MAINTENANCE OF VIEWS

Oded Shmueli

Alon Itai

Computer Science Department
Technion - Israel Institute of Technology,
Haifa, Israel

ABSTRACT

In relational databases a *view definition* is a query against the database, and a *view materialization* is the result of applying the view definition to the current database. A view materialization over a database may change as relations in the database undergo modifications.

In this paper a mechanism is proposed in which the view is materialized at all times. The problem which this mechanism addresses is how to quickly update the view in response to database changes. A structure is maintained which provides information useful in minimizing the amount of work caused by updates.

Methods are presented for handling both general databases and the much simpler *tree* databases (also called *acyclic* database). In both cases adding or deleting a tuple can be performed in polynomial time. For tree databases the degree of the polynomial is independent of the schema structure while for cyclic databases the degree depends on the schema structure. The cost of a sequence of tuple additions (deletions) is also analyzed.

1. INTRODUCTION

Consider a view V over a database D. The view may change as relations in D undergo modifications. Usually, views are not materialized until needed. In many systems views are never materialized. Instead, queries against the view are *modified* to reflect the view definition. When views are materialized, they remain valid as long as the underlying database remains unchanged.

In this paper a mechanism is proposed in which the view is materialized at all times. The problem which this mechanism addresses is how to quickly update the view in response to database changes. A structure is maintained which provides information useful in minimizing the amount of work caused by updates.

The concept of *tree database* (also called *acyclic database*) is utilized. Acyclicity is a property of the database schema which has wide implications in query processing [GS1-GS4, GST, Yan], dependency theory [BFMMUY, BFMY, Fag, FMU, Hul] and schema design [BFMY, MU1-MU2]. Mathematical properties of acyclicity have also been studied [GS4-GS5, GST, MU1, TY].

It has been shown [BC, BG, GS2, Yan] that certain queries which imply acyclic databases, called *tree queries*, appear easier to process than queries which imply cyclic databases (called *cyclic queries*); and that the crux of query processing is constructing a tree (actually an "embedded tree") [GS3, GST].

The proposed mechanism consists of maintaining an acyclic database at all times together with information that may be useful for future additions and deletions. If the database is cyclic, then it is made acyclic by adding relations. The added relations are called *templates*.

During additions (or deletions) base and template relations undergo modifications to reflect changes to base relations. Changes propagate towards the *database root* where relations are viewed as tree nodes.

Consider adding and deleting tuples starting from an initially empty database, the cost of n such operations is illustrated in the table below. (κ, τ and γ are schema dependent parameters, all less than k - the number of relations.)

2. TERMINOLOGY

2.1 Relational Databases

A *relation schema* is a set of *attributes* and a *database schema* (or simply *schema)* is a multi-set of relation schemas *. A *relation state* R for relation schema R is a relation over R's attributes; a *database state* for schema D is an assignment of relation states to D's relation schemas. A relation R is *total* in database D if it contains all its possible tuples composed of values appearing somewhere in the database, i.e. if $R = \times_{A \in \mathbf{R}}(\cup_{R \in D} R[A])$. We use $\mathbf{D} = (\mathbf{R}_1, \ldots, \mathbf{R}_k)$ to denote a database schema and $D = (R_1, \ldots, R_k)$ for a corresponding state.

* All structures in this paper are finite.

	A single addition or deletion	A sequence of n additions
Tree databases	$O(\kappa n \log n)$	$O(n \log n)$
Cyclic databases	$O(\kappa \frac{\tau}{\gamma^{\tau-1}} n^{\gamma} \log n)$	$O(\frac{\tau}{\gamma^{\tau-1}} n^{\gamma} \log n)$

We use $\bowtie$ for natural join and [X] for projection onto attribute set X. Define $J(D) = \bowtie_{R \in D} R_i$, we use J instead of $J(D)$ when D is understood. Tuple t over schema R *matches* tuple s over schema S if $t[R \cap S] = s[R \cap S]$.

A *view definition* is simply a set $X \subseteq U$ of attributes; a *view materialization* V is $V = J[X]$. Our class of views appears to be quite limited; however, as is shown in [BG], this class encodes a much larger class - those views defined by equijoin queries.

2.2 Tree Schemas

A *qual graph** for D is an undirected graph whose nodes are in one-to-one correspondence with the relation schemas of D, such that for each attribute A, the subgraph induced by the nodes whose corresponding relation schemas contain A is connected [BG]. D is a *tree schema* if some qual graph for it is a tree; otherwise D is a *cyclic schema*. See Figure 2.1.

* We use traditional graph theory notation.

A database is a *tree database* (or an *acyclic database*) if the underlying database system is acyclic, otherwise it is a *cyclic database*.

The following simple procedure, discovered independently by [Gra] and [YO], recognizes tree schemas. The procedure applies the following two steps until neither is applicable.

Step 1: Delete any attribute which appears in exactly one relation schema.

Step 2: Find two relation schemas R and S in D such that $R \subseteq S$; delete R from D.

It can be shown that the original schema was a tree schema iff upon termination of the above procedure the database schema consists of a single (empty) relation schema. (A linear time algorithm for recognizing tree schemas appears in [TY].)

3. ADDITIONS INTO A TREE DATABASE

Let us consider a special case. Suppose for some $r \leq n$, $X \subseteq R_r$. Furthermore, assume the view $R_1, \ldots, R_k$ constitute a tree schema. Let T be a qual tree with R_r at its

root. (R_r is called the *root* relation and the relations at the leaves are called *leaf* relations.)

Let R_i be a node in T and R_j its child. Tuple $t \in R_i$ is *supported* by tuple $s \in R_j$ if t matches s. A tuple $t \in R_i$ is *good* if every child R_j of R_i has a good tuple $s \in R_j$ which supports t. Also, all tuples in a leaf relation are considered good. Tuple $t \in R_i$ is *compatible below* with a child relation R_j if there is a good tuple $s_j \in R_j$ which supports t. Hence, $t \in R_i$ is good iff t is compatible below with all of R_i's children.

Intuitively, a tuple $t \in R_i$ is good if it is unanimously supported by all its children, its children's children and so on, i.e. t belongs to the projection onto R_i of all the relations in the subtree rooted at R_i. Observe that $t \in R_i$ may contribute to $J(D)$, and therefore possibly to V, iff t is good. In other words, all non-good tuples, which we call *bad*, will definitely not contribute to $J(D)$ and V.

Consider the database of Fig. 3.1, with $X \subseteq R_1$. The relations R_2, R_4 and R_5 are leaf relations and therefore all their tuples are good. Only the first two tuples of R_3 are good (E.g. <47,8,SF> matches <47,90> $\in R_4$ and <47,White,SF> $\in R_5$; and <99,15,LA> matches <99,Brown,LA> $\in R_5$ but no tuple of R_4.) Tuple <3,8> is the only good tuple of R_1, it matches the good tuples <47,8,SF> $\in R_3$ and <3,L> $\in R_2$. Tuple <9,17> $\in R_1$ is bad because *all* the R_3 tuples it matches are bad.

The partition of each original relation into a good part and a bad part is helpful when processing updates. We start by discussing tuple addition into the tree database of Fig. 3.1. There are three cases to consider - the relation is a root, a leaf or an internal node.

(i) Root: Suppose t_1=<9,8> is added to R_1 to indicate that supplier number 55 now supplies part 8. Tuple t_1 is good since it is supported by the good tuples <55,8,*LA*> $\in R_3$ which indicates that project 55, located at LA, requires part number 5, and <9,L> $\in R_2$ indicating that the service level of supplier number 9 is rated L. On the other hand, adding the tuple t_2=<7,99> to R_1 cannot possibly change the view since it is not supported by any good tuple of R_3. (The fact that it is supported by the good tuple <7,M> $\in R_2$ is immaterial.) Thus t_2 should be added to bad(R_1).

(ii) Leaf: Suppose t_3=<99,30> is added to R_4 indicating that project 9 has been assigned a budget 30K. First, leaves only have good

parts. Thus t_3 is added to good(R_4). Now, it is possible that the new addition may change the good part of R_3 (which is equivalent to changing an internal node and is discussed below). Namely, t_4=<99,15,LA> $\in R_3$, previously supported only by the good tuple <99,Brown,LA> $\in R_5$ is now also supported by t_3; thus t_4 should move to the good part of R_3. This effect might propagate up the tree. On the other hand <99,15,SF>, which also matches t_3, remains in bad(R_3) since even now it is not supported by any R_5-tuple. To summarize, if the new tuple is good, we should check the matching tuples in the bad part of the parent node because now some of them can become good.

(iii) Internal Node: Suppose t_5=<70,18,DC> is added to R_3. Tuple t_5 is good since it is supported both by <70,50> $\in R_4$ and by <70,Black,DC> $\in R_5$. As mentioned above, changes to an internal node may propagate upwards. We now have to check if t_5 is *compatible above* - i.e. matches with tuples in its parent relation. Indeed, t_5 matches t_6=<19,18> and t_7=<20,18> $\in R_1$. Hence t_6 becomes good since it is supported by <19,M> $\in R_2$, while t_7 remains bad since it is not supported by any (good) R_2-tuple.

Consider an empty database over our fixed schema. To this state apply a sequence of n tuple additions (into various relations). Throughout this addition process maintain the database as above - i.e. with good-bad partitions. Compatibility above is checked only when a tuple becomes good. A tuple t is thus compared to all tuples in its parent node, and if we find a matching bad tuple s, s is checked for compatibility below, since potentially s may have become good. Thus, each time a tuple becomes good it initiates $O(n)$ compatibility checks. Each compatibility check compares a tuple with all the tuples in a parent (or child) node. Thus, in the worst case, each tuple is compared to all other tuples, costing $O(n^2)$ time. Thus, the cost of n additions in this naive scheme is $O(n^3)$.

The following *good-bad marking scheme* reduces the number of times t is checked for compatibility below. Consider a tuple t in bad(R_3) (see Fig. 3.1). It may be there because either

(i) t[PROJ#] is not mentioned in R_4, or

(ii) t[PROJ#,LOC] is not mentioned in R_5.

However, we have no information as to which of these cases hold. To remedy this situation, with each tuple in bad(R_3) we associate

marks. For example, an R_4-mark would indicate that t could find no match in R_4; likewise, for an R_5-mark. As relations change marks may need updating.

Data Structures

We now describe the data structures employed and how the insert and delete operations are performed. Consider relation (node) R_i with tree parent R_p and children $R_1, \ldots, R_c$ Define $Z_{im} = R_i \cap R_m$. The following balanced trees* are associated with R_i:

(a) For each child R_m, a tree C_{im} containing all tuples of $R_i[Z_{im}]$. For each $w_m \in C_{im}$ we associate the list of tuples t in R_i with $w_m = t[Z_{im}]$ and a *good-counter* indicating the number of good tuples (in R_m) which support it.

(b) T_i - containing all the tuples (good and bad) of R_i; each tuple has a *mark-counter* which counts the number of bad marks it has and a pointer (called the *up-pointer)* to the tuple $v = t[Z_{pi}] \in C_{pi}$. (t has an R_m-mark iff w_m's good-counter is equal to zero.)

* On a set with n elements, the operations insert, delete and member can be performed in $O(\log n)$ when the set is implemented as a balanced tree; examples for balanced tree schemes include AVL trees and 2:3 trees [AHU].

We should note that that in all the appearances of t in these trees, it is the *same* t, i.e. t has a record structure which allows it to exist concurrently on several lists.

Operations

Consider a tuple t in R_i with $v = t[Z_{pi}]$ and $w_m = t[Z_{im}]$.

Insert (t, R_i)

A. First, t is inserted into the tree T_i and its mark-counter is set to zero.

B. For each child R_m treat C_{im} as follows:

(a) If w_m does not appear in C_{im} then insert it into C_{im} and set its good-counter to zero. Add t to w_m's list and if w_m's good-counter > 0 then add 1 to t's mark-counter.

(b) Once all C_{im}'s have been treated, if t's mark-counter > 0 then t is bad and we are done; otherwise t is good and we set t's up-pointer to v's appearance in C_{pi}. (Of course, if v does not appear in C_{pi} then it is inserted.) Finally, v's good-counter in C_{pi} is incremented by 1.

(c) If incrementing v's counter transformed it form 0 to 1 then v's list is scanned and each tuple on this list has its mark-counter decremented. If now some tuple s on v's list has its

mark-counter equal to zero (i.e. it became good) then stage (b) above need be (recursively) applied to s.

Delete (t, R_i)

A. Delete t from the tree T_i in R_i.

B. For $1 \leq m \leq c$, delete w_m from the tree C_{im}, if t was the only tuple on w_m's list and w_m's good-counter is zero, then delete w_m from C_{im}.

C. If t was good then the good-counter associated with v in R_{pi} is decreased; (if it becomes zero and v's list is empty then v is removed from C_{pi}). This may remove support from R_p-tuples: v's list is scanned and each tuple has its mark-counter incremented. If some tuple's mark-counter changes from 0 to 1 then the tuple is now bad and stage C of *Delete* has to be (recursively) applied to this tuple and R_p.

Addition Analysis Consider adding a tuple t into relation R_i where the database contains n tuples (we use the same notation as above). Entering t into T_i costs $O(\log n)$. Entering t into w_m's list (recall that $w_m = t[Z_{im}]$ belongs to C_{im}) costs $O(\log n)$; as there are c such trees, the overall cost is $O(c \log n)$. The analysis of t's interaction (in case t is good) with R_p in a bit more intricate. First, the good-counter of v in C_{pi} has to be incremented at a cost of $O(\log n)$. Now, if as a result of this the counter has changed from 0 to 1, mark-counters for tuples on v's list are updated. This updating may cause some bad tuples in R_p to become good and the effect propagates up the tree.

The crucial point in the analysis is that the effect propagates on the unique path from R_i to the root and that in each relation node R along the way each tuple can lose at most one mark - the one corresponding to the unique child S of R which also lies on the path from R_i to the root. Turning $s \in S$ from bad to good costs only $O(1)$ time since we use t's up-pointer to access the list in the appropriate C-tree in R's parent. Hence, since there are n tuples in the database, the overall cost of the propagation effect is $O(n)$. Summarizing, the overall cost of inserting t is $O(c \log n + n)$.

Deletion Analysis Finding t and deleting it from T_i and the lists on the C_{im} trees can be done in $O(c + \log n)$ time. However, if t was the only tuple on a list in C_{im} and the value w_m has a zero good-counter then w_m needs to be deleted ($O(\log n)$ time). Thus the overall cost of updating T and the c trees is $O(c \log n)$. If t was bad we are done. Otherwise, v's good-counter in C_{pi} is decremented; if it becomes zero then, effectively,

an R_i-mark is added to the tuples on v's list. If this transforms some tuples in R_p from good to bad the effect might propagate up the tree. Again, the number of marks that can be added to all tuples in the database in the course of a single deletion is bounded by n. Hence, the overall cost of a single deletion is $O(c \log n + n)$.

Theorem 3.1 A single tuple can be added or deleted from a tree database with n tuples in $O(n + c \log n)$ time.

By the above theorem, any sequence of m operations during which the database never contained more than n tuples costs $O(mn)$. Another complexity measure is *amortized cost*, the cost of adding n tuples into an initially empty database. The main observation here is that in the course of n additions at most n tuples can become good and each tuple can lose at most all its marks. Thus the amortized cost for n additions (and no deletions) into a node with c children is $O(cn \log n)$. We summarize this by

Theorem 3.2 Consider a sequence of n additions to an initially empty database or n deletions and no additions applied to a database with n tuples. This sequence can be performed in $O(\kappa n \log n)$ time, where κ is the maximum number of children of a node in the qual tree.

4. ADDITIONS INTO A GENERAL DATABASE

If the view attributes are not contained in any relation schema, or if the database is not a tree database, we transform the database and view to the previous case by adding new relations that we call *templates*. The problem of finding suitable templates will not be addressed here; see [GS4,GST]. One can think of templates as including in principle "all possible tuples". One way to achieve this is to let a template be total w.r.t. the database. This is fairly wasteful and we shall see other ways of maintaining templates in which only relevant tuples are maintained. In general, templates contain tuples which are computed in various ways from database relations; i.e. *template tuples* are generated from *original tuples*.

Observation 1: Let $D=(R_1, \ldots, R_k)$ be a database and let S be a relation such that $S \supseteq J(D)[S]$. Then for all X,

$$(\bowtie_{i=1}^{k} R_i)[X] = ((\bowtie_{i=1}^{k} R_i) \bowtie S)[X],$$

(i.e. a view cannot be affected by adding S).

Proof: By elementary properties of the join operator.

Observation 2: Let $D=(R_1, \ldots, R_k)$ be a database and let S be a relation such that there exists a tuple t in $((\bowtie_{i=1}^{k} R_i)[S]) \setminus S$. Then for some X it is possible that

$(\bowtie_{i=1}^{k} R_i)[X] \supset ((\bowtie_{i=1}^{k} R_i) \bowtie S)[X]$.

Proof: There exists a tuple $u \in J(D)$ such that $u[S]=t$. However, since $t \notin S$, $u \notin (\bowtie_{i=1}^{k} R_i) \bowtie S$. If there is no tuple $v \neq u$ in $J(D)$ such that $v[X]=u[X]$, then $u[X] \in (\bowtie_{i=1}^{k} R_i)[X]$ but $u[X] \notin ((\bowtie_{i=1}^{k} R_i) \bowtie S)[X]$.

[]

Consider first the case of a cyclic database in which the view attributes are contained in some relation; the other cases are similar. Assume the database was *tree*fied by adding some templates. For the good-bad mechanism to function, by Observations 1 and 2, each template S must at least contain $(\bowtie_{i=1}^{k} R_i)[S]$.

Next, we discuss various schemes for extending the good-bad mechanism to templates. Unlike relations where the "base set" of tuples is fixed, templates may undergo changes when base relation tuples are changed: the template base set may grow as a result of adding a tuple to the good set of a base relation, or shrink when such a tuple is deleted. The problem is parametrized according to the treefied schema structure. We use the following parameters:

τ - the number of templates.

γ - the maximum number of generators (defined below) per template.

κ - the maximum number of children of a node in the resulting qual tree.

Let D be a database schema *tree*fied by adding τ templates. Consider the process of adding n tuples to an initially empty database state D. We separate the cost into two parts: that of finding the tuples to be entered into the templates and that of entering all the tuples into the database; the latter consists of the cost of the addition of the n original tuples and the cost of adding template tuples, both using the good-bad mechanism. We have the following theorem

Theorem 4.1: Adding n tuples into an initially empty *tree*fied database requires adding at most $O(\tau \cdot 2^n)$ tuples into templates.

Proof: An addition of a tuple t into a relation R may introduce a "new value" $t[S]$ for template S. Let $s=t[R \cap S]$. To enlarge the template we simply *duplicate* S and in one of the copies replace the $R \cap S$ columns with s. Thus, the addition of a tuple may double the number of tuples in each template. The result follows since there are n original tuples and τ templates.

[]

Corollary: Adding n tuples to an initially empty treefied database requires at most $O(\kappa\tau n 2^n)$ time.

Proof: By Theorem 4.1 $N=\tau 2^n$ tuples are added, and by Theorem 3.2 this costs $O(\kappa N \log N)$ time.

[]

The above result is discouraging since the cost is extremely high even for a small number of tuples. As we shall see, we can substantially improve this result.

The manner in which templates are enlarged determines the cost of extending the good-bad mechanism. Let S be a template over attributes S. One way to generate relevant S tuples is to join enough database relations so as to obtain all of S's attributes. Formally, the relations $R_1, \ldots, R_g$ are a *generator set* for S provided $S \subseteq \cup_{i=1}^{g} R_i$; they generate $S' = (\bowtie_{i=1}^{g} R_i)[S]$. S' can then be partitioned to good(S') and bad(S') by the usual procedure. In other words, we have described a method for instantiating a candidate for containing both the good and the relevant bad tuples in a template. (See Figure 4.1(a).)

The cost of tuple additions is dominated by the correct maintenance of templates, i.e. when a tuple is added to the good part of a generator relation, the templates for which it is a generator might have to be enlarged. This means joining the new tuple with all the other generators, a potentially costly procedure ($O(\tau n^\gamma)$). Since there are at most n such additions the overall cost is $O(\tau n^{\gamma+1})$. (A closer analysis reveals that the cost is $O(\frac{\tau}{\gamma-1^{\gamma-1}} n^\gamma)$.)

The following refinement will enable us to reduce this cost. For each template we shall build a *generator tree* which is a full binary tree, the template is at its root and the generators at its leaves. An internal node consists of the join of its two child relations. (Note that the generator tree is a separate structure which comes in addition to the usual qual tree and the various balanced trees. See Figure 4.1(b).)

In order to compute the cost of n additions into the generator relations of a template S we make the following observations:

(1) When a tuple enters a generator relation, it has to be compared to its sibling in the generator tree in order to populate their parent.

(2) Each leaf contains at most n tuples.

(3) The parent of nodes with m_1 and m_2 tuples has at most $m_1 \cdot m_2$ tuples. Consequently, a node at distance h from the leaves has at most n^{2^h} tuples.

(4) The cost of adding m_1 tuples to a child

and m_2 tuples to its sibling is exactly $m_1 \cdot m_2$, the maximum size of their parent.

(5) The cost of all the additions into a set of generators is equal to the sum of the sizes of all the internal nodes of the generator tree.

Theorem 4.2 Suppose n tuples are added to an initially empty database. The time required to add all template tuples is $O(\tau(\frac{n}{\gamma})^\gamma)$.

Proof: First, consider two sibling nodes in the generator tree with a total of m tuples. The number of tuples in their parent node is maximum when each of the siblings has $\frac{m}{2}$ tuples. Therefore, the number of tuples in a generator tree is maximum when all its leaves have the same number of tuples. The worst case occurs when there are γ leaves and exactly $\frac{n}{\gamma}$ tuples per leaf, in which case the total number of tuples is

$$\sum_{i=1}^{\log\gamma} (\frac{n}{\gamma})^{2^i} \frac{\gamma}{2^i} = O\left[(\frac{n}{\gamma})^\gamma\right].$$

Since there are τ templates the total cost is

$$O\left[\tau(\frac{n}{\gamma})^\gamma\right].$$

[]

Corollary: Adding or deleting a single tuple to a treefied database containing n original tuples requires at most $O(\tau(\frac{n}{\gamma})^\gamma + \kappa\gamma\log n)$ time.

Corollary: Adding n tuples to an initially empty treefied database requires at most $O(\frac{\kappa\tau}{\gamma^{\gamma-1}} n^\gamma \log n)$ time.

This is more encouraging than the corollary to Theorem 4.1 since in many practical applications γ is small.

Finally, we note that the cost of a single deletion can be quite high, since it may cause many tuples in templates to become bad, costing the same as n additions. Practically, it seems better to do the following: each time we delete a tuple we also delete all tuples it helped generating (in templates). Thus, at all time, when n original tuples are in the database, there remain at most $O(\tau(\frac{n}{\gamma})^\gamma)$ tuples in the database.

CONCLUSIONS

The problem of dynamically maintaining a class of views has been examined. A scheme incorporating various structures was proposed as a maintenance mechanism for views in the class.

The complexity of updates is polynomial: for tree schemas the degree of the polynomial is independent of the schema structure while for cyclic schemas the degree depends on the schema structure. We do not know

whether the bounds we found are tight and we leave it as an open problem. Note that if the balanced trees are replaced by hash tables the complexity is reduced by a factor of log$n$ (on the average).

This paper also suggests additional problems such as that of maintaining multiple views, and that of extending the mechanism to an off-line sequence of updates to base relations.

The complexity measure used in the analysis was the number of tuple operations. Thus the analysis is directly applicable to small scale databases whose data, or very large portions thereof, fits into memory. The tuple operations measure is inadequate for large databases in which only a small portion of the data can reside in main memory.

Consider a large scale database environment. First, the balanced trees may be implemented as B-trees or replaced by a suitable hashing scheme. Second, the recursive add and delete operations should be made to recurse on sets of tuples rather than on single tuples. This minimizes the number of relations that are dealt with at any one time, a better use of buffers is achieved and therefore secondary storage access performance is improved.

REFERENCES

[AHU] Aho, A.V., J.E.. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1976.

[BC] Bernstein, P.A., and D.W. Chiu, "Using Semi-Joins to Solve Relational Queries", J. ACM 28 (1): 25-40, January 1981.

[BFMMUY] Beeri, C., R. Fagin, D. Maier, A. Mendelzon, J.D. Ullman, and M. Yannakakis, "Properties of Acyclic Database Schemas", in Thirteenth Annual ACM Symp. on Theory of Computing, 355-362. Association for Computing Machinery, New York, N.Y., May 1981.

[BFMY] Beeri, C., R. Fagin, D. Maier and M. Yannakakis, "On the Desirability of Acyclic Database Schemes", J.ACM, to appear.

[BG] Bernstein, P.A., and N. Goodman, "The Power of Natural Semijoins", SIAM J. of Comput., 10 (4), November 1981.

[Fag] Fagin, R., "Types of Acyclicity for Hypergraphs and Relational Database Systems", Research Report RJ3330, IBM Research Laboratory, San Jose, CA, November 1981.

[FMU] Fagin, R., A.O. Mendelzon, and J.D. Ullman, "A Simplified Universal Relation Assumption and Its Properties", Technical Report RJ2900, IBM, San Jose, CA, 1980.

[Gra] Graham, M.H., On the Universal Relation, Technical Report, University of Toronto, September 1979.

[GS1] Goodman, N., and O. Shmueli, "Tree Queries: A Simple Class of Queries", ACM Transactions on Database Systems, December 1982.

[GS2] Goodman, N., and O. Shmueli, "The Structure of Database Schemas". To appear in J. ACM.

[GS3] Goodman, N., and O. Shmueli, "The Tree Property is Fundamental for Query Processing", in Proc. ACM SIGACT- SIGMOD Conference on Principles of Database Systems, 40-48, Los Angeles, CA, March 1982.

[GS4] Goodman, N., and O. Shmueli, "Transforming Cyclic Schemas into Trees", in Proc. ACM SIGACT-SIGMOD Conference on Principles of Database Systems, 49-54, Los Angeles, CA, March 1982.

[GS5] Goodman, N., and O. Shmueli, "NP-Complete Problems Simplified on Tree Schemas", To appear in Acta Informatica.

[GST] Goodman, N., O. Shmueli and Y.C. Tay, "GYO Reductions, Canonical Connections, Tree and Cyclic Schemas and Tree Projections", in Proc. ACM SIGACT-SIGMOD Conference on Principles on Database Systems, 267-278, Atlanta, Ga., March 1983.

[Hul] Hull, R., "Acyclic Join Dependencies and Database Projections", in Proc. XP2, State College, PA, June 1981.

[MU1] Maier, D., and J.D. Ullman, "Connections in Acyclic Hypergraphs", in Proc. ACM SIGACT-SIGMOD Conference on Principles of Database Systems, 34-39, Los Angeles, CA, March 1982.

[MU2] Maier, D., and J.D. Ullman, "Maximal Objects and the Semantics of Universal Relation Databases", Technical Report #80-016, Dept. of Comp. Science, SUNY at Stonybrook, November 1980.

[PY] Papadimitriou C.H., and M. Yannakakis, "The Complexity of Facets (and some facets of complexity)", in Fourteenth Annual ACM Symp. on Theory of Computing. Association for Computing

Machinery, New York, N.Y., May 1982.

[TY] Tarjan, R.E., and M. Yannakakis, "Simple Linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs", unpublished manuscript, March 1982.

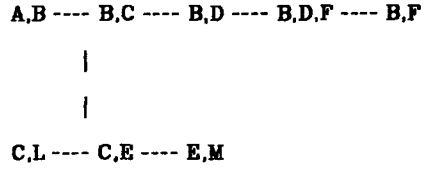
[Yan] Yannakakis, M., "Algorithms for Acyclic Database Schemes", in Proc. VLDB, 82-94, Cannes, France, September 1981.

[YO] Yu, C.T., and M.Z. Ozsoyoglu, "An Algorithm for Tree-Query Membership of a Distributed Query," in Proc. COMPSAC79, IEEE Comp. Society, November 1979.

Consider the schema

$D = (\{A,B\},\{C,L\},\{E,M\},\{C,E\},\{B,F\},\{B,D,F\},\{B,D\},\{B,C\})$.

D is a tree schema viz.



For example, the subgraph induced by attribute C is :

C,L----C,E----B,C.

The following is a cyclic schema:

$D = (\{A,B\},\{B,C\},\{C,A\})$.

The only qual graph for D is

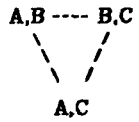
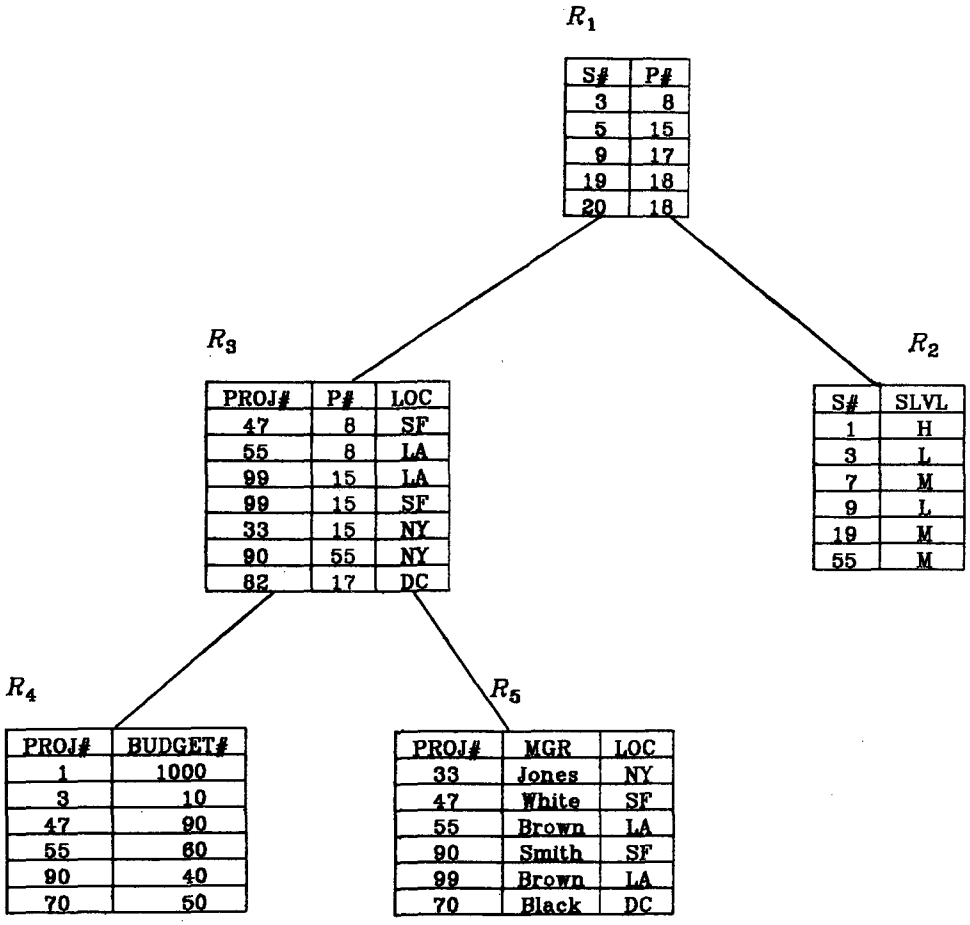


Figure 2.1 Tree and Cyclic Schemas



R_1: supplier $S\#$ supplies part $P\#$;

R_2: each supplier may provide product support (indicated
by SLEVEL);

R_3: project PROJ# may need part P# at location LOC;

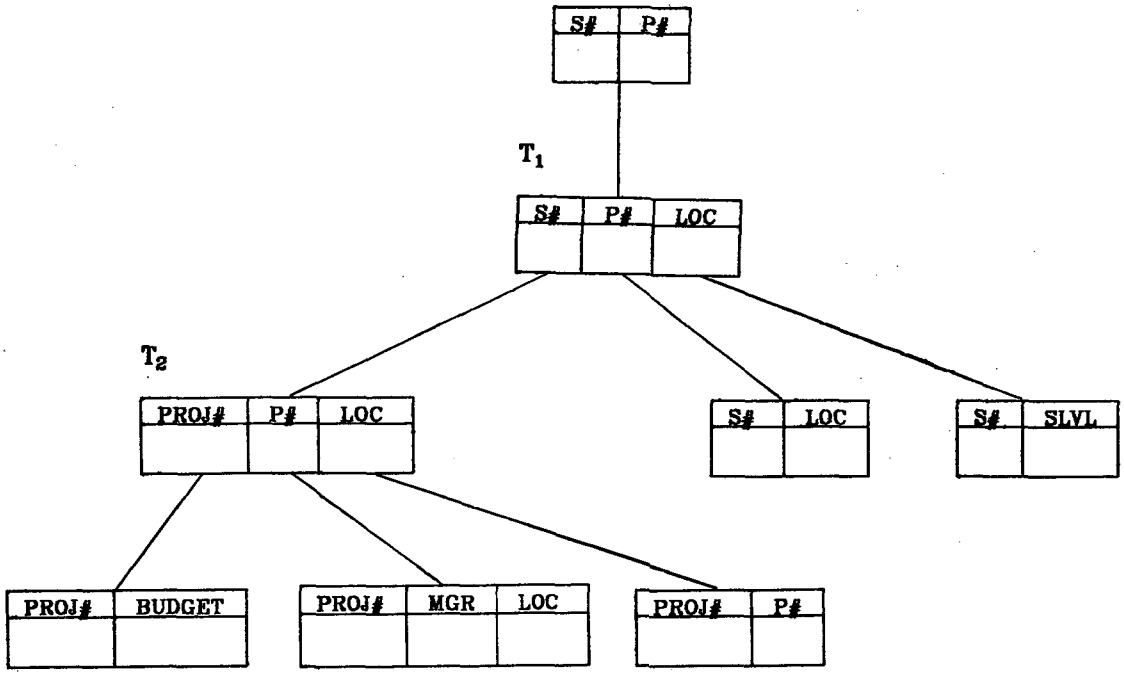
R_4: project PROJ# has an allocated BUDGET;

R_5: project PROJ# is managed by MGR at location LOC.

The view is on $S\#$ and $P\#$.

Figure 3.1 : An Example Tree Database

(a) Adding templates T_1 and T_2 to the original schema:



(b) Generator Trees

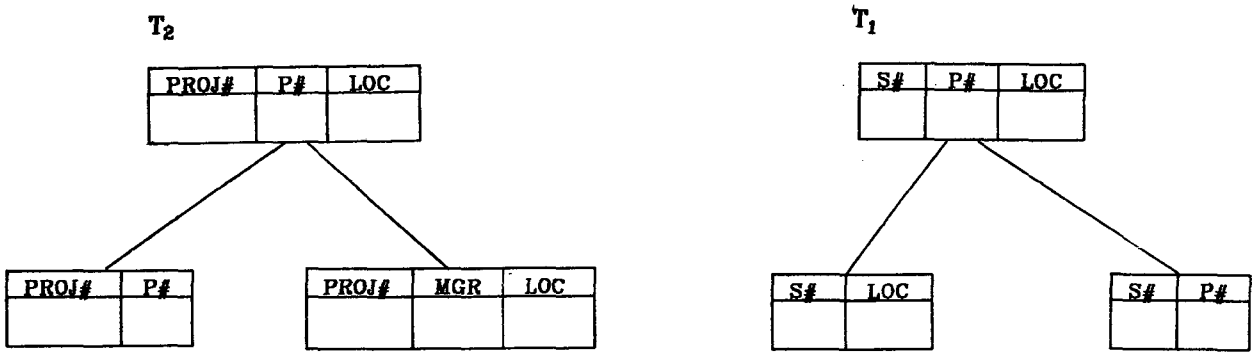


Figure 4.1 Templates