set of edge labels taken from D.
- Incidence is a function from E into N × N.
- LabelNode is a function from N into LN.
- LabelEdge is a function from E into LE.

According to this definition, a graph view consists of a set of nodes and a set of edges with associated labels. The nodes and edges are derived from objects of Ω. Each node and edge must be uniquely identified within a graph view. This does not preclude a single node or edge to be derived from several objects of Ω or a single object of Ω to participate in the definition of several nodes or edges of the same or different graph view(s). The relations between edges and nodes are given by the Incidence function which allows in particular to define the neighbours of a node. Considering edges and their labels apart allows to have multiple edges with the same valuation between a given pair of nodes. Either directed or undirected underlying graphs can be defined depending on whether or not the Incidence function distinguishes between (n, m) and (m, n) (where n, m ∈ N).

The *scheme of a database graph view* is a tuple of the form (NodeIdType, EdgeIdType, NodeLabelType, EdgeLabelType) where,

- NodeIdType is the type of the node identifiers in N.
- EdgeIdType is the type of the edge identifiers in E.
- NodeLabelType is the type of the LN's labels. It has the form $(A_1 : t_1, A_2 : t_2, ..., A_p : t_p)$.
- EdgeLabelType is the type of the LE's labels. It has the same form as NodeLabelType.

We introduce the notion of scheme for graph views to properly define derivation operators involving graph views having same NodeIdType and EdgeIdType but having different NodeLabelType and EdgeLabelType.

2.3 Graph View Operational Definition

The operational definition of a graph view corresponds to the implementation of the graph view model. This definition must fix the rules to identify the nodes and the edges of the underlying graph defined by a graph view. It must also provide a way to traverse the elements of this graph and to access their labels, independently of the graph implementation in the database. Different operational definitions can be given for a graph view, depending on the data model used to define the database and on the graph operators that will be applied to the graph view. The most appropriate way to fix the operational definition of a graph view is a matter of query optimization, discussed in Section 3.

To illustrate the principle, we just give a possible operational definition for the Road graph view according to the database scheme presented in Figure 1. For clarity, this definition is expressed using a SQL-like syntax in Figure 2.

The complete scheme of the graph view is fixed by the DefineGraphView statement. In this example, the nodes (resp. edges) are identified by the identifiers of City instances (resp. RoadSegment instances). N, LN and LabelNode are defined by the function BuildNodeSet() which delivers the set of cities of interest together with their labels. E, LE, LabelEdge and Incidence are defined by the function BuildEdgeSet() which delivers the set of edges with their extremity nodes and their labels. The extremity nodes of each edge are determined by a geometric computation on the database objects. This points out the generality provided by the graph view mechanism. Although quite different in its expression, this definition conforms to the semantics of the formal definition introduced in Section 2.2. Pros and cons of this definition are discussed in Section 3.

2.4 Graph View Derivation

Complex graph views can be built by a recursive application of unary and/or binary operators on already defined graph views. Unary operators allow to select a subset of nodes and edges in a graph view and to project the labels of these nodes and edges on relevant values. Binary operators allow the construction of a graph view resulting from the union, the intersection or the difference of two graph views.

The derivation operators separate the structural treatment from the label treatment of the operand graph views. The structural part of the graph is given by N, E and the Incidence function, while the label part is given by LN, LE, LabelNode and LabelEdge. Separating the structural part of the graph from its label part allow to compose graphs that can have different NodeLabelType and EdgeLabelType. Note that the binary operators are properly defined only for operand graph views having the same type for their structural part (i.e. same NodeIdType and same EdgeIdType). To illustrate the semantics of each derivation operator, we will use comparable graph views, namely CityRoad and CityTrain, having the following schemes:

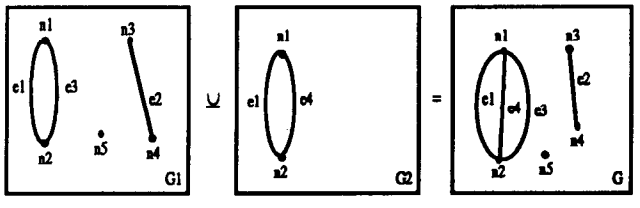
CityRoad (CityName, CityName × CityName, City, (roadtype:RoadType, length:Length))

CityTrain (CityName, CityName × CityName, City, ())

In the following, we define the semantics of the derivation operators in terms of the usual notions in the set theory. N_i, E_i, LN_i, LE_i, $Incidence_i$, $LabelNode_i$ and $LabelEdge_i$ denote the elements of a graph view instance G_i. In the figures, n_1, n_2, ..., n_p denote node identifiers (N elements) while e_1, e_2, ..., e_q denote edge identifiers (E elements).

Union

The union operation is denoted by $\cup$. The union of two graph views consists of the union of the nodes and edges of the graph operands and of the aggregation of their labels. For example, we can derive the complete transportation network between cities by performing the union of the CityRoad and CityTrain graphs. The following example shows the structural result of the union of two graphs. For clarity, labels are not considered in the example but can be easily deduced from the definition.



$G = G_1 \cup G_2$ is defined by:

G (NodeIdType$_1$, EdgeIdType$_1$,
(NodeLabelType$_1$ ∪ {null}) × (NodeLabelType$_2$ ∪ {null}),
(EdgeLabelType$_1$ ∪ {null}) × (EdgeLabelType$_2$ ∪ {null}))

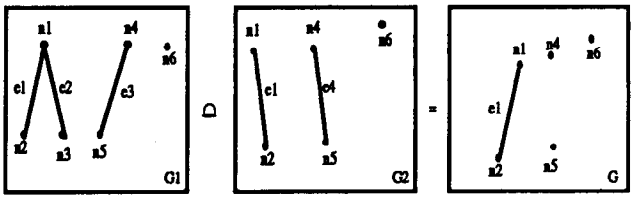
- $N = N_1 \cup N_2$
- $E = E_1 \cup E_2$
- $LN = (LN_1 \cup \{null\}) \times (LN_2 \cup \{null\})$
- $LE = (LE_1 \cup \{null\}) \times (LE_2 \cup \{null\})$
- Incidence ($e \in E$) = if $e \in E_1$ then $Incidence_1$ (e) else $Incidence_2$ (e)
- LabelNode ($n \in N$) = ($LabelNode_1$ (n), $LabelNode_2$ (n))
- LabelEdge ($e \in E$) = ($LabelEdge_1$ (e), $LabelEdge_2$ (e))

Remarks: $LabelNode_i$ (n) = null if $n \notin N_i$ and $LabelEdge_i$ (e) = null if $e \notin E_i$

This definition leads to two remarks. First, the incidence functions of G1 and G2 generally deliver the same result when applied to the same edge. In particular cases, this result may be different due to an ambiguous identification of edges (see Section 2.5 for a detailed example). In such a case, the definition presented above chooses the incidence function of G1. Other possible choices could be to select the incidence function of G2 or to discard the corresponding edge. Second, the labels of the nodes and edges resulting from the union are built by aggregating the labels issued from the two operand graphs into a tuple of the form (label1, label2). This could be meaningless when the two operand graphs have the same NodeLabelType and EdgeLabelType. For conciseness, we do not introduce an extra definition to handle this case. This second remark applies as well to all the binary derivation operators.

Intersection

The intersection operation is denoted by $\cap$. The intersection of two graph views basically consists of the intersection of the nodes and edges of the graph operands and of the aggregation of their labels. For example, we can derive the graph of cities connected both by train and by roads by doing the intersection between the CityRoad and CityTrain graphs. The following example shows the structural result of the intersection of two graphs.



$G = G_1 \sqcap G_2$ is defined by:

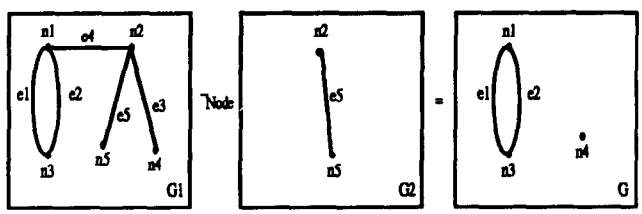
G (NodeIdType$_1$, EdgeIdType$_1$,
NodeLabelType$_1$ × NodeLabelType$_2$,
EdgeLabelType$_1$ × EdgeLabelType$_2$)

- $N = N_1 \cap N_2$
- $E = \{e \in E_1 \cap E_2 / \text{Incidence}_1(e) = \text{Incidence}_2(e)\}$
- $LN = LN_1 \times LN_2$
- $LE = LE_1 \times LE_2$
- Incidence ($e \in E$) = Incidence$_1$ (e) = Incidence$_2$ (e)
- LabelNode ($n \in N$) = (LabelNode$_1$ (n), LabelNode$_2$ (n))
- LabelEdge ($e \in E$) = (LabelEdge$_1$ (e), LabelEdge$_2$ (e))

Note that the E set of the graph view result is not merely the intersection of the edges of the graph view operands. Otherwise, an inconsistent graph may be produced if there is an edge in both graphs for which the respective incidence functions give a different result. An edge with extremity nodes that do not belong to N could be generated (see Section 2.5 for an example).

Node Difference

The node difference operation is denoted by $-_{Node}$. The node difference between two graph views G_1 and G_2 yields a subgraph of G_1 containing only the nodes of G_1 which do not belong to G_2 and the edges connecting these nodes. The node difference between the CityRoad and the CityTrain graphs builds the graph of cities which are only connected by roads (and that could claim for a connection to the railway network). The following example shows the structural result of the node difference of two graphs.



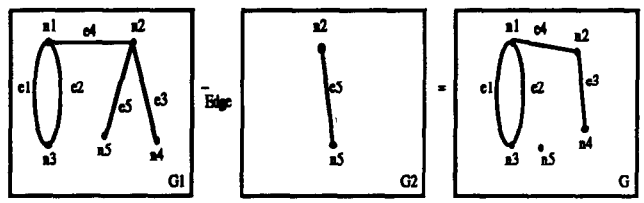
$G = G_1 -_{Node} G_2$ is defined by:

G (NodeIdType$_1$, EdgeIdType$_1$,
NodeLabelType$_1$, EdgeLabelType$_1$)

- $N = N_1 - N_2$
- $E = \{e \in E_1 / \exists n, m \in N, \text{Incidence}(e) = (n, m)\}$
- $LN = LN_1$
- $LE = LE_1$
- Incidence ($e \in E$) = Incidence$_1$ (e)
- LabelNode ($n \in N$) = LabelNode$_1$ (n)
- LabelEdge ($e \in E$) = LabelEdge$_1$ (e)

Edge Difference

The edge difference operation is denoted by $-_{Edge}$. The edge difference between two graph views G_1 and G_2 yields a partial graph of G_1 containing the edges which do not belong to G_2. The edge difference between the CityRoad graph and the graph of blocked roads (i.e. a temporary graph determined by an helicopter pilot) delivers the graph of available roads. The following example shows the structural result of the edge difference of two graphs.



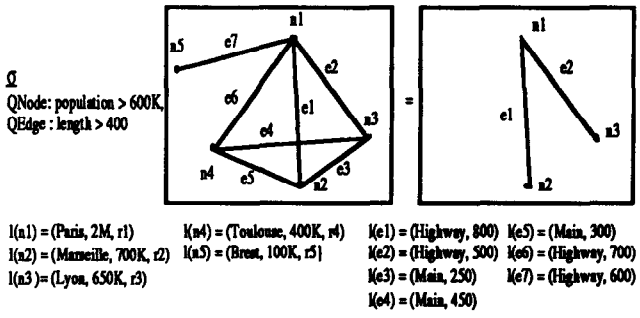
$G = G_1 -_{Edge} G_2$ is defined by:

G (NodeIdType$_1$, EdgeIdType$_1$,
NodeLabelType$_1$, EdgeLabelType$_1$)

- $N = N_1$
- $E = E_1 - E_2$
- $LN = LN_1$
- $LE = LE_1$
- Incidence ($e \in E$) = Incidence$_1$ (e)
- LabelNode ($n \in N$) = LabelNode$_1$ (n)
- LabelEdge ($e \in E$) = LabelEdge$_1$ (e)

Selection

The selection operation is denoted by $\sigma_{QNode, QEdge}$. The selection applied to a graph view yields a graph whose nodes and edges satisfy respectively the predicates QNode and QEdge. The attributes that appear in QNode (resp. QEdge) must be attributes of NodeLabelType$_1$ (resp. EdgeLabelType$_1$). One of these two predicates can be omitted if the selection is applied only to nodes or to edges. These predicates can be expressed by SQL-like predicates or by user-defined functions depending on the way the graph view model is integrated in a system. The following example shows how to obtain from the CityRoad graph the subgraph corresponding to the cities whose population is greater than 600K that are connected by roads longer than 400 Km.



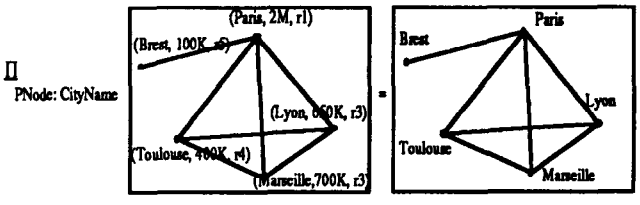
$G = \Omega_{QNode,\ QEdge}\ (G_1)$ is defined by:

G (NodeIdType$_1$, EdgeIdType$_1$,
NodeLabelType$_1$, EdgeLabelType$_1$)

- N = {n ∈ N_1 / QNode (LabelNode$_1$ (n))}
- E = {e ∈ E_1 / QEdge (LabelEdge$_1$ (e)) ∧ ∃ n, m ∈ N, Incidence (e) = (n, m)}
- LN = LN_1
- LE = LE_1
- Incidence (e ∈ E) = Incidence$_1$ (e)
- LabelNode (n ∈ N) = LabelNode$_1$ (n)
- LabelEdge (e ∈ E) = LabelEdge$_1$ (e)

Projection

The projection operation is denoted by $\Pi_{PNode,\ PEdge}$. The projection of a graph view G_1 yields a graph containing the nodes and edges of G_1 whose labels are the projection of the labels of G_1 according to PNode and PEdge. One of these two parameters can be omitted if the projection is applied only to nodes or to edges. π_{PNode} (NodeLabelType$_1$) means that the type of the node label aggregates the types of the NodeLabelType$_1$ attributes referenced in PNode. The same applies to π_{PEdge}(EdgeLabelType$_1$) and PEdge. The following figure shows the result of a projection on the Road graph.

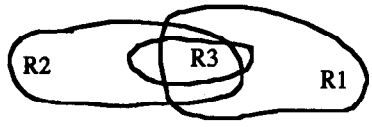


$G = \Pi_{PNode,\ PEdge}\ (G1)$ is defined by:

G (NodeIdType$_1$, EdgeIdType$_1$,
π_{PNode}(NodeLabelType$_1$),
π_{PEdge}(EdgeLabelType$_1$))

- N = N_1
- E = E_1
- LN = {ln / ln = π_{PNode} (ln$_1$), ln$_1$ ∈ LN_1}
- LE = {le / le = π_{PNode} (le$_1$), le$_1$ ∈ LE_1}
- Incidence (e ∈ E) = Incidence$_1$ (e)
- LabelNode (n ∈ N) = π_{PNode} (LabelNode$_1$ (n))
- LabelEdge (e ∈ E) = π_{PEdge} (LabelEdge$_1$ (e))

Example of a derivation expression. Graph views can be derived from already defined ones by complex expressions of derivation operators. Suppose that the transport network in the region R3 is blocked and we want to travel by car in the region R1 and by train in the region R2. We can obtain the graph of available railways and roads by the following expression.



$$\left(\sigma_{QNode:region\subset R1}(CityRoad) - \sigma_{Node\ QNode:region\subset R3}(CityRoad)\right) \cup \left(\sigma_{QNode:region\subset R2}(CityTrain) - \sigma_{Node\ QNode:region\subset R3}(CityTrain)\right)$$

2.5 Nodes and Edges Identification

All the binary derivation operations are based on the node and edge identifiers. These identifiers are defined according to the application needs, resulting in more flexibility in defining graphs over existing databases. The examples given below illustrate how the choice of these identifiers impacts the semantics of different graphs defined on the same data :

1. Road = (CityId, RoadSegmentId, –, –). In this case both identifiers are defined by the database object identifiers. Graph view schemes in which EdgeIdType is different from NodeIdType × NodeIdType allow the definition of multigraphs (i.e. graphs having several edges between two nodes).
2. CityRoad = (CityName, CityName × CityName, –, –). This is an usual definition for graphs that are not multigraphs. The identifiers correspond to some attributes of the database.
3. CityTrain = (CityName, CityName × CityName, –, –). In this case, each node (resp. edge) identifier results from the aggregation of a set of objects at the database level.

This flexibility requires adding conditions in the definition of the binary derivation operators to deal with graphs where EdgeIdType is not NodeIdType × NodeIdType nor object identifiers. Taking no precaution in such cases may lead to an ambiguous identification of edges in the graph resulting from the derivation. Let us consider the following graph view schemes:

G_1 = (Region, Point × Point, –, –)

G_2 = (Region, Point × Point, –, –)

Assume that the database represents the situation pictured in Figure 3. That is, the edge is uniquely identified by (p_1, p_2) in G1 and G2 but $Incidence_1((p_1, p_2)) = (A, B)$ while $Incidence_2((p_1, p_2)) = (C, D)$. When computing $G_1 \cap G_2$, the intersection of nodes is empty while the intersection of the edges gives (p_1, p_2). Thus, only the edges whose extremity nodes belongs to the intersection of nodes have to be kept in the result (see Section 2.4).

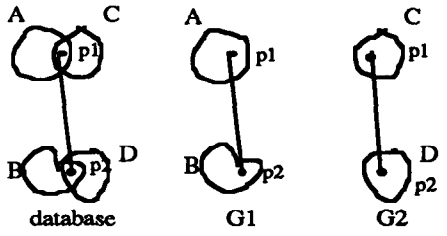


Figure 3: Ambiguous identification of edges

3 Query Processing on Graph Views

Section 3.1 comments on the three abstraction levels identified in the graph view model, namely the database, the base graph view and the derived graph view levels. Section 3.2 presents different alternatives to implement base graph views on top of the database level. The way derived graph views are mapped on the base graph view level is discussed in Section 3.3. Section 3.4 introduces optimization strategies to efficiently handle queries on either base graph views or derived graph views. Finally, Section 3.5 discusses the different alternatives to integrate database graph views in existing database systems or applications. This throws light on the way graph operators interacts with graph views.

3.1 Abstraction Levels

Each derivation operator produces a new derived graph view from one or two existing graph view(s). The derivation rules between graph views can themselves be expressed by a graph where the nodes represent graph views or operators and the edges represent *derivation links* (see Figure 4). Graph views defined upon database objects are called *base graph views* while graph views defined upon already existing graph views are called *derived graph views*. Each graph view defines an *underlying graph* on the objects physically stored in the database.

Graph operators and queries can be applied on either base graph views or derived graph views. Efficient mechanisms are required to extract base graph views from the database and then, derived graph views from base graph views. These translations hide from the graph operators first the physical representation of the underlying graphs in the database and second the derivation rules between graph views. These translation mechanisms are the foundation of the graph view model.

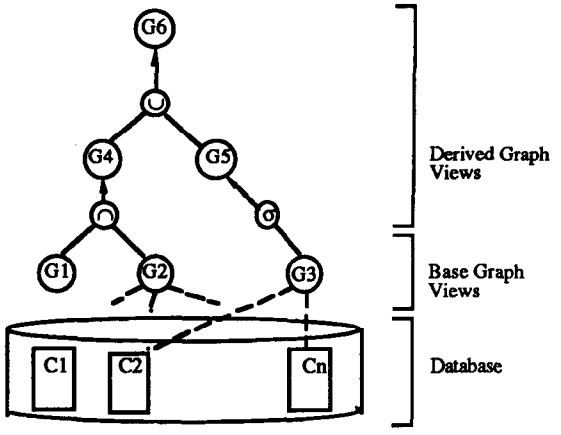


Figure 4 : Derivation tree of a graph view

3.2 Traversing the Underlying Graph of a Base Graph View

There are basically two methods to extract a base graph view from a collection of database objects. The first method consists in materializing the underlying graph defined by the base graph view by building dynamically its collection of nodes and its collection of edges prior to the execution of any graph operator. These two collections must conform to a predefined scheme understandable by all graph operators. The second method consists of defining a function, called Succ(), which delivers for an input node of the underlying graph the set of nodes connected to it. The graph operators can then recursively invoke the Succ() function to traverse the underlying graph. To make these two kinds of translations possible, the following functions have to be included by the DBA in the operational definition of each base graph view G:

BuildNodeSet $\rightarrow$
{(n, LabelNode (n)) / n $\in$ N }

BuildEdgeSet $\rightarrow$
{(e, n, m, LabelEdge (e)) / n, m $\in$ N $\wedge$ e $\in$ E $\wedge$ Incidence (e) = (n, m)}

Succ (n $\in$ N) $\rightarrow$
{(e, m, LabelNode (m), LabelEdge (e) / e $\in$ E $\wedge$ m $\in$ N $\wedge$ Incidence (e) = (n, m)}

As stated in Section 2.3, BuildNodeSet() and BuildEdgeSet() implement all the elements that enter in the formal definition of a graph view (i.e. N, E, Incidence, LN, LE, LabelNode and LabelEdge). These two functions build a pseudo-relational form of the underlying graph. We selected this representation for two reasons. First, most of the graph operators proposed in the database literature make the assumption that the graphs are stored as relations. Thus, all the principles introduced to speed up the recursive process on such graphs can be exploited in the graph view context. Second, this representation is well adapted to the storage

DefineGraphView	Road (CityID, RoadSegmentID, (name : CityName, population : int), (roadtype : RoadType, distance : Length))
BuildNodeSet is	*(see Figure 2)*
BuildEdgeSet is	*(see Figure 2)*
Succ(CityID) is	
Select	R.RoadSegmentID, C2.ID, (C2.name, C2.population), (R.roadtype, R.length)
From	C1 in Cities, C2 in Cities, R in RoadSegments
Where	C1 is CityID
and	R.origin ⊂ C1.shape
and	R.destination ⊂ C2.shape

Figure 5: Succ() definition for the Road graph view

of intermediate results in a query execution plan involving graph operators. One may argue that if the underlying graph edges are materialized by direct links (i.e. object identifiers) in the database, the traversal of these links is less costly than traversing a relational representation of the underlying graph. However, if any derivation operator is applied to the underlying graph - thereby producing an intermediate graph - object identifiers can no longer be used to traverse the graph, as indices referencing tuples on base relations cannot be directly used on intermediate relations.

The Succ() function provides a direct way to access the database objects from the underlying graph defined on them. It avoids the construction of an intermediate representation of the graph. When edges are not materialized by direct links in the database, it incurs an extra calculation each time the successors of a node have to be accessed during the graph traversal. The result of Succ(n) aggregates for each successor m of n, the identifier of the m node, the label of m and the identifier and the label of the edge linking n to m. Any traversal recursion algorithm - even those integrating restrictions and computations on the nodes and edges labels - can be built on the Succ() function basis. However, the graph algorithms having to enumerate all the entry nodes of a graph (e.g. to perform cycle detection or connectivity checking) require the BuildNodeSet() function in addition to Succ(). This is due to the fact that Succ() implements the Incidence, LN, LE, LabelNode and LabelEdge elements of the formal definition of a graph view but considers only the subset of N and E accessed at traversal time. As a consequence, restraining the operational definition of a base graph view to the Succ() function restrains the computation on this graph view to path traversal problems. We give in Figure 5 the definition of the Succ() function that must be part of the operational definition of the Road base graph view introduced in Section 2.3. Obviously, the BuildNodeSet(), BuildEdgeSet() and Succ() functions can be expressed with any programming language to cope with performance or storage structure constraints.

Note that the BuildNodeSet(), BuildEdgeSet() and Succ() functions correspond to the implementation part of the graph view model while the N, E, Incidence, LabelNode and LabelEdge elements defined in Section 2.2 fix the semantics of the underlying graph as well as the semantics of the derivation operators. A DBA attempting to define a graph view has only to be aware of the implementation part of the model (i.e. of the operational definition of the graph view).

3.3 Traversing the Underlying Graph of a Derived Graph View

The translation of a derived graph view G into a collection of base graph views (G1, G2, ...Gn) can be done by deriving the functions BuildNodeSet(), BuildEdgeSet() and Succ() from the corresponding functions defined on the base graph views G1, G2, ...Gn. This translation can be automatically done with no intervention from the DBA part. Each derivation rule between graph views comes with a derivation rule for the three aforementioned functions. The table presented in Figure 6 gives for each derivation operator the way to derive the three functions defining the resulting graph view from the functions of the operand graph views.

To illustrate these derivation steps, let us consider the derivation tree presented in Figure 4. In this example, the Succ() function automatically derived for graph view G6 corresponds to the functional expression given below. Note that the Succ() functions of the base graph views G1, G2 and G3 are the only ones to access the database objects.

$$Succ_6(n) = (Succ_1(n) \cap Succ_2(n)) \cup (\sigma_Q(Succ_3(n)))$$

3.4 Pipelined vs. Set-Oriented Query Processing

Let us now consider the execution of graph operators over database graph views. For the sake of conciseness, we focus on the processing of the generalized transitive closure operator (integrating computations on the nodes and edges labels) considered as a key operator of future database systems. While important research efforts focused on developing new algorithms and new data structures to support it efficiently, less studies addressed query optimization problems. One important contribution in this area [DAJ91] considers the optimization of generalized transitive closure queries of the form:

$$Q = Agg\ (\sigma\ (Con\ (Paths\ (G))))$$

	BuildNodeSet	BuildEdgeSet	Succ (n ∈ N)
∪	$\prod_{1,2+4}(BNS_1 *_{1=1} BNS_2) \cup \prod_{1,2+null}(BNS_1 \propto (\prod_1 BNS_1 - \prod_1 BNS_1)) \cup \prod_{1,null+2}(BNS_2 \propto (\prod_1 BNS_2 - \prod_1 BNS_1))$	$\prod_{1,2,3,4+8}(BES_1 *_{1=1} BES_2) \cup \prod_{1,2,3,4+null}(BES_1 \propto (\prod_1 BES_1 - \prod_1 BES_2)) \cup \prod_{1,2,3,null+4}(BES_2 \propto (\prod_1 BES_2 - \prod_1 BES_1))$	$\prod_{1,2,3+7,4+8}(S_1(n) *_{1=1} S_2(n)) \cup \prod_{1,2,3+null,4+null}(S_1(n) \propto (\prod_1 S_1(n) - \prod_1 S_2(n))) \cup \prod_{1,2,null+3,null+4}(S_2(n) \propto (\prod_1 S_2(n) - \prod_1 S_1(n)))$
D	$\prod_{1,2+4}(BNS_1 *_{1=1} BNS_2)$	$\prod_{1,2,3,4+8}(BES_1 *_{1=1 \wedge 2=2 \wedge 3=3} BES_2)$	$\prod_{1,2,3+7,4+8}(S_1(n) *_{1=1 \wedge 2=2} S_2(n))$
− Node	$BNS_1 \propto (\prod_1 BNS_1 - \prod_1 BNS_2)$	$BES_1 \propto (\prod_1 BES_1 - \prod_1 (BES_1 *_{2=2 \vee 3=3} BES_2))$	$S_1(n) \propto (\prod_1 S_1(n) - \prod_1 S_2(n))$
− Edge	BNS_1	$BES_1 \propto (\prod_1 BES_1 - \prod_1 BES_2)$	$S_1(n) \propto (\prod_1 S_1(n) - \prod_1 S_2(n))$
σ	$\sigma_{QNode(2)} BNS_1$	$\sigma_{QEdge(4)} BES_1$	$\sigma_{QNode(3) \wedge QEdge(4)} S_1(n)$
Π	$\Pi_{1,PNode(2)} BNS_1$	$\Pi_{1,2,3,PEdge(4)} BES_1$	$\Pi_{1,2,PNode(3),PEdge(4)} S_1(n)$

BNS_i, BES_i and S_i stand for the BuildNodeSet(), BuildEdgeSet() and Succ() functions of the graph view operand G_i. * , $\propto$, σ and Π respectively denote the join, semi-join, selection and projection relational operators. When associated to relational operators, 1,2, ..., n denote attribute positions and i+j the concatenation of attributes i and j.

Figure 6: Derivation rules for BuildNodeSet(), BuildEdgeSet() and Succ()

where G is an input relation materializing a graph. Paths() enumerates all the paths of G. For each path, Con() concatenates the labels of the path edges into a unique path label. σ() selects all paths satisfying some predicates applied on the path labels and/or on the origin and destination nodes of paths. Finally, Agg() aggregates the paths having same origin and same destination as well as the labels of these paths. The contribution leads to a precise classification of the selection criteria attached to σ in the query expression. Whenever possible, these selection criteria are applied on G before starting the recursive process or are evaluated as soon as possible during the recursive process to prune unneeded paths. Such optimization rules can be exploited in the database graph view context as well.

Let G be a graph view itself resulting from a potentially complex expression involving graph view derivation operators. If we note D(G1, G2, ...Gn) the derivation expression of G from the base graph views G1, G2, ..., Gn, the query to be optimized becomes:

Q = **Agg** (σ (**Con** (**Paths** (D(G1, G2, ...Gn)))))

In the graph view context, either the graph view G is materialized by composing the functions BuildNodeSet() and BuildEdgeSet() of G1, G2, ...Gn and the transitive closure is applied on this materialization, or the transitive closure is directly applied on G by using a Succ() function derived from the Succ() functions of G1, G2, ...Gn. We will refer to these two strategies respectively as the *set-oriented strategy* and the *pipelined strategy*. Indeed, materializing G leads to a set-oriented evaluation of the expression D(G1, G2, ...Gn) while this expression is evaluated in a pipelined mode for each Succ() invocation performed by the Paths() operator. Roughly speaking, the best strategy depends on the size of the underlying graphs, the cost of each Succ() invocation, the percentage of edges traversed during the recursive process and the presence of indices.

Suppose that a user queries the Road graph view to find the shortest path between node A and node B traversing only the main roads. This can be expressed as:

Q = **Min**(length) σ_C **Add**(length) **Paths** (Road)

where,

C ≡ Origin=A and Destination=B and roadtype="main road"

As stated in [DAJ91], the Paths() operator can take advantage of predicates of the form *Origin=A* to enumerate only the relevant paths ($C_{initial}$ selection criteria). In addition, the predicate *roadtype="main road"* can be pre-processed to reduce the size of the Road graph before invoking Paths() ($C_{preprocess}$ criteria). The query optimizer can naturally exploit this second optimization by dynamically building a temporary graph view:

MainRoad = $\sigma_{QEdge:roadtype="main\ road"}$(Road)

and by applying Paths() on it. The query optimizer can add derivation operators in a query expression for optimization purpose without explicitly creating new graph view schemes in the database.

```
SuccMainRoad(CityID) = σQEdge(roadtype="main road") G.Succ(CityID)
which semantically corresponds to
SuccMainRoad(CityID) is
  Select   R.RoadSegmentID, C2.ID, (C2.name, C2.population), (R.roadtype, R.length)
  From     C1 in Cities, C2 in Cities, R in RoadSegment
  Where    C1 is CityID
  and      R.origin ⊂ C1.shape
  and      R.destination ⊂ C2.shape
  and      R.roadtype = "main road";
```

Figure 7: Pipelined version of the Succ() function

```
Succ(CityID) is
  Select   R.RoadSegmentID, C2.ID, (C2.name, C2.population), (R.roadtype, R.length)
  From     C1 in Cities, C2 in Cities, R in MainRoadSegment
  Where    C1 is CityID
  and      R.origin ⊂ C1.shape
  and      R.destination ⊂ C2.shape
```

Figure 8: Hybrid version of the Succ() function

```
Succ(n) is
  Select   G.EdgeId, G.NodeId2, G.LabelNode2, G.LabelEdge
  From     G in Materialized_Graph_View      /* result of the materialization phase */
  Where    G.NodeId1 = n
```

Figure 9: Set-oriented version of the Succ() function

We present below three strategies to evaluate the optimized version of Q:

Min(length) $\sigma_{C'}$ Add(length) **Paths** (MainRoad).

where,

C' ≡ Origin=A and Destination=B

In a pure pipelined evaluation of Q, the query optimizer invokes the Paths() operator with as parameter the derived $Succ_{MainRoad}$() function detailed in Figure 7. This semantically corresponds to pushing down the predicate roadtype="main road" to the Succ() function of the Road graph view. This solution should be the best if few edges are traversed during the processing of the shortest path, especially if the restriction can be speeded up via an index (e.g. index on roadtype in the example).

If no index permits to speed up the restriction, a more efficient solution is to evaluate the restriction on the RoadSegment class in a set-oriented way before invoking the Paths() operator. The restriction will produce an intermediate object set called MainRoadSegment and the Succ() function given as parameter to the Paths() operator has to be slightly changed by the query optimizer (see Figure 8). Note that this solution is practical only in the case where the query optimizer understands the programming language used to define Succ() (e.g. SQL). This strategy is a mix between a set-oriented and a pipelined evaluation of the query.

Finally, suppose that the spatial inclusion predicates (R.origin ⊂ C1.shape) and (R.destination ⊂ C2.shape) are very costly to evaluate and that a spatial index is defined on these attributes. It could be more efficient to materialize the view in a set-oriented way in order to produce an intermediate graph that can be efficiently treated by the Paths() operator. This can efficiently be done by a join operation between the result of invoking the BuildEdgeSet() function attached to the graph view definition (doing the assumption that the implementation of BuildEdgeSet() exploits the spatial index) and the result of the BuildNodeSet() function. In this case, the Succ() function given as parameter to the Paths() operator is built from the standard one defined on graphs stored in a relational form {(NodeId1, NodeId2, EdgeId, LabelNode1, LabelNode2, LabelEdge)} (see Figure 9).

The query optimizer has the ability to perform either pure pipelined evaluation or pure set-oriented evaluation for the same query and can even mix both in the same execution plan. If the queried graph view is derived by a more complex expression than the one presented above, the query optimizer may choose to materialize any

nodes of the derivation graph. More precise hints for accurate query optimization remain to be defined.

3.5 Integrating the Graph View Model in Existing Systems

The graph view model can be tightly integrated in existing database systems. As pictured in Figure 10, different components have to cooperate in order to query database graph views. First, the scheme of the graph views as well as their operational definitions have to be declared and registered in a metabase. For DBMS's supporting SQL-like languages, the examples of graph view declarations given throughout the paper illustrate well this step. The predicates involved in the selection and projection derivation operators can be expressed as formulas of this language. The query language has to be extended with operators dedicated to graph management (e.g. [MS90, DA93]) that can be applied on graph views. The query optimizer must also be extended to deal with graph operators (e.g. alpha-algebra operator [Agr87]) and to exploit the different evaluation strategies defined in Section 3.4. Finally, the graph operators themselves must traverse the underlying graphs through a Succ() function given as parameter by the query optimizer.

One may ask if the database graph view mechanism can be directly supported by a standard view mechanism without impacting the underlying DBMS. Three observations are important to answer this question. First, a standard view mechanism may be used to define the database graph views (the examples given along the paper are expressed using a SQL style). Second, we could express the derivation operators using it (Figure 6 shows that they can be expressed using relational operators) but the complex algebraic expressions have to be written by the application programmer. Database graph views provide a tool specific to an application domain. Finally, to use the graph operators on the database graph views and to take advantage of the given evaluation strategies, the extensions mentioned above are inevitably needed.

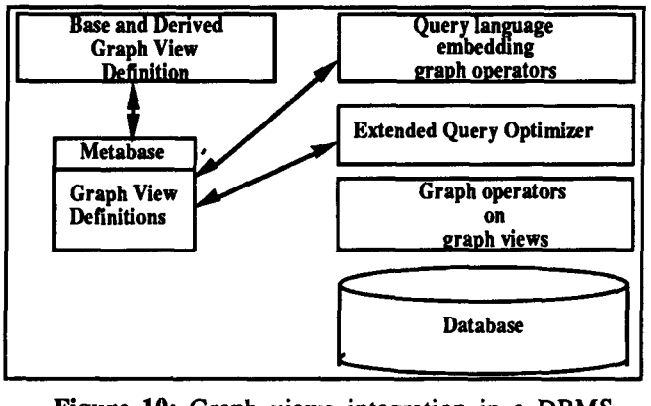


Figure 10: Graph views integration in a DBMS architecture

The graph view model can also be the basis of a graph operator library dedicated to applications managing graphs defined using a particular data model and stored in any repository (e.g. files). We are defining such a library integrating path traversal algorithms in the IMPRESS project [CGPT93]. This requires the management of an independent metabase to register the graph view definitions and to define rules to translate these definitions into the types of the data managed by the application (e.g. C types). Although we do not address this issue, such a library could be completed with high level languages to query graphs [CM90].

4 Conclusion

The database graph view model addresses three important requirements of technical applications managing large and complex graph structures. First, the model implements an abstraction mechanism which provides the application designer with the ability to define various underlying graphs on top of objects stored in databases or in files. Connections between nodes and edges may be either represented by physical links between objects or dynamically computed at traversal time. This enables to cope with complex graph organizations that cannot easily be mapped on predefined storage structures for graphs. Second, the model comes with a collection of powerful derivation operators. Graphs having different node and edge types can be combined using these operators to derive ad-hoc underlying graphs satisfying specific application requirements. The semantics of these operators integrates both the set of nodes, the set of edges and the labels of the operand graphs. Third, the model supports different execution strategies for graph operators exploiting graph views. The pipelined, set-oriented and hybrid execution strategies may outperform each other depending on the size of the queried graph and on the number of edges visited at traversal time. This opens new perspectives in the optimization of graph queries.

Our future work will focus on performance analysis of the proposed operators in the context of the three different technical applications we are working on. Our objective is to define precise rules for a query optimizer to fully exploit the different execution strategies identified in this paper.

Acknowledgements

We wish to thank Joël Ducassou and Eloi Chabaud for their active participation in the definition of the graph view derivation operators. Special thanks are also due to Cristina Cornes and Raul Ruggia for their precious help in improving the presentation of this paper. The first author was partly supported by PEDECIBA (Uruguay).

References

[ADJ88] Agrawal R., Dar S., Jagadish H.V., "On Transitive Closure Problems Involving Path Computations", AT&T Bell Laboratories Technical Memorandum, 1988.

[Agr87] Agrawal R., "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", Proc. IEEE 3rd Int'l Conf. on Data Engineering, Los Angeles, Feb. 1987, pp. 580-590.

[AJ87] Agrawal R., Jagadish H.V., "Direct Algorithms for Computing the Transitive Closure of Database Relations", Brighton, Proc. 13th Int'l Conf. on Very Large Data Bases, Sept. 1987, pp. 255-266.

[AS92] Amann B., Scholl M., "Gram: A Graph Data Model and Query Language", ACM ECHT'92, Milan, Dec. 1992, pp. 201-211.

[CGPT93] Cuerva J., Gutiérrez A., Pucheral P., Thévenin J.M., "Specification of Graph Views and Graph Operators", IMPRESS Technical Report Nº W7-005-R75, July 1993.

[CM90] Consens M.P., Mendelzon A.O., "Graphlog: a Visual Formalism for Real Life Recursion", Proc. ACM Symposium on Principles of Database Systems, Nashville, 1990, pp. 404-416.

[CN89] Cruz I.F., Norvell T.S., "Aggregative Closure: An Extension of Transitive Closure", Proc. IEEE 5th. Int'l Conf. on Data Engineering, Los Angeles, Feb. 1989, pp. 384-393.

[Cruz88] Cruz, I.F., "Domains of Application for the G^+ Query Language", Office and Database Systems Research, ed. F.H. Lochovsky, CSRI, Univ. of Toronto, 1988, pp. 141-159.

[DA93] Dar S., Agrawal R., "Extending SQL with Generalized Transitive Closure", IEEE Transactions on Knowledge and Data Engineering, Vol. 5, N° 5, Oct. 1993, pp. 799-812.

[DAJ91] Dar S., Agrawal R., Jagadish H.V., "Optimization of Generalized Transitive Closure Queries", Proc. 7th. Int'l Conf. on Data Engineering, Kobe, April 1991, pp. 345-354.

[Impr93] IMPRESS partners, "Description of the Dispatching Technical Information System", IMPRESS Technical Report Nº W5-001-R1, May 1993.

[IRW93] Ioannidis Y., Ramakrishnan R., Winger L., "Transitive Closure Algorithms Based on Graph Traversal", ACM Transactions on Database Systems, Vol. 18, N° 3, September 1993, pp. 512-576.

[Lu87] Lu H., "New Strategies for Computing the Transitive Closure of Database Relations", Proc. 13th Int'l Conf. on Very Large Data Bases, Brighton, Sept. 1987, pp. 267-274.

[MS90] Mannino M., Shapiro L., "Extensions to Query Languages for Graph Traversal Problems", IEEE Transactions on Knowledge and Data Engineering, Vol. 2, N° 3, Sept. 1990, pp. 353-363.

[PTV90] Pucheral P., Thévenin J.M., Valduriez P., "Efficient Main Memory Data Management Using the DBGraph Storage Model", Proc. 16th Int'l Conf. on Very Large Data Bases, Brisbane, August 1990.

[VB86] Valduriez P., Boral H., "Evaluation of Recursive Queries Using Join Indices", Proc. 1st Int'l Conf. on EDBT, Venice, 1986, pp. 197-208.