COMPLEXITY OF VIEWS: TREE AND CYCLIC SCHEMAS.

by

O. Shmueli and A. Itai

Technical Report #312
January 1984
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.

Several problems concerning views are considered, many of which are shown to be hard (NP-complete or even \( \Sigma_2^p \)-complete). Each problem was treated for general databases and for the much simpler tree databases (also called acyclic database).

View related problems over fixed schemas, in which only the data is allowed to vary, were examined. Methods to handle this case were presented, their complexity 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. These methods may present a practical possibility for dynamic view maintenance.
1. INTRODUCTION

A relational database [Cod, Ull] is a collection of tables called relations, each containing a set of data rows called tuples. We differentiate between the structure of the database (the schema) and the time varying data (the state). A database schema \( D = (R_1, \ldots, R_n) \) is simply a multiset of finite subsets of a set \( U \) of attributes. A schema can be viewed as the edge-set of a hypergraph over \( U \) [Ber].

One may partition the class of database schemas into tree and cyclic schemas. A schema is a tree schema if there is a tree whose nodes correspond to the schema's sets, and for each \( A \) in \( U \), the subtree induced by nodes containing \( A \) is connected. Some equivalent characterizations of tree schemas are mentioned in Section 2.

The partition above appears to be a good dividing line for database problem analyses. Acyclicity 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 [GS2, GS4-GS5, GST, MU1, TY].

It has been shown [BC, BG, GS2, Yan] that a class of queries (called tree queries) which imply acyclic databases 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 above results all hinge on the simple structure of tree schemas. In this paper we examine the relationship between schema structure and view related problems.

A view definition is a query against the database. 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. When views are materialized, they remain valid as long as the
underlying database remains unchanged. 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 (a process called query modification).

The main difference between an "ordinary" query and a view definition has to do with the frequency of use. A view is either a query that is often posed or one which delimits a relevant portion of the database for a group of users. Hence, maintaining a correct view materialization over time may prove beneficial.

The view definitions we consider are all of a simple form: perform the natural join of all the relations in the database and project the result on a set of attributes $X$. This simple form actually encodes a much larger class of views [BG]. We examine various problems associated with these views and their materialization maintenance over time. We note that view related problems were mainly treated in the past under the guise of query-processing [Cos,Yan].

View maintenance includes a variety of problems concerning the tuples in the view, equivalence of views, how changes in the underlying database affect the view and which kind of information is useful in maintaining a view. For example, one of the problems we treat is the following: Given a database $D$, a view definition $X$, a tuple $t$ and a relation schema $R_i$ would the view materialization change when $t$ is added to $R_i$?

Terminology is presented in Section 2; our problem classification scheme is introduced, and a summary of the results is tabulated. Section 3 is devoted to "join problems", i.e. we consider the case $X=U$, (the view is the natural join of all the database relations) "Genuine" views, where $X$ is a proper subset of $U$, are treated in Section 4. In Section 5 we consider view complexity over a fixed schema.
2. TERMINOLOGY

2.1 Relational Databases

A universe \( U \) is a finite set of attributes. A relation schema \( R_i \) is a subset of \( U \), and a database schema \( D \) (or simply schema) is a multi-set of relation schemas. Clearly, a database schema may be viewed as the set of edges of a hypergraph over \( U \) [Ber]. Associated with each \( A \in U \) is a possibly infinite domain, \( \text{dom}(A) \).

The domain of a relation schema \( R_i = \{A_{i1}, \ldots, A_{i_h}\} \) is \( \text{dom}(R_i) = \times_{k=1}^{h} \text{dom}(A_{ik}) \).

A relation state \( R_i \) for relation schema \( R_i \) is a finite subset of \( \text{dom}(R_i) \); one can think about the state as a table of data with columns \( A_{i1}, \ldots, A_{ih} \). A database state for schema \( D \) is an assignment of relation states to \( D \)'s relation schemas. We use \( D = (R_1, \ldots, R_n) \) to denote a database schema and \( D = (R_1, \ldots, R_n) \) for a corresponding state. Elements in a relation state are called tuples. Tuple \( t \) over schema \( R \) matches tuple \( s \) over schema \( S \) if for all \( A \in R \cap S \), the values of tuples \( t \) and \( s \) for attribute \( A \) are identical.

The projection of relation state \( R \) over attribute set \( X \subseteq R \), denoted \( R[X] \), is the maximal subset of \( \text{dom}(X) \) containing tuples that match some tuple in \( R \). The (natural) join of relation states \( R \) and \( S \), denoted \( R \bowtie S \), is defined as the maximal subset in \( \text{dom}(R \cup S) \) containing tuples that match a tuple in \( R \) and a tuple in \( S \). 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 \( R = \times_{A \in R} (\cup_{R \in D} \text{dom}(A)) \).

For a database \( D \) over schema \( D \) define \( J(D) = \bowtie_{R \in D} R_i \), we use \( J \) instead of \( J(D) \) when \( D \) is understood; define \( J(R_i + t) \) to be the natural join of all the relations in \( D \) except that \( R_i \) is augmented with the tuple \( t \). A view definition is simply a set \( X \subseteq U \) of attributes; a view materialization \( V \) is \( V = J[X] \); also let

\* All structures in this paper are finite.
$V(R_i + t) = J(R_i + t)[X]$. 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.

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

Define the *adjacency graph* (or 2-section [Ber]) for schema $D$ to be the graph $AG(D)$ with node set $U$, where $(u,v)$ is an edge of $AG$ iff there is some $R \in D$ such that $\{u,v\} \subseteq R$. A graph is *chordal* if for each cycle of length four or more there is a *chord* - an edge not in the cycle which joins two nodes of the cycle. A schema $D$ is *conformal* [Ber] iff each maximal clique in $AG(D)$ is a relation schema in $D$. It can be shown that $D$ is a tree schema iff $AG(D)$ is chordal and $D$ is conformal [Bun, GS2, BFMMUY].

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].)

---

*We use traditional graph theory notation.*
2.3 Complexity Classes

We shall characterize the complexity of various view related problems using three complexity classes: P, NP and \( \Sigma^P_2 \). The problems we treat are all language recognition problems: given a string as input we are asked whether the string belongs to a language. (For example, the language may consist of strings encoding graphs having a Hamiltonian cycle.)

A problem (language) \( L \) is in \( \text{NP} \) if given a string \( x \), determining \( x \in L \) can be done by a (non)deterministic Turing machine within time polynomial in the size of the input \( x \). A problem is in \( \Sigma^P_2 \) if it can be solved by a nondeterministic Turing machine, which may use an oracle for a set in NP, in time polynomial in the size of the input. (An oracle for a language \( L \) can be thought of as a "subroutine" which when given some string \( x \) answers in one time unit "yes" if \( x \in L \) and "no" otherwise. The subtle point is that the Turing machine can make use of the fact that a string does not belong to the oracle set.) For more details, the reader is referred to [GJ]

A problem \( A \) is complete for the complexity class \( C \) if for any other problem \( B \) in \( C \), there is a polynomial time bounded Turing machine \( M \) which transforms, a string \( x \) into a string \( M(x) \), such that \( x \in B \) iff \( M(x) \in A \); \( M \) is said to reduce \( B \) to \( A \). Intuitively, if \( A \) is a complete problem in a class then an efficient algorithm for solving \( A \) will provide an efficient algorithm for all problems in the class.

We now exhibit some known complete problems:

1. 3CNF - Given: a boolean formulas in conjunctive normal form having three literals per clause [AHU].

   Question: Is \( F \) satisfiable? i.e. is there an assignment to the boolean variables in \( F \) which makes it true.

   3CNF is \( \text{NP} \)-complete.
(2) Let $L$ be the language

$$L = \{ F(X,Y) \mid \exists X \forall Y F(X,Y) \text{ is a valid formula} \}.$$ $L$ is complete in $\Sigma_2^p$ (see [GJ]).

2.4 Problem Classification

We shall classify problems according to the following criteria:

(1) The object in question:

$J$ - A problem concerning the join of a given database $D$.

$V$ - A problem concerning the view of a given database $D$ on the given attributes $X$.

(2) The type of data supplied (optional).

$C$ - Change: given a tuple $t$ and an index $i$, consider the new join $J(R_i + t)$ (or new view $V(R_i + t)$).

$G$ - Given: in addition to $C$, the old join (or old view materialization) is part of the input.

(3) The question:

$E$ - Emptiness: is the join (or the view) not empty?

$M$ - Membership: given a tuple $t$ does $t$ belong to the join (view)?

$I$ - Intotality: is the join (view) not total?

$N$ - Not equal: is the new join (view) not equal to the old one? (This question is meaningful only if $C$ or $G$ are present.)

For example: the problem $JE$ is defined as follows:

Given a database $D$ is $J \neq \emptyset$?

And the problem $VGN$ is:

Given a database $D$, a view definition $X$, the view materialization $V$, an index $i$ and the tuple $t$ is $V \neq V(R_i + t)$?
Thus a total of 22 different problems are defined, results concerning these problems are shown in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>M</th>
<th>I</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>NP-C</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>JC</td>
<td>NP-C</td>
<td>P</td>
<td>P</td>
<td>NP-C</td>
</tr>
<tr>
<td>JG</td>
<td>NP-C</td>
<td>P</td>
<td>P</td>
<td>NP-C</td>
</tr>
<tr>
<td>V</td>
<td>NP-C</td>
<td>NP-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>NP-C</td>
<td>NP-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VG</td>
<td>NP-C</td>
<td>NP-C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1.

If D is a tree schema the above problems are sometimes easier as seen in Table 2.2.

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>M'</th>
<th>I</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>P*</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>JC</td>
<td>P*</td>
<td>P</td>
<td>P</td>
<td>P*</td>
</tr>
<tr>
<td>JG</td>
<td>P*</td>
<td>P</td>
<td>P</td>
<td>P*</td>
</tr>
<tr>
<td>V</td>
<td>P</td>
<td>P [Yan]</td>
<td>NP-C</td>
<td></td>
</tr>
<tr>
<td>VC</td>
<td>P</td>
<td>P</td>
<td>NP-C</td>
<td>NP-C</td>
</tr>
<tr>
<td>VG</td>
<td>P</td>
<td>P</td>
<td>NP-C</td>
<td>P</td>
</tr>
</tbody>
</table>

Table 2.2.

**Remark:** For tree databases all the problems are polynomial if the view definition \( X \) is contained in any one of the relations \( R_i \).

The results of [BG, BG] imply that these problems are polynomial.
3. JOIN PROBLEMS

3.1 Polynomial Problems

JI, JCI and JGI - The join is total iff its projection on the attributes of each \( R_t \) is total. Since when the join is total for all \( i \), \( J[R_t] = R_t \), the join is total iff each \( R_t \) is total. The latter condition can be checked by counting the number of distinct tuples in each \( R_t \).

3.2 NP-complete problems

JE - This problem was first shown to be NP-complete by Chandra and Merlin [CM]. The problem is in NP since a nondeterministic Turing machine can guess a tuple \( s \) and check whether \( s \in J \) in polynomial time. We show completeness by a folk theorem using the following

Standard Database Construction:

Given a boolean formula \( F(X) \) in 3CNF we show how to construct a database \( D \) such that \( J \neq \phi \) iff \( F \) is satisfiable. With each clause of \( F \) we associate a relation schema, whose attributes are the variables appearing in the clause. W.l.o.g. we may assume that each relation schema thus constructed consists of three attributes. Each relation consists of the seven boolean assignments which make the original clause evaluate to true. It can be easily seen that each tuple in the natural join of all the relations in the database "spells out" an assignment to \( F \)'s variables satisfying \( F \). Hence, \( J \neq \phi \) iff \( F \) is satisfiable.

JCE - The problem is in NP since it suffices to guess a tuple \( s \) and check whether it belongs to \( J(R_t+t) \). The completeness follows by a reduction from JE: Let \( D=(R_1, \ldots, R_k) \) be a instance of JE. Let \( D'=(R_0', R_1', \ldots, R_k') \) where \( R_0' \) consists of a single attribute \( C \), \( R_0'=\{<b>\} \) and the attributes of \( R_i' \) are those of \( R_i \) and the additional attribute \( C \).

Let \( R_t' = \{<a_1, \ldots, a_{m_t}, a> | <a_1, \ldots, a_{m_t}> \in R_t\} \)
by construction \( J(D') = \phi \). We augment \( R_0' \) by the tuple \( t = \langle a \rangle \), clearly
\[
J(R_0' + t) \neq \phi \iff J(D) \neq \phi.
\]

JGE - In the previous construction the old join was empty, thus we may assume it
was given.

JCN and JGN - The problem is in NP since a nondeterministic Turing machine can
guess a tuple \( s \) such that \( s \notin J \) and \( s \in J(R_0 + t) \). (Recall that JM is polyno-
mial.) Completeness follows by a reduction from JGE (as in the completeness
proof of JCE - we add a new column \( C \) and a new relation \( R_0 \) such that
\( J(D') = \phi \), however after adding a new tuple \( t \) \( J(R_0 + t) = \phi \iff J(D) = \phi \).

4. VIEW PROBLEMS

4.1 The VI problem over general databases

Lemma 4.1: Let \( F(X) \) be a boolean formula over the boolean variables
\( X = \{X_1, \ldots, X_m\} \). There exists a boolean formula \( F^{\text{cook}}(X, Z) \), where \( Z = \{Z_1, \ldots, Z_n\} \),
such that for any boolean vector \( t_X \) (of length \( m \))
\[
[ \exists Z F^{\text{cook}}(t_X, Z)] \leftrightarrow \neg F(t_X).
\]
Moreover, \( F^{\text{cook}} \) is in 3CNF and can be constructed in polynomial time.

Proof: Given \( F(X) \), construct a nondeterministic Turing machine \( M \): \( M \) guesses
true/false values \( t_1, \ldots, t_m \) for \( X_1, \ldots, X_m \). \( M \) puts its guess on the first \( m \)
squares of its work tape and never modifies these values. \( M \) then acts determi-
nistically, it evaluates \( F \) on the \( t_i \)’s and accepts iff \( F(t_1, \ldots, t_m) = \text{false} \).
Clearly both \( M \)’s description and running time are small polynomials in the size of \( F(X) \).

We now utilizes the construction used in Cook’s proof that satisfiability of
boolean formulas is NP-complete (see the exposition in [AHU]). From \( M \)’s
description and its running time we cook a boolean formula \( F^o \) (over boolean vari-
ables \( Z_1, \ldots, Z_l \) in 3CNF which is satisfiable iff \( M \) has an accepting computation.
In Cook's construction the content of a tape square at a point in time during
the computation is represented by a boolean variable. Therefore, we can "put
our hand" on those variables representing the first \( m \) squares on \( M \)'s tape at a
point in time following \( M \)'s guesses. We augment \( F^c \) with \( m \) clauses of the form
\[ X_i = C(i, \text{true}, T) \]
where as in Cook's construction \( C(i, \text{true}, T) \) is a boolean variable
with the meaning "the content of tape square \( i \) at time \( T \) is the symbol \text{true}" and
\( T \) is a sufficiently "late time" in \( M \)'s computation.

The added clauses can easily be expressed in 3CNF. We end up with a for-

mula \( F^{\text{cook}}(X,Z) \) in 3CNF where the \( X \)'s are semantically the same \( X \)'s as in \( F(X) \)
and the \( Z \)'s are variables introduced in the construction. The claim of the lemma
follows by construction.

**Theorem 4.1:** \( \text{VI} \) is \( \Sigma^P_2 \)-complete.

**Proof:** The fact that \( \text{VI} \) is in \( \Sigma^P_2 \) is straightforward: a nondeterministic Turing
machine \( M \) can guess \( v \) and then consult an oracle for \( v \not\in V \), the oracle set is in
\( \text{NP} \) since determining whether \( v \in V \) is \( \text{NP} \)-complete.

Let \( L \) be the language

\[ L = \{ F(X,Y) \mid \exists X \forall Y. F(X,Y) \text{ is a valid formula} \} \]

\( L \) is complete in \( \Sigma^P_2 \) (see [GJ]). Given a string of the form \( F(X,Y) \) we show how to
construct a database \( D \) and view definition \( X \), such that \( \exists X \forall Y. F(X,Y) \) is valid if
(\( \exists v \not\in V \).

By Lemma 4.1, given a boolean formula \( F(X,Y) \) we construct in polynomial
time, in the size of \( F \), a boolean formula \( F^{\text{cook}}(X,Y,Z) \) in 3CNF such that for all
boolean vectors \( t_X, t_Y \),

\[ \exists Z F^{\text{cook}}(t_X,t_Y,Z) \iff [-F(t_X,t_Y)], \]

which implies that for any boolean vector \( t_Z \),
Build a standard database $D$ for $F^{\text{cook}}$ as described in Section 3. Let $X$ be the view definition on $D$.

Claim:

$$[\exists v \forall v \notin V] \rightarrow [\exists X \forall Y F(X,Y)].$$

Proof of Claim:

Assume $[\exists v \forall v \notin V]$:

Each tuple $t$ "spells out" an assignment $t_X \cup t_Y \cup t_Z$ to the boolean variables in $F^{\text{cook}}(X,Y,Z)$. Suppose $v \notin V$, then for all tuples $t$ such that $t[X]=v$, $F^{\text{cook}}(t_X \cup t_Y \cup t_Z)=\text{false}$ (otherwise, if $F^{\text{cook}}(t_X \cup t_Y \cup t_Z)=\text{true}$ then $t$ would be in $J$ and $v$ in $V$). Let $v$ spell out the assignment $t_X$ to the $X$ variables in $F^{\text{cook}}(X,Y,Z)$.

It follows that

$$\forall Y \forall Z [F^{\text{cook}}(t_X,Y,Z)=\text{false}].$$

In other words

$$\forall Y \forall Z - F^{\text{cook}}(t_X,Y,Z).$$

This gives

$$\forall Y - [\exists Z F^{\text{cook}}(t_X,Y,Z)].$$

Consider any boolean vector $t_Y$. By the above we have

$$\neg[\exists Z F^{\text{cook}}(t_X,t_Y,Z)].$$

But, by Lemma 4.1

$$[(\exists Z)F^{\text{cook}}(t_X,t_Y,Z)] \rightarrow [\neg F(t_X,t_Y)].$$

We conclude that for any boolean vector $t_Y$

$$\neg F(t_X,t_Y),$$

or equivalently, $F(t_Y,t_X)$:

which means that indeed $\exists X$ (namely $t_X$) such that for all $Y$, $F(X,Y)$.

We now assume $[\exists X \forall Y F(X,Y)]$: Let $t_X$ be a boolean vector such that $\forall Y F(t_X,Y)$. Choose any $t_Y$, we have $F(t_X,t_Y)$. By Lemma 4.1,

$$\neg[\exists Z F^{\text{cook}}(t_X,t_Y,Z)].$$
i.e.

\[ \forall Z \neg P_{\text{cook}}(t_X, t_Y, Z). \]

Since \( t_Y \) was chosen arbitrarily it follows that,

\[ \left[ \forall Y \forall Z \neg P_{\text{cook}}(t_X, Y, Z) \right] = \text{true}. \]  

We now show that (4.1) implies \( \exists \forall u \in V \).

It suffices to show that

\[ \left[ \exists \forall u \in V \right] \rightarrow \neg \left[ \forall Y \forall Z \neg P_{\text{cook}}(t_X, Y, Z) \right] \]

Or equivalently that \( \left[ \forall u \in V \right] \rightarrow \neg \left[ \forall Y \forall Z \neg P_{\text{cook}}(t_X, Y, Z) \right] \).

In other words we have to show

\[ \left[ \forall \forall u \in V \right] \rightarrow \exists Y \exists Z P_{\text{cook}}(t_X, Y, Z). \]

From \( \forall \forall u \in V \) follows the existence of a boolean vector \( t_X \) in \( V \). But if \( t_X \) is a tuple in the view, then there must exist a "parent" tuple \( t=t_X \cup Y \cup Z \) such that \( t_X=t[X] \). The existence of \( t \) implies the existence of \( t_Y=t[Y] \) and \( t_Z=t[Z] \) such that \( P_{\text{cook}}(t_X, t_Y, t_Z) = \text{true} \). But this simply states that

\[ \exists Y \exists Z P_{\text{cook}}(t_X, Y, Z). \]

Thus concluding the proof of Theorem 4.1.

4.2 The VI problem over tree databases

The following problems are useful in proving the NP-completeness result:

**SET COVER:**

Given \( n \) sets \( C_1, \ldots, C_n \) and an integer \( k \), are there \( C_1, \ldots, C_k \) whose union equals \( \bigcup_{i=1}^{n} C_i \)? (NP-complete [GJ].)

**FAMILY COVER:**

Given \( k \) families \( S_1, \ldots, S_k \), where each \( S_i \) contains \( n \) sets \( C_{i1}, \ldots, C_{in} \). Are there \( S_{1i_1}, \ldots, S_{ni_n} \) such that \( \bigcup_{j=1}^{n} S_{ij} = \bigcup_{i=1}^{k} \bigcup_{j=1}^{n} S_{ij} \)?
NCP (Non Cartesian Product):

Given families $F_1, \ldots, F_h$, where $F_i = (C_{i1}, \ldots, C_{ik})$.

Does the NCP inequality condition

$$\bigcup_{i=1}^h (C_{i1} \times \cdots \times C_{ik}) \neq \big( \bigcup_{i=1}^h C_{i1} \big) \times \cdots \times \big( \bigcup_{i=1}^h C_{ik} \big)$$

hold?

Claim 1: FAMILY COVER is NP-complete.

Proof: Let $C_1, \ldots, C_n$ and $k$ be an instance of SET COVER. Form an instance of FAMILY COVER with families $S_1, \ldots, S_k$, where each $S_i$ contains a copy of $C_1, \ldots, C_h$.

Claim 2: NCP is NP-complete.

Proof: The problem is obviously in NP. Let $S_1, \ldots, S_k$ be an instance of FAMILY COVER, where $S_i = \{S_{i1}, \ldots, S_{in}\}$. w.l.o.g. each $S_i$ has a unique element associated with it that is a member of each set in the $S_i$ family, and is not a member of any set in any other family. Let $A = \bigcup_{i=1}^k \bigcup_{j=1}^n S_{ij}$. We form an instance of NCP by associating with each $a \in A$ a family of sets $F_a = \{F_{a1}, \ldots, F_{ak}\}$ where $F_{aj} = \{<i,j> \mid a \not\in S_{jp} \text{ for } 1 \leq p \leq n\}$.

We claim that there is a FAMILY COVER iff the NCP inequality condition is satisfied.

First, observe that $(\bigcup_{a \in A} F_{a1}) \times \cdots \times (\bigcup_{a \in A} F_{ak}) = \times_{i=1}^k \{<i,1>, \ldots, <i,n>\}$, because there is a unique element associated with each family $S_i$ which appears in no other family $S_j$.

Suppose there is a family cover $S_{i1}, \ldots, S_{ik}$. We claim that

$$t = <<1,i_1>, \ldots, <k,i_k>> \not\in \bigcup_{a \in A} (F_{a1} \times \cdots \times F_{ak}).$$

Otherwise, there must exist $a \in A$ such that $t \in (F_{a1} \times \cdots \times F_{ak})$; i.e. $<1,i_1> \in F_{a1}, \ldots, <k,i_k> \in F_{ak}$, which means $a \not\in S_{i1}, \ldots, a \not\in S_{ik}$, which in turn implies that $S_{i1}, \ldots, S_{ik}$ is not a FAMILY COVER. A contradiction.
Suppose that \( t = \langle \langle 1, i_1 \rangle, \ldots, \langle k, i_k \rangle \rangle \notin \bigcup_{\alpha \in A} (F_{\alpha 1} \times \cdots \times F_{\alpha k}) \). We claim that \( S_{1 i_1} \cdots S_{k i_k} \) is a FAMILY COVER. Consider \( \alpha \in A \). Since \( t \notin (F_{\alpha 1} \times \cdots \times F_{\alpha k}) \), it follows that for some \( 1 \leq j \leq k \), \( \langle j, i_j \rangle \notin F_{\alpha j} \), i.e. \( \alpha \in S_{j i_j} \). This reasoning holds for all \( \alpha \in A \) and hence \( S_{1 i_1} \cdots S_{k i_k} \) is a FAMILY COVER.

Theorem 4.2: For tree schemas, VI is NP-complete.

Proof: For tree schemas \( t \in V \) is in P, hence the problem is in NP. Consider an NCP instance \( F_1, \ldots, F_m \) with \( F_i = \{ F_{i1}, \ldots, F_{ik} \} \). We now show how to build a database and view definition (forming a VI instance) such that the NCP inequality holds for the NCP instance iff there is a tuple which is not in the materialized view of the VI instance.

Construct \( k+1 \) relations \( R_0, R_1, \ldots, R_k \). \( C \) is the sole attribute of \( R_0 \). For \( 1 \leq i \leq k \), \( R_i \) has two attributes: \( C \) and \( A_i \). These relations constitute a tree database, with root \( R_0 \). The database state is constructed as follows:

\[
R_0 = \{ <i> \mid i = 1, \ldots, k \}
\]

\[
R_i = \{ <j, e> \mid e \in F_{j1}, \ldots, F_{jk} \} \quad h = 1, \ldots, k
\]

Clearly,

\[
V = (R_0 \bowtie \cdots \bowtie R_k)[B_1 \cdots B_k] = \bigcup_{j=1}^{m} (F_{11} \times \cdots \times F_{1k}).
\]

So, if there is \( v \notin V \) then the NCP inequality holds. Conversely, if the NCP inequality holds, then there is a tuple \( v \notin V \).

4.3 The VCI and VGI problems

Given a database \( D \), a view definition \( X \) and a tuple \( t \), the VCI problem is to determine whether there exists \( v \notin V(R_X+t) \). The problem is easily seen in \( \Sigma_2 \) for general databases and in NP for tree databases; we show that it is complete in these
respective cases.

The proof is done by reducing a VI instance to a VCI instance. The reduction is based on adding a new column, say $N$, to $R_1$ and setting each tuple entry in this column to $a$. Also, add a new relation $R_N$, whose sole column is $N$, containing no tuples. Clearly, $J$ on this new database is empty and so is $V$. However, adding $<a>$ to $R_N$ will yield the original view. Thus, there exists $v \notin V$ in the original database iff there exists $v \notin V(R_N+<a>)$.

The reduction shows that VCI is $\Sigma^p_2$-complete for general databases. Observe that if the original database is a tree database, then so is the new database. Hence, VCI is NP-complete for tree databases.

Since in the previous reductions the view materialization was empty before adding $<a>$, the same results hold for VGI.

4.4 The VCN problem

The VCN problem is to determine whether $V(R_1+t) \neq V$. This problem is clearly in $\Sigma^p_2$ since a nondeterministic Turing machine can guess a "new" join tuple, extract from it a "supposed" new view tuple, and consult an oracle as to whether the new view tuple belongs to the original view. We show that VCN is $\Sigma^p_2$- complete by reducing VI, which was previously shown $\Sigma^p_2$-complete, to VCN.

The VI completeness proof actually yields that VI over databases with at most three attributes per relation schema, whose attributes all have the domain $\{\text{true}, \text{false}\}$, is $\Sigma^p_2$-complete. Given such a database $D$ and some view definition $X$, we construct a database $D'$ by adding to each relation schema in $D$ a new column, say $N$. We also add a new relation $R_N$ with the sole attribute $N$.

The database $D'$ is populated thus. To each of the original tuples in $D$ the $N$ column entry is set to $a$. We also put the tuple $<a>$ in $R_N$. To each relation in $D'$ we add all the eight $\{\text{true}, \text{false}\}$ combinations for the original columns, with the $N$
column entry set to b.

Note that, by construction, \( V \) over \( D \) is identical to \( V \) over \( D' \); the \( a \) values preserve the original view and the \( b \) values add nothing as \( b \) does not appear in \( R_N \). We claim that \( V \neq V(R_N + <b>) \) in \( D' \) iff \( \exists v \in V \) in \( D \). Since all \{true, false\} combinations are present in \( D' \) in conjunction with the \( N \) column value \( b \), the addition of \(<b>\) to \( R_N \) will make the view total. So, if the view with \(<b>\) added is different then the view without \(<b>\), we conclude that the original view "missed" a tuple. Conversely, if the view prior to \(<b>\)'s addition "missed" a tuple, certainly \( V \neq V(R_N + <b>) \) following \(<b>\)'s addition. Hence we have proved the following theorem.

**Theorem 4.3:** \( VCN \) is \( \Sigma^p_2 \)-complete.

We now treat the \( VCN \) problem over tree databases.

**Theorem 4.4:** For tree schemas, \( V(R_i + t) \neq V \) is NP-complete.

**Proof:** For tree schemas, \( t \in V \) is in P, hence VI is in NP. By Theorem 4.3, VI is NP-complete over tree databases. In addition the reduction from VI to \( VCN \) preserves "treeness" because attribute \( N \) is uniformly added to each relation. Hence \( VCN \) is NP-complete over tree databases.

### 4.5 VCE and VGE

These problems are in NP, since a nondeterministic Turing machine can guess a tuple \( s \) and then check (nondeterministically) that \( s \in V(R_i + t) \). There exists an accepting computation iff \( V(R_i + t) \neq \phi \).

The problem is NP-complete by a reduction from JCE, since we may choose the view definition to be \( X = \cup R_i \).
4.6 The VGN problem for general databases

Given a database, a tuple $t$, a view definition $X$ and the materialization $V$, the VGN problem is to determine whether $t$'s addition to the database results in a different materialization. The VGN problem is in NP as one can guess a new join tuple "matching" $t$, extract a view tuple $v$ from it and check that $v$ is not already in $V$ (which is explicitly given).

**Theorem 4.5**: VGN is NP-complete.

**Proof**: Start with a 3CNF formula $F$ and construct a standard database $D$. Add a new column, $N$, to each relation in $D$ and set the $N$ entry in each tuple to $a$. Add a new tuple $<b, b, b, b>$ to each relation. Also add a new relation $R_N$ with the single column $N$. Populate $R_N$ with a single tuple $<b>$. Define a view $X'$ on the database.

As $<b>$ is in $R_N$ and $<b, b, b, b>$ is in all other relations, it follows that $V' = \{<b>\}$, and we consider it as given. Let $t = <a>$. Consider the addition of $t$ to $R_N$. Clearly, $<a> \in V(R_N + t)$ iff $F$ is satisfiable. We conclude that VGN is NP-complete.

4.7 VGN for tree databases

Yannakakis [Yan] has shown that the view materialization of a tree database can be computed by an algorithm which is polynomial in the size of the database and the size of the view materialization. Let $p$ be that polynomial.

Let $D'$ be the database resulting from the addition of $t$ to $R_i$ and $V = V(R_i + t)$. We now start producing $V'$ from $D'$ using Yannakakis' algorithm and abort the algorithm if it does not halt within $p(|D| + 1, |V|)$ steps. Since $V' \supseteq V$, the views are different iff $|V'| > |V|$. If $V' = V$ then Yannakakis' algorithm must finish within $p(|D'|, |V'|) = p(|D| + 1, |V|)$ steps. Thus if the algorithm aborted then the views...
are different. On the other hand, if the algorithm terminated then we have produced the materialization $V'$ and can easily check whether $|V| = |V'|$. The entire procedure requires essentially $p(|D|+1, |V|)$ steps.

5. FIXED SCHEMAS

In the problems previously analyzed, the database schema was part of the problem instance. This section treats the case in which the database schema is fixed over all problem instances, i.e. problem instances differ only in the tuples in the relations.

5.1 Additions into a tree database with the view contained in one of the relations

Let us consider a special case. Suppose for some $x \leq \eta$, $X \subseteq R_x$. Furthermore, assume $R_1, \ldots, R_e$ constitute a tree schema. Let $T$ be a qual tree with $R_e$ at its root.

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 $s \in R_j$ 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 the join 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 $V$.

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. 5.1. There are three cases to consider - the relation is a
root, a leaf or an internal node.

(i) Root: Suppose <55,9> is added to $R_1$ to indicate that supplier number 55 now supplies part 9. This should appear in the view provided that part 9 is mentioned in $\text{good}(R_3)$ and supplier 55 is mentioned in $\text{good}(R_2)$. In the case that <55,9> does not appear in the view, it is made part of $\text{bad}(R_1)$.

(ii) Leaf: Suppose <99,30> is added to $R_4$. First, leaves only have a good part. 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). This may happen if project 99 is mentioned in $\text{bad}(R_3)$ and in $R_5$. We first check $\text{bad}(R_3)$; if project 99 is not mentioned there, we are done. Otherwise, the next step is to check $R_5$ to verify whether project 99 is mentioned. If so, then for all tuples $t$ in $\text{bad}(R_3)$ with $t[\text{proj#}] = 99$, $t$ is moved from $\text{bad}(R_3)$ to $\text{good}(R_3)$. A change in an internal relation, such as $R_3$, may cause further effects up the tree.

(iii) Internal Node: Suppose <55,18> is added to $R_5$. First, we must decide whether it belongs to $\text{good}(R_5)$. This happens if it is compatible below; i.e. project 55 appears in both $R_4$ and $R_5$. Note that if any tuple $t$ with $t[\text{proj#}] = 55$ is in $\text{bad}(R_5)$ then $t$ is added to $\text{bad}(R_5)$ as well. If <55,18> is compatible below, it is added to $\text{good}(R_5)$. As mentioned before, changes to an internal node may propagate. We now have to check if <55,18> is compatible above - i.e. matches with tuples in both its parent and siblings relations. If part 18 is mentioned in $\text{good}(R_1)$ then no work is needed. If part 18 is mentioned only in $\text{bad}(R_1)$ then for each tuple $t$ in $\text{bad}(R_1)$ such that $t[\text{P#}] = 18$ we have to check whether $t[\text{S#}]$ is mentioned in $R_2$; if it is mentioned then $t$ is moved from $\text{bad}(R_1)$ to $\text{good}(R_1)$. Again, had $R_1$ been an internal node changes to it might have caused propagation up the tree.
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 \( \text{bad}(R_3) \) (see Fig. 5.1). It may be there because either

(i) \( t[\text{PROj}] \) is not mentioned in \( R_4 \), or

(ii) \( t[\text{PROj}] \) 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 \( \text{bad}(R_3) \) 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.

Let \( R_i \) be a node and \( R_j \) its parent in the query tree. To facilitate compatibility checks we set several balanced trees (for example the 2-3 trees described in [AHU]) and auxiliary data structures: First, the balanced tree \( T_{ij} \) which consists of \( R_i[R_i \cap R_j] \). With each entry \( v \in T_{ij} \) we associate a doubly linked list of pointers to all the good tuples \( t \in R_i \) for which \( v=t[R_j] \). We call this list the good-list of \( v \) (in \( T_{ij} \)). A similar list pointing to all the bad tuples of \( v \) is called the bad-list (of \( v \) in \( T_{ij} \)). With each tuple we
also associate a counter that tells us how many bad-lists point to this tuple, a
tuple is good iff its counter equals zero.

For the parent node $R_j$ we define the balanced tree $T_j$ whose entries are
$R_j \cap R_j$. Its lists are defined similarly. Thus, if $R_i$ has $c$ qual tree children and
a qual tree parent, then each of its tuples will be pointed at from $c+1$ lists (good
or bad).

A tuple $s \in R_j$ is compatible below if the good-list of $s[R_i]$ in $T_j$ is nonempty.
Since there are $n$ tuples in the database, this can be verified in $O(\log n)$ time.
Similarly, to find all good (bad) tuples which a tuple $t \in R_i$ supports (in $R_j$) we
access the good-list (bad-list) of $t[R_j]$ in $T_j$. The time for finding $\sigma$ such tuples
is $O(\sigma \log n)$.

To add a tuple into a relation $R_i$ we must first insert it into the $c+1$ balanced
trees of $R_j$. Then do a compatibility below check to find out whether it is good. If
it is we should remove the $R_i$-mark from all the $R_j$ tuples it supports. Thus possibly
making some tuples lose their last mark and become good. The new good
tuples can support tuples in their parent node and the effect propagates up the
tree. Observe that when an $R_j$-mark is removed form a tuple $s \in R_j$, a tuple is
moved from the bad list of $T_j$ to the good list.

The crucial point in the analysis is that in the course of a single addition
every tuple can lose at most one mark (that of its child in the unique qual tree
path from $R_i$ to the root). Thus, each tuple is accessed at most once through its
bad-list. A compatibility above test is performed for tuples that have become
good, this costs $O(\log n)$ per tuple and $O(1)$ time for each bad tuple it supports.
However, the latter cost is charged to the bad tuple found, and their costs sum
up to $O(n)$ for the entire addition. Consequently, a single addition costs
$O(n \log n)$. 
Before discussing deletions, we introduce the notion of subsumption. A tuple \( t \in R_i \) is subsumed if there is a good tuple \( t' \in R_i \) such that for \( R_j \), the parent node of \( R_i \), \( t[R_i \cap R_j] = t'[R_i \cap R_j] \). To check whether a tuple \( t \) is subsumed, we simply check whether the good-list of \( t[R_i] \) in \( T_y \) contains a tuple other than \( t \). This requires \( O(\log n) \) time.

If a tuple \( t \) is subsumed, then there is no need to check for compatibility above since the tuple \( t' \) which is already in the database supports the same tuples \( t \) does.

Deletions are similar to additions. When a tuple \( t \in R_i \) is deleted we first have to remove it from all balanced trees to which it belongs \( (O(c \log n) \) time). If \( t \) was a bad tuple we are done. If \( t \) was good, we check whether it is subsumed. If it is not, we put an \( R_i \)-mark on all (good and bad) tuples it supports. This might turn some good tuples into bad and the effect propagates up the tree. But the same reasoning as before leads us to conclude that a single deletion costs at most \( O(n \log n) \) time.

**Theorem 5.1** A tuple can be added or deleted from a tree database with \( n \) tuples in \( O(n \log n) \) time.

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) is \( O(cn + n \log n) \). We summarize this below:

**Theorem 5.2** A sequence of \( n \) additions to an initially empty database can be performed in \( O(cn + n \log n) \) time.
5.2 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 some of the tuples are only maintained implicitly.

Observation 1: Let \( D=(R_1, \ldots, R_k) \) be a database and let \( S \) be a relation such that \( S \supseteq (\bigcup_{i=1}^{k} R_i)[S] \). Then for all \( X \),
\[
(\bigcup_{i=1}^{k} R_i)[X] = (\bigcup_{i=1}^{k} R_i) \bowtie S)[X],
\]
(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 \( ((\bigcup_{i=1}^{k} R_i)[S]) \setminus S \). Then for some \( X \) it is possible that
\[
(\bigcup_{i=1}^{k} R_i)[X] \neq (\bigcup_{i=1}^{k} R_i) \bowtie S)[X].
\]

**Proof:** There exists a tuple \( u \in J(D) \) such that \( u[S]=t \). However, since \( t \not\subseteq S \),
\[
u \not\subseteq (\bigcup_{i=1}^{k} R_i) \bowtie S.\]
If there is no tuple \( v \neq u \) in \( J(D) \) such that \( v[X]=u[X] \), then,
\[
u[X] \in (\bigcup_{i=1}^{k} R_i)[X] \text{ but } u[X] \not\subseteq (\bigcup_{i=1}^{k} R_i) \bowtie S)[X].
\]

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 "treeified" by adding some templates. For the good-bad mechanism to function, by
Observations 1 and 2, each template $S$ must at least contain $\bigcap_{i=1}^{k} R_i[S]$.

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 one of the relations is changed. The template base set may grow as a result of adding a tuple to the good set of a relation, or shrink when such a tuple is deleted. The problem is parametrized according to the treelied schema structure. Let $\tau$ be the number of templates.

Let $D$ be a database schema treelied by adding $\tau$ 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 tuples into templates both using the good-bad mechanism. We have the following theorem.

Theorem 5.3: Adding $n$ tuples into an initially empty treelied database requires adding at most $O(\tau 2^n)$ 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 $s = t[R \setminus S]$. To enlarge the template we simply duplicate $s$ and in one of the copies replace the $R \setminus S$ 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 $\tau$ templates.

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

Proof: By Theorem 5.3 $N = \tau 2^n$ tuples are added, and by Theorem 5.2 this costs $O(\sqrt[N]{N})$ 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 $S$, let $R_1, \ldots, R_p$ be relations in its subtree of its qual tree. The relations $R_1, \ldots, R_p$ are a generator set for $S$ provided $S \subseteq \bigcup_{i=1}^{p} R_i$. Observe that they generate $S' = (\bigcup_{i=1}^{p} R_i)[S]$. $S'$ can then be partitioned to $\text{good}(S')$ and $\text{bad}(S')$ 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.

Let $\gamma$ be the maximum number of generators per template. The cost of tuple additions is dominated by the correct maintenance of templates. This means joining the new tuple with all the other generators, a potentially costly procedure $O(n\gamma)$. Since there are at most $n$ such additions the overall cost is $O(n\gamma^2)$.

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.)

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 $n^{2h}$ tuples.
(4) The cost of adding, in any order, \( m_1 \) tuples to a child and \( m_2 \) tuples to its sibling is exactly \( m_1m_2 \), 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 5.4** Suppose \( n \) tuples are added to an initially empty database. The time required to find the tuples which should be added to the templates is \( O\left( \tau \left( \frac{n}{\gamma} \right)^\gamma \right) \).

**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 \( \gamma \) leaves and exactly \( \frac{n}{\gamma} \) tuples per leaf, and since in the generator tree there are \( \frac{\gamma}{2^h} \) vertices at height \( h \), the total number of tuples is

\[
\sum_{h=1}^{\log_\gamma \frac{n}{\gamma}} \left( \frac{n}{\gamma} \right)^h \frac{\gamma}{2^h} = O\left( \left( \frac{n}{\gamma} \right)^\gamma \right)
\]

Since there are \( \tau \) templates the total cost is \( O\left( \tau \left( \frac{n}{\gamma} \right)^\gamma \right) \).

**Corollary:** Adding \( n \) tuples to an initially empty treefiled database requires at most \( O\left( \frac{\gamma}{\gamma-1} n \log n \right) \) time.

This is more encouraging than the corollary to Theorem 5.3 since in many practical applications \( \gamma \) 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 generate (in templates). Thus, when \( n \) non-template tuples are in the database, there remain at most \( O(\tau \left( \frac{n}{\gamma} \right)^{\gamma}) \) tuples in the database.

6. CONCLUSIONS

Several problems involving views were considered. It turns out that many view related problems are hard (\( \Sigma^P_2 \)-complete) for arbitrary databases. Even when the database structure is relatively simple (tree databases), many problems remain \( \text{NP-complete} \).

Each problem was treated for general databases and for the much simpler tree databases. We noticed the following "complexity reduction phenomenon" - NP-complete (\( \Sigma^P_2 \)-complete) problems over general schemas become polynomial (NP-complete) over tree schemas. It is also interesting to note that while query processing over tree databases is polynomial, in the sense that intermediate results can be bounded by a polynomial in the input and the final result, such is not the case for view related problems. There seems to be an inherent "information loss" which makes view problems hard even on tree databases.

We have also examined view related problems over fixed schemas, in which only the data is allowed to vary. We have presented methods to handle this case. Their complexity 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. Our results concerning fixed schemas are summarized in table 6.1 below.
<table>
<thead>
<tr>
<th></th>
<th>A single addition or deletion</th>
<th>A sequence of $n$ additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree databases</td>
<td>$O(cn + n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Cyclic databases</td>
<td>$O\left(\frac{n}{2^{t-1}} n \log n\right)$</td>
<td>$O\left(\frac{n}{2^{t-1}} n \log n\right)$</td>
</tr>
</tbody>
</table>

Table 6.1

The $\log n$ factor arises from using balanced trees. We can eliminate it by using hashing, but then the results bound the average behavior, not the worst case. We do not know whether the bounds we found are tight and we leave it as an open problem.
REFERENCES


Figure 2.1 Tree and Cyclic Schemas

The following is a tree schema:

\[ D = (R_1, R_2, R_3, R_4) \]

where

\[ R_1 = \{A\}, \ R_2 = \{A, B\}, \ R_3 = \{B, C\}, \ R_4 = \{C, D\} \]

viz. \( A \rightarrow AB \rightarrow BC \rightarrow CD \)

The following is a cyclic schema:

\[ D = (R_1, R_2, R_3) \]

where

\[ R_1 = \{A, B\}, \ R_2 = \{B, C\}, \ R_3 = \{C, A\} \]

the only valid graph for \( D \) is

\[ AB \rightarrow BC \]

\[ AC \]
$E_1$: supplier $S\#$ supplies part $P\#$;

$E_2$: each supplier may provide product support (indicated by SLEVEL);

$E_3$: project $PROJ\#$ may need part $P\#$;

$E_4$: project $PROJ\#$ has an allocated BUDGET;

$E_5$: project $PROJ\#$ is managed by MGR at LOCATION.