Extraction of Data Dependencies

Malú Castellanos
Fèlix Saltor

Report LSI–93–2–R

Extraction of Data Dependencies

Malú Castellanos, Fèlix Saltor
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
{castellanos,saltor} @ lsi.upc.es

Abstract

Relational database schemas must be semantically enriched to reflect knowledge about the data, as needed by many applications. One technique is the analysis of the database extension, extracting existing data dependencies. In this paper, new algorithms to extract functional and inclusion dependencies are presented; they are designed to minimize the number of disk accesses.

1. Introduction

The relational data model has gained lots of popularity during the last years, one key factor for this success being the simplicity of its *structure*. However, the price for this simplicity is its being almost devoid of semantics. In the relational model, semantics are specified not in the structure, but thru *constraints*. Data dependencies are one kind of these constraints, where *functional* and *inclusion dependencies* [Maier83, Thalheim91] are of particular interest.

Nowadays relational databases are spread everywhere, and many applications like reverse engineering, integrated access for interoperable databases, knowledge based interfaces, migration to newer database management systems (DBMS), and improvement on the effective utilization of the data of the organization require a deep knowledge on the semantics of these existing databases. Unfortunately, many relational DBMSs do not provide much support for specifying data dependencies, and consequently this semantics cannot be found in the schemas. However, since data dependencies are implicit in the extension of the databases, a solution to this problem is to extract the dependencies by analyzing the extensions. This is a particular case of *knowledge acquisition* from the database, which is part of a *semantic enrichment* process where extracted semantics is made explicit in an enriched representation of the database [Castellanos91].

Our approach to the extraction of data dependencies is the following: 1) If the DDL of the DBMS supports the specification of some semantic constraints -for example primary keys or foreign keys-, and the DB designer has made use of it, then the corresponding dependencies -functional and inclusion, respectively-, are taken as the initial set of dependencies.

2) Then the extensions of the relations are analyzed, looking for additional dependencies. This is an incremental process, and each time a new dependency is discovered, it is added to the existing set and, by applying implication rules, new, derived dependencies are computed, so that these are not checked by extension analysis. The order in which possible dependencies are analyzed is therefore very important.

3) In theory, the fact that the extension of the DB at a specific time satisfies a constraint, is no guarantee that this constraint is part of the semantics (the intension) of the DB and should be satisfied at all times; it could happen just by chance. In practice, in a real DB with relations of thousands or millions of tuples, this possibility is hardly probable. In our approach, in order to be sure, the DB administrator is prompted for confirmation of every dependency discovered by extension analysis.

The second step is the crucial one. The problem is that analyzing extensions has a high cost given mainly by the numerous accesses to disk that are required to retrieve the tuples of the relations. In this paper we develop some algorithms for extracting functional (Section 2) and inclusion dependencies (Section 3) where the number of accesses to disk is minimized.

2. Functional Dependencies

A functional dependency (FD) is a constraint imposed on a relation that specifies that the value for a given set of attributes X uniquely determines the value of another attribute Y. A formal definition follows:

Let R be a relation schema, and X and Y be subsets of its attributes.
Let *r* be a relation over R.
A FD: $X \rightarrow Y$ specifies that for every pair of tuples t1, t2 of *r*

$$t1[X] = t2[X] \Rightarrow t1[Y] = t2[Y]$$

From all FDs, the ones of interest here are those called *elementary FDs*. A FD $X \rightarrow Y$ is *elementary* if and only if: Y is a single attribute, $\neg\exists X' \subset X \mid X' \rightarrow Y$ (X is minimal), and $X \rightarrow Y$ is non trivial. A FD $X \rightarrow Y$ is *trivial* if $Y \subset X$.

In any algorithm that discovers FDs from the extension of the relations, it is important to minimize redundant work, that is, the work of analyzing the tuples to obtain redundant FDs, because they can be derived from other ones by applying *Armstrong rules* [Armstrong 74]. The focus is on the following derived FDs:

- non elementary FDs where Y is composite (more than one attribute) since they can be derived from the elementary ones by applying the Armstrong rule of *additivity*:

if $X \rightarrow Y$ and $W \rightarrow Z$, then $XW \rightarrow YZ$.

A particular case is where W is X: if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.

- non elementary FDs where X is not minimal because they can be derived by the *augmentation* rule of Armstrong:

$\forall X' \supset X$, if $X \rightarrow Y$, then $X' \rightarrow Y$.

- transitive FDs because they can be derived by the *transitivity* Armstrong rule:

if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Note that the *Unicity assumption*, under which a given FD $X \rightarrow Y$ can exist at most once , is not required in our approach, since we analyze the extensions.

Non elementary FDs where the right hand side is composite can be completely avoided by proposing always single attributes of the relation as possible right hand sides. With respect to non elementary FDs where the left hand side is not minimal, these can be eliminated by establishing an appropriate order of the possible left hand sides to be analyzed. The difficulty is with transitive FDs because there is no way to completely eliminate the redundant work performed on the extension of a relation that gives as result transitive FD's; at most, only some of them can be discovered by applying the transitivity axiom each time that a new FD is obtained. The problem is that there is no information to determine an order of possible FD's by which it is guarantied that no transitive FD is analyzed before the FDs from which it derives are analyzed. In section 3 it is shown that this is not the case for inclusion dependencies.

Let ***lhs*** be the left hand side (simple or composite) of a FD, that is, the determinant, and ***rhs*** the right hand side (always simple), for every pair (lhs,rhs) all the tuples of the relation must be analyzed to test if every time that:

$$t_i[lhs] = t_j[lhs] \text{ also } t_i[rhs] = t_j[rhs].$$

The cost of analyzing the extension of a relation is very high because of the necessary accesses to disk, thus, it is important to minimize them by generating the candidate pairs (lhs, rhs) in an appropriate order. There are two approaches: for each rhs, analyze its possible lhs, as in [Mannila91], or for each lhs, analyze its possible rhs, which is our approach. Since every combination of any number of attributes is a candidate lhs, the problem is to find the most adequate order of generating lhs's. We present four alternatives where each one is an improvement of the previous one.

Alternative A.1.

The possible lhs's are ordered by number of attributes (degree), and lexically within degree, i.e. numerical sequence and not lexical sequence. For a relation with attributes A, B, C, and D, the order would be: A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD. This order guarantees that no lhs is analyzed until any subset of its component attributes has been analyzed.

For every possible lhs X that is analyzed, each attribute of R - X is candidate as a rhs of a possible FD (trivial FDs are not considered). However, each such attribute is proposed as a possible rhs only if there is no previously found FD whose left hand side is contained by this lhs and whose right hand side is equal to this rhs, otherwise, the pair (lhs, rhs) represents a redundant FD (by augmentation) and rhs is positively a right hand side for lhs. In this way only FD's with minimal lhs's are obtained. Furthermore, once a FD has been discovered, all transitive FDs that can be derived at that moment are computed. This mechanism makes possible to avoid having to analyze the extension of the relation for many redundant FD's.

For each pair (lhs, rhs) representing a possible FD, the extension of the relation is analyzed to test if whenever t_i[lhs] = t_j[lhs] also t_i[rhs] = t_j[rhs] (called *'FD condition'* from now on). Furthermore, this condition is checked simultaneously for all possible FD's (i.e. all possible rhs) on a lhs while the pairs of tuples t_i, t_j are being retrieved. The problem is that to test the FD condition, for each different value *b* for lhs, every tuple in the relation must be retrieved in search of the same value *b* for the combination of attributes composing lhs. If the search succeeds, then for each possible rhs the corresponding attributes of both tuples are checked to see if their values match too. This process implies that too many accesses to disk have to be performed: *all the tuples of the relation must be retrieved for each different value of each possible lhs!.*

Now, let us make some considerations that apply to all our alternatives.

i) The only case when a possible lhs can be discarded from the extension analysis is when previously found FDs show that it functionally determines every single attribute of the relation redundantly by augmentation. This means that for each attribute of the relation, a FD whose left hand side is contained by this lhs has already been found. Just in this case no single attribute can be proposed as a *possible* rhs for lhs. Consequently, since there are no *possible* FDs for lhs, it is not analyzed against the extension of the relation.

ii) Since keys can be obtained directly from the catalog (by looking for primary key definitions or for indexes defined to be unique) and a key functionally determines every single attribute of the relation, the set of FDs is initialized to these *key based* FDs. Then, when the list of possible lhs's is being generated, every possible lhs is checked against the keys and if it is a key, it is not inserted into the list. This fact is important because most of the FD's that hold in a relation are *key based* ones and obtaining them directly from the keys is less costly than obtaining them from the extensions.

iii) If the dictionary contains metadata about the cardinality of each single attribute, this information can be used to determine if a given attribute is a possible rhs for a given lhs, that is, whether a pair (lhs, rhs) is a possible FD or not. If the cardinality of lhs is greater or equal than the cardinality of rhs, it is possible that there is a FD, otherwise lhs cannot functionally determine rhs. Unfortunately this necessary condition is not useful for composite lhs's because the saving in the cost given by the discarded rhs's does not compensate the cost of computing the cardinality (which requires a sort on the projection on lhs).

The algorithm is presented next:

```
Algorithm extract_FDs_for_lhs_A.1
 {* possible_rhs's_for_lhs: contains all the single attributes that are possible rhs's for the lhs *}
 {* possible_lhs's: contains all the subsets of R that are possible lhs *}
 {* checked_values: contains the values of lhs that have already been checked *}
 initialize FDs      {* with those derived from the keys *}
 obtain_possible_lhs's      {* in numerical order *}
 for each lhs in possible_lhs's do
    obtain_possible_rhs's_for_lhs
    if possible_rhs's_for_lhs not empty then
      for each tuple t_i do     {* i := 1 to n-1 where n is the cardinality of the relation *}
         if t_i[lhs] not in checked_values then      {* current value of lhs has not been checked yet *}
```

```
            insert t_i[lhs] in checked_values
            search/check_tuples_with_same_value_for_lhs
            generate_FDs_from_remaining_rhs's
          end_if
        end_for
      end_if
  end_for
end_algorithm

 Procedure initialize_FDs          {* with key based FD's *}
{* keys: contains the keys obtained previously from the catalog *}
{* FDs: contains the FDs that have been obtained *}
   for each element K in keys do
        for each attribute A_j ∈ R-K do
            insert K → A_j into FDs      {* key based FD *}
        end_for
   end_for
 end_procedure

 Procedure obtain_possible_lhs's
{* keys: contains the keys obtained previously from the catalog *}
{* possible_lhs's: contains all the subsets of R that are possible lhs *}
  possible_lhs's := ∅
  for each subset S of R do              {* considered in numerical order *}
     if S is not in keys then
         insert S into possible_lhs's
     end_if
  end_for
 end_procedure

 Procedure obtain_possible_rhs's_for_lhs
{* possible_rhs_for_lhs: contains all the single attributes that are possible rhs's for the lhs *}
{* FDs: contains the FDs that have been obtained *}
   possible_rhs's_for_lhs := ∅
   for each attribute A_j∈ (R - lhs) do
        if X → Y| lhs ⊇ X and A_j = Y not in FDs then {* possible FD (lhs,rhs) is not redundant by augmentation nor
                                                        has been derived by transitivity *}
           if degree (lhs) = 1 then  {* next 'if ' only useful for single attribute lhs's *}
              if card (lhs) ≥ card (A_j) then   {*necessary condition for the existence of FD lhs → A_j *}
                  insert A_j into possible_rhs's_for_lhs
              end_if
           else
              insert A_j into possible_rhs's_for_lhs
           end_if
        end_if
  end for
end_procedure

Procedure search/check_tuples_with_same_value_for_lhs
{* possible_rhs_for_lhs: contains all the single attributes that are possible rhs's for the lhs *}
{* t_i: current tuple in outer loop to be checked against the rest of the tuples t_j *}
   j:= i + 1
   while possible_rhs's_for_lhs not empty and j < n do   {* n: number of tuples *}
        if t_i[lhs] = t_j[lhs] then
           for each rhs in possible_rhs's_for_lhs do
              if t_i[rhs] ≠ t_j[rhs] then          {* evidence of no existence of the possible FD *}
                  delete rhs from possible_rhs's_for_lhs
              end_if
           end_for
        end_if
        j:=j+1
   end_while
end_procedure
```

```
Procedure generate_FDs_from_remaining_rhs's
{* possible_rhs_for_lhs: contains all the single attributes that are possible rhs's for the lhs *}
{* FDs: contains the FDs that have been obtained *}
  if possible_rhs_for_lhs not empty then
    for each rhs in possible_rhs_for_lhs for
        insert lhs → rhs into FDs
        delete rhs from possible_rhs's_for_lhs
        compute transitive FDs        {* and insert into FDs*}
    end_for
  end_if
  delete lhs from possible_lhs's
end_procedure
```

Now, let us develop formulas for the cost of this alternative in terms of accesses to disk. We use the following notation:

- v : the number of different values of lhs,
- t : the tuple size,
- C : the number of tuples (cardinality) of the relation,
- d : the number of attributes (degree) of the relation,
- p : the page size,
- b : the blocking factor (tuples per page) and
- P : the number of pages occupied by the relation. We assume the extension is stored in a file on its own, and therefore $b = \lceil p/t \rceil$, and $P = \lceil C/b \rceil$.

First, we consider the cost for determining all the FDs on a single lhs and then the cost for all the FDs of the relation.

I) Cost for a single lhs: the search for a pair of tuples that contradicts the FD condition, for every possible rhs, continues as long as no such a pair is found. There are two extreme cases:

- in the worst case, at least one FD on this lhs exists, and thus the whole relation has to be scanned entirely for each different value of the lhs because no evidence that contradicts the FD is found. The number of accesses to disk is: $vP = v\,[C/b] = v\,[C/(p/t)]$.

- in the best case, no FD exists on lhs and the evidence that contradicts the candidate FD's is found for the first value considered for lhs and for pairs of tuples (for which the attributes composing lhs have the same corresponding values) that belong to the first page. As soon as this is found, the search stops. Thus, *only one* access to disk is required.

  In general, we can say that if no FD exists, it is very likely that this fact will be found soon, not requiring to access all the pages occupied by the relation.

II) Cost for the whole relation: the number of possible lhs's for a given relation is equal to the number of non redundant subsets of its attributes, except empty and R (R represents all the attributes of the relation and it is a lhs because it determines each one of its attribute), that is: $N = 2^d - 2$. Then, depending on the number of possible lhs's that we can discard, N can range between two extremes:

- in the best case, when every attribute in the relation directly determines the following one, N can be reduced to $2d$ because of all the possible lhs's that can be discarded. For example, if the attributes in R are A, B, C, and D, and the FD's $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$ hold, the following lhs's are analyzed:

  1- lhs = A; possible rhs = B, C, D. FDs discovered: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$. Any lhs X such that $X \supset A$ can be discarded.

2- lhs = B; possible rhs = A, C, D. FDs discovered: B → C, B → D (we now find that A → C, discovered in 1-, is derived). Any FD with lhs X such that X ⊃ B and rhs = C or D (i.e. BC → D, BD → C) can be discarded.

3- lhs = C; possible rhs = D, A, B. FDs discovered: C → D (A → D and B → D, already discovered, are derived). Any FD with lhs X such that X ⊃ C and rhs = D (none non trivial exists) can be discarded.

4- lhs = D; possible rhs = A, B, C. FDs discovered: None.

5- lhs = BC; possible rhs = A. FDs discovered: None.

6- lhs = BD; possible rhs = A. FDs discovered: None.

7- lhs = CD; possible rhs = A, B. FDs discovered: None.

8- lhs = BCD; possible rhs = A. FDs discovered: None.

thus, when 2d lhs's have been analyzed, there are no proposed rhs's for any further lhs because they would be redundant (by augmentation and transitivity axioms).

- in the worst case, no possible lhs can be discarded (see consideration i). In this case $N = 2^d-2$ is maintained. For each of these lhs's we know (from I) that the number of accesses ranges between 1 and v [C/b]. Thus, the upper bound is given by:

$$(2^d-2)\ (v\ [C/b])$$

If v is proportional to C, then this formula is quadratic in the number of tuples, but exponential in the degree.

Now let us make some more considerations that apply to all our alternatives:

iv) The number of accesses will never reach the upper bound (given in the 'worst case' for the whole relation in each alternative) because when it is the worst case for the number of possible lhs's (no lhs can be discarded because the FDs for each attribute of R that would make it redundant do not exist, see consideration i), it is not the worst case for each possible lhs (the fact that a FD does not exist is found soon in general).

v) The cost of analyzing one possible lhs (with all its possible rhs's) is the cost of checking only one pair (lhs, rhs). This is because when two tuples with the same value for the current lhs are found (t_i[lhs] = t_j[lhs]), the values of these tuples for every possible rhs are tested to see if there is a matching too (t_i[rhs] = t_j[rhs]).

vi) Checking a candidate pair (lhs, rhs) is generally done fast for the following reasons:
- in general, relations are not highly unnormalized, so not many *non key* FDs (where lhs is not a key) exist
- *key based* FDs are obtained directly from the keys right at the beginning, thus, the possible FD's given by the pairs (lhs, rhs) are *non key based* ones
- the failure of the *FD condition* is generally discovered after checking just few pairs of tuples.

A numerical example that illustrates this alternative is found in the appendix.

The conclusion is that even though no time is wasted in checking FD's which are redundant by augmentation, too many accesses to disk are necessary with this alternative.

Alternative A.2.

The most serious deficiency of the previous alternative is that for each existing different value of the candidate lhs being analyzed it is necessary to search for all tuples with that same value. Thus, the whole relation is examined over an over again for every possible lhs. To overcome this deficiency the relation could be ordered by using the *lhs as the sort key*, so that only consecutive pairs of tuples will have to be compared.

Hence, for each possible lhs the whole relation will be scanned *only once*. However, this alternative introduces the cost associated to the sorts. Let us formulate the cost in a similar way as in A.1, that is, first we consider the cost for a single lhs and then for all the lhs's of the relation.

I) Cost for a single lhs: on one side we have the cost of the sort, which is

$$S = P \log P = (C/b \log (C/b)) \text{ accesses;}$$

on the other, the cost of the search on the sorted file which continues as long as no pair of tuples that contradicts the FD condition for every possible rhs (on this lhs) is found:

- in the worst case, a FD on this lhs exists and the whole relation has to be scanned entirely because the evidence that contradicts the FD condition is not found; however, this scan is performed *only once* giving a cost of *P* accesses (instead of vP as in A.1) because the file is sorted by lhs. The total cost for a lhs is:

$$S + P = (C/b \log (C/b)) + C/b \text{ accesses to disk}$$

- in the best case, no FD exists, the relation is sorted and the evidence that contradicts the candidate FD's on lhs is found in the first page leading to *only one* access to disk. This gives a total of accesses equal to $S + 1$.

II) Cost for all the lhs's: analogously to alternative A.1, the number of possible lhs's given by $N = 2^d - 2$ can be reduced by discarding some of them:

- in the best case, with a chain of FDs such as $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $N = 2d$.

- in the worst case, no possible lhs can be discarded, since there are no FD's besides the trivial ones, so $N = 2^d - 2$ is maintained. From i) we know the cost for each lhs, and thus, the upper limit (never reached as explained in consideration iv) in A.1) is given by:

$$(2^d - 2)\ (C/b \log (C/b) + C/b$$

see the appendix for an example.

The conclusion is that here again every non elementary FD where the determinant (lhs) is not minimal is deduced without having to analyze the extension of the relation. The cost of accesses is high but not as much as in option A.1), because it is not quadratic but *O (ClogC)* on the cardinality. The algorithm of this alternative is given next. From now on, when we reuse a procedure given in some previous alternative, we indicate this by a comment: {* given in A.# *}.

```
Algorithm extract_FDs_for_lhs_A.2
 {* possible_rhs_for_lhs: contains all the possible rhs's for the lhs *}
 {* FDs: contains all the obtained FDs *}
 {* keys: contains the keys obtained from the catalog *}
 initialize_FDs    {* with those derived from the keys *}                    {* given in A.1*}
 obtain_possible_lhs's                                                       {* given in A.1*}
 for every lhs in possible_lhs's do     {* in numerical order *}
        obtain_possible_rhs's_for_lhs                                        {* given in A.1*}
        if possible_rhs's_for_lhs not empty then
           sort by lhs
           check_extension_for_possible_FD's_on_lhs
        end_if
        generate_FDs_from_remaining_rhs's                                    {* given in A.1*}
   end_for
end_algorithm

  Procedure check_extension_for_possible_FD's_on_lhs
        {* analysis of pairs of consecutive tuples to check the existence of FDs for each possible lhs *}
        {* possible_rhs_for_lhs: contains all the possible rhs's for lhs *}
        i := 1
        n := card (r)          {* r is the extension of R *}
        while possible_rhs's_for_lhs is not empty and i < n do
             if t_i[lhs] = t_i+1[lhs] then
                for each element rhs ∈ list_possible_rhs's_for_lhs do        {* all FD's on lhs are checked at once *}
                     if t_i[rhs] ≠ t_i+1[rhs] then
                          delete rhs from possible_rhs's_for_lhs {* FD lhs → rhs does not hold *}
```

```
                end_if
            end_for
        end_if
        i:= i+1
    end_while
end_procedure
```

Alternative A.3.

This alternative tackles the main obstacle of the previous one, that is, the excessive number of sorts that have to be performed: one for every possible lhs. The idea is to perform only sorts where the *sort key is maximal*, this means that the sort is useful for checking not only one but various possible lhs's. For example, if the relation has attributes A, B, C and D, the sort keys will be ABC, BCD, CDA, DAB, AC, and BD because with these sorts all the possible lhs's A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, and BCD can be analyzed. The number of sorts *N* for this example is six instead of fourteen (from 2^d-2) as would be required using option A.2. If the degree were five, the number of sorts N would be ten instead of thirty. In general, the number of sorts is

$$N = \max_i \binom{d}{i} = \binom{d}{d/2} \cong \frac{2^d}{\sqrt{(\pi/2)\sqrt{d}}}$$

Once the relation has been sorted by one sort key, all the possible lhs's that can use it are completely analyzed successively before sorting the relation by another sort key.

Now, the order of the possible lhs's is determined by the sort key and not by the need of eliminating non minimal lhs's as in the previous alternatives. This means that the property of alternatives A.1 and A.2 by which it was guarantied that only minimal lhs's were analyzed is lost. Consequently not only elementary FDs will be obtained but *also some redundant non elementary ones*. For example, from the relation sorted by ABC, the possible lhs's A, AB, and ABC are analyzed successively and if the FD ABC $\rightarrow$ D is found, it may be a redundant one if B, C, BC, or AC, which have not been analyzed yet, determine D. For some of these redundant dependencies the analysis work can be eliminated by checking each possible FD (lhs, rhs) against the FDs already obtained: if a FD X $\rightarrow$ Y where X $\subset$ lhs and Y = rhs exists, then lhs $\rightarrow$ rhs is a redundant FD. Consequently rhs is not proposed as a possible rhs for the current lhs.

The cost of this alternative in terms of accesses to disk for a relation of degree *d* is: $NS + (2^d - 2)L$ where $S = (C/b)\ log\ (C/b)$ is the cost of each sort and *L* the cost of checking each possible lhs ranges between:

- The best case, where tuples that contradict all rhs for the current lhs are found in the first page, so $L = 1$.

- The worst case, where at least one FD on lhs exists; thus, all pages have to be read, giving $L = P = C/b$.

The upper bound is therefore given by the formula: $\binom{d}{d/2}\left(\frac{C}{b}\log\left(\frac{C}{b}\right)\right) + (2^d - 2)\left(\frac{C}{b}\right)$

This is still O (2^d) on the degree, but now its factor is linear on the cardinality of the relation; the other term remains as in A.2, and is O (C log C) on the cardinality. An example is found in the appendix.

In some cases it is possible to avoid the sort by applying the following consideration (applicable to A.3 and A.4): for a lhs that is not a determinant, tuples contradicting all possible rhs´s can in most cases be found in one page. Therefore, before performing a sort, the first page is read and analyzed for all lhs´s of the sort key; if all these lhs´s can be discarded (because evidence of no existence of all possible FDs on them has been found), the sort is not performed. This step is useful even if not all the possible lhs's have been discarded because at least some lhs's or some rhs's have been.

The algorithm for alternative A.3) is presented next:

```
Algorithm extract_FD_for_lhs_A.3
 {* possible_lhs's: contains all the possible left hand sides grouped by sort key *}
 {* G: group of possible lhs that use the same sort key for their analysis *}
      initialize_FDs                                                          {* given in A.1*}
      obtain_possible_lhs's_ grouped_by_sort_key
      for each group G of possible_lhs's do
          read 1st page
          for each lhs ∈ G do
              obtain_possible_rhs's_for_lhs   {* each lhs has its own list of possible_rhs's_for_lhs *}   {* given in A.1*}
              check_possible_FDs_on_lhs
              if possible_rhs's_for_lhs ≠ ∅ then
                  delete lhs from possible_lhs's of G
              end_if
          end_for
          if possible_lhs's of G ≠ ∅ then
              sort r by sort_key of G                  {* one sort is used to check all possible lhs's of the group *}
              for each lhs ∈ G do
                  obtain_possible_rhs's_for_lhs
                  check_possible FD's_on_lhs           {* by analyzing consecutive pairs of tuples *}
                  prepare_effects_of_FDs
              end_for
          end_if
      end_for
end_algorithm

 Procedure obtain_possible_lhs's_grouped_by_sort_key
      {* every subset of attributes of R that is not a key is made a possible lhs and it is inserted into the group for
         which it is a prefix of the sort key *}
      {* possible_lhs's: contains all the possible lhs's to be tested *}
      {* keys: contains the keys obtained from the catalog        *}
      generate_appropriate_set_of_combinations_for_sort_keys
      for each subset S of attributes of R do      {* in such an order that inappropriate permutations are eliminated*}
          if S not in keys then
                insert S into corresponding group of possible_lhs's
          end_if
      end_for
end_procedure

 Procedure check_possible FD's_on_lhs
      {* analysis of pairs of consecutive tuples to check the existence of FDs for each possible lhs,
         only consecutive tuples because r is sorted by the sort key of the group to which lhs belongs *}
      {* possible_rhs_for_lhs: contains all the possible rhs's for the lhs *}
      {* t_i: i^th tuple of the relation *}
      i := 1
      n := card (r)        {* r is the extension of R *}
      while possible_rhs's_for_lhs ≠ ∅  or i < n do
            if t_i[lhs] = t_i+1[lhs] then
               for each element rhs ∈ list_possible_rhs's_for_lhs do          {* all FD's on lhs are checked at once *}
                    if  t_i[rhs] ≠ t_i+1[rhs] then
                          delete rhs from possible_rhs's_for_lhs  {* FD lhs → rhs does not hold *}
                    end_if
               end_for
            end_if
            i:= i+1
      end_while
 end_procedure
```

```
Procedure prepare_effects_of_FDs
        {* insert FDs, compute transitivities and eliminate redundant FDs whose left hand side is not minimal *}
        {* possible_rhs_for_lhs: contains all the possible rhs's for the lhs *}
        {* FDs: contains all the obtained FDs                                *}
        if possible_rhs's_for_lhs is ≠ ∅ then   {* there is at least one FD *}
           for each rhs ∈ possible_rhs's_for_lhs do
                insert FD  lhs → rhs into FDs
                compute_transitive FDs        {*also insert them in FDs and mark them as transitive *}
                for each FD X→Y ∈ FDs | (X ⊃ lhs and Y = rhs) do {* FD is redundant by augmentation *}
                    delete FD from FDs
                end_for
                delete rhs from possible_rhs's_for_lhs
           end_for
        end_if
        delete lhs from possible_lhs's
end_procedure
```

Alternative A.4

In alternative A.3, once a sort is done, each possible lhs in the group is analyzed successively, requiring a pass over the file for each lhs. This can be enhanced by analyzing all lhs´s of a group of a sort key in a single pass. The cost of the sorts is maintained because the number of sorts N remains the same, but the cost of analyzing the extension decreases because *the number of passes is reduced from the number of lhs's (2^d-2) to the number of sorts*. The upper bound is therefore given by the formula

$$NS + NP = N(S + P) = \binom{d}{d/2}\left(\frac{C}{b}\log\left(\frac{C}{b}\right) + \frac{C}{b}\right)$$

(where P = C/b in the worst case because at least one FD exists on one lhs of the group.)
This upper bound is no longer O (2^d) on the degree, but (O ($2^d/\sqrt{d}$).

The algorithm is similar to the previous one:

```
Algorithm extract_FD_for_lhs_A.4
  {* possible_lhs's: contains all the possible left  hand sides grouped by sort key *}
  {* G: group of possible lhs that use the same sort key for their analysis *}
        initialize FDs                                                          {* given in A.1 *}
        obtain_possible_lhs's_grouped_by sort_key                               {* given in A.3 *}
        for each group G of possible_lhs's do
           for each lhs ∈ G do
              obtain_possible_rhs's_for_lhs  {* each lhs has its own list of possible_rhs's_for_lhs *}  {* given in A.1*}
           end_for
           read 1st page
           n := number of tuples in 1st page
           check_possible_FDs_on_a_group          {* only on 1st page *}
           if possible_lhs's of G ≠ ∅ then
                sort r by sort_key of G       {* one sort is used to check all elements (possible lhs's) of the group *}
                n := card(r)
                check_possible FD's_on_a_group      {* by analyzing consecutive pairs of tuples *}
                prepare_effects_of_FDs
           end_if
        end_for
end_algorithm
```

```
Procedure check_possible FD's_on_a_group
        {* analysis of pairs of consecutive tuples to check the existence of FDs for each group,
           only consecutive tuples because r is sorted by the sort key of the group *}
        {* possible_lhs's: contains all the possible lhs's for the group*}
        {* possible_rhs's_for_lhs: contains all the possible rhs's for a lhs *}
        {* G: group of lhs's for the current sort key *}
        {* t_i: i^th tuple of the relation *}
        i:= 1
        while possible_lhs ≠ ∅ or i < n do
           for each lhs ∈ possible_lhs's do                {* all lhs in the group are checked in this pass *}
              if t_i[lhs] = t_i+1[lhs] then
                 for each rhs ∈ list_possible_rhs's_for_lhs do      {* all FD's on lhs are checked at once *}
                    if t_i[rhs] ≠ t_i+1[rhs] then
                          delete rhs from possible_rhs's_for_lhs {* FD lhs → rhs does not hold *}
                          if possible_rhs_for_lhs = ∅ then
                                delete lhs from possible_lhs of G
                          end if
                    end_if
                 end_for
              end_if
           end for
        i := i+1
        end_while
end_procedure

Procedure prepare_effects_of_FDs
        {* insert FDs, compute transitivities and eliminate redundant FDs whose left hand side is not minimal *}
        {* possible_lhs: contains all the possible lhs's for the group*}
        {* possible_rhs_for_lhs: contains all the possible rhs's for a lhs *}
        {* FDs: contains all the obtained FDs *}
        {* G: group of lhs's for the current sort key *}
        if possible_lhs of G ≠ ∅ then   {* there is at least one FD *}
          for each lhs in possible_lhs do
            for each rhs ∈ possible_rhs_for_lhs do
                insert lhs → rhs into FDs
                compute_transitive FDs        {* insert them in FDs, mark them as transitive and *}
                        {* delete rhs *}
                for each FD X→Y ∈ FDs | (X ⊃ lhs and Y = rhs) do {* FD is redundant by augmentation *}
                     delete FD from FD's
                end_for
                delete rhs from possible_rhs's_for_lhs
            end_for
            delete lhs from possible_lhs's
          end for
        end_if
end_procedure
```

3. Inclusion Dependencies

An inclusion dependency is a constraint imposed on the database that specifies that the values of a given subset of attributes of a relation must exist as values of a given subset of attributes of another (or the same) relation. A definition follows:

> Let R and S be two relations, and X and Y be subsets of their attributes respectively.
> An IND: R.X ⊂ S.Y exists iff R [X] ⊆ S [Y]

The naive method of checking, for each pair of relations in the database, all possible INDs, one by one, is practically impossible for two reasons: one is that for any pair of relations, with degrees *D* and *d* (d <= D), there are

$$\binom{D+d}{d} - 1 = \frac{(D+d)!}{D!\,d!} - 1$$

possible non equivalent IND, and the second one is that in general it is not possible to check the existence of a long IND fast. However, we can use some heuristics to reduce the set of possible INDs:

A. *The corresponding attributes in the left and right hand side of an IND must be of the same type.*

All those possible INDs that do not satisfy this rule are not meaningful, so they are disregarded. For this purpose, lists of attributes of the same type are generated. The ideal would be to compare only attributes of the same semantic domain (as specified in the SQL2 standard), but unfortunately there is still no support for domains in most relational DBMS, only a very primitive notion of (syntactic) domains as basic data types is provided. Therefore, we cannot avoid having to compare attributes that pertain to different semantic domains when they have the same syntactic domain.

B. *Only domains used for identifiers have to be considered.*

Since INDs constitute the relational mechanism for referencing related entities thru their identifiers, only syntactic domains used for identification purposes have to be considered, that is, strings of characters and numbers without decimal part, i.e. integers and fixed point reals (m, n) with n = 0.

C. *A limit must be imposed on the length of INDs considered.*

Typically the INDs that hold in a database are short, so it is not useful to consider INDs of arbitrary length. A limit of three seems rather reasonable.

D. *Start with unary INDs, and then disregard all those possible non unary INDs for which there does not exist a unary IND for each pair of the corresponding attributes.*

This heuristic is based on the projection rule of INDs:

if there is an IND: R [a1, a2,..., an] $\subset$ S [b1, b2,..., bn]
then the INDs: R [a1] $\subset$ S [b1], R [a2] $\subset$ S [b2], ... , R [an] $\subset$ S [bn] exist.

Thus, unary INDs constitute necessary conditions for the existence of non unary ones, in particular for binary ones. For ternary ones the heuristic becomes:

E *Disregard all those possible ternary INDs for which there does not exist a binary IND for each pair of any of the corresponding pairs of attributes.*

Thus, binary INDs between any pair of corresponding attributes is a necessary condition for the existence of ternary ones.

Now, let us discuss the problem of extracting inclusion dependencies, but since it is attacked in a different way depending on whether the IND is unary, binary or ternary, it will be exposed in three parts.

Algorithm **extract_IND**
    extract_unary_IND;
    extract_binary_IND;
    extract_ternary_IND;
end_algorithm

3.1 Unary INDs.

In principle, every pair of attributes, no matter from which relations, form a possible IND. But, in addition to the heuristics above, two criteria will help us reduce the complexity of the problem. They constitute the core of the algorithm for extracting unary inclusion dependencies.

F. *Compare an attribute only with those of the same type with greater of equal cardinality.*

It makes no sense to look for an IND where the attribute on the right hand side (rhs) of the IND has lower cardinality than the one on the left hand side (lhs), since the values of this last cannot be a subset (neither proper or improper) of the other one. For this purpose the list of attributes of the same type is ordered according to the cardinalities of the attributes.

G. *Never compare an attribute Ai with another one Aj with greater (or equal) cardinality before the attributes whose cardinality is greater than the one of Ai but lower than that of Aj, have been compared with Aj.*

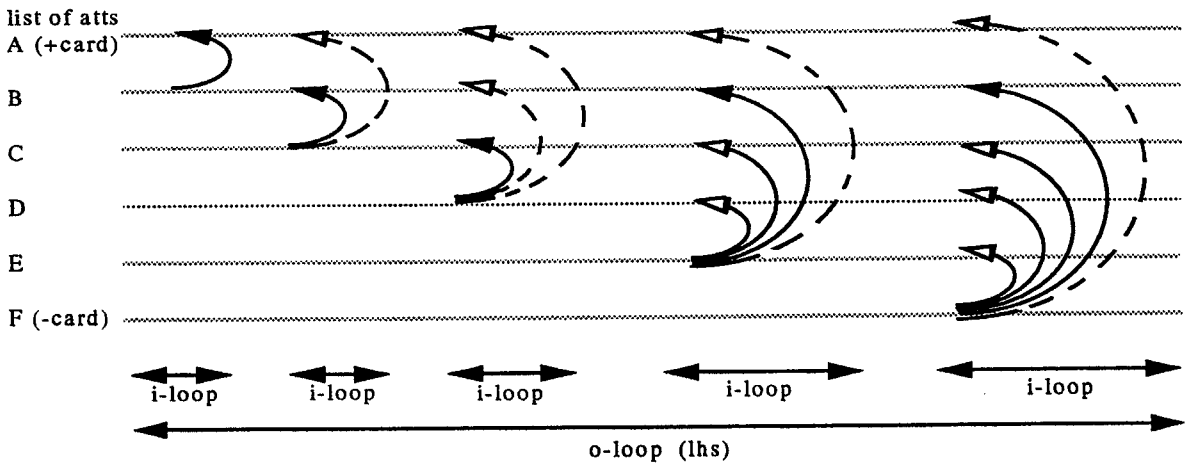
The comparison of attributes is a costly operation because it requires to project them and perform a difference in order to check whether the extension of one of them is contained in the extension of the other. Thus, it is mandatory to minimize the comparison work by avoiding transitive INDs, which are redundant since they can be obtained thru the simple application of the transitivity rule. The *analysis of extensions must discover only direct INDs*. This is achieved by making sure that an IND Ai $\subset$ Aj is never considered in the comparison process unless it is guarantied that it is not a transitive dependency.

Three facts follow from these criteria:

- Attributes of the list are considered as the lhs of possible INDs in *descending* order according to their cardinalities.
- For each possible lhs attribute, those with greater cardinality are considered as possible rhs in *ascending* order according to their cardinalities.
- As soon as an IND Ai $\subset$ Aj is found, all transitive dependencies of Ai are computed immediately. This can be done because all the attributes with higher cardinalities have already been considered as lhs of possible INDs.

The following example illustrates the ideas. Each arrow represents a check whether the attribute on the start of the arrow is a subset of the one on the end of the arrow. If there is a direct IND, the end of the arrow is filled, and if the IND is transitive then the line is dotted.

B $\subset$ A;
C $\subset$ B; C $\subset$ A;
D $\subset$ C; D $\subset$ B; D $\subset$ A;
E $\subset$ B; E $\subset$ A;
F $\subset$ B; F $\subset$ A;



The following facts about the procedure for inferring INDs are a result of the criteria given above:

- There is an *outer* loop (o-loop) where the attribute **Ai** considered in each iteration is on the left hand side of the searched IND. In each iteration we take the attribute with the immediate next lower cardinality with respect to the one considered in the previous iteration (*descending* order). The loop begins with the second attribute with the highest cardinality and ends when all the attributes have been considered.

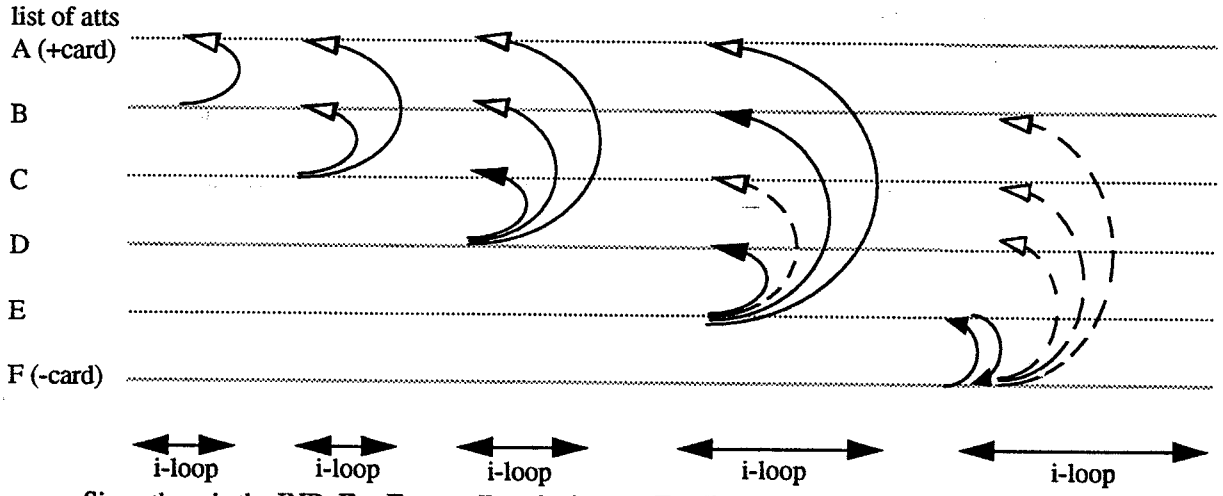
- There is an *inner* loop (i-loop) where the attribute **Aj** considered in each iteration is on the right hand side of the possible IND, thus, it must have greater (or equal) cardinality than the attribute on the left hand side (Ai). The loop begins with the attribute with the immediate next greater cardinality than Ai, and continues in *ascending* order until there are no more attributes to compare with (or an inverse IND exists, as will be explained later). In each iteration, attribute Aj is compared with Ai in order to detect if Aj is a subset (proper or not) of Ai. In this loop , transitive INDs are computed as soon as a direct IND is found. Therefore, it is possible that when Aj is to be considered in an i-loop, the IND Ai $\subset$ Aj has already been obtained by transitivity. Thus, the first thing to be done when entering an iteration of i-loop, is to check whether the IND already exists.

- Once an IND for the attribute Ai being considered in o-loop, on the attribute Aj being considered in i-loop, has been found, i-loop still continues looking for more INDs for Ai because there can be more than one direct IND for Ai.

- The order for considering attributes in each loop, as well as the exact moment to compute transitivities, are *mandatory* to completely eliminate the redundant work of analyzing extensions to extract transitive INDs, since they will be inferred by the transitivity rule.

When an IND Ai $\subset$ Aj is found, if the inverse IND Aj $\subset$ Ai exists, this implies that Ai and Aj have the same values. Therefore, Ai has *exactly* the same INDs as Aj, and since the INDs for Aj have already been obtained (in a previous iteration of o-loop) , then by transitivity Aj simply adopts them and the search for INDs for Ai ends. The following example illustrates the idea:

D $\subset$ C;
E $\subset$ D; E $\subset$ C; E $\subset$ B; E $\subset$ F;
F $\subset$ E; F $\subset$ D; F $\subset$ C; F $\subset$ B;



Since there is the IND F $\subset$ E, as well as the inverse E $\subset$ F, the rhs's of the IND's for F are exactly those of the IND's for E. Notice that F was not compared with A, not with B, because E has already been compared with them and no IND was found. This leads to the following fact on the algorithm: i-loop ends either if there are no more elements to compare with (as stated before), or else *when an inverse IND was found* in its last iteration.

If the IND Ai $\subset$ Aj exists and the cardinalities of both attributes is the same, then, Ai and Aj are identical, and as consequence, the inverse IND Aj $\subset$ Ai also exists.

Next, the algorithm for the extraction of ***unary*** inclusion dependencies is presented.

```
Procedure extract_unary_IND
        obtain_lists_attrs_same_type
        for each list L do
              obtain_unary_IND_for_list
        end_for
end_procedure

Procedure obtain_lists_attrs_same_type
        for each domain D do
             for each attribute Ai  belonging to D do
                  obtain_cardinality;     {by querying the catalog or by computing it with an SQL statement}
                  insert_in_decreasing cardinality_ordered_list;
             end_for
        end_for
        initialize INDs     {from Foreign Keys in the catalog}
end_procedure

Procedure obtain_unary_IND_for_list
 {lhs: attribute considered as the left hand side of the possible IND,
  rhs: attribute considered as the right hand side of the possible IND,
  IND_found: indicates that an IND has been found,
  IND_inv_found: indicates that the inverse IND has been found}
 lhs := second attribute of the list                {second attribute with the lowest cardinality}
 while  not end of list do                          {o-loop:  an iteration for each element of the list}
        rhs := attribute previous to lhs in the list          {its cardinality is greater or equal than the one of lhs}
        IND_inv_found := false
        while not IND_inv_found or not end of list  do                  {i-loop: each attr with higher or equal cardinality
                        than lhs is considered as rhs of a possible  IND of lhs, except when an inverse IND is found}
          if (lhs, rhs) not in INDs then          {if the IND has not already been obtained by initialize/transitivity}
             compare_extensions_attrs (lhs, rhs, IND_found)        {see below}
             if IND_found  then
                  insert (lhs, rhs) into INDs              {lhs ⊂ rhs}
                  if cardinality(lhs) = cardinality (rhs)  then
                          insert (rhs, lhs) into INDs                {inverse IND}
                          insert (rhs, X) into INDs for all (lhs, X) in INDs
                  end_if
                  compute_transitive_INDs     { insert into INDs and mark them as transitive }
             end_if
          end_if
          rhs := attribute previous to rhs in the list      {attr with next greater cardinality than current rhs}
                                                                    {end i-loop}
        lhs := attribute next to current lhs in the list    {the new lhs is the attr with next lower card than the actual lhs}
 end_while                                                          {end o-loop}
end_procedure
```

Procedure *compare_extensions_attrs (lhs, rhs, IND_found)* can be performed in different ways. One possible algorithm is to sort the relation R (containing lhs) on the lhs values, sort the relation S (containing rhs) on the rhs values, and then match the sorted files: as soon as a lhs value in R not present as a rhs in S is found, the process stops, and IND_found is set to false. In the best case, this would happen with the first page of R and the first page of S. If the IND exists, then both sorted files will be read up to the end of R, and IND_found is set to true, giving a cost

$$P \log P + P' \log P' + P + P'$$

where P and P′ are the number of pages of R and S, respectively. Actually, there is no need to sort whole records, just the lhs and rhs attributes, and duplicate values can be eliminated during the internal phase of the sorts, so that real cost should be lower than this formula.

Another algorithm would be: read S and build a structure in memory with all values of rhs found; then read R, and for each lhs value look if it is present in the structure: if yes, keep going on; if not, stop and set IND_found to false. The kind of the memory structure (bit array, hashed table, search tree) depends on the domain of rhs. This algorithm is only possible if the domain allows for such a structure to be kept in main memory. The cost is $P + P'$

The number of time this comparison procedure is performed is, in the worst case where no IND is found, the number of combinations of 2 elements taken out of l, where l is the length of the list, and this for each list. Assuming the first algorithm for the comparison procedure is used, and the worst cost in each case, the upper bound for the total cost is (taking the sort of R out of the i-loop into the o-loop):

$$\sum_{lists}(l-1)\,P\log P + \binom{l}{2}(P'\log P' + P + P')$$

Actual cost will be lower, not only because of previous considerations, but also because the worst case for the cost of each comparison (many INDs) is best for the number of comparisons, and vice versa, similarly to what happens with FDs.

3.2 Binary INDs.

The problem with binary INDs is that there is no criteria to configure the ordered lists of possible *binary* INDs required to maximize the use of transitivity as was done for unary INDs. Thus, instead of generating these lists, heuristic D will be used: *a necessary condition for the existence of a binary IND is that there exists a unary IND for each pair of the corresponding attributes*. Now, the question is: in which order the unary dependencies are to be considered? To take advantage of this heuristic while at the same time maximize the use of transitivity the process is performed in three steps:

i) In the first one, for every pair of relations, all pairs of unary *direct* INDs between them are searched. For each such pair of dependencies, the *binary direct* IND is tested by analyzing the extension of the involved relations.

ii) The transitivity rule is applied to compute *binary transitive* INDs from the binary direct ones obtained in i).

iii) In this last step, *unary transitive* dependencies are considered, but only those pairs for which the corresponding binary one has not been computed in ii). Since not every pair of unary direct INDs, considered in step i), gives rise to a binary direct one, some pairs of unary transitive INDs give rise not to a binary transitive one (computed in ii), but to a direct one (obtained here in iii).

To illustrate the ideas, let us assume that we have the following pairs of unary INDs:

1) R1.X ⊂ R2.W; R1.Y ⊂ R2.Z; (direct dependencies)
2) R2.W ⊂ R3.P; R2.Z ⊂ R3.Q; (direct dependencies)
3) R1.X ⊂ R3.P; R1.Y ⊂ R3.Q; (transitive dependencies)

two different situations may occur:

Situation A:

Step i): the pairs 1) and 2) of unary direct INDs are considered and the corresponding binary ones are tested successfully:

1) R1.X ⊂ R2.W; R1.Y ⊂ R2.Z → R1.X,Y ⊂ R2.W,Z (direct binary dep.)
2) R2.W ⊂ R3.P; R2.Z ⊂ R3.Q → R2.W,Z ⊂ R3.P,Q (direct binary dep.)

Step ii): the binary transitive IND R1.X,Y ⊂ R3.P,Q is computed from the binary direct ones R1.X,Y ⊂ R2.W,Z and R2.W,Z ⊂ R3.P,Q obtained in i).

Step iii): the pair 3) of transitive unary dependencies is not considered because it would give rise to the same transitive binary IND obtained in step ii).

Situation B:

Step i): the pairs 1) and 2) of unary direct INDs are considered, but at least one of the corresponding binary ones fails to hold in the database:

Step ii): therefore, the transitivity rule cannot be applied.

Step iii): the binary IND corresponding to the pair 3) of transitive unary dependencies is now tested:

3) $R1.X \subset R3.P$; $R1.Y \subset R3.Q \rightarrow R1.X,Y \subset R3.P,Q$
Notice that this is a binary direct dependency and not a transitive one as in situation A.

The procedure to extract binary IND's guarantees that every transitive dependency that exists, is obtained by merely applying the transitivity rule. Thus, no redundant work is performed and the analysis of extensions of relations is required only for direct dependencies. Next, the procedure according to the ideas given above is presented.

```
Procedure extract_binary_IND
    {step i: binary direct deps. from unary direct ones}
    for every pair of unary direct INDs between two relations do
       check_binary_IND (lhs's, rhs's)    {test extensions for existence of IND and its inverse}
    end_for
    {step ii: binary transitive deps. from binary direct ones}
    compute_transitive_INDs                {apply transitivity rule}
    {step iii: binary direct deps. from unary transitive ones}
    for every pair of unary transitive INDs between two relations do
       if corresponding binary IND not exists then   {if not found as transitive in step ii}
          check_binary_IND (lhs's, rhs's)
       end_if
    end_for
end_procedure


Procedure check_binary_IND (lhs's, rhs's)
       lhs := lhs's        {lhs's of unary INDs}
       rhs := rhs's        {rhs's of unary INDs}
       compare_extensions_attrs (lhs, rhs, IND_found)    {analysis of extensions of relations}
       if IND_found then
          insert (lhs, rhs) into INDs
          if cardinality (lhs) = cardinality (rhs) then      {inverse IND exists}
                insert (rhs, lhs) into INDs
          end_if
       end_if
end_procedure
```

3.3 Ternary INDs.

Heuristic E gives the necessary condition for their existence: *binary INDs must exist between any pair of the corresponding pairs of attributes*. For example, if the ternary inclusion dependency $R1.X,Y,Z \subset R2.P,Q,R$ exits, then the binary INDs $R1.X,Y \subset R2.P,Q$, $R1.Y,Z \subset R2.Q,R$, and $R1.Z,X \subset R2.R,P$ also exist. Therefore, since binary dependencies have been obtained before, it is possible to apply this heuristic to select the ternary ones to be analyzed.

```
Procedure extract_ternary_IND
        for every pair of relations do
           for every triplet T (lhs, rhs) formed by 3 binary INDs between these relations do
                compare_extensions_attrs (lhs, rhs, IND_found)
                if IND_found then
                        insert (lhs, rhs) into INDs
                end_if
          end_for
        end_for
end_procedure
```

For both binary and ternary INDs, the same considerations on algorithms and costs of procedure *compare_extensions_attributes,* given for unary INDs, apply. No easy formula can be given for the number of comparisons.

4. Related and future work

The only other work that we know in this area is the one by Mannila (and Räihä), leading to [Mannila91]. A number of similarities exist, as well as big differences. He focuses on the number of operations, while we consider disk accesses as the primary factor of the cost. His algorithm for FDs extraction orders candidates by rhs, while our approach uses a lhs sequence, which minimizes disk cost. These and other considerations produce quite different algorithms.

This paper considers FDs and INDs *separately.* Even if no axiomatization of FDs and general INDs taken together can exist [Casanova84], some benefit could be obtained from their joint consideration, and new algorithms to extract both kinds of dependencies could be developed. We intend to pursue work on this topic.

5. Conclusions

Semantic knowledge about a database is needed for many uses, and relational catalogs are usually rather poor in representing this intensional meaning One form of knowledge acquisition from a database is the extraction of constraints from its extension. Data dependencies, particularly Functional and Inclusion Dependencies, are important constraints.

In this paper, we have presented algorithms to extract FDs and INDs from relational databases. We have shown how, by adopting intelligent choices in the order of candidate dependencies to be tested, derived dependencies can be found by applying implication rules and their testing is avoided, and the number of disk accesses is minimized. Even if the cost of constraint extraction by extension analysis remains high, with these algorithms it becomes more affordable.

Acknowledgments

This research has been partially supported by the Spanish PRONTIC program, under project TIC89/0303. We thank R. Casas for assistance in a combinatorial approximation formula.

References

[Armstrong74] W. W. Armstrong: "Dependency structures of data base relationships". *Information Processing 74* (Proceedings, IFIP World Computer Congress, Stockholm, 1974). North Holland, Amsterdam, 1974, pages 580-583.

[Casanova84] M.A.Casanova, F.Fagin & C.H.Papadimitrou: "Inclusion Dependencies and their Interaction with Functional Dependencies".*JCSS, Vol.28, No.1,* Feb.1974, pages 29-59.

[Castellanos91] M.G. Castellanos & F. Saltor: "Semantic Enrichment of Database Schemas: an Object Oriented Approach". In Kambayashi et al (eds.): *Proceedings, 1st Intnl. Workshop on Interoperability in Multidatabase Systems* (Kyoto, 1991). IEEE-CS Press, 1991.

[Maier83] D. Maier: *The Theory of Relational Databases*. Computer Science Press, Rockville, MD, 1983.

[Mannila91] H. Mannila & K-J. Räihä: "Algorithms for Inferring Functional Dependencies". Dept. of Computer Science, Univ. of Helsinki, *Report C-1991-41*.

[Thalheim91] Bernhard Thalheim: *Dependencies in Relational Databases*. Teubner, Stuttgart-Leipzig, 1991.

Appendix

We show one example that is used to illustrate the upper bound in the number of accesses to disk for a relation of degree 10 with 1000 tuples of size 100 (bytes). The size of the page is 1024 (bytes), and the average of the number of distinct values is 100. For estimating the time invested in accessing disk, we assume an access time of 15 msecs.

$d = 10$
$C = 1000$
$v = 100$
$b = [p/t] = [1024 / 100] = 10$

Alternative 1.

$$(2^d\text{-}2)\ (v\ [C/b]) = (1024\text{-}2) \times 100 \times 100 = 10{,}220{,}000 \text{ accesses}$$

then the time invested in accesses to disk is 153,300 secs, that is, 42.58 hours !

Alternative A.2

$$\begin{aligned}(2^d\text{-}2)\ (S + P) &= (2^d\text{-}2)\ (C/b \log (C/b) + C/b) \\ &= 1022\ (100 \log 100 + 100) \\ &= 1022 \times 764.3856 \\ &= 781{,}202 \text{ accesses}\end{aligned}$$

then the time invested in accesses to disk is 11,718 secs, that is, 3.25 hours !

Alternative A.3

$$\begin{aligned}NS + (2^d\text{-}2)L &= (2^d / (\sqrt{(\pi / 2)}\ \sqrt{d}))\ (C/b \log (C/b) + (2^d\text{-}2)\ (C/b) \\ &= (1024/(\sqrt{(\pi / 2)}\ \sqrt{10}))\ (100 \log (100)) + (1022)\ (100) \\ &= (1024/3.9633)\ (664.385) + 102{,}200 \\ &= 273{,}857 \text{ accesses}\end{aligned}$$

then the time invested in accesses to disk is 4,107 secs, that is, 1.14 hours!

Alternative A.4

$$\begin{aligned}NS + NP = N\ (S + P) &= (2^d / (\sqrt{(\pi / 2)}\ \sqrt{d}))\ (C/b \log (C/b) + C/b) \\ &= (1024/3.9633)\ (764.3856) \\ &= 197{,}495 \text{ accesses}\end{aligned}$$

then the time invested in accesses to disk is 2,962 secs, that is, 0.82 hour !

Departament de Llengatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

List of research reports (1993).

LSI–93–1–R "A methodology for semantically enriching interoperable databases", Malú Castellanos.

LSI–93–2–R "Extraction of data dependencies", Malú Castellanos and Fèlix Saltor.

LSI–93–3–R "The use of visibility coherence for radiosity computation", X. Pueyo.

LSI–93–4–R "An integral geometry based method for fast form-factor computation", Mateu Sbert.

LSI–93–5–R "Temporal coherence in progressive radiosity", D. Tost and X. Pueyo.

LSI–93–6–R "Multilevel use of coherence for complex radiosity environments", Josep Vilaplana and Xavier Pueyo.

Internal reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics (U.P.C.)
Pau Gargallo 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es