New Methods and Fast Algorithms for Database Normalization

JIM DIEDERICH and JACK MILTON
University of California, Davis

A new method for computing minimal covers is presented using a new type of closure that allows significant reductions in the number of closures computed for normalizing relations. Benchmarks are reported comparing the new and the standard techniques.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*normal forms*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Functional dependency, normalization, relations, third normal form

1. INTRODUCTION

Functional dependencies and normalization lie at the heart of relational database theory [8, 9]. Until recently they have generally not been used in a formal way for database design, but have been used as guidelines to assist in avoiding schemas that exhibit various well-known types of anomalies, even for nonrelational databases.

With the increased use of relational database management systems, automated tools are being developed. Hasan and York [7] have reported on a tool for helping users to specify functional dependencies. Recently, Ceri and Gottlob [5] developed a Prolog-based system for normalization, through Boyce–Codd normal form, by uniting several heretofore uncorrelated results. They also incorporated a new algorithm for projecting functional dependencies on to subrelations. Another tool written in Prolog, which gives the designer a means of prototyping a small but representative database and of executing queries and transactions, has been described by Bjornerstedt and Hulten [3]. A number of commercial database design systems support normalization. An informal survey [4] suggests that up to 80 percent of the design tools for which responses were obtained use some form of normalization, though it is not clear to what extent the designer specifies functional dependencies. Indications are that some allow this, and the standard algorithms are employed for normalization.

Tools must be responsive if they are to attract users. In the course of working with the system reported in [5], we found that in handling a modest-sized set of

Authors' address: Department of Mathematics, University of California, Davis, CA 95616.

about 40 dependencies it required over a quarter of an hour to produce a minimal cover prior to synthesis into subrelations. Performance was not the objective of this system since it was designed for small database applications and as a teaching aid. Nevertheless, its performance suggested possible problems with the standard methods. Clearly, for larger sets of dependencies, for an iterative design process where additional dependencies are added incrementally and renormalization may be required, or for database integration, better performance is required.

This paper presents new methods and fast algorithms for computing minimal covers and synthesizing relations into third normal form using a new type of closure, called an r-closure. These techniques overcome several fundamental inefficiencies in the standard algorithms for producing minimal covers. We verify our methods by presenting their theoretical foundations and validate them through benchmarks on several generic and one slightly larger set of dependencies derived from a "real" database. Even in these small sets, the benchmarks reveal up to a five-fold improvement in performance in removing redundant dependencies. For larger sets of dependencies arising in a typical database design, one would expect an even greater improvement, perhaps an order-of-magnitude or more. Significant gains are realized in other stages of normalization as well.

The complexity of the standard algorithms used in normalization has been well characterized [8]. In actual system building, complexity results often do not provide sufficiently precise estimates to determine a priori if an implementation of a given algorithm will have acceptable performance for the task at hand. Algorithms may have the same order of complexity, but vary widely in performance; or they may have different orders of complexity and not vary significantly over a certain range of input. In one example, replacing a familiar algorithm for computing the closure of a set of attributes, which is $O(n^2)$ in the number of dependencies, by a linear time algorithm does not always improve performance. In general, this leads to improvements that are far less than expected, even for n in the range of 40–100. System and language factors can also affect the relative performance of algorithms. This may be more pronounced when languages with inference engines, such as Prolog, are used. However, as our results demonstrate, a significant measure of the poor performance in normalizing relations can be traced to the standard algorithms.

Our techniques will yield improvements in computing minimal covers that are not generally captured by complexity arguments, but can be captured in terms of relative improvements of the value of the bounding coefficient k, which is implicit in the term $O(n^b)$. There are other enhancements included in our techniques that yield significant improvements that are not characterized by complexity arguments either. For example, in the standard methods for synthesizing relations, most dependencies have to be checked a second time for redundancy after grouping dependencies with equivalent left-hand sides. Our methods characterize dependencies that do not have to be rechecked. However, it is not clear that the improvements can be stated in terms of n, the total number of dependencies. (In practice, we would expect that very few dependencies would need to be retested, in which case the improvement in performance would be tantamount to eliminating this second check altogether, apart from testing whether to make the second check.)

In the remainder of this paper, terminology is presented and standard algorithms for computing minimal covers are examined in terms of their computa-

tional weak points in Section 2. In Section 3 the motivation for our approach is presented, the main ideas are illustrated in the context of an example, and our algorithms for eliminating redundant dependencies and extraneous attributes are stated. Section 4 contains statements and proofs of our main results as well as our modification of Bernstein's algorithm for synthesizing third-normal-form relations [1]. Section 5 contains the benchmark databases and benchmark results for eliminating redundant dependencies. In addition, theoretical estimates are given of the expected improvement in our techniques over the standard ones.

2. TERMINOLOGY AND STANDARD ALGORITHMS

In this section we introduce some basic terminology and examine some of the computational weak points of the standard algorithms that are used for computing minimal covers. These algorithms will be included in this paper for the sake of completeness, but for a more detailed discussion we suggest [8] and [9]. Except for the linear closure algorithm, these are the algorithms used in [5].

In the following, X, Y, and Z will be used to denote sets of attributes, and A, B, and C will be used to denote the individual attributes. We will not distinguish between the attribute A and set $\{A\}$. The union of sets X and Y will be denoted XY. Lowercase letters will be used when generic example sets of functional dependencies are considered.

A set of attributes X *functionally determines* A in a relation R, denoted $X \rightarrow A$, if whenever two tuples agree on X, then they also agree on A. For example, $\text{emp}^{\#} \rightarrow$ birthdate is a *functional dependency*, since each employee can have only one birthdate. Functional dependencies are determined by semantics and not by any individual instance of the database. Although the usual definition of functional dependency allows right-hand sides to be sets of attributes, we will restrict the right-hand side to a single attribute without loss of generality. As in [5], our discussion will be restricted to functional dependencies and so we will sometimes simply use the term dependency when we mean functional dependency. Multivalued dependencies will not be treated in this paper. F, G, and H will be used to denote sets of dependencies. Often we will use rhs to signify the right-hand side of a dependency and lhs to signify the left-hand side. F will be called *simplified* if it does not contain trivial dependencies, i.e., dependencies $X \rightarrow A$ with $A \in X$, and F has no duplicate dependencies, i.e., no dependency appears more than once.

The *closure of a set of attributes X with respect to F* will be denoted X_F^+. (Often, when no ambiguity arises, the subscript F will be omitted.) It is the set of all attributes derived from X using the dependencies F and Armstrong's axioms [8, 9]. The standard method, which can be stated independent of the axioms, for computing the closure of X is given by Algorithm A1.

Algorithm A1. Standard Closure of X with Respect to F.

Input: a set of dependencies F, a set of attributes X.
Output: CLO, the closure of X with respect to F.

0. Set CLO = X.
1. Repeat until no more attributes are added to CLO:
   For each dependency $Y \rightarrow A$ in F, if $Y \subseteq$ CLO and $A \notin$ CLO, then set CLO = CLO $\cup$ $\{A\}$.

In Algorithm A1, X_F^+ is computed starting with CLO = X and by making one or more passes over a set of dependencies F, adding the rhs of a dependency to CLO whenever the lhs is in CLO and the rhs is not. In the worst case, Algorithm A1 is $O(a*n^2)$, where n is the number of dependencies in F and a is the number of attributes, since each pass over F adds but one rhs to CLO. For example, given the set of dependencies F:

$$b \rightarrow a$$
$$c \rightarrow b$$
$$d \rightarrow c$$
$$e \rightarrow d$$

it requires four passes over F to compute $\{e\}^+$. The first pass adds d to CLO, the second adds c, and so forth.

One performance enhancement, not used in [5] because of inherent difficulties in implementing it in Prolog, is a method for computing closures in linear time, Algorithm A2 [2].

Algorithm A2. Linear Closure of X with Respect to F

Input: same as Algorithm A1.
Output: same as Algorithm A1.

1. Set CLO = X and UPDATE = X.
2. for each functional dependency fd
   let count[fd] = size of lhs of fd,
3. for each attribute A,
   let list[A] be a list of pointers to dependencies in which A is in lhs of fd.
4. while UPDATE is not empty
   5. select and remove an attribute A from UPDATE.
   6. for each dependency fd in list[A]
      7. decrement count[fd]
      8. if count[fd] = 0 then
         add rhs of fd to UPDATE and CLO if rhs is not already in CLO.

The main idea is to add the rhs of a dependency fd to CLO only when it is known that all attributes on the lhs of fd are already in CLO. Thus each dependency is accessed only one time. To discover when the lhs of a dependency is contained in CLO, lists are maintained associating each attribute with the dependencies in which it occurs on the lhs and counts are maintained to determine how many attributes on the lhs of each dependency have not yet been added to CLO.

Our benchmarks will evaluate standard and linear closure both in the standard and the new methods presented here for computing minimal covers. However, as the benchmarks demonstrate, in many if not most cases, the poor performance of the standard methods lies mainly with the number of closures computed rather than the method for computing the closures. Consequently, the benefits of linear closure are not sufficient to overcome the performance problems of the standard methods for computing minimal covers, whereas our methods reduce

the number of closures computed and thereby show substantial performance improvements.

It should be noted that the structure list[A], the list of dependencies in which A occurs on the left-hand side, required in linear closure, can also be exploited in our methods for computing minimal covers. Consequently, the additional cost of using this can be amortized over more than just linear closure.

One weak point of the definition of closure is that it loses information by allowing all attributes functionally determined by X, including X itself, into the closure of X. This will be evident in Section 3, where we introduce and discuss a new type of closure that is more restrictive and retains more information than the normal closure, but is as easily understood and implemented as the normal closure.

Given a set of dependencies F, an attribute B is *extraneous* in $X \rightarrow A$ with respect to F if $X = ZB$, $X \neq Z$, and $A \in Z_F^+$. We distinguish two kinds of extraneous attributes. If $X = ZB$, $X \neq Z$, and $B \in Z_F^+$ then B is called an *implied extraneous attribute.* If B is extraneous, but not implied, then it is called *nonimplied*, F is called *partially left-reduced* if the only extraneous attributes are nonimplied, and F is called *left-reduced* if no attributes are extraneous.

A set of dependencies H is called a *minimal cover* for a set F if each dependency in H as exactly one attribute on the right-hand side, if no attribute on the left-hand side is extraneous, and if no dependency in H can be derived from the other dependencies in H. This definition of minimal cover is the one used in [9]. Other types of covers are defined in [8].

The method used in [5] to produce third normal form relations is a synthetic method developed in [1]. Another method using decomposition can be found in [9]. The advantage of the synthetic approach is that the functional dependencies are embodied in the keys of the synthesized relations and the set of relations satisfying this property is minimized, but the computations and the algorithm are more complicated than with decomposition. Both approaches use the same initial steps to find a minimal cover. The first step eliminates extraneous attributes and the second eliminates redundant dependencies. The standard algorithms for these are given in Algorithms A3 and A4, respectively.

Algorithm A3. Eliminate Extraneous Attributes

Input: F, a set of dependencies.
Output: F', a left-reduced set of dependencies.

1. For each dependency $X \rightarrow B$ in F
   2. Let $L = X$.
   3. For each attribute $A \in X$
      if B is in $(L - A)_F^+$, then set $L = (L - A)$.
   4. Replace $X \rightarrow B$ by $L \rightarrow B$.

Algorithm A4. Eliminate Redundant Dependencies

Input: F, a left-reduced set of dependencies.
Output: H, a minimal cover for F.

0. Set $H = F$.
1. For each dependency $X \rightarrow A$ in F, let $G = H - \{X \rightarrow A\}$
   if $A \in X_G^+$ then set $H = G$.

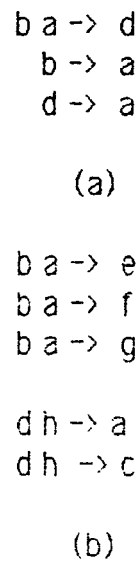

Fig. 1. Removing extraneous attributes first.

In the standard approach to producing minimal covers, elimination of extraneous attributes must precede elimination of redundant dependencies. An example illustrating this [8] is given in Figure 1(a). Until the extraneous attribute a is eliminated in the first dependency, the second dependency will not be detected as being redundant. Note that a is an implied extraneous attribute in Figure 1(a).

We will present an alternative approach in which this established order is somewhat modified, and in so doing we avoid many computations performed in Algorithm A3. In our approach it is possible to postpone eliminating nonimplied extraneous attributes until the time that redundant dependencies are eliminated. Implied extraneous attributes are removed prior to eliminating redundant dependencies in a wholesale fashion rather than one dependency at a time, as in Algorithm A3. Furthermore, our approach will also reduce the number of attributes that have to be tested as possibly being extraneous.

In particular, if the dependencies in Figure 1(a) and (b) are combined, then eight closures are computed in Algorithm A3 to remove the extraneous attribute a from the dependencies with lhs $X = \{b, a\}$, assuming that the attributes are tested in order from left to right. Most of these are redundant computations. As we shall illustrate, our approach will detect that a is the only candidate for an extraneous attribute in $a\ b$, so that only one closure will be computed for this set of four dependencies, rather than one for each dependency and each attribute on the lhs. The nonimplied extraneous attribute h in the next to last dependency of Figure 1 will automatically be handled in removing redundant dependencies.

A major weakness of Algorithm A4, for eliminating redundant dependencies, is that a good deal of work is repeated. For example, given the set dependencies F in Figure 2(a), the closure of a is computed four times, each time with a different dependency removed from F. Many of the steps in computing these closures are similar or the same, and so are the results when the first, third, and

```
a -> b
a -> c
a -> d
a -> e

c -> e
c -> f
```

(a)

```
g -> b
g -> d
```

(b)

Fig. 2. Set including redundant dependencies.

fourth dependencies are removed, respectively. Another computational weakness is that this process does not always compute the closure of the left-hand sides since a dependency is always removed from the set prior to the computation. Consequently, the closure of the lhs would have to be computed separately when needed for grouping dependencies in one of the subsequent stages of synthesizing the relations. However, in our approach the computation of the normal closure follows directly from the closure we introduce and does not need to be repeated.

Some savings in effort can be effected by ignoring any dependency for which the attribute on the rhs appears nowhere else in the set F, for such dependencies cannot possibly be redundant. Consequently, the first and third dependencies in Figure 2(a) could be bypassed in eliminating redundant dependencies. However, this simple approach is not the best. For example, if the dependencies in Figure 2(b) are added to those in (a), then no dependency could be skipped, since any attribute appearing more than once could be in a redundant dependency, such as e in the fourth dependency. No savings would occur even though none of the four dependencies involving b or d is redundant. As we shall demonstrate in the next section, instead of computing eight closures—four for a and two each for c and g—to eliminate the single redundant dependency, only one closure needs to be computed for this entire set of dependencies using our techniques.

3. MOTIVATION, DEFINITIONS, AND NEW MINIMAL COVER ALGORITHMS

In this section we present, prior to the main algorithms, some motivation for our definitions and for our methods of finding minimal covers in a more efficient fashion.

3.1 Degree of Fullness of a Set of Dependencies

Our techniques work best when the set of dependencies is a type one would expect to encounter in practice rather than a pathological type sometimes

encountered in theory. Consequently our methods optimize for the expected sets of dependencies rather than unusual cases. To characterize sets of dependencies occurring in practice we need a few definitions.

Given F, a set of dependencies $DepForLHS(X)$ will denote the nonempty subset of dependencies in F with left-hand side X. Also we let $\mathrm{LSH}(F)$ denote the set of all left-hand sides of dependencies in F, that is, $\mathrm{LHS}(F) = \{X \mid \text{there is a set DepForLHS}(X)\}$. We will assume that no duplicate dependencies appear in DepForLHS(X). However, in practice, after extraneous attributes have been removed, duplicate dependencies can arise. For example, if F contains $ab \rightarrow c$ and $a \rightarrow c$, then b will be extraneous and when removed from the lhs leaves $a \rightarrow c$ duplicated. Since duplicate (and trivial) dependencies can easily be removed whenever the sets DepForLHS(x) are created, we will assume in subsequent discussions that the set of dependencies is simplified. Nevertheless, we will indicate how our algorithm for eliminating redundant dependencies can easily be modified to handle duplicate and trivial dependencies. Let $|S|$ denote the cardinality of the set S.

If an attribute A occurs only on the right-hand size of dependencies in F, A will be called an *ro-attribute*, the prefix *ro* signifying *right only*. Attributes that are not ro-attributes will be called *rl-attributes*, since they appear on the left side of at least one dependency and may appear on the right-hand side of other dependencies. The structure used for the linear closure Algorithm A2 is sufficient for determining whether an attribute A is an ro or rl-attribute, since list[A] is empty for the former and not empty for the latter.

In addition we define the sets

$$\mathrm{RO}_X = \{A \mid X \rightarrow A \text{ is in } F \text{ and } A \text{ is an ro-attribute}\},$$
$$\mathrm{RL}_X = \{A \mid X \rightarrow A \text{ is in } F \text{ and } A \text{ is an rl-attribute}\},$$
$$\mathrm{R}_X = \mathrm{RO}_X \cup \mathrm{RL}_X.$$

The subscript X will be suppressed when no ambiguity arises from doing so.

To characterize databases normally found in practice, we first note that at the conceptual level databases consist of entities and relationships. Usually an entity has a set of attributes, X, representing its primary key and other attributes representing its properties. For example, in Figure 3, emp# is a key for the entity employee, and the right-hand sides in DepForLHS(emp#) are employee's attributes. In practice, one would therefore expect that many of the sets DepForLHS(X) would represent entities in the database, and those that did would contain several dependencies, one for each nonkey property. Even for a DepForLHS(X) corresponding to a relationship rather than an entity, one might find that it contains several dependencies since relationships often have properties also. For example, $X = \{student^{\#}, class^{\#}\}$ represents a relationship between student entities and class entities and might have attributes for *semester-taken* and *grade-received*. Certainly some of the sets DepForLHS(X) will consist of a single dependency in case the relationship has no attributes and in the case of a second key or a lexicon in the terminology of [10]. The last dependency in Figure 3 is an example of a second key with a single dependency in DepForLHS(dept-name).

In summary, then, we would expect that on the average the sets DepForLHS(X) represent entities and therefore contain several dependencies. The average

```
emp# -> emp-name
emp# -> address
emp# -> title
emp# -> salary
emp# -> dept#
emp# -> manager

dept# -> dept-name
dept# -> manager
dept# -> building#
dept# -> office#

dept-name -> dept#
```

Fig. 3. Example set of dependencies for Section 3.

number of dependencies per left-hand side is ($|F|/|\mathrm{LHS}(F)|$). This is the first factor needed to characterize sets of dependencies found in practice. For any database that occurs in practice we would expect this average to be at least 2, and more likely in the range of 5 to 50. For the dependencies in Figure 3, the average is 11/3.

A second factor stems from the expectation that most attributes are facts about entities or relationships, but do not represent entities or relationships by participating as key attributes. In our terminology, we expect most attributes to be ro-attributes and not rl-attributes. We let ro be the average number of dependencies with ro-attribute right-hand sides per DepForLHS(X) and rl be the average with rl-attribute right-hand sides. In a typical database rl $\neq$ 0. Now dividing the average number of attributes per left-hand side by the average number of rl attributes gives a measure we call *the degree of fullness of a set of dependencies F:*

$$\mathrm{degf}(F) = \frac{|F|}{\mathrm{rl}^* \,|\, \mathrm{LHS}(F)\,|}.$$

Clearly, the more dependencies per left-hand side and the fewer rl-attributes per left-hand side, the higher the value of degf(F). In other words, we measure, on the average, how full DepForLHS(X) is with dependencies and ro-attributes.

For the example in Figure 3, rl = 3/3 and so degf(F) = 11/3. In Section 5, on the benchmarks, we will show that the improvements of our algorithms over the standard algorithms for eliminating redundant dependencies can be estimated by the degf(F). Since, by definition (ro + rl) $*$ $|\,\mathrm{LHS}(F)\,| = |\,F\,|$, we can reformulate degf(F) to be

$$\mathrm{degf}(F) = \frac{\mathrm{ro} + \mathrm{rl}}{\mathrm{rl}}.$$

3.2 r-closures

In examining Algorithm A4, we observed that many computations of the closure for dependencies with the same lhs X are redone each time. In the example in Figure 3 the closure of emp# will be computed six times, each time with a different dependency deleted from F. In order to reduce these computations, we introduce

the following concept:

An *r-closure of X with respect to a set of dependencies G*, denoted $X_G^{\wedge}$, is the set of all attributes A such that $Y \rightarrow A$ is in G and Y is in X_G^+. $X_G^{\wedge}$ is the set of all right-hand sides derived from X with respect to G. Often G will be a proper subset of a set of dependencies F.

As an example, let $G = F$, the set of dependencies in Figure 3. Let $X = \{\text{emp}^\#\}$. Then the usual closure is

$\{\text{emp}^\#\}_G^+ = \{\text{emp}^\#, \text{emp-name, address, title, salary, dept}^\#, \text{manager, dept-name, building}^\#, \text{office}^\#\}$

while the r-closure is

$\{\text{emp}^\#\}_G^{\wedge} = \{\text{emp-name, address, title, salary, dept}^\#, \text{manager, dept-name, building}^\#, \text{office}^\#\}$.

Note that $\text{emp}^\#$ is contained in the normal closure but not in the r-closure since for no Y in $\{\text{emp}^\#\}_F^+$ is $Y \rightarrow \text{emp}^\#$ in G.

As another example,

$\{\text{dept}^\#\}_G^+ = \{\text{dept-name, manager, building}^\#, \text{office}^\#, \text{dept}^\#\}$
$\{\text{dept}^\#\}_G^{\wedge} = \{\text{dept-name, manager, building}^\#, \text{office}^\#, \text{dept}^\#\}$

the r-closure includes $\text{dept}^\#$ because dept-name is in $\{\text{dept}^\#\}_G^+$ and dept-name $\rightarrow$ $\text{dept}^\#$ is in G.

A simple modification of standard and linear closure, Algorithms A1 and A2, can be used to generate the r-closure of X. Start with $X^{\wedge}$ empty, and each time a dependency is examined that has its left-hand side in CLO, add its rhs to $X^{\wedge}$, if it is not currently in $X^{\wedge}$. It is easy to see that for any set of attributes X, $X_G^{\wedge} \subseteq X_G^+$, and the only elements of X_G^+, possibly missing from $X_G^{\wedge}$ are elements of X. This will prove very useful in eliminating implied extraneous attributes. For eliminating redundant dependencies, it is the manner in which r-closures are used that is important, as illustrated next.

The examples above do not reveal the best use of r-closures. The r-closure of a set of attributes with respect to the complete set of dependencies is not the interesting application of this concept. To illustrate how r-closures can be used to eliminate redundant dependencies, let F be the set of dependencies in Figure 3, let $X = \{\text{emp}^\#\}$, and let $G = F - \text{DepForLHS}(X)$. Then compute the r-closure of X and its rl-attributes with respect to G, i.e., for

$(X \cup \text{RL}_X) = \{\text{emp}^\#, \text{dept}^\#\}$

we get

$(X \cup \text{RL}_X)_G^{\wedge} = \{\text{dept-name, manager, building}^\#, \text{office}^\#, \text{dept}^\#\}$.

Note that adding ro-attributes to $(X \cup \text{RL}_X)$ has no effect on the result. The r-closure just computed has an interesting comparison with

$\{\text{emp}^\#\}F^+ = \{\text{emp}^\#, \text{emp-name, address, title, salary, dept}^\#, \text{manager,}$
$\text{dept-name, building}^\#, \text{office}^\#\}$

since they both iuse the dependencies in DepForLHS(emp#), but in quite different ways. With the r-closure they are only used to determine the set of attributes over which the r-closure is computed, while with the regular closure they are used in its computation.

Two important properties of r-closures of $(X \cup \mathrm{RL}_X)$ for removing redundant dependencies are: If F is simplified, i.e., F has no duplicate or trivial dependencies, then

(P1) $X \rightarrow A$ is not redundant in F if $A \notin (X \cup \mathrm{RL}_X)\hat{}_G$

(P2) $X \rightarrow A$ is redundant if $A \in (X \cup \mathrm{RL}_X)\hat{}_G \cap \mathrm{RO}_X$

The proofs of (P1) and (P2) are postponed until Section 4, Theorem 4.4. Using these properties in our example for the set DepForLHS(emp#) in Figure 3, the first four dependencies are not redundant since they satisfy (P1), the sixth is redundant as it satisfies (P2), and only the fifth dependency, where $A \in (X \cup \mathrm{RL}_X)\hat{}_G \cap \mathrm{RL}_X$, needs to be checked using the standard technique of Algorithm A4, step 1. Hence, six dependencies have been checked computing one r-closure and one normal closure.

Likewise, for the set of dependencies in Figure 2, only one r-closure needs to be calculated, that of $\{a, c\}$. The result yields the fourth dependency as redundant. Since the remaining sets of DepForLHS(X), for $X = \{c\}$ and $X = \{g\}$, have only ro-attributes, no r-closures need to be calculated for removing redundant dependencies.

In addition to detecting redundant dependencies and reducing the number of computed closures, r-closures can be used to (1) reduce the number of attributes and the number of dependencies considered in eliminating extraneous attributes; (2) derive the normal closure of an lhs X; (3) detect those left-hand sides that may have equivalent left-hand sides; and (4) reduce the number of closures required after grouping when synthesizing relations.

In essence, the r-closure of a left-hand side and its rl-attributes is the analytical embodiment of a heuristic that reduces the number of computed closures but does not guarantee in every case that one can avoid checking a dependency as redundant using the normal method of Algorithm A4, step 1. Also the r-closure cannot detect trivial or duplicate dependencies. (Examples are easy to construct and are left to the reader.) But since these are easily handled in other ways, little is sacrificed given the power of r-closures.

It is interesting to note that the r-closure, as it is used above, is not invariant over equivalent sets of dependencies. Figure 4(a) and (b) give two equivalent sets of dependencies and the r-closures of $\{c, a\}$, with $X = \{c\}$ and $\mathrm{RL}_X = \{a\}$, in each case with respect to $G = F - \mathrm{DepForLHS}(c)$. In Figure 4(a), the r-closure of $\{c, a\}$ contains the ro-attribute b, which signifies that $c \rightarrow b$ is redundant, while in Figure 4(b) the dependency is not redundant and the r-closure does not contain b.

3.3 Fast Methods for Eliminating Redundant Dependencies

In this section we first illustrate and then present our algorithm, Algorithm 3.1, for eliminating redundant functional dependencies under the assumption that extraneous attributes have been eliminated. The algorithm for eliminating extra-

Fig. 4. r-closures are not invariant.

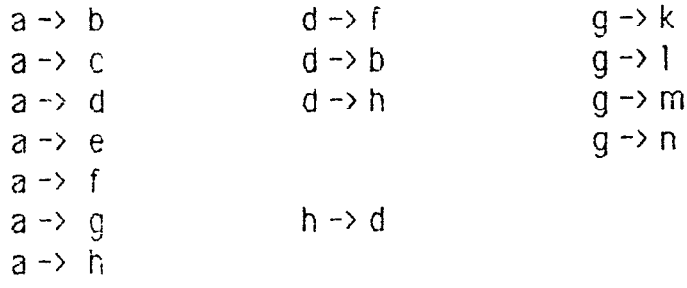

Fig. 5. Example set of dependencies for Section 3.3.

neous attributes will be presented in the next subsection. The example set of dependencies F in Figure 5 is simplified and left-reduced.

Instead of computing a single r-closure of $X \cup \mathrm{RL}_X$ for each DepForLHS(X), as suggested in the example of the previous section, we will take a slightly different approach. Both approaches will be benchmarked in Section 5.

Since each DepForLHS(X) will be treated in turn, we first select DepForLHS(a). We let crc(a) represent a cumulative r-closure in the course of treating the dependencies one at a time. Initially we set crc(a) = $\varnothing$, the empty set. Let $G = F -$ DepForLHS(a). Two passes will be made over DepForLHS(a), with each dependency considered one at a time. Figure 6 shows the actions taken and the resulting value of crc(a) from handling each dependency. An explanation follows.

On the first pass for each $X \rightarrow A$:

—If $A \in$ crc(X), the current value of crc(X), then the dependency is redundant and is deleted from F.

—If $A \notin$ crc(X) and A is an ro-attribute, then go to the next dependency.

—If $A \notin$ crc(X) and A is an rl-attribute, then compute the r-closure of X and all rl-attributes encountered so far including A and set crc(X) equal to the result.

Pass 1:

fd	rhs type	test/result	action(s)	resulting crc(a)
a -> b	ro	b ∈ crc(a)? no	-	∅
a -> c	ro	c ∈ crc(a)? no	-	∅
a -> d	rl	d ∈ crc(a)? no	compute $\{a,d\}_G^{\wedge}$	{f,b,h,d}
a -> e	ro	e ∈ crc(a)? no	-	"
a -> f	ro	f ∈ crc(a)? yes	delete a ->f	"
a -> g	rl	g ∈ crc(a)? no	compute $\{a,d,g\}_G^{\wedge}$	{f,b,h,d,k,l,m,n}
a -> h	rl	h ∈ crc(a)? yes	delete a -> h	"

At the end of the first pass the final value of crc(a) has been computed.

Pass 2.

a -> b	ro	b ∈ crc(a)? yes	delete a -> b
a -> c	ro	c ∈ crc(a)? no	
a -> d	rl	d ∈ crc(a)? yes	compute $\{a\}_{F'}^{+}$, where F' = F - {a ->d}, and delete a -> d if d is in this closure.
a -> e	ro	e ∈ crc(a)? no	
a -> g	rl	g ∈ crc(a)? no	

Fig. 6. Removing redundant dependencies using r-closures.

On the second pass for each $X \rightarrow A$:

—If $A \notin \text{crc}(X)$, then go to the next dependency.

—If $A \in \text{crc}(X)$ and A is an ro-attribute, then delete the dependency from F.

—If $A \in \text{crc}(X)$ and A is an rl-attribute, then the dependency must be tested for redundancy using the standard or linear method of computing the closure of X with $X \rightarrow A$ deleted from F.

Note that both b and f are ro-attributes, but that on the first pass only $a \rightarrow f$ is detected as redundant because the r-closure for the rl-attribute d is handled after b but before f. Consequently, the second pass is required to eliminate the redundant dependency $a \rightarrow b$.

This process is repeated for DepForLHS(X) for $X = d$, $X = g$, and $X = h$. One r-closure will be computed for $X = d$, none when $X = g$, and one when $X = h$. Upon completion only 5 closures total, including r-closures and normal closures, will have been calculated, versus 15 for Algorithm A4. For sets of dependencies for which degf(F) is large, the savings can be substantial. As will be demonstrated in Section 4, the number of closures computed in the remainder of the synthesis of 3NF relations will be drastically reduced.

Algorithm 3.1. Fast Elimination of Redundant Dependencies

Input: F, a simplified, left-reduced set of dependencies.
Output: H, a minimal cover for F.

0. Let $H = F$.
1. For each DepForLHS(X) do
   2. Let $G = H -$ DepForLHS(X).
      Let crc(X) = [].
      Let Attr = X.

(Pass 1)

   3. For each dependency $X \rightarrow A$ in DepForLHS(X) do
      If $A \in$ crc(X), delete $X \rightarrow A$ from H, otherwise
        if A is an rl-attribute, then
          let Attr = Attr $\cup \{A\}$,
          let crc(X) = $\text{Attr}_{\hat{G}}$

(Pass 2)

   4. For each dependency $X \rightarrow A$ in DepForLHS(X) $\cap$ H do
      5. If $A \in$ crc(X), then
           if A is an ro-attribute
           or
           if A is an rl-attribute and
             A is in X^+ (with respect to $H - \{X \rightarrow A\}$),
               then delete $X \rightarrow A$ from H.

It should be observed that Algorithm 3.1 can be modified in several ways. For example, in the first pass, keeping a running r-closure using linear closure techniques can reduce the number of r-closures to one. As each new rl-attribute is encountered, it is added to UPDATE in the linear closure algorithm A2. We did not test this in our benchmarks.

Another approach is simply to compute $\text{crc}(X) = (X \cup \text{RL}_X)_{\hat{G}}$ as the sole activity of the first pass, replacing step 3. The advantage is its simplicity and the elimination of additional r-closures on the first pass. The disadvantage is the possibility of needing to compute additional closures during the second pass. For instance, in Figure 5, $a \rightarrow h$ would not be deleted on the first pass, and since h is an rl-attribute, another closure would have to be calculated on the second pass, as was the case with the dependency $a \rightarrow d$. This approach is benchmarked in Section 5 and is denoted as Algorithm 3.1(a).

Algorithm 3.1(a). Fast Elimination of Redundant Dependencies

Same as Algorithm 3.1 except replace step 3 by

(Pass 1)

   3. a. For each dependency $X \rightarrow A$ in DepForLHS(X) do
           if A is an rl-attribute, then
             let Attr = Attr $\cup \{A\}$.
      b. If Attr $\neq \emptyset$, then let crc(X) = $(X \cup \text{Attr})_{\hat{G}}$.

To allow for eliminating trivial and duplicate dependencies if they have not been removed when forming DepForLHS(X), modify Algorithm 3.1 as follows, to give Algorithm 3.1(b).

Algorithm 3.1(b). Fast Elimination of Redundant Dependencies

Same as Algorithm 3.1 except:
Input: F, a left-reduced set of dependencies.

In step 2 add the statement:
let CLO = X.

Replace step 3 by
(Pass 1)
3. For each dependency $X \rightarrow A$ in DepForLHS(X) do
if $A \in$ CLO, delete fd from H,
otherwise,
let CLO = CLO $\cup$ $\{A\}$.
If A is an rl-attribute, then
let Attr = Attr $\cup$ $\{A\}$,
let crc(X) = $\text{Attr}_G^{\wedge}$,
let CLO = CLO $\cup$ crc(X).

It is easy to see that trivial dependencies will be removed because X is in CLO, and duplicate dependencies will be removed because every rhs A is added to CLO.

Finally, if F is partially left-reduced instead of having all extraneous attributes removed, it is necessary to add a step in Algorithm 3.1. The additional step computes an r-closure when DepForLHS(X) has no rl-attributes. This allows dependencies that have nonimplied extraneous attributes to be detected as redundant. This will be illustrated in an example in the next subsection. The resulting algorithm is Algorithm 3.1(c).

Algorithm 3.1(c). Fast Elimination of Redundant Dependencies

Same as Algorithm 3.1 except:
Replace Input by:
Input: F, a simplified, partially left-reduced set of dependencies.
Insert the following step between 3 and 4:

If Attr = $\emptyset$, at the end of step 3, then no r-closure has been computed,
so
let crc(X) = $X_G^{\wedge}$.

3.4 Fast Methods for Eliminating Extraneous Attributes and Redundant Dependencies Combined

In light of Algorithm 3.1, we can state an algorithm that combines eliminating extraneous attributes and redundant dependencies. An example will follow the statement of the algorithm.

Two important properties of r-closures of $(X \cup \text{RL}_X)$ for eliminating extraneous attributes, are

(P3) If $C \in X$ and C is an implied extraneous attribute, then $C \in (X \cup \text{RL}_X)_G^{\wedge}$.

(P4) If $C \in X$ and $C \notin (X \cup \text{RL}_X)_G^{\wedge}$, then either C is not extraneous in $X \rightarrow A$ or $X \rightarrow A$ is redundant in F.

The proofs of (P3) and (P4) are postponed until Section 4.3, Theorem 4.11.

The main idea is to remove all implied extraneous attributes from each lhs. Individual dependencies do not have to be treated when removing implied

extraneous attributes, and only attributes that satisfy (P3) have to be checked as possible extraneous attributes. After all implied extraneous attributes have been eliminated from F, then, according to property (P4), if a dependency has an extraneous attribute, which must be nonimplied extraneous, that dependency is redundant and will be detected as a candidate for deletion when redundant dependencies are removed. Therefore the only attributes in X that need to be removed prior to eliminating redundant dependencies are the implied extraneous attributes.

Algorithm 3.5 Fast Elimination of Extraneous Attributes and Redundant Dependencies

Input: F, a simplified set of dependencies.
Output: H, a minimal cover for F.

0. For each DepForLHS(X) do
   If $|X| > 1$ then
   1. Compute $rc(x) = (X \cup RL_X)\hat{}_G$. "compute a single r-closure"
   2. While $|X| > 1$, for each $B \in X$
      if $B \in X \cap rc(X)$, then
      set $X' = X - B$
      compute X'^{+}_{F}
      if $B \in X'^{+}_{F}$ then set $X = X'$.
   3. If X has changed in step 2, then
      replace X by its new value in all dependencies in DepForLHS(X).
4. Reconstitute the sets DepForLHS(X), eliminating duplicate dependencies.
5. Remove redundant dependencies using Algorithm 3.1(c).

To illustrate Algorithm 3.5, let F be the dependencies in Figure 7.

For DepForLHS(a b)

In step 1: $rc(a\ b) = \{a, b, d\}\hat{}_G = \{a, h, g\}$

In step 2: only a is tested for being extraneous and is removed.

In step 3: a b is replaced by b for all dependencies in DepForLHS(a b).

For DepForLHS(a b c)

In step 1: $rc(a\ b\ c) = \{a, b, c, d\}\hat{}_G = \{d, e, f, g, a, h\}$

In step 2: only a is tested for being extraneous and is removed.

In step 3: a b c is replaced by b c for all dependencies in DepForLHS(a b c).

For DepForLHS(b n)

In step 1: $rc(b\ n) = \{b, n\}\hat{}_G = \{d, e, f, g, a, h\}$

In step 2: no attribute is tested for being extraneous.
(note that n is nonimplied extraneous)

In step 3: no change in the lhs.

In step 4 new sets DepForLHS(X) are formed, eliminating duplicates and giving the set shown in Figure 8.

Note that $b\ c \rightarrow d$ has a nonimplied extraneous attribute c that will be removed as redundant in step 5 since d will be in the r-closure of $X \cup RL$, where $X = \{b\ c\}$ and $RL = \{d\}$. In similar fashion, n is nonimplied extraneous. But note that since DepForLHS(bn) has no rl-attributes, it is necessary to compute an r-closure as done in Algorithm 3.1(c) to eliminate the redundant dependency

```
a b -> d        b -> a
a b -> e        b -> h
a b -> f        b -> g
a b -> g
                a b c -> d
d -> a          a b c -> j
                a b c -> k
bn -> h
```

Fig. 7. Dependencies with some extraneous attributes.

```
b -> a      b c -> d
b -> d      b c -> j
b -> e      b c -> k
b -> f
b -> g      d -> a
b -> h      bn -> h
```

Fig. 8. Implied extraneous attributes removed.

$bn \rightarrow h$. Also, note that 19 closures are computed using Algorithm A3 but only 5 are needed using Algorithm 3.5.

It is possible to remove some redundant dependencies that have ro-attribute right-hand sides in the course of eliminating implied extraneous attributes. This can be done between steps 1 and 2 in Algorithm 3.5, but it is not clear that there would be sufficient savings to warrant an additional pass.

4. THEOREMS

In this section we will establish the theorems needed to justify the algorithms already presented. In addition, our modification of Bernstein's algorithm for synthesizing 3NF relations and related theorems will be presented. The subsections contain theorems pertaining to specific algorithms.

In the following let F denote a set of functional dependencies and let $G = F - \text{DepForLHS}(X)$. F is not assumed to be the initial set of dependencies in the theorems, as it was in the examples.

4.1 Basic Theorems and Theorems for Algorithm 3.1

LEMMA 4.1

$$(X \cup \text{RL}_X)_G^+ \cup \text{RO}_X = X_F^+. \tag{1}$$

PROOF. RO attributes do not participate in producing new elements for the closure and each $X \rightarrow A$, A an rl-attribute, can be discarded after A is added to the closure. □

The next lemma gives a slight variation on the definition of r-closure for the context in which it is used in the algorithms and theorems.

LEMMA 4.2. $A \in (X \cup \text{RL}_X)_{\hat{G}}$ *if and only if*

for some Y, $Y \neq X$, $Y \rightarrow A$ *is in* DepForLHS(Y), *and* $Y \subseteq X_F^+$. (2)

PROOF. If $A \in (X \cup \mathrm{RL}_X)^{\hat{}}_G$, then by definition (a) there is a $Y \rightarrow A$ in G and (b) $Y \in X_G^+$. By (a) $Y \neq X$ and by (b) $Y \subseteq X_F^+$, and thus (2) holds.

Conversely, assume (2) holds. Thus the attributes of Y are rl-attributes, and since $Y \subseteq X_F^+$ by (1) of Lemma 4.1, $Y \subseteq (X \cup \mathrm{RL}_X)_G^+$. Since $Y \neq X$, $Y \rightarrow A$ is in G, and by definition, then $A \in (X \cup \mathrm{RL}_X)^{\hat{}}_G$. □

Although G contains no dependency with left-hand side X, $(X \cup \mathrm{RL}_X)^{\hat{}}_G$ may differ from $(\mathrm{RL}_X)^{\hat{}}_G$ since G may contain dependencies with left-hand sides strictly in X or that intersect X. The first theorem shows how to derive closures from r-closures.

THEOREM 4.3. *The closure of X can be recovered directly from the* r*-closure of $(X \cup \mathrm{RL}_X)$ with respect to G, that is,*

$$X_F^+ = (X \cup \mathrm{RL}_X)^{\hat{}}_G \cup X \cup \mathrm{R}_X. \tag{3}$$

PROOF. By Lemma 4.2, if $A \in (X \cup \mathrm{RL}_X)^{\hat{}}_G$, then there is a $Y \rightarrow A$ in F and $Y \subseteq X_F^+$ so $A \in X_F^+$. Thus the left-hand side of (3) contains the right-hand side.

To show containment in the opposite direction, let $A \in X_F^+$, $A \notin X$ and $A \notin \mathrm{R}_X$. Thus there is a $Y \rightarrow A$ in F such that $Y \subseteq X_F^+$. Since $A \notin \mathrm{R}_X$, it must be that $Y \neq X$ and, by Lemma 4.2, $A \in (X \cup \mathrm{RL}_X)^{\hat{}}_G$. □

The next theorem is used to determine when a dependency is redundant and establishes properties (P1) and (P2). It presumes that dependencies will be removed one at a time and that F will be modified to reflect each deletion.

THEOREM 4.4. *Let F be simplified. A dependency $X \rightarrow A$ is redundant in F if and only if either*

$$A \in \mathrm{RO}_X \cap (X \cup Z)^{\hat{}}_G, \quad \textit{for some} \quad Z \subseteq \mathrm{RL}_X. \tag{4}$$

$$A \in \mathrm{RL}_X \cap (X \cup Z)^{\hat{}}_G, \quad \textit{for some} \quad Z \subseteq \mathrm{RL}_X - \{A\}. \tag{5}$$

PROOF. Let $\mathrm{RL}' = \mathrm{RL} - \{A\}$, where the X subscript is suppressed. Suppose that (4) or (5) holds. Then $A \in (X \cup \mathrm{RL}')^{\hat{}}_G$, since if $A \in \mathrm{RO}$, then $Z \subseteq \mathrm{RL} = \mathrm{RL}'$ or if $A \in \mathrm{RL}$, then $Z \subseteq \mathrm{RL}'$. By definition of the r-closure, there is a $Y \rightarrow A$ in G and $Y \in (X \cup \mathrm{RL}')_G^+$, and therefore $A \in (X \cup \mathrm{RL}')_G^+$. Let $F' = F - \{X \rightarrow A\}$. Clearly, $(X \cup \mathrm{RL}')_G^+ \subseteq (X \cup \mathrm{RL}')_{F'}^+ = X_{F'}^+$ since $A \notin \mathrm{RL}'$. Thus $A \in X_{F'}^+$, and $X \rightarrow A$ is redundant.

Conversely, let RL' and F' be defined as above and suppose that $X \rightarrow A$ is redundant in F. Since trivial dependencies are not allowed, there is some $Y \subseteq X_{F'}^+$, with $Y \rightarrow A$ in F'. Also $Y \neq X$, since F contains no duplicates. Thus $Y \rightarrow A$ is in G.

Since $(X \cup \mathrm{RL}')_{F'}^+ = X_{F'}^+$, $Y \subseteq (X \cup \mathrm{RL}')_{F'}^+ = (X \cup \mathrm{RL}')_G^+$ and since $Y \rightarrow A$ is in G, then by definition, $A \in (X \cup \mathrm{RL}')^{\hat{}}_G$. □

THEOREM 4.5. *Let H be the cover produced by Algorithm* 3.1. *Then H is a minimal cover for F.*

PROOF. Let H denote the minimal cover as it has evolved from F just prior to treating the dependencies in DepForLHS(X).

First we will show that no nonredundant dependencies are removed. Suppose that $X \rightarrow A$ is not a redundant dependency but is removed at some point during consideration of the set DepForLHS(X). Then it must be the case that $A \in \text{crc}(X) \subseteq (X \cup \text{RL}_X)^{\hat{}}_G$ in order for this dependency to be removed by the algorithm. If A is an ro-attribute, then (4) of Theorem 4.4, where $F = H$, holds with $Z = \text{RL}$, and so $X \rightarrow A$ is redundant contrary to the assumption. If A is an rl-attribute, then if the dependency is removed in pass 1, step 3, $A \in (X \cup Z)^+_G$, where $Z \subseteq \text{RL} - \{A\}$, since A is not included in the set Attr at this point in the algorithm. Thus by (5) of Theorem 4.4, $X \rightarrow A$ is redundant. If the dependency is removed in pass 2, step 5, then it is removed as redundant in the standard way, and therefore is redundant.

Next we show that redundant dependencies are removed.

Suppose $X \rightarrow A$ is redundant in H. Then (4) or (5) holds with $F = H$. If either (4) or (5) holds when this dependency is encountered in the first pass of the algorithm, step 3, it would be removed. At the end of the first pass $\text{crc}(X) = (X \cup \text{RL}_X)^{\hat{}}_G$, and so during the second pass either case (4) or (5) would guarantee that $A \in \text{crc}(X)$. If A is an ro-attribute this dependency is removed, and if not, the dependency is treated in the standard way and would be removed from H. □

4.2 Theorems and Synthesis of Relations

In this section we present our modification of Bernstein's method for synthesizing 3NF relations, Algorithm A5.

Algorithm A5. Bernstein's 3NF Algorithm

Input: F, a set of functional dependencies.
Output: A collection of relations in third normal form.

1. Eliminate extraneous attributes from the left-hand sides of F.
2. Find a minimal cover H for F.
3. Partition H into groups such that the dependencies in each group have the same left-hand sides.
4. Let $J = [\,]$. Regroup by combining old ones if their left-hand sides are equivalent in H. If X and Y are left-hand sides that have been combined, then add $X \rightarrow Y$ and $Y \rightarrow X$ to J and delete from H $X \rightarrow A$ if $A \in Y$ and $Y \rightarrow B$ if $B \in X$.
5. To avoid reintroducing transitive dependencies in the relations formed by the attributes in the groups in (6), find an $H' \subset H$ such that $(H' \cup J)^+ = (H \cup J)^+$ and no proper subset of H' has this property. Add each dependency in J to its group.
6. For each group form a relation consisting of attributes appearing in that group.

The next two theorems will be used to justify our modification of Algorithm A5.

THEOREM 4.6. *Let F be simplified and partially left-reduced. If X has an equivalent left-hand side Y, that is, if*

$$\text{DepForLHS}(Y) \neq \emptyset,\ Y \neq X,\ Y \subseteq X^+_F \quad \textit{and} \quad X \subseteq Y^+_F, \tag{6}$$

then

$$X \cap (X \cup \text{RL}_X)^{\hat{}}_G \neq \emptyset. \tag{7}$$

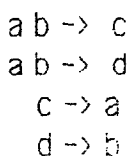

Fig. 9. Equivalent keys but no equivalent left-hand sides.

PROOF. Suppose (6) holds. Then there is an A with $A \in X$ and $A \notin Y$, since F has no dependencies with implied extraneous attributes and $Y \neq X$. Since $A \in Y_F^+$, $A \notin Y$, there is a $Z \subseteq Y_F^+$ such that $Z \rightarrow A$ is in F. Also, $Z \neq X$ since $A \in X$ would make $Z \rightarrow A$ a trivial dependency, and such dependencies are presumed removed from F. Thus $Z \rightarrow A$ is in G. By (6), $X_F^+ = Y_F^+$, and so $Z \subseteq X_F^+$ and thus, by Lemma 4.1, since no attribute in Z is an ro-attribute, $Z \subseteq (X \cup \mathrm{RL}_X)_G^+$. Then, by definition, $A \in (X \cup \mathrm{RL}_X)\hat{}_G$. Since $A \in X$, (7) holds. □

Theorem 4.6 will be used to determine which left-hand sides in the Bernstein 3NF algorithm may possibly have equivalent left-hand sides. If (7) does not hold, then X has no equivalent key. If (7) is true, then X may have an equivalent key Y. No closure needs to be calculated to find Y because Theorem 4.3 shows how to recapture it from the r-closure. It can happen that (7) holds and X has no equivalent left-hand side. For example, let F be the set of dependencies in Figure 9.
In this example, with $X = \{a, b\}$, $(X \cup \mathrm{RL})\hat{}_G = \{a, b\}$, but there is no left-hand side Y equivalent to X.

The following theorem is used in synthesizing third normal forms. It is used to reduce the number of dependencies that have to be tested for redundancy in the subrelations when left-hand sides are grouped. The definitions of H and J are stated in step 5 of Algorithm A5.

THEOREM 4.7. *Suppose X has an equivalent left-hand side and $X \rightarrow A$ is in H. If*

$$A \notin (X \cup \mathrm{RL}_X)\hat{}_G, \tag{8}$$

then $X \rightarrow A$ is not redundant in $H \cup J$.

PROOF. To show the contrapositive, suppose that $X \rightarrow A$ is in H and redundant in $E = H \cup J$. So with $E' = E - \{X \rightarrow A\}$, $A \in X_{E'}^+$. Since $A \notin X$, there is a $Z \subseteq X_{E'}^+$ such that $Z \rightarrow A$ is in E' and $Z \neq X$. Since Z has no ro-attributes and since $Z \subseteq X_{E'}^+ \subseteq X_E^+ = X_F^+$, it follows by Lemma 4.1 that $Z \subseteq (X \cup \mathrm{RL}_X)_G^+$. Thus if $Z \rightarrow A$ is in G, then by definition, $A \in (X \cup \mathrm{RL}_X)\hat{}_G$, and (8) does not hold.

Suppose that $Z \rightarrow A$ is not in G. Then it is not in H since $Z \neq X$, and so it is in J. But this dependency is in J only if there is some left-hand side W that is equivalent to Z and $A \in W$.

Now if $X \rightarrow A$ is required in F in order to establish that $Z \rightarrow W$ is in F^+, then $Z \rightarrow X$ is in F^+ by Lemma 1 of [1]. But $X \rightarrow Z$ is in F^+ and so X, Z, and W are equivalent left-hand sides. But then $X \rightarrow A$ would not be in H since $A \in W$

would require that it be removed from H and placed in J, contrary to the assumption of the theorem.

If Z does not require $X \rightarrow A$ in F in order to establish $Z \rightarrow W$ is in F^+, then, since $A \in W$, there is a $Y \rightarrow A$ in $F' = F - \{X \rightarrow A\}$ and $Y \subseteq Z^+_{F'}$. But since $Z \subseteq X^+_F$, $Y \in (X \cup \mathrm{RL}_X)^+_G$ and $Y \rightarrow A$ is in G. By definition, $A \in (X \cup \mathrm{RL}_X)\hat{}_G$, and (8) does not hold. □

Note that condition (8) holds for all ro-attributes, since $X \rightarrow A$ would have been removed and cannot be in H. So, in the worst case, (8) will fail for the set of the rl-attributes, and only those dependencies need to be checked for redundancy. For full sets of dependencies this will significantly prune the dependencies that need to be tested in the last stages of synthesizing 3NF relations. Also note that property P, as defined and discussed in [1], does not provide a computationally useful means for determining which dependencies are not redundant in step 5 of Algorithm A5. It only provides a theoretically sufficient condition for guaranteeing that a relation is in third normal form. However, (8) provides means for ruling out certain dependencies as redundant and, when applied to the two examples in Figure 3 of [1], guarantees in the first example that $X \rightarrow A$ is not redundant and in the second example suggests that $X \rightarrow A$ needs to be tested as possibly being redundant, which it is. It is easy to find examples where property P and (8) fail to hold simultaneously, that is, neither implies the other.

The following is our modification of Bernstein's algorithm for synthesizing third normal forms.

In Algorithm A5, it should be observed that closures X^+ for each left-hand side need to be computed in step 4, since step 2 does not do so. In contrast, our implementation, Algorithm 4.9, will not require any closures to be calculated in step 4, since they can be recovered from the r-closures using (3) of Theorem 4.3.

Also, elimination of transitive dependencies in step 5 of Algorithm A5 requires retesting each dependency remaining in H after step 4. In contrast, our implementation will prune the space of dependencies to those with right-hand sides that are rl-attributes and do not satisfy (8) in Theorem 4.7.

Our algorithm will parallel Bernstein's, using the same step numbers but including specific methods for implementation.

Algorithm 4.9. Fast 3NF Algorithm

Input: F, a set of functional dependencies.
Output: A collection of relations in third normal form.

1. *Eliminate implied extraneous attributes.* Use Algorithm 3.5, steps 1–4.
2. *Find a minimal cover H for F.* Use Algorithm 3.1(c), storing crc(X) for each lhs X.
3. *Partition dependencies.* Done as part of previous step.
4. *Regroup dependencies.* For each lhs X satisfying (7) of Theorem 4.6, determine if there are equivalent left-hand sides Y. By Theorem 4.3, no closures need to be calculated; simply test X in Y^+ and Y in X^+ for each DepForLHS(Y). Regroup according to Bernstein's step 4.
5. *Recheck some dependencies for redundancy.* For each dependency $X \rightarrow A$ in H, A, an rl-attribute not satisfying (8) of Theorem 4.7, test using the standard method that $X \rightarrow A$ is not redundant in $H + J$.
6. *Form relations.* Same as Bernstein's step 6.

4.3 Theorems for Algorithm 3.5

The final theorems and lemmas are given to justify Algorithm 3.5.

The next lemma classifies dependencies with extraneous attributes, in that they are either implied by the other attributes of the lhs or they are extraneous in a dependency that is redundant, that is, a dependency that can be derived from the other dependencies. We assume that F is simplified.

LEMMA 4.10. *Let $CZ \rightarrow B \in F$ and suppose that $B \in Z_F^+$, then either $CZ \rightarrow B$ is redundant or $C \in Z_F^+$.*

PROOF. Suppose $C \notin Z_F^+$ and $B \notin (\mathrm{CZ})_{F'}^+$, where $F' = F - \{CZ \rightarrow B\}$. Since $B \in Z_F^+$, $CZ \rightarrow B$ is required to establish B is in the closure of Z. By Lemma 1, [1], $Z \rightarrow CZ$ is in F^+, a contradiction. □

The next theorem demonstrates that any attribute C of an lhs X that is not in the r-closure of $(X \cup \mathrm{RL}_X)$ with respect to $F - \mathrm{DepForLHS}(X)$ need not be checked for being extraneous, since it either is not extraneous or it is in a redundant dependency that will be eliminated using Algorithm 3.1(c). Property (P4) is a restatement of Theorem 4.11, and (P3) is a direct corollary.

THEOREM 4.11. *If $CZ \rightarrow B \in F$ and $C \notin (X \cup \mathrm{RL}_X)\hat{}_G$, where $X = CZ$ and C is extraneous, then $CZ \rightarrow B$ is redundant.*

PROOF. Suppose the conditions of the theorem hold and $CZ \rightarrow B$ is not redundant. Since C is extraneous, $B \in Z_F^+$. By Lemma 4.10, $C \in Z_F^+$. Thus there is a dependency $U \rightarrow C \in F$ and $U \subseteq Z_F^+ \subseteq CZ_F^+$. Since there are no trivial dependencies, $U \neq CZ$. Thus (2) of Lemma 4.2 is satisfied, giving $C \in (X \cup \mathrm{RL}_X)\hat{}_G$, a contradiction. □

5. BENCHMARKS

In this section we present the benchmark databases and the benchmark results for Algorithms A4 and 3.1, using both standard and linear closure. In addition, before presenting the benchmarks, we give a theoretical estimate of the expected improvement in Algorithm 3.1 over Algorithm A4.

5.1 Expected Improvement in Performance

First we give some general estimates on the expected improvement of our Algorithm 3.1 for removing redundant dependencies over the standard Algorithm A4, both using either normal or linear closure techniques. Both algorithms have the same order of complexity, so we focus on the number of closures computed in each as a measure of the relative difference in performance.

Recall that ro is average number of dependencies with ro-attribute right-hand sides per DepForLHS(X) and rl is the average with rl-attribute right-hand sides. We define the P-factor, short for Performance-factor, to be

$$\frac{2 * \mathrm{rl}}{\mathrm{ro} + \mathrm{rl}} = 2 * \mathrm{degf}(F)^{-1}. \tag{9}$$

Estimate 5.1. Let F be a set of functional dependencies with extraneous attributes removed and no trivial or duplicate dependencies. The time required

by Algorithm 3.1, excluding fixed overhead per dependency, is less than the time required by Algorithm A4, also excluding fixed overhead per dependency, by the P-factor (9), assuming that all closures require the same amount of time.

Justification. Since most of the time is spent in computing closures, we ignore the relatively small overhead in accessing dependencies. Since at most rl r-closures will be computed on the first pass and potentially rl standard closures will be computed on the second pass of Algorithm 3.1, then at most $2 * rl$ closures are computed. Since Algorithm A4 computes $ro + rl$ closures on the average for each left-hand side, the maximum fraction of closures computed in our method versus Algorithm A4 is given by (9).

Clearly, whenever $rl \leq ro$, (9) will be less than or equal to one, which we would expect for most actual databases.

5.2 Test Dependency Sets

We created six generic sets of dependencies, each having 40 to 50 functional dependencies, which are given below as F1, . . . , F6. We also created one other more practical set of dependencies for order processing with 103 functional dependencies. Five of the six generic sets have characteristics that we would expect to find in typical databases in various stages of development and with a degree of fullness typically in the range of 5 to 10, except in two cases. The other, F3, is a pathological set of dependencies [8], considered to be worst case for standard closure, which indeed it is if only closures are being computed. Somewhat paradoxically, it is not worst case in the context of producing minimal covers.

F1. *Clustered Dependent (CD)*

first cluster:

$a1 \rightarrow b1, c1, d1, e1, f1, g1, h1,$
$e1 \rightarrow j1, k1, m1, n1, p1, r1, s1,$
$p1 \rightarrow d1, t1, u1, v1, h1, w1.$

transitional dependency:

$w1 \rightarrow a2.$

second cluster:

same as the first cluster with each suffix 1 changed to 2.

There are four redundant dependencies $a1 \rightarrow d1, h1$; $a2 \rightarrow d2, h2$.

This is characteristic of groups of entities like employee, department, and office that are more tightly linked as a cluster by sharing attributes such as employee's manager, department's manager, employee's office, offices in departments, plus a second cluster consisting of products, inventory, and warehouses linked to the first cluster through an attribute such as an employee who is the sales representative for a product.

F2. *Clustered Independent (CI)*

This is the same as F1 with the transitional dependency removed. This corresponds to independent clusters of entities.

F3. *Linear Reversed (LR)*

$a1 \rightarrow a0,\ a2 \rightarrow a1,\ a3 \rightarrow a2,\ \ldots,\ a50 \rightarrow a49.$

This set is the pathological case where computing $\{a50\}^+$ requires 50 passes over F3 using the standard closure. For this set degf(F3) is approximately one, the lowest possible value for the degree of fullness of a set.

F4. *Full Linear Reversed (FLR)*

$a0 \rightarrow a1, a2, a3, a4.$ $\quad b0 \rightarrow a0, b1, b2, b3, b4.$
$c0 \rightarrow b0, c1, c2, c3, c4.$ $\quad d0 \rightarrow c0, d1, d2, d3, d4.$
...
$h0 \rightarrow g0, h1, h2, h3, h4.$ $\quad i0 \rightarrow h0, i1, i2, i3, i4, i5.$

This set retains some of the pathology of F3 by forcing standard closure to return over the set to compute closures, but here degf(F4) is approximately 5.5. It also reflects the configuration of a clustered set of entities after removal of redundant dependencies. This could happen if the addition of new dependencies to an existing minimal cover required recomputing a new minimal cover.

F5. *Full Independent (FI)*

This is the same as F4, except there are no links between the a's, b's, c's, etc. This corresponds to independent entities in the database.

F6. Independent Equivalent (IE)

$a0 \rightarrow a1, a2, a3, a4, a5.$ $\quad a4 \rightarrow a1, a2, a3, a0, a5.$
$b0 \rightarrow b1, b2, b3, b4, b5.$ $\quad b4 \rightarrow b1, b0, b3, b2, b5.$
$c0 \rightarrow c1, c2, c3, c4, c5.$ $\quad c4 \rightarrow c0, c2, c3, c1, c5.$
$d0 \rightarrow d1, d2, d3, d4, d5.$ $\quad d4 \rightarrow d1, d0, d3, d2, d5.$

This set corresponds to equivalent keys with many redundant dependencies.

F7. *Order Processing (OP)*

This set is developed in part from a sizeable subset of a real order processing system with the standard employee database tacked on. It exhibits many of the individual characteristics found in the generic sets above.

5.3 Benchmark Results

The following tests were conducted on each set of dependencies:

StdStd: Standard elimination of redundant dependencies using standard closure. Algorithm A4 using Algorithm A1.

StdLin: As previous, but with linear closure.

FstStd: Our algorithm for eliminating redundant dependencies using standard closure. Algorithm 3.1 using Algorithm A1.

FstLin: As previous, but with linear closure.

Fast2Lin: This is the alternative Algorithm 3.1(a) described at the end of Section 3.3 where only a single r-closure is computed on the first pass.

The results are shown in Table I.

Table I. Performance Comparisons for Algorithms A4 and 3.1

	Benchmarks (Time: seconds[1])						
	CD	CI	FLR	FI	IE	OP	LR
StdStd:	25.8	18.3	81.0	14.2	11.5	322.5	6.1
StdLin:	39.4	21.6	45.8	10.4	10.3	227.9	6.5
FstStd:	10.1	5.6	24.8	3.0	4.3	66.2	619.7
FstLin:	11.6	5.7	15.5	3.0	4.1	50.1	87.4
Fst2Lin:	11.0	5.6	15.3	3.9	4.0	65.9	88.0
P-factors							
computed	0.30	0.20	0.36	0.04	0.40	0.33	—
measured (FstLin/StdLin)	0.29	0.26	0.34	0.28	0.40	0.22	—

[1] The benchmarks were conducted on a Tektronix 4404 using Version 2.1.2 of Smalltalk 80. All tests were standalone. For the most part, the times cited are cpu. System run-time facilities for distinguishing between cpu and I/O were not available but indications are that an extremely small fraction of the time was I/O time. Most Smalltalk systems exhibit variations due to the vagaries of the background storage management procedures including garbage collection. Consequently, the best time is given and is very close to the average time, except in those cases where garbage collection obviously took place.

The times for setting up the additional data structures such as grouping dependencies in DepForLHS(X) for Algorithm 3.1 and count[d] and list[A] in linear closure are included in the results for the tests that required these.

We can make some general observations about these results:

(1) Algorithm 3.1 has significantly more impact on improving performance than does adding linear closure to Algorithm A4, which in the OP case still takes nearly 4 minutes for 103 dependencies.

(2) At worst, the linear closure overhead adds slightly to the time for our algorithm. Linear closure can help significantly in real situations where the dependencies are in reversed order for computing the closure, and can prevent disasters in pathological cases, discussed below.

(3) P-factors give a reasonably accurate estimate of the improvement in performance of Algorithm 3.1 over Algorithm A4, based on the degree of fullness of the set of dependencies.

(4) The synergistic effects of two algorithms, standard minimal cover and standard closure, can have some surprising results in pathological situations.

Specifically, Algorithm 3.1 using standard closure, FstStd, is from 2.5 to 5 times faster than Algorithm A4 with standard closure (StdStd), with one notable exception, to be discussed shortly. The higher value is achieved in the larger and more practical OP dependencies where, in absolute terms, our algorithm takes a little over 1 minute and StdStd takes over 5 minutes. But using linear closure (StdLin) in Algorithm A4 in OP reduces the time to just under 4 minutes, which is still quite high in comparison to Algorithm 3.1, even using standard closure. When we add linear closure (FstLin), the time for OP drops to 50 seconds, still over 4.5 times faster than (StdLin).

There does seem to be an overhead cost to using linear closure in that in some cases, such as CD, and CI, it is slower than standard closure. But it is apparent

from comparing FstStd and FstLin results that it is almost always beneficial to use linear closure with our algorithm, which is not too surprising. The overhead of linear closure barely hurts in CD, and reduces the time in OP by 25%, and by 40% in FLR. In the pathological case LR, it indeed saves the day by reducing the time from over 10 minutes to about 1.5 minutes.

The paradoxical case occurs with standard closure and Algorithm A4 (StdStd), with the linear reversed (LR) dependencies, where StdStd should be far worse than all other tests but is indeed the best at 6.1 seconds. If one were simply computing closures it would be the worst, but in eliminating redundant dependencies with LR no closures are in effect computed, which explains the paradox. For example, to determine if $a_i \rightarrow a_{i-1}$ is redundant in F3, this dependency is deleted and the closure of a_i is computed. But a_i does not appear on the left side of any other dependency in F3. On the other hand, comparing FstStd and FstLin in LR and in FLR illustrates the benefits of linear closure in one pathological and one not-so-pathological case, respectively.

The two rows at the bottom of the benchmark table indicate that the computed P-factors using (9) are close to the measured values using the benchmark results. In the one exception, FI, the time for FstLin is so small that most of it is due to overhead in setting up the structures used by our method. Also, in this case rl = 0, so we use rl = 1 instead to reflect the overhead of the method. In practical databases we would expect the P-factor to indicate at least up to an order-of-magnitude improvement using our algorithm, since one would expect a high ratio of ro to rl-attributes and many dependencies per left-hand side. Since the standard methods used in synthesizing third normal forms require recomputing closures for the dependencies in H, and our method does not, we expect comparable improvements in the later synthesis steps. Naturally, it is more efficient to maintain multiple attributes on the right-hand side of dependencies, rather than treating multiple dependencies with single attributes on the right-hand side. But this efficiency benefits all techniques uniformly and does not affect Estimate 5.1. Consequently, the conclusions drawn from the benchmarks should be the same.

In the case where rl $\ll$ 1 we replaced rl by 1 in (9). For our alternative algorithm FstLin2, one could use

$$\frac{\text{rl} + 1}{\text{ro} + \text{rl}}$$

to estimate the P-factor, because at most one r-closure is computed on the first pass and at most rl are computed on the second pass. It would appear that for large values of rl this alternative algorithm might yield better performance, but we did not test this in the benchmarks.

6. CONCLUSION

We have presented a new type of closure called an r-closure that can significantly reduce the number of closures computed in all stages of producing third normal form relations. Some future directions for research involving r-closures include the use of r-closures with multivalued dependencies and the use of r-closures to

manage sets of dependencies that are created incrementally in the course of an iterative design process.

ACKNOWLEDGMENTS

We have benefitted from the comments of several individuals, and are particularly indebted to Stefano Ceri and Georg Gottlob for access to their Prolog-based system for normalization. We also wish to thank the referees for their helpful comments.

REFERENCES

1. BERNSTEIN, P. A. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst. 1*, 4 (Dec. 1976), 277–298.
2. BERRI, C., AND BERNSTEIN, P. A. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst. 4*, 1 (Mar. 1979), 30–59.
3. BJORNERSTEDT, A., AND HULTEN, C. RED1: A database design tool for the relational model of data. *Database Eng. 7*, 4 (Dec. 1984), 34–39.
4. BRODIE, M. Automating database design & development: A SIGMOD 87 tutorial. *ACM SIGMOD 1987* (San Francisco, May 1987), ACM, New York, 1987.
5. CERI, S., AND GOTTLOB, G. Normalization of relations and Prolog. *Commun. ACM 29*, 6 (June 1986), 524–545.
6. GALIL, Z. An almost linear time algorithm for computing a dependency basis in a relational database. *J. ACM 29*, 1 (Jan. 1982), 96–102.
7. HASAN, W., AND YORK, B. W. A database design tool. AITG Tech. Rep. 003, Digital Equipment Corp., Hudson, Mass., 1985.
8. MAIER, D. *The Theory of Relational Databases.* Computer Science Press, Rockville, Md., 1983.
9. ULLMAN, J. D. *Principles of Database Systems.* Computer Science Press, Rockville, Md., 1982.
10. WIEDERHOLD, G., AND ELMASRI, R. The structural model for database design. In *Proceedings of the International Conference on Entity-Relationship Approach* (Los Angeles, Dec. 1979). pp. 247–267.

Received July 1986; revised June 1987; accepted November 1987