A Storage Structure for Graph-Oriented Databases Using an Array of Element Types

Teruhisa Hochin　　Tatsuo Tsuji
Department of Information Science
Fukui University
3-9-1 Bunkyo, Fukui-Shi, Fukui 910 Japan
{ hochin, tsuji }@plum.fuis.fukui-u.ac.jp

Abstract

This paper proposes a storage structure for graph-oriented databases called the flattened separable directory method. In this method, a data representing graph, which is a unit of representing graph, is primarily represented with an array of edge or node types. As every node or edge can be accessed without navigation. the values of nodes and/or edges can be quickly evaluated. Experimental evaluations support this characteristics, and clarify that the performance of inserting data is high, and less storage overhead is needed in the case of the graphs consisting of many node and edge types.

1 Introduction

In the VLSI design process, a lot of diagrams are used[1]. These diagrams are block diagrams, logic ones, and so on. A diagram is composed of a lot of elements connected with each other. For example, a block diagram of a CPU may include the elements for a memory, an ALU, a bus, latches, and so on. There are a lot of connections between the elements, e.g., the connection between a memory and a bus. Therefore, a diagram has very complex structure. Representing and handling complex structure of a diagram are the most important functionalities required in the VLSI design applications. Semantic, object-oriented, and graph-oriented data models have been proposed for these functionalities. Semantic data models have introduced wide varieties of relationships among entities in order to represent rich semantics of data in the applications[2]. Two kinds of relationships, i.e., IS-A and IS-PART-OF relationships, have been introduced in object-oriented data models[3, 4]. As relationships can be drawn with directed edges, data may be represented with a directed graph. Graph-oriented data models[5–13] have been proposed in order to capture data directly as a graph. Most of these models are based on directed labeled graphs. That is, a database consists of nodes and directed edges between them. Each node or edge has a label, which may include a type name and a data value. Accessing a descendant entity/object means navigating a graph under the data models described above.

Several storage structures and models have been proposed for these data models. Linked list methods[14, 15], directory methods[16, 17], and storage models for complex objects[18, 19, 3] have been proposed. In these storage models, searching an element must be based on navigation. An element except for the directly accessible ones can not be accessed until another one having the pointer to it is accessed. Almost all of the nodes and the edges on the path from the root to the node having the required data value must be visited in spite that they are not significant in evaluating a retrieval condition. This will cause the worst retrieval performance. Indexing nodes may not solve this problem because the access paths can not be identified in the design applications. Indexing nodes may result in the worst performance of insertion of data in these applications. It is required that a retrieval condition is evaluated without accessing the unnecessary elements for the purpose of high retrieval performance.

In this paper, a storage structure for graph-oriented databases is proposed. In the data model assumed in this paper, a data representing graph (DRG), which is a directed labeled graph, is a unit in representing data. A collection of DRGs can be captured into a data graph. A data graph is also a directed labeled graph, but it has DRGs as its components. The proposed storage structure represents each DRG in a data graph with an array of edge or node types. Each edge or node is pointed from the entry of the array. Every node or edge can be accessed via that entry without navigation. This may cause the values of nodes and/or edges to be quickly evaluated, and may result in the high performance in evaluating a retrieval condition. The characteristics of the proposed storage structure is experimentally evaluated.

This paper is organized as follows: In Section 2, after the data model assumed is described, conventional storage structures for graph-based databases are surveyed. Section 3 proposes the storage structure for graph-oriented databases. It is called the *flattened separable directory method.* In Section 4, the proposed method is evaluated. Lastly, Section 5 concludes this paper.

0-8186-8409-7/98 $10.00 © 1998 IEEE

2 Data model and survey

2.1 Data model

The data model assumed in this paper is based on the directed labeled graphs. A *data representing graph* (DRG) is a fundamental unit in representing an object. A set of DRGs is represented as a *data graph.*

Definition 1 A *data representing graph* is a septuple $(V, E, L_v, L_e, \phi_v, \phi_e, \phi)$, where V is a set of nodes, E is a set of edges, L_v is a set of labels of nodes, L_e is that of edges, $\phi_v : V \rightarrow L_v$, $\phi_e : E \rightarrow L_e$, $\phi : V \times V \rightarrow E$. A label $l \in L_v \cup L_e$ is a triplet (d_{ID}, N, d), where d_{ID} is an identifier, N is a name, and d is a data and is a tuple of a data type and a value.

A *data graph* is a set of data representing graphs. It is also represented with a septuple $(V, E, L_v, L_e, \phi_v, \phi_e, \phi)$. However, it is the graph composed of one or more data representing graphs. □

Example 1 An organization of computer elements shown in Fig. 1 is a directed graph. The computer elements are nodes of the graph. The direction of data flow between two elements is an arrow of the graph. An example of a DRG of this organization is shown in Fig. 2. In this figure, for example, "Memory:m1" represents that the element (node or edge) name is "Memory" and the data is "m1." The edges in Fig. 2 have similar labels. For example, "MB:mb1" represents that the edge name is "MB" and the data is "mb1." Although the edges of the original diagram in Fig. 1 have no labels, the labels can be put to edges like in Fig. 2 in order that an edge has the information, e.g. the bit length.

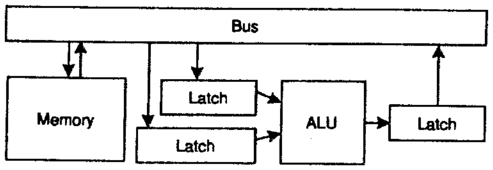

Figure 1. An organization of computer elements

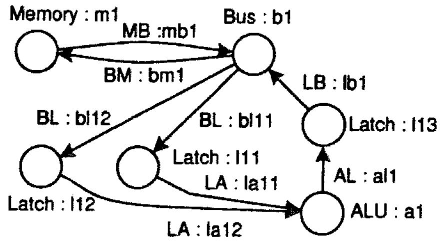

Figure 2. An example of a data representing graph.

Example 2 An example of a data graph is shown in Fig. 3. A data graph is surrounded by a bold line. Its name is "CPU." A data graph can gather DRGs. A DRG is surrounded by a dashed line in order to be able to distinguish one another. There are two DRGs in the data graph. These are alternative organizations of computer elements in this example. The upper DRG is the one shown in Fig. 2. The lower one is another DRG having two Buses.

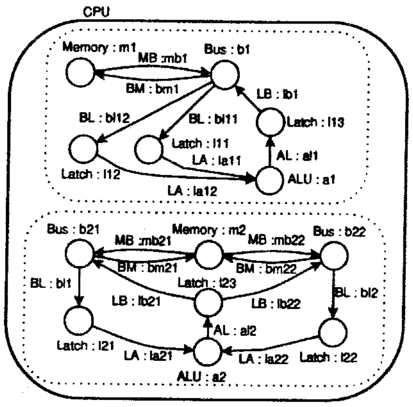

Figure 3. An example of a data graph.

2.2 Survey on the storage structure

Several storage structures and models have been proposed for data models having graph structure. Linked lists have been used for the databases based on semantic data models[14, 15]. A node has the pointers to the next nodes as well as its data value in this method. A required value can be obtained through navigation. Nodes can be visited one by one through the next pointers. As a node has data values which may often be large volume, the number of disk accesses tends to be large.

Directories have also been used for the same kind of databases[16, 17]. A directory represents only the structure of a graph. Each node of a directory has the pointer to its data value as well as the pointers to the next nodes. As a node in a directory does not have any data value, the number of nodes which can be stored in a page is larger than that in the linked list methods. The larger number of the nodes can be obtained with one disc access than in the linked list method. Therefore, the number of disc accesses in the directory method tends to be less than in the linked list method.

Several storage models including the normalized storage model[3] and the decomposed one[18] have been proposed for complex objects in the context of object-oriented databases[19]. These models except for the direct storage model decompose a complex object into smaller chunks of data. Decomposition levels vary according to the storage models. For example, the decomposed storage model decomposes a complex object into a set of tuples of an object identifier (OID), and a data value or an OID of a child object. The normalized storage model decomposes a complex object into a set of n-tuples of atomic values and/or object identifiers of child objects. On the other hand, the direct storage model does not decompose a complex object. The decomposed storage model can easily be extended for edges to have labels. However, this storage model is not good for navigation in order to evaluate the retrieval condition.

As we have seen, searching an element must be based on navigation through the storage models described here. After one element is evaluated, the next element directly connected to it is then accessed and evaluated. In the case that almost all nodes of a graph can not be directly accessed, and the distance of a graph is long, the evaluation cost of a retrieval condition may become very high, where a distance of a graph is the largest number of the numbers of the edges, each of which is the smallest one of those of one or more paths from a node to another one in a graph. For example, consider the tree, only whose root can directly be accessed. In evaluating a retrieval condition, a leaf node, which is far from the root, may have to be visited. All of the nodes and edges on the path from the root to that node must be visited in spite that they are not significant in evaluating a retrieval condition. Indexing nodes may not solve this problem because the access paths can not be identified in the design applications. Indexing nodes may result in the worse performance of insertion of data. For example, if a lot of indexes relate to the value of a node type, these indexes have to be updated when a node of this type is inserted, modified, or deleted. Therefore, it is required that a retrieval condition is evaluated without accessing the unnecessary elements for the purpose of high retrieval performance.

3 Flattened separable directory method

The flattened separable directory method (FSDM) uses an array of node or edge types of a DRG. This array is called a *primary array* (PA). The data format of a *PA* is shown in Fig. 4. A PA is an array with a number of array entries. An array entry is for a node or edge type. An array entry is represented with a tuple of a number k, which represents a number of instances, and a pointer p. If k is equal to zero, p points to no instances. If k is equal to one, p points directly to an instance. If k is more than one, p points to the root of a B^+ tree, which manages more than one instance.

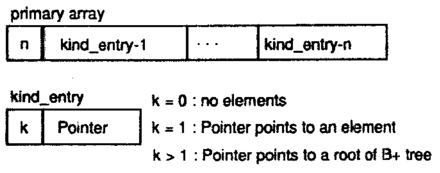

Figure 4. Data format of primary array.

There are two kinds of PA: edge PAs and node PAs. Every array entry of an edge PA is for edge types, while that of a node PA is for node types. Figure 5 shows an example of an edge PA. This figure represents the data structure of the DRG in Fig. 2.

The primary array represents the whole structure of a DRG rather than the parts of it. As the whole structure is represented with a single data structure, the

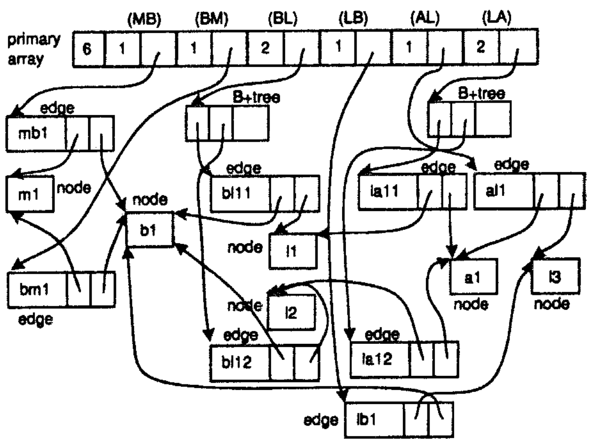

Figure 5. An example of the proposed storage structure.

proposed storage structure belongs to the directory method. However, a graph structure is represented with a flattened format. That is, an array of edge or node types is used in representing a graph structure. This is the distinguished difference between the ordinary directory method[16, 17] and the proposed one. This enables every node or edge to be able to be accessed directly rather than navigationally.

Nodes and edges do not have to be stored in continuous area in FSDM. These can be stored as separate data chunks. Therefore, a large continuous space is not required to store a whole of a DRG. In this sense, FSDM resembles the decomposed storage model[18] and the methods representing nodes as relations. If primary arrays were not adopted, and nodes and edges were separately stored into files, then FSDM becomes the decomposed storage model. In this case, searching a node must be based on navigation. Performance of evaluating retrieval conditions becomes drastically worse according to the position of the node evaluated (See Section 4). It is the primary array that prevent the performance of evaluating retrieval conditions from being worse. Because of the primary array, it is possible that every node or edge in a DRG can directly be accessed.

4 Evaluation

4.1 Evaluation method

FSDM is evaluated on the performance and the storage overhead by comparing with the linked list method (LLM), the directory method (DIM), and the decomposed storage model (DSM), which are extended in order that an edge can have a label.

In FSDM, edge PAs are used. In LLM, variable length arrays are used in representing the element types and the elements (See Fig. 6 (a)). In DIM, nodes are stored in a former part, and edges in a latter part of a directory. A node or edge is accessed by using an offset in a directory. The value of a node or edge is accessed through a pointer (See Fig. 6 (b)). In DSM, an edge is represented with a triplet of pointers to the initial node, to the terminal node, and to its value. These triplets are managed by using B^+ tree. These for an edge type are stored in the file separated from the one storing values.

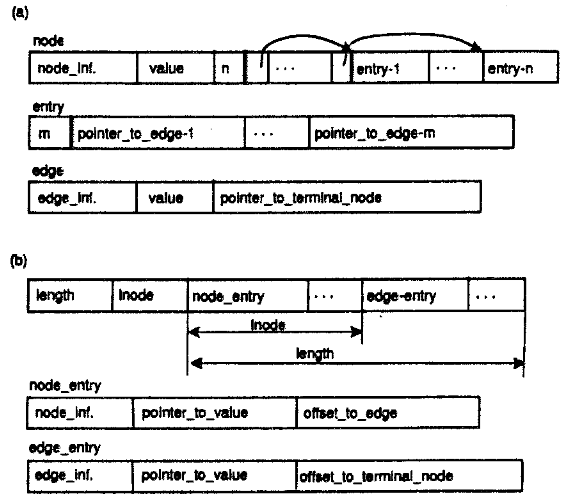

Figure 6. Data formats of the linked list method (a) and the directory method (b).

Several assumptions are taken as follows. A node (edge) type has only one node (edge) in order not to be influenced by a set of elements. A starting node is decided in advance. There is no isolated nodes. A graph is a line in order to make the evaluation simple. An element is addressed through an OID. An OID is transformed to a tuple (page#, slot#) by using a transformation table, which is on memory. Buffers are replaced with LRU scheme.

Performance is measured on the Ultimal workstation (Axil, Solaris 2.5.1, 256MB memory). The page and the buffer sizes are 4K bytes. The number of buffers is 100 for excluding the effect of buffering pages.

4.2 Experimental results

Retrieval and insertion performance is experimentally evaluated. Performance in evaluating a retrieval condition and that in returning data to a user are evaluated as the retrieval performance. The storage overhead is also evaluated.

4.2.1 Performance of evaluating retrieval condition

First, the effect of the retrieval condition is evaluated. A retrieval condition is assumed to be the conjunction of predicates, and only the predicates for equality test are used. An example of a retrieval condition is the form "$N_0 = 1$ AND $\cdots$ AND $N_m = 1$," where N_i denotes the ith node. In this evaluation, the number of predicates in a retrieval condition and the position of the farthest node are varied, where the farthest node means the node that is the farthest from the directly accessed node. The position of the farthest node corresponds to the distance between the directly accessed node and that node. The retrieval condition used in this evaluation fails at the farthest node. For example, if N_i is an ith node, and every node has a value 1, a retrieval condition is the form "$N_0 = 1$ AND $N_1 = 1 \cdots N_8 = 1$ AND $N_9 = 0$." Every DRG has eleven nodes and ten edges. The number of DRGs is 100. Figure 7 shows the experimental result. The values in this figure are those averaged of 100 times of experiments. In FSDM, the time in evaluating a retrieval condition is independent of the farthest node position. It depends only on the number of predicates in a retrieval condition. On the other hand, it depends on the farthest node position in the other methods, i.e., DSM, LLM, and DIM, because navigation is inevitable to evaluate a retrieval condition in these methods.

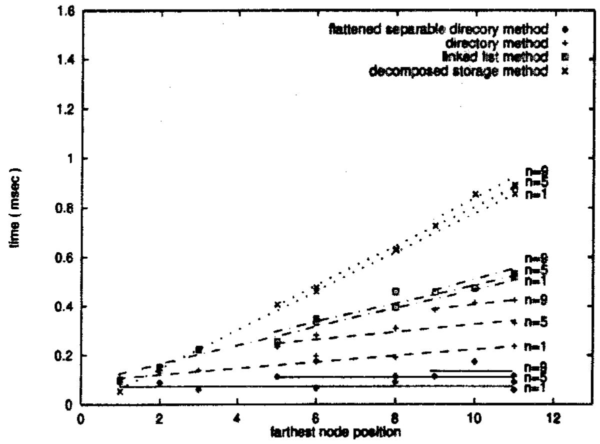

Figure 7. Effect of the farthest node position and the number of predicates on the performance of retrieval condition evaluation. n : the number of predicates.

Next experiment evaluates the effect of the number of node and edge types. The number is varied from 21 to 99. A retrieval condition consists of only one predicate. The number of DRGs is also 100. The result is shown in Fig. 8. The time in evaluating a retrieval condition increases according to the number of node and edge types in all of the methods. The characteristic lines have the same incline for each method.

The effect of the number of DRGs is then evaluated. The number of DRGs is varied from 100 to 10000. Every DRG has eleven nodes and ten edges. A retrieval condition has only one predicate on the node whose position is 11 (fixed). Figure 9 shows this result. The evaluation time is independent of the number of DRGs.

4.2.2 Performance of returning DRGs

Next, performance in returning a DRG is evaluated. The number of node and edge types is varied. No retrieval conditions are specified. The number of DRGs is 100. The result of this experiment is shown in Fig. 10. The time in returning a DRG depends on the number of elements in a DRG.

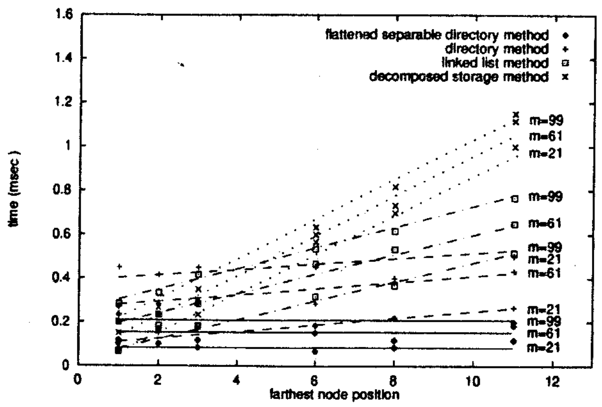

Figure 8. Effect of the number of node and edge types on the performance of retrieval condition evaluation. m : the number of node and edge types.

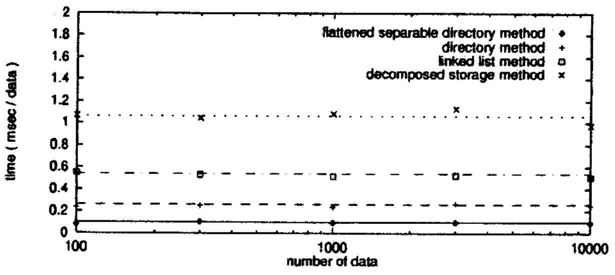

Figure 9. Effect of the number of instances on the performance of retrieval condition evaluation.

4.2.3 Performance of insertion

Next, the performance in inserting data is evaluated. The number of node and edge types is varied. The number of DRGs is 100. Figure 11 shows this result. FSDM has better insertion performance than the other methods.

4.2.4 Storage overhead

Lastly, the storage overhead is evaluated. The number of node and edge types is varied. The overheads per one element are calculated. This result is shown in Fig. 12. In the case that a DRG has a large number of elements, FSDM and DSM need less storage overhead than the other methods. FSDM and DIM decreases the storage overhead according to the number of node and edge types. This is because there exists the data structure per a DRG: a PA for FSDM and a directory for DIM. The overhead per an element where a DRG has a large number of elements is less than that where it has a small number of elements.

4.3 Consideration

The performance in evaluating a retrieval condition is independent of the farthest position of the node appearing in a retrieval condition in FSDM. It depends only on the number of predicates in a retrieval condition. This makes it possible to estimate the time of a retrieval condition evaluation. This is the good characteristics for the retrieval condition evaluation. The performance in returning and inserting data of FSDM is the best among the methods experimented.

DSM has good performance in evaluating a retrieval condition consisting the predicates on the node directly accessible because the evaluating targets are the decomposed values, and they may be stored in the small number of pages. However, The farther the position of the node evaluated is, the worse the evaluation performance becomes drastically. This result agrees with the characteristics of DSM[19].

In the VLSI design applications, a diagram may have a lot of nodes and edges. This is mainly categorized into two cases. First is that a diagram has many graphs, each of which has a few nodes and edges. Second is that a diagram has one or several graphs, each of which has many nodes and edges. In the first case, indexing graphs will be required in order to process queries efficiently. This kind of index may be created by using the values of nodes and/or edges.

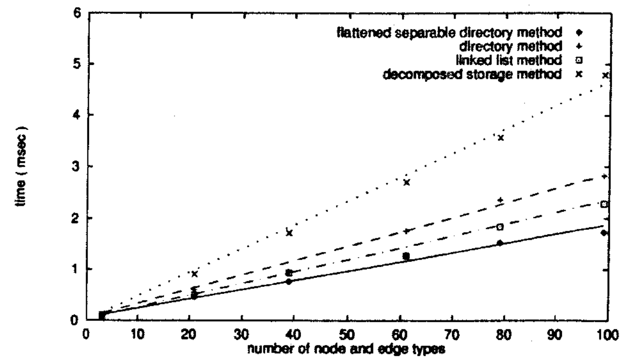

Figure 10. Effect of the number of node and edge types on the performance of returning data.

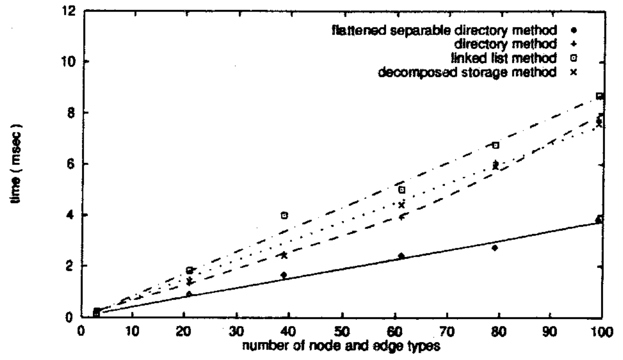

Figure 11. Effect of the number of node and edge types on the performance of inserting data.

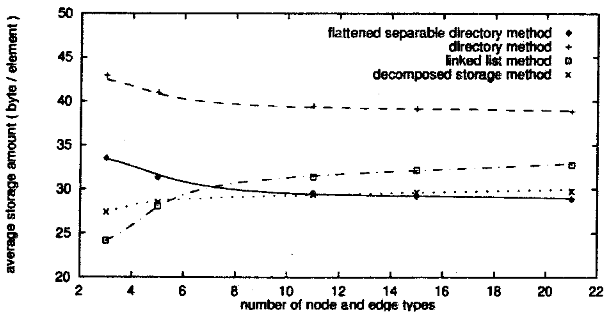

Figure 12. Effect of the number of node and edge types on the storage overhead.

When evaluating a retrieval condition on the elements in a DRG is not needed like in this case, the proposed storage structure may not contribute the performance. The second case is further categorized into two subcategories. The first is that a graph has many types of nodes and edges, and each type has a few instances. For this case, the proposed storage structure is effective as shown through the experiments. The second is that a graph has a few types of nodes and edges, whereas there are many instances of nodes and edges. A tree is an example of this case. Although a tree may have only one type for nodes, and only one for edges, it may have many nodes and edges. The contribution of the proposed storage structure to this case is not clear. We have a plan to make it clear through a benchmark[20].

5 Concluding remarks

This paper proposed the storage structure for graph-oriented databases called the flattened separable directory method. In this method, a data representing graph, which is a unit of representing graph, is primarily represented with an array of edge or node types. As every node or edge can be accessed without navigation, evaluating the values of nodes and/or edges can be fast. Furthermore, the inserting performance is high. These characteristics are revealed by the experiments. The storage overhead is less than in the other methods evaluated when a data representing graph consists of many node and edge types.

The graph structure evaluated is a line. The experiment for more complex structures through a benchmark, e.g., the 007 benchmark[20], is a subject for a future research. Although the primary array may be less in size than the directory, the primary array must be in a continuous space. Relaxing this restriction is another subject for a future research. In this paper, the data model assumed is based on a simple directed labeled graph. A study on the storage structure for the databases based on the data models introducing the concepts of hypergraphs and recursive graphs[11, 12, 13] is another future work.

Acknowledgement

This work is supported in part by the Ministry of Education, Science, Sports and Culture of Japan under a Grant-in-Aid for Scientific Research No. 09780258.

References

[1] S. Muroga. *VLSI SYSTEM DESIGN*, John Wiley & Sons, Inc., 1982.

[2] R. Hull, and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Survey*, 19(3):201-260, 1987.

[3] W. Kim, *et al.* Architecture of the ORION Next-Generation Database System. *IEEE Trans. Know. and Data Eng.*, 2(1):109-124, 1990.

[4] S. Y. W. Su, *et al.* Association Algebra: A Mathematical Foundation for Object-Oriented Databases. *IEEE Trans. on Know. and Data Eng.*, 5(5):775-798, 1994.

[5] M. Gyssens, *et al.* A Graph-Oriented Object Database Model. *IEEE Trans. on Know. and Data Eng.*, 6(4): 572-586, 1994.

[6] M. P. Consens, *et al.* GraphLog: a Visual Formalism for Real Life Recursion. *Proc. of 9th ACM PODS*, pp. 404-416, 1990.

[7] H. S. Kunii. *Graph Data Model and Its Data Language*. Springer-Verlag, 1990.

[8] A. L. Rosenberg. Addressable Data Graphs. *J. ACM*, 19(2):309-340, 1972.

[9] D. Lucarella, *et al.* A Graph-Oriented Data Model. *Proc. of DEXA '96*, pp. 197-206, 1996.

[10] R. H. Guting. GraphDB : Modeling and Querying Graphs in Databases. *Proc. of the 20th VLDB Conf.*, pp. 297-308, 1994.

[11] M. Levene, and G. Loizou. A Graph-Based Data Model and its Ramification. *IEEE Trans. on Know. and Data Eng.*, 7(5):809-823, 1995.

[12] H. Boley. Directed Recursive Labelnode Hypergraphs: A New Representation-Language. *Artificial Intelligence*, 9:49-85, 1977.

[13] Y. Fujiwara, *et al.* Representation Model for relativity of Concepts. *Int'l Forum on Information and Documentation*, 20(1):22-30, 1995.

[14] A. Chan, *et al.* Storage and access structures to support a semantic data model. *Proc. of 8th VLDB*, pp.122-130, 1982.

[15] D. Jagannathan, *et al.* SIM: A database system based on the semantic data model. *Proc. of ACM SIGMOD 1988*, pp.46-55, 1988.

[16] H.-J. Scheck, *et al.* The DASDBS Project. *IEEE Trans. Know. and Data Eng.*, 2(1):25-43, 1990.

[17] Härder, T. *et al.* PRIMA - a DBMS Prototype Supporting Engineering Applications. *Proc. of 13th VLDB*, pp.433-442, 1987.

[18] G. P. Copeland, and S. N. Khoshafian. A Decomposition Storage Model. *Proc. of ACM SIGMOD 1985*, pp.268-279, 1985.

[19] P. Valduriez, *et al.* Implementation Techniques of Complex Objects. *Proc. of 12th VLDB*, pp.101-110, 1986.

[20] M. J. Carey, *et al.* The 007 Benchmark. *Proc. of ACM SIGMOD 1993*, pp.12-21, 1993.