データベース・システム 58－5
（1987. 3. 16）

履歴データベースにおける更新機能とビュー機能について

徐 海燕　　上林 彌彦
九州大学 工学部

データベースの応用分野の広がりに伴って、バージョン管理が非常に重要となってきた。バージョン間の一般的な履歴関連を表現するためには直線や木構造ではなく、有向非巡回グラフが必要である。しかし、バージョンの分割や併合操作を導入することにより多くの新しい問題が生じる。本稿ではその中の次のような問題に対して検討する。
(1)過去のデータの訂正やバージョングラフの変更操作
(2)バージョングラフに対するビューの定義問題及びビュー上の変更操作問題
(3)有向非巡回バージョングラフにより生じる並行処理問題
現在、履歴情報を扱うための種々のモデルが提案されているが、最新のデータのみが変更でき、過去のデータや履歴グラフを変更することは許されていない。しかし、誤りの訂正や設計の変更のためにこの機能が必要であると共に、バージョングラフを変更する必要もある。また、中間的なバージョンが見えない利用者に対しては、推移的なバージョン関連をビューに提供しなければならないので、関係データベースのビューとは異なる機能が必要である。そして、バージョンの分割や併合機能が必要なので、直列可能性、即ち、並行処理メカニズムの正当性の規準を修正しなければならない。

Update and View Operations for Historical Databases

Hai-yan XU and Yahiko KAMBAYASHI

Department of Computer Science and Communication Engineering

Kyushu University 36, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812, Japan

Due to the increasing importance of database applications to CAD systems, it becomes very important to handle versions in database systems. In order to handle general cases, we need to use directed acyclic graphs to represent historical relationships among versions instead of linear sequences or tree structures. By permitting splitting and merging of versions, we have new problems to be solved. In this paper the following problems are discussed:
(1) Update of past data as well as update of version history graphs.
(2) View definition problem and view update operations on version history graph.
(3) Concurrency control problems when versions are presented by acyclic directed graphs.
In conventional proposal, past versions are assumed to be not changeable, but we need such operations due to errors or some design changes. Functions to change version graph structure are also necessary. Since we need to generate a version graph to be seen by users who cannot use intermediate versions, views of version graphs are different from relational database views. Due to the existence of splitting and merging operations, serializability condition used for the correctness of concurrency control mechanisms should be modified.

1. Introduction

Due to the increase of application areas of database systems, version management is getting very important. In order to manage various versions, there are problems such as how to represent and manage relationships among versions, which are not encountered in conventional database systems. Version management approaches proposed so far do not provide enough functions to handle such problems. In this paper we discuss the following important problems of version management:

(1) Updating of past data as well as updating of the version history graph.
(2) View operations on the version graph.
(3) Concurrency control problem when versions are presented by directed acyclic graphs.

In order to manage data of different states in general cases, first historical relationships among them must be recorded by directed acyclic version graphs. Second the following flexible version operations must be provided: (1) Modify: When there are mistakes in a version (past or current), it needs to be modified. Furthermore, since mistakes in a past version may be propagated to its successors, it is necessary to modify several versions at a time. (2) Delete: If a version is not wanted any more, the delete command is very useful. When a version is deleted, version graph must be modified, too. (3) Merge: Since versions record the different states of objects, in some cases, it is better to combine several versions into a global one. (4) Split: In some cases it may be necessary to split a version to several ones, which is an inverse operation of merge.

Since a user only can see a part of versions, for each user the view appropriated to him should be provided. However, since versions are related to each other, the problem of how to handle relationships among them must be considered. In relational databases, since data are handled independently, this problem is not encountered. Furthermore, for view there are view update problems which are very critical issues. Therefore view update problems of version graphs must be handled, too.

When versions are presented by directed acyclic graphs, serializability, the correctness of conventional concurrency control must be modified. Serializability severs quite well for conventional data-processing applications that update is taken place in linear order and the sequence of action is important. However, in general case serializable consistency is not a suitable criterion for concurrency control. For example, transaction T_1 updated version v_1 of object A to v_2 (of same object) and then committed. After T_1 being committed, transaction T_2 updates v_1 and v_2 to a new version v_3 (of same object A) by merging v_1 and v_2 (Fig.1.1). It is clear that this execution is non-serializable on the object level, for v_1, v_2 and v_3 are only the different states of an identical object. This kind of execution schedule, however, is necessary in design environment and other applications.

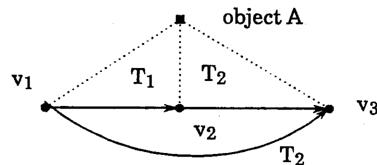

Fig.1.1 A non-serializable execution schedule

2. Versions

In this section, we discuss the version representation and version operations which treat the following cases: (1) correction of historical data, (2) merge and split of versions, (3) modification of several versions at a time, (4) use of a version of an object to create other version(s) of different objects.

Generally version concept is introduced to represent the following situations [CHOU86, BATK85, KATL84, KATZ85, KATZ86, KLAH86, KRUS84, LOCK85]:

(1) Alternative design or parallel design for an object. For example, in software development, a module may be coded in several ways using a language at a time (alternative design) or can be coded in several languages simultaneously (parallel design).

(2) Historical data. Generally states of an object changes over time and they are recorded by versions.

In this paper, similar to [CHOK86, KLAH86, BATK85], we use version concept to represent a state of an object and the object is only the unit which comprises all versions.

[Version Representation]

(1) Version graph G(V, E): Historical relationships among versions are represented by directed acyclic version history graphs (for example, Fig.2.1(a)). Nodes V represent versions and edges E show the derivative or historical relationships among versions (historical relationship (v_i, v_j) is created when version v_j is derived from version v_i or the results of update of version v_i is in version v_j).

(2) Class: Each class corresponds to a required design method (for example, a required programming language). By the concept of class, alternative versions or parallel versions can be known in the following way:

(a) Parallel versions are those which belong to different classes.

(b) Alternative versions are those which are valid at same time and within same class.

[Version Update Operations]

(1) Modification of a version. When a version is modified, result is recorded in a new version and updated version only became invalid. Furthermore if a past version is modified, it is necessary to add edges. Such cases are explained by Fig.2.1: If version v_3 is modified and its result is recorded in version v_6 and v_6 represents v_3, then doted lines which shown in Fig.2.1(c) are required. For this purpose, we introduce a new kind of relationship: a version is a "parent-in-law" of another one. Formally, suppose version v_i of version graph G(V, E) is modified into v_j and v_j represents v_i, then if $v_k \in V$ and (v_k, v_i)$\in$E ((v_i, v_k)$\in$E), then v_k is defined as a "parent-in-law" of v_i (v_i is defined as a "parent-in-law" of v_k). Here notion (v_k, v_i)$\in$E represents v_k is a parent of v_i, rather a "parent-in-law" of v_i. For a "parent-in-law" v_t of v_i (v_i is a "parent-in-law" of v_t), from transitive relationships the information that v_t is a "parent-in-law" of result version v_j (v_j is a "parent-in-law" of v_t) can be known.

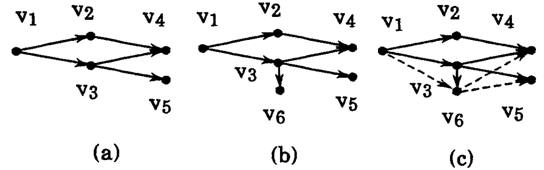

Fig.2.1 An example of modification of a intermediate version

We point out that the relationship "parent-in-law" is also very important for the purpose to provide appropriate view for users. Suppose a user A only can access versions v_1, v_4, v_5, v_6 for the case of Fig.2.1(b), then the version graph in his view should be as {{v_1, v_4, v_5, v_6}, {(v_1, v_6), (v_6, v_4), (v_6, v_5)}}. When the relationship "parent-in-law" is introduced, as will be described in next section, this kind of view can be provided.

(2) Modification of several versions at one time. Since mistakes in a past version may be propagated into its successive versions, it is necessary to update them together. Suppose version $v_1,\cdots, v_n$ of version graph G(V, E) are corrected together and results are in $v'_1,\cdots, v'_n$ each other, then if (v_i, v_j)$\in$E, then v'_i is defined as a "parent-in-law" of v'_j; and if $v_k \in V$ and (v_k, v_i)$\in$E ((v_i, v_k)$\in$E) but $v_k \notin \{v_1,\cdots, v_n\}$, then v_k is defined as a "parent-in-law" of v_i (v_i is defined as a "parent-in-law" of v_k) (Fig.2.2).

(3) Derivation of a new version from an existing one. In this case, a new version is created without changing the valid time of old one.

(4) Deletion of an existing version. When a version is deleted, in principal its successive versions should be connected to its predecessors directly. Since algorithm in section 3.1 will solve the

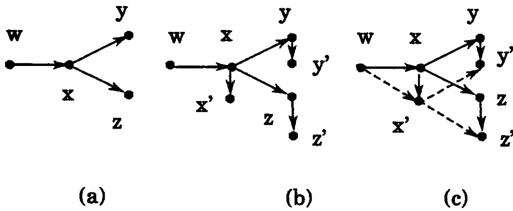

Fig.2.2 An example of modification of several versions together

problem of how to distinguish predecessor and "parent-in-law", we do not discussed it here.

(5) Merge of versions. In this case, a new version can be obtained while some merged versions remain valid and others became invalid (see Example 2.1)

(6) Split of a version. New versions can be obtained and the split version can remain valid or become invalid.

(7) Updating of metadata of version. It is necessary to permit users to change metadata such as creator, valid time, version name or version identification number, and class name.

(8) Updating of relationships. Although historical relationships record the development of versions, users may want to have rights to change them. In the version representation of this paper, it is necessary to let users change kinds of relationships: from a "parent-in-law" to a general predecessor. For example, in the case of Fig.2.2 if only versions x', y', and z' are remained, users may want that x' becomes a predecessor of versions y' and z' directly, rather than a "parent-in-law".

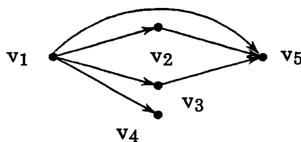

Fig.2.3 An example of merging of versions

Example 2.1: Suppose three versions v_2, v_3, v_4 are derived from version v_1 and now a user wants to merge versions v_1, v_2, v_3 into new version v_5. Furthermore, the user wants to keep v_1 valid and make v_2, v_3 invalid (Fig.2.3). This is the case when a user wants to get a design by integration of versions v_2, v_3 and only a part of v_1 which is not included in v_2 and v_3 is used.

Furthermore, derivation or update operation can be performed not only within the versions of the same object, but also on the versions of different objects. Since in the cases of creating of a new object or merging of existing objects to a new one, existing object's version(s) may be used to create the versions of new object. That is, there may be edges between versions belonging to different objects.

3. View Functions

In this section, the following two problems are discussed.: (1) how to provide views, (2) view update problems.

3.1 Definition of View

For a user, generally only a part of versions can be accessed. For this purpose, there is the problem of providing appropriate views to different users. For this problem, first, users should be allowed to select relationships of required kinds for a given set of versions, since a user may only need a part of historical relationships. For example, a user may not select: (1) relationships among versions of different objects, (2) "parent-in-law" relationships.

Second, it is inappropriate to define the relationship set of view as a subset of base schema. The problem will be shown in the following example.

Example 3.1: A history version graph of an object is shown in Fig.3.1 (a). Suppose user A selects only versions v_1 and v_3 into his view. Since versions represent different states of objects, a user may not be permitted to see some intermediate versions. If the history version graph Fig.3.1(b) is shown for user A, then the information that version v_3 is a successor of v_1 is lost. For this case, history version graph Fig.3.1(c) is an appropriate one. That is, to calculate history relationship transitively: if v_2 is a successor of v_1 and v_3 is a successor of v_2, then for the user who only selects versions v_1 and v_3 into his view, v_3 is regarded as a successor of v_1 in that view.

However, as described above there are two kinds of edges in version graph: (1) historical or derivative

relationship, (2) a version is set as a "parent-in-law" of another one. Therefore we need to consider how to treat this two kinds of edges. (a) If there is a directed path connecting v_i and v_j and every edge on this path is of kind (1), then (v_i, v_j) can be defined as kind (1) without any problems. There is, however, a difficult problem in the following case. (b) If there is an edge of kind (2) on the path connecting v_i and v_j and there is not a path which satisfies condition (a). For this case we think what is important is to keep the following thing true: If in version graph G(V', E') of view, a version $v_1 \in V'$ is shown as a one gotten by merging of two versions v_2 and v_3, then v_2 or v_3 must not be only an "ancestor-in-law" of v_1. Therefore we define the edge of case (b) as kind (2). In the following an example and an algorithm are shown.

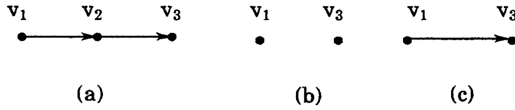

Fig.3.1 View of history version graph

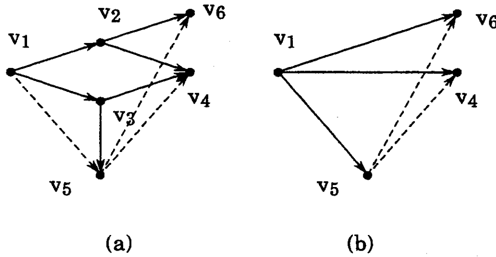

Fig.3.2 An example of view of version graph

Example 3.2: Suppose the version history graph kept in the base schema is shown in Fig.3.2(a) and a user B only selects the versions v_1, v_4, v_5, v_6 into his view, then version graph in his view is shown in Fig.3.2(b).

[Algorithm] (we distinguish two kinds of edges by representing of edge (v_i, v_j) as $(v_i, v_j)^i$ $(i=1$ or $2)$)

$V'=\{$Versions that user has right to access$\}$; $E'=\phi$;
$S=V'$;
Do while $S\neq\phi$
  Let $s\in S$; $S=S-\{s\}$; $T=\{(s, x)^i (i=1 \text{ or } 2) | x\in V \text{ and } (s,x)^i\in E\}$;
  Do while $T\neq\phi$
    Let $(s, t)^j\in T$;
    If $t\in V'$ then begin $E'=E'\cup(s,t)^j$;
        if $((s, t)^1\in E'$ and $(s, t)^2\in E')$
        then $E'=E'-(s, t)^2$;
        $T=T-\{(s, t)^j\}$;
        end;
    $T'=T-\{(s, x)^i | \neg\exists(y\in V)(x, y)^k\in E\}$;
    $T=\{(s, x)^k (k = \max(i, j)) | \exists((s, y)^i\in T') \text{ and } \exists((y, x)^j\in E\}$;
  end;
end;

3.2 View Update Problems

In general, view update problems are very critical issues and only if a view is a subset of base schema, view update operations can be implemented. Although the version set in view is a subset of base schema, a view defined here is not a subset of base schema due to the existence of transitive edges. Addition of such edges will not cause difficult problems as described below.

(1) The problem that the relationships in view are calculated transitively from base schema. Since there is not one-to-one mapping from view and its base schema, operations on view generally cannot be mapped into the operations on base schema. However, this is not a critical problem. First, generally the information on relationship between versions v_i and v_j is as follows: kinds of relationship, valid time, creator, ID of version v_i, ID of version v_j. For historical relationship, however, valid time and creator are unnecessary, because valid time and creator of it are as same as these of version v_j. Furthermore, the possible update operation on a historical relationship is only to change the kind of it. Moreover, when a user cannot see some intermediate states, the method to calculate relationship transitively is required. That is, when the method to calculate relationship transitively is used, it means that the user does not have rights to access some intermediate versions. Therefore it is natural that the user cannot update the kinds of relationships which are calculated transitively.

(2) Updating operations on versions: Since a version set in view is a subset of its base schema,

operations on the version set in view can be completely mapped into the operations on the version set in base schema. Although by such operations relationships among versions on base schema may be changed, by the algorithm described in Section 3.1 relationships on view can be obtained. Therefore it seems that update operations on the version set are carried out on view directly.

Some examples of view update are shown as follows.

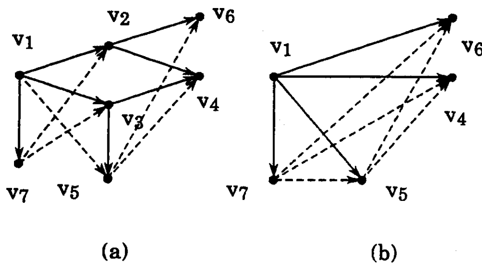

Fig.3.3 View update operation: modification of a intermediate version

Example 3.3: In the case of Fig. 3.2 if the user updates version v_1 to a new version v_7 and v_7 represents v_1, then the update which is carried out on the base schema is shown in Fig.3.3(a). By mappings between view and base schema, the user can get his updated view (Fig.3.3(b)). For the user it seems that update is carried out on his view directly.

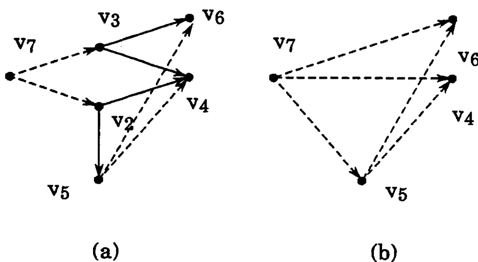

Fig.3.4 View update operation: deletion of a version

Example 3.4: If base schema and view are shown in Fig.3.4 and the user issues delete v_1 command to his view, then base schema becomes the one shown as Fig.3.4(a) and the view changes into the one of Fig.3.4(b).

4. Concurrency Control

For traditional transaction schema the notion of correctness for an execution of a set of transactions is serializability. Serializability requires that an execution should be equivalent to one that would have occurred without any concurrency in the system. In other words, users (transactions) must not see the effects of concurrent execution. However serializability is restricted by the following reasons:

(1) In real world, changes of states of an object are not only in linear order. For example, as described in Section 2, in design environment, there may be alternative or concurrent states for a design object.

(2) In some applications such as design environment, the primary concern is the correctness of the design rather than the sequence of steps that led to the design. In a banking environment that motivated the traditional schema, consistency involves not only the final state, but also the sequence of steps through which the final state has reached. In a design environment, consistency only involves final results and serializability is not the necessary condition for consistency.

In order to relax the restriction of serializability for general cases, various kinds of locks which always have the following two meanings are required:

(1) The rights of owner of lock
(2) The rights of other transactions (users)

For rights, as described in Section 2, we have to consider the following five kinds:

A retrieval of a version (versions)
B update of metadata of a version
C change of relationship
D derivation of a new version
E modification of a version

[Property of the above rights]:
$A<B<E$, $A<C<E$, $A<D<E$

It is clear that for any given version the right E includes the others rights and the right A is the weaker one. The rights B, C, and D, however, cannot be compared with each other. In order to make different transactions (users) use above five kinds of operations flexibly, the following twenty-five basic lock modes are required: Ee, Ed, Eb, Ec, Ea, E, De, ..., Ae, Aa. Here the capital character represents the right of owner and the small character describes the right of others. For example, lock Ee represents both owner and other transactions have the right E. The compatibility relationships of these locks imposed by different transactions are shown in Fig.3.5 (the locks in row are set earlier than the locks in the columns). For example, the second column of Fig.3.5 represents that if a user sets Ed lock then the other users only can do operations of kind D (since the right D is stronger than A, the other users also can do operations of kind A).

By using these locks and version level's authorization mechanism, various kinds of concurrent work can be carried out. Here, serializability is a special case satisfying the following conditions: (1) Only locks of E and Aa can be used. They correspond to write-lock and read-lock respectively. (2) Users only have right to access current versions. (3) Merge, split, and derive operations cannot be used.

Another special case is that only locks of Ee, De, Ce, Be, Ae can be used. That is, concurrent work can be done arbitrarily. Between these two special cases, many kinds of concurrent work can be carried out.

Locks of Ee, De, Ce, Be, Ae are unnecessary, because they do not restrict any concurrent work. Furthermore, in practice twenty kinds of locks may confuse users. In such cases many simpler methods can be used. For example, only to provide rights A, C, and E, if a user wants to update metadata of a version, he must have right to modify that version.

5. Concluding Remarks

In this paper we discussed three basic and important problems of version management. Since versions record different states of objects, it is necessary to permit users to operate them flexibly. To the

	Ee	Ed	Eb	Ec	Ea	E	De	Dd	Db	Dc	Da	D	Ce	Cd	Cc	Cb	Ca	C	Be	Bd	Bc	Bb	Ba	B	Ae	Aa
Ee	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
Ed	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
Eb	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
Ec	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
Ea	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
E	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
De	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	X
Db	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	X
Dc	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	X
Da	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	X
D	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	O	X	X	X	X	O	X
Ce	O	X	X	O	X	X	O	X	X	O	X	X	O	X	O	X	X	X	O	X	O	X	X	X	O	X
Cd	O	X	X	O	X	X	O	X	X	O	X	X	O	X	O	X	X	X	O	X	O	X	X	X	O	X
Cc	O	X	X	O	X	X	O	X	X	O	X	X	O	X	O	X	X	X	O	X	O	X	X	X	O	X
Cb	O	X	X	O	X	X	O	X	X	O	X	X	O	X	O	X	X	X	O	X	O	X	X	X	O	X
Ca	O	X	X	O	X	X	O	X	X	O	X	X	O	X	O	X	X	X	O	X	O	X	X	X	O	X
C	O	X	X	O	X	X	O	X	X	O	X	X	O	X	O	X	X	X	O	X	O	X	X	X	O	X
Be	O	X	O	X	X	X	O	X	O	X	X	X	O	X	X	O	X	X	O	X	X	O	X	X	O	X
Bd	O	X	O	X	X	X	O	X	O	X	X	X	O	X	X	O	X	X	O	X	X	O	X	X	O	X
Bc	O	X	O	X	X	X	O	X	O	X	X	X	O	X	X	O	X	X	O	X	X	O	X	X	O	X
Bb	O	X	O	X	X	X	O	X	O	X	X	X	O	X	X	O	X	X	O	X	X	O	X	X	O	X
Ba	O	X	O	X	X	X	O	X	O	X	X	X	O	X	X	O	X	X	O	X	X	O	X	X	O	X
B	O	X	O	X	X	X	O	X	O	X	X	X	O	X	X	O	X	X	O	X	X	O	X	X	O	X
Ae	O	X	X	X	X	X	O	X	O	X	X	X	O	X	X	X	X	X	O	X	X	X	X	X	O	X
Aa	O	O	O	O	O	X	O	O	O	O	O	X	O	O	O	O	O	X	O	O	O	O	O	X	O	O

Fig.3.5 Compatibility relationships of locks

contrary, since huge numbers of versions may confuse users, it is also necessary to manage different versions in a uniform way.

REFERENCES

[BATK85] D.S.Batory and W.Kim, "Modelling Concepts for VLSI CAD Objects," ACM Tran. Database Syst. 10, Setp. 1985, pp.322-346.

[CHOK86] H-T.Chou and W.Kim, "A Unifying Framework for Version Control in a CAD Environment," Proc. of the 12nd Inter. Conf. on Very Large Data Bases, Kyoto, Aug. 1986, pp.336-344.

[DALW84] P.Dadam, V.Lum, and H-D.Werner, "Integration of Time Version into a Relation Database System," 10th International Conference on VLDB, Aug. 1984, pp. 27-31.

[GINT86] S.Ginsburg and K.Tanaka, "Computation-Tuple Sequences and Object Histories," ACM Tran. Database Syst. 11, No.2 June 1986 pp.186-212.

[HASK82] R.L.Haskin, "On Extending the Functions of Relational Database System," International Conference on Management of Data, June 2-4, Orlando, Florida, 1982, pp. 207-212.

[KATL84] R.H.Katz and T.J.Lehman, "Database Support for Versions and Alternatives of Large Design Files," IEEE Tran. Soft. Eng. Vol.SE-10, No.2, March 1984, pp.191-200.

[KATZ85] R.H.Katz, "Information Management for Engineering Design," Springer-Verlag Berlin Heidelberg New York 1985.

[KATZ86] R.H.Katz, E.Chang, and R.Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," ACM-SIGMOD Inter. Conf. on Management of Data, 1986, pp.379-386.

[KLAH86] P.Klahold, G.Schlageter, and R.Unland, W.Wilkes, "A General Model for Version Management in Databases," Proc. of the 12nd Inter. Conf. on Very Large Data Bases, Kyoto, Aug. 1986, pp.319-327.

[KRUS84] V.Kruskal, "Managing Multi-Version Programs with an Editor", IBM J. Res. Develop., Vol. 28, No. 1, Jan. 1984.