COMPARISON OF FUNCTIONAL DEPENDENCY EXTRACTION METHODS AND AN APPLICATION OF DEPTH FIRST SEARCH

by

KANIKA SOOD

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2014

THESIS APPROVAL PAGE

Student: Kanika Sood

Title: Comparison of Functional Dependency Extraction Methods and an Application of Depth First Search

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Christopher Wilson	Chair
Boyana Norris	Member

and

Kimberly Andrews Espy	Vice President for Research & Innovation/ Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2014

© 2014 : Kanika Sood

THESIS ABSTRACT

Kanika Sood

Master of Science

Department of Computer and Information Science

June 2014

Title: Comparison of Functional Dependency Extraction Methods and an Application of Depth First Search

Extracting functional dependencies from existing databases is a useful technique in relational theory, database design and data mining. Functional dependencies are a key property of relational schema design. A functional dependency is a database constraint between two sets of attributes. In this study we present a comparative study over TANE, FUN, FD_Mine, FastFDs and Dep_Miner, and we propose a new technique, KlipFind, to extract dependencies from relations efficiently. KlipFind employs a depth-first, heuristic driven approach as a solution. Our study indicates that KlipFind is more space efficient than any of the existing solutions and highly efficient in finding keys for relations.

CURRICULUM VITAE

NAME OF AUTHOR: Kanika Sood

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Mody Institute of Technology & Science, Rajasthan, India

DEGREES AWARDED:

Master of Science, Computer & Information Science, 2014, University of Oregon
Bachelor of Technology, Computer & Information Science, 2011, Mody Institute of Technology & Science

AREAS OF SPECIAL INTEREST:

Databases, Automata Theory

PROFESSIONAL EXPERIENCE:

Associate Systems Engineer, IBM, India, 2011-2012

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1

II. MAIN TECHNIQUES/TOOLKIT 4

III. TANE 8

SearchStrategy 9
Pruning the Search Space 9
Challenges 9
Computing with Partitions 11
Algorithm 13
Advantages and Disadvantages 16
Time Complexity 16

IV. FUN 17

Embedded FDs 17
Free Sets 17
Closure and Quasi-closure 18
Approach 18
Algorithm 19
Advantages 21
Disadvantages 21

V. FD_MINE 22

Classification of FDs 22

Chapter Page

Equivalences 22

Approach 23

Algorithm 25

VI. FASTFDS 28

Canonical Cover 28

Agree Sets 28

Algorithm 29

Advantages 30

Disadvantages 30

Time Complexity 31

VII. DEPMINER 32

Armstrong Relation 32

Approach 32

Algorithm 34

Advantages 34

Time Complexity 36

VIII. KLIPFIND 37

Heuristic of KlipFind 37

Conjecture 39

Algorithm 39

Time Complexity 42

Worst Case Scenarios 42

Advantages and Disadvantages 43

Chapter Page

IX. COMPARISON OF FD EXTRACTION METHODS 44

Categorization 44

Differences among Candidate Generate-and-test Strategies 45

Illustration 46

Result of the Bernoulli Example 67

Time Complexity 67

X. CONCLUSION AND FUTURE WORK 69

LIST OF FIGURES

Figure Page

1. Bernoulli Example on TANE: Stage 1. Level 0 has only one element, the empty set and level 1 has six candidates which are all the singleton attributes of the relation. The numbers show the cardinality of the candidates. 47

2. Bernoulli Example on TANE: Stage 2. Level 2 candidates are generated by combining all the possible sets of candidates at level 1. At each level the number of attributes in a candidate increase by one in count. 47

3. Bernoulli Example on TANE: Stage 3 At every level the cardinality of each canddate is compared with the cardinality of its parents in the previous level. If the cardinality is the same for any of the cases this indicates that there is a functional dependency existing there. 48

4. Bernoulli Example on TANE: Stage 4 The same step as in previous stage is followed to get the complete lattice 48

5. Bernoulli Example on TANE: Stage 5 At this stage keys are identified and this becomes the final stage of the lattice. 48

6. Bernoulli Example on FUN: Stage 1 Here also in level 0 there is only the empty set and level 1 has all the singleton elements of the relation. 49

7. Bernoulli Example on FUN Stage 2 At every level the candidates are generated by combining candidates from the previous level and the cardinality is checked to look for non-free sets and keys. 49

8. Bernoulli Example on FUN: Stage 3 If a key or a non-free set is obtained then it is not included in the next level as a subset of any candidate. 50

9. Bernoulli Example on FUN: Stage 4 51

10. Bernoulli Example on FUN: Stage 4(Cleaner version) 52

11. FastFDs: Stage 1 The difference set shown in the upper portion of the figure above is found by comparing each tuple with every other tuple in the the relation and combined on the basis of the same value of the attributes, Then difference set for each of the attributes(shown in the table in the above figure) is found by combining those values that contain that attribute and remove that attribute from that set of candidates. If there is a singleton attribute like C for example then the difference set of C would contain an empty set. 53

12. FastFDs: Stage 2 This stage is obtained from the previous stage by reducing it to a set of minimal candidates , by removing supersets of the candidates present in the respective difference sets. 54

13. FastFDs: Stage 3 The table shows the minimum cover of the difference set of all the attributes, which can be obtained from combining those attributes that alone can represent the entire set of attributes shown in the previous figure. The minimal cover of each of the attribute gives us the functional dependencies as shown in the figure above. 55

Figure Page

14. FastFDs: Stage 4 The lattice for FastFDs follows the lexicographic order at every level. 56

15. Bernoulli Example on KlipFind: KlipFind also starts off the same way as FUN, just that it follows a depth first approach.The above figure shows a stage where the leftmost candidate at level 2 is been explored and it goes down until it comes to the leaf nodes of the tree. After this stage pruning starts. 57

16. Bernoulli Example on KlipFind: This stage shows the lattice before the first prune i.e. AB and its children: Stage 2 58

17. Bernoulli Example on KlipFind: Stage 3 Before pruning ABDF and ABD candidates from the lattice. 59

18. Bernoulli Example on KlipFind: Stage 4 After pruning ABD from the lattice. 59

19. Bernoulli Example on KlipFind: Stage 5 After pruning ABEF and ABE and before pruning ABF and AB. 60

20. Bernoulli Example on KlipFind: Stage 6 After pruning ABF and AB and before pruning ACD. 61

21. Bernoulli Example on KlipFind: Stage 7 After pruning ABF and AB and before pruning ACEF , ACE, AC one by one. 62

22. Bernoulli Example on KlipFind: Stage 8 The stage obtained after pruning AC. 63

23. KlipFind In a similar fashion it is done for all the singleton attributes with the expansion as given in Figure 19. Not all the nodes shown in this figure are in the memory all the time.It is just showing the full expansion of the tree. The algorithm halts once all the attributes are done. 63

24. KlipFind Example 2: Stage 1 64

25. KlipFind Example 2: Stage 2 65

26. KlipFind Example 2: Stage 3 66

LIST OF TABLES

Table Page

1. Student database 2
2. Test for checking if a FD(B$\rightarrow$ A) holds or not 9
3. Stripped partitions 12
4. Categorization table 45
5. Bernoulli Example 56
6. Random Example 64

CHAPTER I

INTRODUCTION

In today's era with huge data inflow it is necessary to have large databases. In building large databases and maintaining them efficiently, a good relational schema design plays an important role.The problem addressed in this paper is to suggest an efficient algorithm for extracting functional dependencies from a relation. Functional dependency is a key property of the relational schema design. It is a constraint of a database between two sets of attributes. Having crisply defined functional dependencies will make querying on a database system efficient. Given any random database, it is important to find out the functional dependencies among the data to know how efficient the database is. It is also a key technique in database design, database analysis and in normalization of databases.

The discovery of functional dependencies from the relationships present in the database has been an active topic of research for the past few years and efficient solutions for finding dependencies have been provided by researchers which has proved to be relevant in the past. An important use of this could be for instance, we have a dataset for the people in US who have been detected with cancer. The dataset tries to capture as many factors responsible for the disease as possible. Some examples could be hereditary within the family, smoking or drinking. In this case, based on millions of records of the patients who have been struck by the disease, there can be some factors identified that are present in most of the cases, or whenever they were present their effect had been adverse. Let us say the dataset has one million records out of which it seems like 95 % of the people detected with lung cancer used to smoke. Acquiring such knowledge from the relation depending on the relationship of set of attributes among themselves can be very helpful in this field. This research topic is not just confined to this area but can have contributions in almost any field.

Another example could be in a student database as shown in Table 1 .From the student id attribute we can get other relevant details of the students like their names, year of admission etc. Initial work in the field involved comparing tuples and confirming whether the functional dependency is satisfied or not.They did not make use of the dependencies discovered in the earlier stages to obtain new knowledge. But it makes this approach non-scalable and impractical for large

databases with more number of attributes and data. Later work has taken into consideration this drawback and developed more scalable approaches which keeps track of information collected in the previous stages to gather further information in the later stages.

Student_id	First Name	Last Name	Date of birth	Year of enrolment	Department name
916768	Jessica	Kidman	09-09-1988	2011	Psychology
928999	Crystal	Brown	19-12-1980	2014	CIS
91111	Himani	Sood	15-05-1988	2012	English Literature
967800	Isma	Hamid	24-06-1983	2013	Chemistry
912345	Minoo	DeRaj	21-01-1991	2013	CIS
923455	Akash	Agnihotri	09-09-1988	2013	CIS
919191	Shashank	Rao	21-09-1989	2013	CIS
919992	Neeraj	Chaudhary	11-08-1983	2012	CIS

TABLE 1. Student database

Based on the strategy that these algorithms identify functional dependencies we can broadly classify them into two categories: Breadth first search and depth first search. The first of these approaches is to traverse through the attributes and their supersets in a breadth first fashion. Few of these algorithms discussed in the paper are : TANE Huhtala et al. [1999], FUN Novelli and Cicchetti [2001],FD_Mine Yao et al. [2002] and Dep-Miner Lopes et al. [2000]. The other classification based on the approach is a depth first traversal over the attributes. One such algorithm is FastFDs Wyss et al. [2001]. Although there are other algorithms as well that fall in this class but we shall cover in detail FastFDs alone.

In this study we do an in-depth study of the above stated five algorithms and consider the various aspects which each algorithm revolves around.These aspects have been the grounds for the next upcoming algorithm in the chronological order. The contribution of this study also involves a new algorithm suggestion that we call KlipFind [1]. One of the algorithms FastFDs is a depth first search version of the other algorithm Dep-Miner. Chapter II lists the main techniques/ toolkit used in the study. We covered in detail the TANE algorithm in chapter III followed by FUN algorithm in the next chapter. In chapter V, FD_Mine is discussed.Fast FD is explored in the following chapter. Chapter VII has the description of Dep-Miner algorithm.

The following talks about the suggested new algorithm which is more space efficient than the algorithms given so far. While FUN uses a breadth first search strategy on the other hand our

[1]The word "klip" means "deep" in Chinook jargon language

algorithm suggests a depth first search approach on the FUN algorithm. The next chapter makes a comparison of above mentioned algorithms and explains by illustrating the algorithm over an example. The final chapter concludes as to what is the goal of this study and its contribution.

We did not include an implementation of the algorithm given the time constraint and have restricted the paper to mention the algorithm and cover the worst cases and worst time complexity.

CHAPTER II

MAIN TECHNIQUES/TOOLKIT

1. Functional Dependency(FD): In a relation R, a functional dependency $X \rightarrow A$ where $X \subseteq R$, $A \in R$ holds if each X value is related to only one value of Y.

   We say a FD $X \rightarrow Y$ holds on a relation r if it is supported by the tuples of r. Sometimes this is expressed as $\models X \rightarrow Y$.

2. Minimal FD: A functional dependency $X \rightarrow A$ is said to be minimal if A is not functionally dependant on any proper subset of X, i.e. if $Y \rightarrow A$ does not hold in R for any $Y \subset X$.

3. Trivial FD: A functional dependency $X \rightarrow A$ is said to be trivial if $A \in X$

4. Approximate FD: A functional dependency that almost holds on all the tuples in a relation. This involves removing those tuples from the relation which do not satisfy.

5. Approximateness of a FD: Minimum no. of tuples that need to be removed from relation r for $X \rightarrow A$ to hold in r.

6. Embedded FD: Dependencies that hold over a subset of the attribute set initially considered. Novelli and Cicchetti [2001]

7. Minimal cover: A minimal cover of F is a set of dependencies such that F logically implies all dependencies in the canonical cover of F and the canonical cover of F logically implies all dependencies in F.

8. Canonical cover:The canonical cover can be found by following the steps below:

   (a) Start with the closure of all the attributes.

   (b) Rewrite with a single attribute on right hand side.

   (c) Cross out trivial dependencies.

   (d) There are dependencies in the list that can be implied by other dependencies in the list. Strike such dependencies out.

   (e) Eliminate redundant dependencies.

(f) Combine dependencies with same left hand sides.

9. Free Set: Let X $\subseteq$ R be a set of attributes. X is a free set in r, an instance of relation over R, if and only if: $\exists$ X $\in$ X, $|\pi'_X| = |\pi_X|$ where $|\pi_X|$ stands for the cardinality of the projection of r over X. Every single attribute is a free set. The left hand side of any minimal dependency is also a free set.

10. Closure: The set of all values that can be determined from X using FD X $\rightarrow$ A

11. Quasi-Closure: The grouping of the closure of X and all its maximal subsets.

12. Partition: A set of tuples that agree on an attribute value.

13. Rank of a partition : The rank of a partition is the number of equivalence classes in that partition. It is denoted by : $|\pi|$

    For example: $\pi_A = \{\{1,2,4\},\{3,5,7\},\{6\}\}$ Here $\pi_A = 3$

14. Stripped Partition: A partition with equivalence classes of size one removed.

15. Agree sets:Every pair of tuple , the attributes for which they have the same value. Agree sets are the complement of difference sets.

16. Disagree sets: If t1 and t2 do not appear together in some stripped partition , then t1 and t2 disagree on every attribute.Such tuples that disagree form a set and such sets are said to be disagree sets.

17. Equivalences: A relation is said to be equivalent if and only if it is reflexive, symmetric and transitive.

18. Partition Refinement: A partition π refines another partition π if every equivalence class in π is a subset of some equivalence class of π

19. Armstrongs Axioms: Below are the 3 axioms given by Armstrong.

    (a) 1. Reflexivity : If Y $\subseteq$ X , then X $\rightarrow$ Y

    (b) 2. Augmentation: If X $\rightarrow$ Y then XZ $\rightarrow$ YZ for any Z

    (c) 3. Transitivity: If X $\rightarrow$ Y and Y $\rightarrow$ Z then X $\rightarrow$ Z

20. Trivial dependencies: Dependencies of the form X $\rightarrow$ X or $XY \rightarrow X$ or $XY \rightarrow Y$ where right hand side of the dependency is a part of the input(or left hand side of the dependency). Such dependencies are said to be trivial dependencies.

21. Proper subset : All dependencies of the form X $\rightarrow$ Y except the ones given below form the proper subset:

    (a) $\emptyset \rightarrow$ Y

    (b) X $\rightarrow$ Y

22. Key : A set of attributes whose value uniquely defines each row in the relation.

23. Superkey: Set of attributes for a relation which has a key(s) as its subset.

24. Search space: In a relation, for a dependency X $\rightarrow$ Y, the possible values of X can be any attribute or set of attributes of the relation. The possible values of X form the search space of the functional dependency.It reduces the search space using pruning.

25. Closure: Let F be a set of functional dependencies. The closure of a functional dependency is the set of all the functional dependencies that can be deduced from F including F itself. It is denoted by F^+.

26. Cardinality of a partition: The number of groups in a partition is called its cardinality. It is denoted by $|\pi_X|$

27. Embedded dependencies: Given a set of functional dependencies that hold in the relation, embedded dependencies are the dependencies that are valid in the projection of the relation over a subset of its attributes. The set of embedded functional dependencies id given as below:

    $\mathcal{F}[X] = \{Y \rightarrow Z / \mathcal{F} \models Y \rightarrow Z \wedge YZ \subseteq X\}$

28. Projection of Functional Dependency: The functional dependencies that hold for an attribute subset X of a relation R.It is another term used for Embedded functional dependencies. The set of embedded functional dependencies id given as below:

    $\mathcal{F}[X] = \{Y \rightarrow Z / \mathcal{F} \models Y \rightarrow Z \wedge YZ \subseteq X\}$

29. Armstrong relation: An Armstrong relation for a set of functional dependencies (FDs) is a relation that satisfies each FD implied by the set but no FD that is not implied by it.

30. Maximal set: The largest possible set of attributes for an attribute X which does not determine X.

31. Level wise approach: Level wise in this study refers to the level by level exploration of the lattice/tree. It can also be defined as the level-wise view of the lattice of the candidates.

CHAPTER III

TANE

TANE algorithm was presented in 1999. It was the first efficient algorithm for discovering functional,non-trivial dependencies in a relation.It gives a new way of finding if a functional dependency holds in a relation or not. This new approach mainly involves the representation of the attribute sets by equivalence class partitions.TANE also gives a way of searching the space of functional dependencies efficiently. It partitions the rows with respect to its attribute values.This scheme has an advantage that it makes the validity of functional dependency fast.Also discovery of approximate functional dependencies becomes faster. TANE is favourable for relations with a large number of tuples.

It represents attribute sets by equivalence class partitions. Functional dependencies are discovered by taking into consideration those tuples that agree for some set of attributes. Whenever they have the same value on the left hand side of a dependency then its checked if they have the same value on the right hand side too.

The algorithms for finding functional dependencies can be classified into two categories:

1. Candidate generate and test approach
2. Minimal cover approach

TANE is a candidate-generate-and-test approach which uses level wise search to explore the search space.It reduces the search space using pruning. TANE starts with a small left hand side and prunes the search space as and when possible.For this it uses partitioning the tuples based on the attribute values.

TANE provides two tests for checking if a functional dependency holds or not.

1. A dependency holds if π_X refines π_A

   A partition B refines another partition A if and only if every equivalence class in B is a subset of some equivalence class in A. For example if we have the case as in Table 2 :

   Here $\pi_A = \{\{1,2,4\},\{3,5,7\},\{6\}\}$ and $\pi_B = \{\{1,2,4\},\{3,5\},\{7\},\{6\}\}$

   Every equivalence class in B , i.e. $\{1,2,4\},\{3,5\},\{7\}and\{6\}$ is a subset of either $(\{1,2,4\},or\{3,5,7\}or\{6\}\})$ in A.

Row	A	B
1	2	4
2	2	4
3	3	5
4	2	4
5	3	5
6	4	6
7	3	3

TABLE 2. Test for checking if a FD(B $\rightarrow$ A) holds or not

2. A dependency holds if $|\pi_X| = |\pi_{X \cup A}|$

   It is a simpler test to check if a functional dependency holds or not. If case (1). If the LHS and RHS of case (2) are not equal then for sure the dependency involved does not hold.

Search strategy

TANE looks for functional dependencies starting from a single element and increments one attribute at each level. When TANE tests for a dependency, it looks for dependencies of the form

X $\setminus\{A\}$ $\rightarrow$ A where A $\in$ X. That way TANE guarantees that it only considers non-trivial dependencies as trivial dependencies are not relevant. The small-to-large direction of the algorithm makes sure that only minimal dependencies are generated by the algorithm.

Pruning the search space

TANE forms a lattice of candidates formed by the attributes of the relation. At every level it keeps moving down by adding one more attribute to the candidate for that level. It keeps going down the lattice until there are no more candidates that can be generated for the next level and it finds all the minimal dependencies.

Challenges

One of the biggest challenge in finding functional dependencies is to confirm that the functional dependencies that the algorithm outputs are minimal. If not then there could be many extra dependencies that could be generated which cannot be counted since there exists another functional dependency with a smaller left hand side for the same functional dependency.

In order to test the minimality of a functional dependency X $\rightarrow A$ what really needs to be done is to check the existence of a functional dependency of the form Y $\rightarrow A$ where Y $\subseteq$ X. TANE stores this information in C(Y) of right hand side candidate of Y. In other words, C(Y) stores the initial right hand side candidates of Y.

In order to check if the dependencies are minimal, below are the steps taken by TANE. Let the dependency to be checked for minimality be X $\rightarrow$ Z. The steps taken are:

1. Rhs candidate pruning

   For simplicity sake, let the relation in consideration be $\{A, B, C, D, Z\}$. Let the left hand side of the dependency, X be AB. The first check should be if there exists a Y such that Y $\subseteq$ X and Y $\rightarrow$ Z. If such a dependency exists, then X $\rightarrow$ Z is not minimal. TANE takes care of this by storing a set C(Y) such that it has all those attributes that can be derived from Y. For example, if there exists Y $\rightarrow$ Z exists then C(Y) should contain Z.

   So for a dependency X $\rightarrow$ Z to be minimal, $\forall$ Y $\in$ X, C(Y) = $\emptyset$

   TANE takes into consideration only proper subsets.

   Also, if X $\supset$ Y and C(X) = $\emptyset$ then $\forall$ Y , C(Y) = $\emptyset$

2. Rhs$^+$ candidates C(X) has the initial set of right hand side candidates. TANE provides an improved version which is given as C$^+$(X).

   C$^+$(X) = {A $\in$ R | $\forall$ B $\in$ X : X $\setminus\{A, B\} \rightarrow \{B\}$ does not hold }

   It is important to have a check that the left hand side of the dependency should not have any internal functional dependencies. For example : for AB $\rightarrow$ C should not have a dependency like A $\rightarrow$ B or B $\rightarrow$ A or else AB $\rightarrow$ C would not be minimal.

   The Lemmas given by authors of TANE make sure that they remove internal functional dependencies from each functional dependency that they find out. Huhtala et al. [1999]

   Let A $\in$ X and let X $\setminus\{A\} \rightarrow A$ be a valid dependency. The dependency X $\setminus\{A\} \rightarrow$ A is minimal if and only if, $\forall B \in X, A \in C^+(X\setminus\{B\})$.

   For the dependency ABC $\rightarrow$ D, the above lemma gets rid of the functional dependencies of the form AB $\rightarrow$ B , AB $\rightarrow$ C , B $\rightarrow$ C and others of the same form.

The below lemma removes additional attributes from C(X). Let B $\in$ X and let X $\backslash\{B\}$ be a valid dependency. If X $\rightarrow$ A holds, then X $\backslash\{B\}\rightarrow$ A holds.

$$C'\bar{(}X) = \begin{cases} R \setminus X \; if \; \exists \, B \; \in \; X : X \setminus \{B\} \rightarrow B \; holds & (3.1) \\ \emptyset \quad otherwise. & (3.2) \end{cases}$$

Now if X has a proper subset Y such that Y $\backslash\{B\}\rightarrow B$ holds for some B $\in$ Y then further we can remove from C(X) all A $\in$ X $\setminus$ Y. The set removed from the above stated rule is :

$C''\bar{(}X) = \{A \; \in \; X \mid \exists \, B \; \in X \setminus \{A\} : X \setminus \{A, B\} \rightarrow B \; holds\}$

3. Key pruning

   ”An attribute is a superkey if no two tuples agree on X i.e. partition π_X consists of singleton equivalence classes only.The set X is a key if it is a super-key and no proper subset of it is a superkey.” Huhtala et al. [1999]

   This rule makes sure than anything that has the super key in it does not go to the next level.

Computing with partitions

TANE gives 2 ways to reduce time and space requirement :

1. Replace partitions with stripped partitions

   Instead of considering all the partitions, TANE uses stripped partitions as shown in Table 3. The reason being singleton equivalent classes do not contribute in checking the validity of a dependency dependency single values never break any dependency.

2. Approximate the error

   The error e(X) is the minimum fraction of tuples that should be removed from the relation to make a dependency valid. This property can also be extended to other parameters, for example a set to be a super-key can be decided using the error. The minimum fraction of tuples that should be removed from a relation to make X a super-key will be the error.This is given as e(X). If the error is small then X can be said as an approximate super-key. The error can be calculated by the following formula:

Row	A	B
1	2	4
2	2	4
3	3	5
4	2	4
5	3	5
6	4	6
7	3	3

TABLE 3. Stripped partitions

e(X) = 1 - $|\pi_X|$ / $|r|$

Since TANE replaces partitions with stripped partitions, the error can be given as:

e(X) = $||\hat{\pi_X}||$ - $|\hat{\pi_X}|$ / $|r|$

In the above equation, $||\hat{\pi_X}||$ is the sum of the size of all the equivalence classes in $\hat{\pi_X}$

Below are the operations on the partitions followed in TANE.

1. Stripping the partitions

   For every partition the equivalence classes with a single element in them are removed. A partition for an attribute X is denoted by π_X and a stripped partition is denoted by $\hat{\pi_X}$. For instance for the attribute A,B in Table 2

   $\pi_A = \{\{1,2,4\},\{3,5,7\},\{6\}\}$ and $\pi_B = \{\{1,2,4\},\{3,5\},\{7\},\{6\}\}$

   and $\hat{\pi_A} = \{\{1,2,4\},\{3,5,7\}\}$ and $\hat{\pi_B} = \{\{1,2,4\},\{3,5\}\}$

2. Computing partitions

   The partitions are calculated for each attribute set. Staring from level 0 in the lattice, it begins with finding partitions for single attributes of the relation. $\forall A \in R,\ \pi_A$ is calculated from the database. As it goes further in the lattice,for the attribute set with size > 2, π_X it calculates the partitions using the two partitions computed in the previous level i.e. the from the subsets of X.

   The lemma in Huhtala et al. [1999] states that the partitions can be calculated as follows:
   $\forall X, Y \subseteq R, \pi_X.\pi_Y = \pi_{X \cup Y}$

In order to find all the non-trivial minimal dependencies, TANE searches the lattice in a level-wise fashion.Starting from an empty set at level zero, it advances the lattice. Each level L_l

is a collection of attribute sets of size l. These attribute sets are used to get the dependencies in L_{l+1}. L_1 uses L_0, L_2 uses L_1 and so on and so forth.

Algorithm

Algorithm 1 TANE

1: $L_0 := \{\emptyset\}$
2: $C^+(\emptyset)$:=R
3: $L_1 := \{\{A\}\}|A \in R\}$
4: L := 1
5: **while** $L_l \neq \emptyset$ **do**
6: $\quad COMPUTE_DEPENDENCIES(L_l)$
7: $\quad PRUNE(L_l)$
8: $\quad L_{l+1} := GENERATE_NEXT_LEVEL(L_l)$
9: $\quad$ L := L+ 1

Below are the methods implemented by TANE .

Algorithm 2 TANE

$GENERATE_NEXT_LEVEL(L_l)$
$COMPUTE_DEPENDENCIES(L_l)$
$PRUNE(L_l)$
$STRIPPED_PRODUCT$
e

$GENERATE_NEXT_LEVEL$ method computes the level L_{l+1} using the previous level L_l. At each level only one attribute is added and hence the size increases by only one.Only those sets are generated which has all its subsets in the previous level. The $PREFIX_BLOCKS(L_l)$ method partitions the level L_l into blocks such that they do not have anything in common.They follow the lexicographic order. All the elements which have the same attributes except the last one belong to the same prefix block. They have only one attribute different in the attribute set.

Algorithm 3 Generates the next level

1: **procedure** $GENERATE_NEXT_LEVEL(L_l)$
2: $\quad L_{l+1} := \emptyset$
3: $\quad$ **for all** K $\in PREFIX_BLOCKS(L_l)$ **do**
4: $\quad\quad$ **for all** $\{Y, Z\} \subseteq K, Y, Y \neq$ Z **do**
5: $\quad\quad\quad$ X := Y $\cup$ Z
6: $\quad\quad\quad$ **if** for all A $\in X, X\ \{A\} \in L_l$ **then**
7: $\quad\quad\quad\quad L_{l+1} := L_{l+1} \cup \{X\}$
$\quad$ **return** L_{l+1}

In the algorithm $COMPUTE_DEPENDENCIES$, $C^+(X)$ is calculated.Pruning takes place here. The difference between the initial set of right hand side candidates $C^+(X)$ and C(X) is calculated and removed from the set. This procedure ensures that minimal functional dependencies are output by the algorithm.

Algorithm 4 Generates the functional dependencies for the next level

```
procedure COMPUTE_DEPENDENCIES(set L_l)
    for all X ∈ L_l do
        C+(X) := ∩_{A∈X} C+(X {A})
    for all X ∈ L_l do
        for all A ∈ X ∩ C+(X) do
            if X\{A} → A is valid then
                output X \ {A} → A
                remove A from C+(X)
                remove all B in R \ X from C+(X)
```

TANE implements pruning rules that are applied in this method PRUNE.Line 4 check for a key and removes it. Also, if $C^+(X)$ is empty it removes it from L_l. Step 7 outputs the dependencies found by this method.

Algorithm 5 Pruning the lattice

```
procedure PRUNE( L_l )
    for all X ∈ L_l do
        if C+(X) = ∅ then
            delete X from L_l
        if X is a (super)key then
            for all A ∈ C+(X)\ X do
                if A ∈ ∩_{B∈X} C+(X ∪ {A}\{B} ) then
                    output X → A
            delete X from L_l
```

Initially stripped partitions for singleton attributes are calculated from the relation directly. A hash table or trie data structure is used for doing this. Then the partitions with single values are stripped off to get stripped partitions. While generating next level candidates in *Generate_Next_Level* when an attribute is added to the attribute set, that is when the partition is computed for the new attribute set. The table needs to be set to 0 initially.

Compute_Dependencies finds minimal approximate dependencies using the error bounds mentioned in the section before. If that fails then the exact error is calculated using the e method. The entire table here is initialised to 0 but does not need a re-initialization.

Algorithm 6 Stripped Product

```
procedure STRIPPED_PRODUCT( inoutL_k, inL_{k-1})
    Input : Stripped partitions π̂ := {c'_1, ....., c'_{|π'|}} and π̂'' = {c''_1, ....., c''_{|π̂''|}}
    Output: Stripped partition π̂ = π'.̂π''
    π̂' := ∅
    for  i:= 1 to π̂'  do
        for all t ∈ c'_i do T[t] := i
        S[i] := ∅
    for  i:= 1 to π̂'' do
        for all t ∈ c''_i do
            if T[t] neq NULL  then S[T[t]] := S[T[t]] ∪{t}
        for all t ∈ c''_i do
            if |S[T[t]]| ≥ 2 then π̂ := π̂ ∪ {S[T[t]]}
            S[T[t]] := ∅
    for i := 1 to π̂' do
        for all t ∈ c'_i do T[t] := NULL
    return π̂
```

Algorithm 7 Exact error

```
procedure E
    Input : Stripped partitions π̂_X and π̂_{X∪A}
    Output: e(X → A)
    e := 0
    for all c ∈ π_{X∪̂{A}} do
        choose ()arbitrary t ∈ c
        T[t] := |c|
    for all c ∈ π̂_X do
        m := 1
        for all c ∈ π̂_X do
            m := 1
    for all t ∈ c  do m := max {m, T[t]}
    e := e + |c| − m
    for all c ∈ π_{X∪̂{A}}  do
        choose t ∈ c (same t as on line 3)
        T[t]= 0
        return e/ |r|
```

Advantages and Disadvantages

Advantages

1. Applicable on large databases

2. The method is at its best when the dependencies are relatively small.

Disadvantages

1. TANE does repeatedly sorting and comparing of tuples to determine FDs which increases time complexity .

2. Heavy manipulation of attribute sets and numerous tests that are performed

Time complexity

Worst case: Exponential with respect to the number of attributes but this is inevitable since the number of minimal dependencies can be exponential in the number of attributes. If the set of dependencies do not change with increase in the number of tuples, then time complexity is linear.

CHAPTER IV

FUN

FUN is a level-wise algorithm that explores the attribute set lattice of a relation level wise. It is proved to be more efficient than TANE which was the best solution then. It also extracts embedded functional dependencies without adding to the execution time, which none of the previous algorithms does. They identify keys at an earlier stage than the other solutions. One more thing that FUN does differently than others is handling the partitions over a relation. The general approach is to store all the partitions, however FUN stores only the number of partitions Novelli and Cicchetti [2001] .

Embedded FDs

FUN introduces the concept of embedded functional dependencies. Given a set of functional dependencies that hold in the relation, embedded dependencies are the dependencies that are valid in the projection of the relation over a subset of its attributes. The set of embedded functional dependencies is given as below:

$\mathcal{F}[X] = \{Y \rightarrow Z / \mathcal{F} \models Y \rightarrow Z \wedge YZ \subseteq X\}$

The aim of FUN like other approaches is finding a solution to problem of discovering the set of minimal functional dependencies. Novelli and Cicchetti [2001] refers to this set as canonical cover of functional dependencies. FUN introduced a new concept of Free set which can be defined as:

Let X $\subseteq$ R be a set of attributes. X is a free set in r, an instance of relation over R, if and only if: $\exists$ X $\in$ X, $|\pi'_X| = |\pi_X|$ where $|\pi_X|$ stands for the cardinality.

Free sets

Free sets are represented as $\mathcal{FS}_r$. The source of any minimal functional dependency is necessarily a free set. The concept of free sets says that free sets do not have any functional dependency within the set. Due to this a free set cannot capture any sort of functional dependency at all. On the other hand, non-free sets at least have one dependency or else they would have been in the free set as well.The combination of attributes not belonging to free set

is called as a non-free set. If the cardinality of any one subset for any set is also equal to the cardinality of the set then it is not a free set.

The lemmas given in FUN state that :

1. Any subset of a free set is a free set itself : $\forall$ X $\in \mathcal{FS}_r$, $\forall$ X' $\subset$ X , X' $\in \mathcal{FS}_r$

2. Any superset of a non-free set is non-free : : $\forall$ X $\notin \mathcal{FS}_r$, $\forall$ X $\subset$ Y , Y $\notin \mathcal{FS}_r$

Closure and quasi-closure

FUN also maintains closure and quasi-closure of attributes for relations. The closure of a set of attributes includes all the other attributes that can be obtained from the attributes in the set and all the attributes with in the set itself. The closure is given as X_r^+ which is given as :

$X_r^+ =$ X $\cup \{A \in R - X \ / \ |X|_r = |X \cup A|_r\}$

Here X is a set of attributes of the relation, X $\subseteq$ R

The quasi-closure of a set of attributes is denoted by X_r° and is given as:

$X_r^\circ =$ X $\cup$ $\cup_{A \in X}$ $(X - A)_r^+$

As per the monotonicity and extensibility properties , the relation between closure and quasi closure of an attribute set can be given as:

X $\subseteq X_r^\circ \subseteq X_r^+$

Approach

FUN is a level-wise approach. Starting from level zero each level k is provided as input a set of possible free sets of length k. These are called candidates. At level k+1 two free sets are combined to obtain the candidates for this level.

At each level there is a managed set maintained for each candidate.This set is described as a quadruple; candidates, count, quasi-closure and closure.The quadruple can be given as:

L_k = (candidate,count,quasi-closure,closure)

Candidates are the candidates at each level which is a set of attributes of the relation.All candidates should be free sets only.In order to make sure that candidates at each level are free sets, the cardinality of the candidate is compared with the cardinality of all its maximal subsets. If the cardinality is the same then it is not free as there is a dependency that exist.If it is proved to be a free set by this comparison check method then it can be a potential functional

dependency. Closure and quasi-closure are for the attribute set at that level. Count is the cardinality of the candidate that is counted by the count function. If for any attribute X, $|X|$ = 1 then it can be thrown away as it has the same value for all the tuples which means $\emptyset \rightarrow$ holds true.Also, if the count of X is equal to the number of tuples in the relation then it is identified as a key. Initially for level zero candidate = $\emptyset$, count =1, quasi-closure = $\emptyset$, closure = $\emptyset$.

The result after each level is the dependencies identified in that level. In the FUN algorithm mentioned below , DisplayFD procedure yields the minimal functional dependencies at each level. The algorithm gets over when there can be no further candidates that can be generated for the next level.

Algorithm

The functions and procedures in FUN are described below:

Algorithm 8 FUN

$L_0 := <\emptyset,1,\emptyset,\emptyset >$
$L_1 := \{<\text{A},\text{Count(A)},\text{A},\text{A} >| \text{ A} \in \text{R} \}$
$R' :=$ R - {A | A is a key }

for (**do**k := 1; $L_k \neq \emptyset$; k:= k+1)
ComputeClosure(L_{k-1}, L_k)
ComputeQuasiClosure(L_k, L_{k-1})
DisplayFD(L_{k-1})
PurePrune(L_k, L_{k-1})
L_{k+1} := GenerateCandidate(L_k)
DisplayFD(L_{k-1})

Algorithm 9 FUN

GenerateCandidate(L_k)
ComputeClosure(L_{k-1}, L_k)
ComputeQuasiClosure(L_k, L_{k-1})
PurePrune(L_k, L_{k-1})
FastCount($inoutL_{k-1}, inL_k$)

The function GenerateCandidate generates candidates from the set of free sets for the next level,k+1 from the candidates at level k.

The function below computes the closure of the free sets obtained from the previous level.Initially it is set to the quasi-closure of l in the below function and later updated for non-keys. For keys the closure in updated in the ComputeQuasiClosure procedure. The procedure

Algorithm 10 Generate the candidate for the next level

```
procedure GENERATECANDIDATE(set Y)
    L_{k+1} := l | ∀ l' l, | l | = | l' | + 1, l' ∈ L_k, and | l' |_r ≠ | r |
    for all l ∈ L_k + 1 do
        l.count := Count(l.candidate)
    return L_{k+1}
```

checks if l → A holds or not by checking the cardinality of l and l ∪ A. The comparison is done by calling another procedure FastCount which is mentioned later in the chapter. The attributes that can be reached from l are added to the closure here in this procedure.

Algorithm 11 Compute the closure

```
procedure COMPUTECLOSURE( inoutL_{k-1}, inL_k )
    for all l ∈ L_k − 1 do
        If l is not a key then
        l.closure := l.quasiclosure
        for all A ∈ R' - l.quasiclosure do
            if FastCount(L_{k-1}, L_k, l.candidate ∪ A = l.count ) then
                then l.closure := l.closure ∪ A
```

The procedure ComputeQuasiClosure computes the quasi-closure of all the candidates for that level. It is initialized with the candidate and later updated by computing the union of its maximal subsets closures. This procedure also computes the closure of keys.

Algorithm 12 Compute the quasi-closure

```
procedure COMPUTEQUASICLOSURE( inoutL_k, inL_{k-1})
    for all l ∈ L_k do
        l.quasiclosure := l.candidate
        for all s ⊂ l.candidate and s ∈ L_{k-1} do
            l.quasiclosure := l.quasiclosure ∪ s.closure

        if l is a key then l.closure := R
```

This method looks for internal functional dependencies and removes the non-free sets from L_k. This check is done by comparing the cardinalities of the candidate and its maximal subsets which are free sets.

This function gives the cardinality of a candidate. Count function does the same thing but it is used in case of generating the candidates. Whereas FastCount is faster than the Count method and does the count for even those which are not yet generated but their count is required.

Algorithm 13 Pruning

procedure PUREPRUNE($inoutL_k, inL_{k-1}$)
  for all l $\in L_k$ **do**
    for all s **do** $\in$ l.candidate and s $\in L_k - 1$
      if l.count = s.count **then** delete l from L_k

Algorithm 14 Fast count

procedure FASTCOUNT(inout L_{k-1} , in L_k,in l.candidate)
  if l.candidate $\in L_k$ return l.count **then**
  return $Max(l'.count|l'.candidate \cup l.candidate, l'.candidate \in L_{k-1})$

Advantages

1. It give embedded dependencies which is innovative as none of the previous algorithms do that. It is more efficient than the best available solution then (TANE).
2. For data that is high correlated FUN is very efficient. (Better than TANE)
3. Search space explored by FUN is smaller than TANE

Disadvantages

1. In case of equivalences in relations, it has more candidates which makes this solution poor.
2. Instead of storing the partitions it just stores the number of partitions, which may not be so good as the algorithm has to keep going back because of this approach.

CHAPTER V

FD_MINE

FD_Mine is another rule discovery algorithm that was given in the year 2002. FD_Mine identifies equivalences in the dataset. It applies Armstrong's axioms to generate equivalences. This is something that had not been done in any of the past work.

Pruning is the process of removing unwanted data and the result of pruning is a reduced, relevant dataset. FD_Mine suggests 4 pruning rules in the algorithm and gives an implementation of the same. These rules help in reducing the search space. No data is lost in the process of pruning.

Classification of FDs

The algorithm discusses about two types of rules: Implication and functional dependency. "An implication describes a relationship between one and specific combination of attribute-value pairs." Yao et al. [2002] An implication X $\Rightarrow$ Y means that whenever X is true, Y is also true. The other rule mentioned in the paper Yao et al. [2002] is functional dependency.Also, it discusses functional dependencies to have mainly two classifications: One set of functional dependencies are those that can be obtained from the pre-discovered functional dependencies. These are redundant dependencies and hence are not taken into consideration in the algorithm. For example, we have a dependency . Now the new dependency XY $\rightarrow$ AY is redundant because according to Axiom of Augmentation for a given dependency,the same attribute(s) can be added to both the sides of the dependency.

The other set of classification includes those dependencies that cannot be inferred from the already discovered dependencies. Such dependencies are to be found in the dataset using this algorithm. This is where the pruning rules kick in as this process is costly to check for dependencies against the dataset. The pruning procedure reduces the number of dependencies to be checked by skipping the redundant dependencies.

Equivalences

With an increase in the number of attributes the number of candidates also increase exponentially.Fd_Mine identifies equivalences in the dataset which help in reducing the attributes in the candidates. When an equivalence is seen the attribute which appears later in the lexicographic order is not included in the further candidates.For any two attributes, X,Y if X $\rightarrow$ Y and Y $\rightarrow$ X then we can say that X and Y are equivalent and are represented as X $\leftrightarrow$ A

Approach

A lattice of attributes is formed. If there are n attributes and there are 2^n possible subsets of attributes out of which 2^n - 2 are non-empty and proper subsets of the candidates. $\emptyset$ and the set of all attributes of the relation are excluded from here. The number of edges = n 2^{n-1}. The lattice starts from level 1 because it does not include $\emptyset$. The size of the search space is exponential to the number of variables in the relation in a database.

The algorithms for finding functional dependencies can be classified into two categories:

1. Generate and test approach

   TANE is a candidate-generate-and-test approach which uses level wise search to explore the search space.It reduces the search space using pruning as and when possible.For this it uses partitioning the tuples based on the attribute values. FD_Mine also falls in the candidate-generate-and-test approach.

   TANE differs from others in this category by the pruning rules it uses. These rules are more effective and this makes it a faster and more efficient algorithm that the rest.

2. Minimal cover approach

   This approach discovers the minimal cover of the set of dependencies. From the original relation, a stripped partition database is extracted. Then using these partitions agree sets (pairs of tuples) are calculated and maximal sets are generated. Then a minimum FD cover is found.

For a dependency X $\rightarrow$ Y FD_Mine refers to X as the antecedent and Y as the consequent. The authors of FD_Mine state that Armstrong's axioms are sound and complete. Soundness indicates that given a set of functional dependencies satisfied by a relation, any functional

dependency inferred from the Armstrong's axioms is valid for that relation as well. Completeness states that all the implied functional dependencies can be inferred from Armstrong's axioms.

Finding functional dependencies

For a dependency X $\rightarrow$ Y to hold,the algorithm states that the cardinality of X and XY should be the same. Otherwise the dependency does not hold true.

Closure and Non-trivial closure

Let X be an attribute or set of attributes of a relation of a database. The closure of X is the set of all the attributes that can be derived from X including X as well. It is denoted by X^+.

The non-trivial closure of X is denoted by X^*. It is given as the closure of X without X in it.

$X^* = X^+$ - $\{X\}$

Equivalent attributes

For two attributes X and Y they are said to be equivalent by comparing the closures of X and Y. Theorem in Yao et al. [2002] states that :

X,Y $\subseteq$ U, if Y $\subseteq X^+$ and X $\subseteq Y^+$, then it can be safely said X $\leftrightarrow$ Y

Here X^+ and Y^+ are the closures of X and Y respectively.

Pruning rules

The possible candidates for the antecedent are suggested and then checked for consequent s. At each level in the lattice candidates are generated by adding one attribute. The number of candidates are given by the formula n2^{n-1} where n is the number of attributes. In big-sized database where relations have large number of attributes, this number can be very large. Pruning rules help to reduce the number of dependencies that we need to check.FD_Mine gives 4 pruning rules given below and is valid when $Y \neq \emptyset$ and $X \neq \emptyset$. These pruning rules are applied to remove the candidates on a particular level before this information is used by the next level.

1. If X $\rightarrow$ Y,Y $\rightarrow$ X then candidate Y can be deleted.

The below rule removes the redundant candidates.

2. $Y^+ \subseteq X$ then candidate X can be deleted provided $Y^* \neq \emptyset$

   If X is a key, then any superset XY of X does not need to be checked.

3. Given X^* and Y^*, then when attempting to determine whether or not the set of functional dependencies XY $\rightarrow v_i$, where $v_i \in U - X^+Y^+$, needs to be checked in r(U).

   Because X^* and Y^* are the non-trivial closures of attributes X and Y, respectively, then XY Closure(X) U Closure(Y) does not need to be checked.

4. If $\forall v_i \in U - X, X \rightarrow v_i$ is $\rightarrow$ satisfied by r(U), the candidate X can be deleted

Fd_Mine is similar to TANE algorithm but TANE only uses 2 of the above 4 rules (pruning rule 2 and 4) used by FD_Mine.

FD_Mine does a level-wise search along with the application of the four pruning rules mentioned above.

Algorithm

Algorithm 15 FD_Mine

TO DISCOVER ALL FUNCTIONAL DEPENDENCIES IN A DATASET.
INPUT: DATASET D AND ITS ATTRIBUTES $X_1, X_2, X_3, ..., X_m$
OUTPUT: FD_SET, EQ_SET AND KEY_SET
1. INITIALIZATION STEP
SET R = $X_1, X_2, ..., X_m$, SET FD_SET = ϕ
SET SET EQ-SET = ϕ, SET KEY_SET = ϕ
SET CANDIDATE_SET = X_1, X_2, ... , X_m
for all $X_i \in$ CANDIDATE_SET **do**
    SETCLOSURE'[X_i] = ϕ
2. ITERATION STEP
while CANDIDATE_SET ≠ ϕ **do**
    for all $X_i \in$ CANDIDATE_SET **do**
        COMPUTENONTRIVIALCLOSURE(X_i)
        OBTAINFDANDKEY(X_i)
    OBTAINEQSET(CANDIDATE_SET)
    PRUNECANDIDATES(CANDIDATE_SET)
    GENERATECANDIDATES(CANDIDATE_SET)
3. DISPLAY(FD_SET,EQ_SET,KEY_SET)

FD_Mine has the below methods that it uses to find out the dependencies for the relations in the database.

Algorithm 16 FD-Mine

ComputeNonTrivialClosure(X_i)
ObtainEQSet($Candidate_set$)
PruneCandidates($Candidate_set$)
GenerateCandidates($Candidate_Set$)
PurePrune(L_k)
FastCount(L_k)

Algorithm 17 Compute the non-trivial closure

procedure ComputeNonTrivialClosure(X_i)
  for all $Y \subset R - X_i - Closure'[X_i]$ **do**
    if $| \prod_{X_i} |=| \prod_{X_i Y} |$,add Y to Closure'$[X_i]$

Algorithm 18 Obtain FD and keys

procedure ObtainFDandKey(set X_i)
  add $X_i \rightarrow$ Closure'$[X_i]$ to FD_SET
  if R = $X_i \cup$ Closure'$[X_i]$ **then** add X_i to KEY_SET

Algorithm 19 Obtain the Equivalent Set

procedure ObtainEQSet(CANDIDATE_SET)
  for all $X_i \in$ CANDIDATE_SET **do**
    for all X $\rightarrow$ Closure'(X) $\in$ FD_SET **do**
      set Z = X $\cap$ X_i
      if (Closure'(X) $\supseteq$ X_i - Z and Closure'$[X_i]$ $\supseteq$ X - Z) **then**
        add X $\leftrightarrow$ X_i to EQ_SET

Algorithm 20 Prune the candidates

procedure PruneCandidates(CANDIDATE_SET)
  for all $X_i \in$ CANDIDATE_SET **do**
    if $\exists$ $X_j \in$ CANDIDATE_SET and $X_j \leftrightarrow X_i \in$ EQ_SET **then**
      delete X_i from CANDIDATE_SET
    if $\exists$ $X_i \in$ KEY_SET **then**
      delete X_i from CANDIDATE_SET

Algorithm 21 Pruning

procedure GenerateCandidates(CANDIDATE_SET)
  for all $X_i \in$ CANDIDATE_SET **do**
    for all $X_j \in$ CANDIDATE_SET and i < j **do**
      if (X_i[1] = X_j[1]), ..., X_i[k-2]= X_j[k-2], X_i[i-1] < X_j[k-1] **then**
        set $X_{ij} = X_i$ join X_j
        if ($\exists X_i \rightarrow X_j[k-1] \notin FD_SET$) **then**
          compute the partition $\prod_{X_{ij}} of X_{ij}$
          setClosure'(X_{ij}) = Closure'(X_i) $\cup$ Closure'(X_j)
          if (R = $X_i \cup Closure'[X_{ij}]$) **then** add X_{ij} to KEY_SET
          else add X_{ij} to CANDIDATE_SET
      Delete X_i from CANDIDATE_SET

Advantages

1. Identifies equivalences in dataset.
2. It reduces the size of the search space.
3. It reduces the number of functional dependencies to be checked.
4. More effective pruning rules.

Time Complexity

For m attributes, O(n.2^m) is the theoretical complexity where n is the number of tuples.

CHAPTER VI

FASTFDS

FastFDs is a depth-first, heuristic-driven search strategy which determines the functional dependencies that hold over an instance r of the relation R. It finds a canonical cover of the set of functional dependencies. Dep_Miner algorithm views the functional dependency discovery problem as finding minimal covers for hypergraphs. Once the minimal covers are found it applies a level-wise search strategy to determine these minimal covers. Experiments show that level-wise strategy that is a typical approach for all the functional dependency discovery algorithms, is outweighed by the depth-first,heuristic-driven strategy.

FastFDs is based on a result showing that finding the canonical cover of the set of FDs is equivalent to finding the minimal covers of each of a set of hypergraphs (one for each attribute) constructed from the difference sets of the relation instance. Wyss et al. [2001]

Canonical cover

A canonical cover F' for a set of functional dependencies F is a set of dependencies such that F logically implies all dependencies in F' , and F' logically implies all dependencies in F. It can be given as :

$F_r = \{X \rightarrow A | X \subseteq R, A \in R, r \models X \rightarrow A, A \notin X, and X \rightarrow$ A is minimal. $\}$ Wyss et al. [2001]

For t_1 and $t_2 \in$ r. The difference sets of t_1 and t_2 is : D$(t_1, t_2) = \{B \ \in \ R \mid t_1[B] \neq t_2[B]\}$

The difference sets of r are $\mathfrak{D}_r = \{D(t_1, t_2) \mid t_1, t_2 \ \in \ r, D(t_1, t_2) \neq \emptyset\}$

$\mathfrak{D}_r^A = \{D - \{A\} \ | D \in \mathfrak{D}_r \ and \ A \in D\}$

Agree sets

Like the previous algorithms, this one considers stripped partitions.Agree sets are generated from stripped partitions instead of tuples from the relation directly. Agree sets are the natural dual of difference sets. The agree sets are given as:

A$(t_1, t_2) = \{B \in R | t_1[B] = t_2[B]\}$

The agree sets of R are: $\mathfrak{A}_r = \{A(t_1, t_2) | t_1, t_2 \in r\}$

X $\rightarrow$ A is a minimal functional dependency if and only if X is a minimal cover of $\mathfrak{D}_r^A$.

The elements of $\mathfrak{D}_r^A$ form edges in a hypergraph. So if we calculate the minimal covers, that gives the functional dependencies.

FastFD is good for :

1. random integer valued instances of varying correlation factors,
2. random Bernoulli instances
3. real-life ML repository relation instances

FastFD is space efficient in all the above 3 cases.

Algorithm

Algorithm 22 FASTFDs

procedure FastFd($inoutL_{k-1}, inL_k$)
  input: relation instance r wth schema R
  output : canonical cover of minimal FDs over r , F_r
  $\mathfrak{D}_r$:= genDiffSets(R,r);
  for A $\in$ R **do**
    compute $\mathfrak{D}_r^A$ from $\mathfrak{D}_r$;
    if $\mathfrak{D}_r^A = \emptyset$ **then**
      output $\emptyset$ -> A;
    else if $\emptyset \notin \mathfrak{D}_r^A$ then
      $>_{init}$ is the total ordering of R - { A } according to $\mathfrak{D}_r^A$;
      findCovers(A, $\mathfrak{D}_r^A$, $\mathfrak{D}_r^A$, $\emptyset$,$>_{init}$);

The below algorithm generates the difference set of functional functional dependencies of the instance of the database.

Algorithm 23 generate difference sets

genDiffSets($\emptyset$)
findCovers(L_k)

The method findCovers uses a search method to find the covers. So for finding the cover of an attribute A, it considers every subset of R which does not contain A in it is a likely candidate for the minimal cover of $\mathfrak{D}_r^A$. The below algorithm finds minimal covers of the difference sets in a depth-first, left-to-right manner. Each node in the tree is a subset of R which does not have A in them. The heuristic chosen includes ordering the subsets lexicographically according to which B >

Algorithm 24 Calculate the cardinality of candidates

```
procedure genDiffSets(R,r)
    input: schema R and r a relation instance over R
    output : difference sets for r, 𝔇_r
    //Initialize :
    resDS := ∅ ;
    strips := ∅ ;
    tmpAS := ∅ ;
    //Compute stripped partitions for all attributes
    for A ∈ R do
        compute stripped partitions for all attributes
    //Compute agree sets from stripped partitions:
    for ∏ ∈ strips do
        for t_i ∈ ∏ do
            for t_j ∈ ∏,j >i  do
                add A(t_i,t_j) to tmpAS
    //Complement agree sets to get difference sets
    for X ∈ tmpAS do
        add R - X to resDS;
```

C > D > E ... The heurisitic is greedy in nature as it picks the one with the most difference sets at each node. This is the heuristic that is chosen by KlipFind as well.

The optimized search method below constructs a search tree which has an attribute ordering which changes as they keep going down the search. This ordering is done on the basis of how many difference sets they cover i.e. those which have not been covered at a node above this level. If there exists a tie then the lexicographic ordering is used to deal with the tie cases.

Two of the situations can arise:

1. If we are at a node and no more attributes but difference sets are still there: then it can be said there are no functional dependencies down this branch. It is marked as fail.

2. If we are at a node and there are no difference sets left then either the dependency may not be minimal, so mark it as fail case; or output the subset that this node and the path to it represents.

Advantages

It is space efficient because of the depth-first search approach which is not there in the levelwise search approach algorithms.

Algorithm 25 Generate the candidate for the next level

```
procedure FINDCOVERS(L_k)
    input: attribute A ∈ R(RHS)
    original dfference sets, 𝔇_r^A
    difference sets not thus far covered, 𝔇_curr
    the current path in the search tree , X ⊆ R
    the current partial ordering of the attributes >_curr
    output : minimal FDs of the form Y -> A
    Base Cases:
    if >_curr = ∅ but 𝔇_curr ≠ ∅ then
        return; // no FDs here
    if 𝔇_curr = ∅ then
        if no subset of size | X | - 1 of X covers 𝔇_r then
            output X -> A and return;
    elsereturn; // wasted effort, non-minimal result
    Recursive case:
    for attributes in >_curr in order do
        𝔇_next := difference sets of 𝔇_curr not covered by B;
        >_next is the total ordering of B̂ ∈ R | B̂ >_curr B according to 𝔇_next;
        findCovers(A, 𝔇_r^A, 𝔇_next, X ∪ B, >_next )
```

Disadvantages

Although the heuristic used is greedy in nature yet it can be doing extra work. The dependency has to be checked for minimality by checking the left hand side of the dependency.

Time complexity

Calculating the difference sets $\mathfrak{D}_r$ takes O(nm^2).

To compute $\mathfrak{D}_r^A$ from $\mathfrak{D}_r$ the minimization of X ∈ $\mathfrak{D}_r$ takes O(d log(d)) where d = $|\mathfrak{D}_r|$. It is given as O(nm^2 log(n m^2))

$|F_r|$ = K, the complexity of findCovers is O((1+w(n))K), where w(n) is a function representing the wasted work due to the imperfect search heuristic.

As findCovers is called for each attribute A ∈ R, FastFDs takes time O(n(1 + w(n))K).

The worst time complexity can be given as : O(nm_2 + nm_2 log(nm2) + n(1 + w(n))K) where w(n) is the wasted work due to imperfect search heuristic. Wyss et al. [2001]

CHAPTER VII

DEPMINER

This is a level-wise algorithm that discovers the functional dependencies and constructs the real world Armstrong relations. The execution times is the same as far as comparison with other algorithms is concerned though this algorithm also finds out the the Armstrong relations in addition. This algorithm characterizes the left hand sides of the dependencies as the traversal of a simple hypergraph. A hypergraph is a graph in which an edge can connect to any number of vertices.

Armstrong relation

Armstrong relation is a relation that is separate from the original relation such that all the functional dependencies that are implied by the set of dependencies in a relation are also satisfied by this relation. The dependencies that the original relation are not satisfied by this relation either. This relation might be smaller in size that conveys the same information as the original relation. Due to this it is preferred over the original relation. An Armstrong relation can be thought of as a subset of original relation or a cleaner version of the original relation.

Approach

This paper uses the concept of agree sets. From agree sets maximal sets are arrived and from maximal sets minimal functional dependencies are generated. This can be done by finding the complement of the maximal sets and this complement is used to derive the left hand side of the dependencies.

dep(r) represents the set of all functional dependencies that exist for the relation r. Trivial dependencies are not included in this set. Trivial dependencies are the dependencies of the form X $\rightarrow$ A where A $\in$ X. The algorithm can be seen as the below steps:

1. Finding agree sets

   The tuples t_i and t_j are said to agree on an attribute X if $t_i[X] = t_j[X]$. It is given as:

   ag(r) $= \{ag(t_i, t_j) = A \in R \mid t_i[A] = t_j[A]\}$

For a relation R, ag(r) = $\{ag(t_i, t_j) \mid t_i, t_j \in r, t_i \neq t_j\}$

2. Finding maximal sets

   A maximal set being the largest attribute set for an attribute which does not determine that attribute. It is represented in the algorithm as max(dep(r),A) where A is the attribute in consideration. It can be given as:

   max(dep(r),A) = $\{X \subseteq R/r \not\models X \rightarrow A \; and \; \forall Y \subseteq R, X \subset Y, r \models Y \rightarrow A\}$ and

   MAX(dep(r)) = $\cup_{A \in R}$ max(dep(r),A)

   max(dep(r),A) represents Maximal sets for A w.r.t dep(r)

3. Finding left hand sides of the functional dependencies From the maximal sets functional dependencies can be generated

   The left hand side of the functional dependencies can be generated from maximal sets using hypergraphs. The hypergraph is represented as $\mathcal{H}$. Elements of the hypergraph are called as edges of the hypergraph and elements of R are the vertices of the hypergraph.

   We find the complement of the maximal set and it is represented as cmax(dep(r),A). A traversal T is a subset of R intersecting all the edges of $\mathcal{H}$ i.e. T $\cap E \neq \emptyset, \forall E \in \mathcal{H}$.

   A minimal traversal is a traversal T such that there does not exist a transversal T', T' $\subset T$

4. Generating real world Armstrong relations

   Some functional dependencies could hold just coincidently on the relation. There is no guarantee of these dependencies with a relation extension. Lopes et al. [2000]

   Functional dependencies may also just represent an integrity constraint which is not of interest to us.

   A real-world Armstrong is identified as :

   (a) $\bar{r}$ is a real-world Armstrong relation satisfying dep(r):

   (b) $|\bar{r}| = |MAX(dep(r))| + 1$;

   (c) $\forall A \in Rt_i \in \bar{r}, t_i[A] \in \pi_A$ (r) is the projection of r on A.

Algorithm 26 Dep_Miner

Discovering minimal functional dependencies and real-world Armstrong relations
Input: a relation r
Output: minimal functional dependencies and real-world Armstrong relation for r
1. AGREE SET: computes agree sets from r
2: CMAX SET: derives complements of maximal sets from agree sets
3: LEFT HAND SIDE: computes lhs of functional dependencies from complements of maximal sets
4: FD OUTPUT: outputs functional dependencies
5. ARMSTRONG RELATION: builds real-world Armstrong relation from maximal sets and r

Algorithm 27 Dep_Miner

AGREE_SET
AGREE_SET 2
CMAX_SET
LEFT_HAND_SIDE

Algorithm

The below algorithm computes the maximal equivalence classes from a stripped partition database followed by the computation of agree sets.

In a naive algorithm when the number of couples reach a threshold value, then the corresponding agree sets are computed for those couples. Then this set is deleted and for the rest of the couples the agree sets are computed. This computation can take a lot of time and make the algorithm inefficient. Dep_Miner gives another characterization of agree sets which can avoid this. This characterization is to preserve the identifiers of equivalence classes for each tuple in which the tuple appears.

In order to find the agree sets the below algorithm is given.It gives the relationship between tuples and equivalence classes and then computes the agree sets after that. Agree_Set and Agree_Set2 are doing the same thing but there is Agree_Set2 because Agree_Set takes more time in case of more tuples. Hence there came Agree_Set2.

Advantages

1. It is faster than TANE

2. It is the only solution that give Armstrong's relation along with functional dependencies.

Algorithm 28 Computes agree sets from stripped partition databases

```
procedure Agree_Set
    Input: the stripped partition database r̂ of a relation r
    Output :the agree sets of r : ag(r)
    MC: = Max ⊂ {c ∈ π̂_A/π̂_A ∈ r̂}
    ag(r) = ∅
    couples: = ∅
    for all maximal equivalence classes c ∈ MC do
        for all (do)couple(t,t') ∈ c
            couples : = couples ∪ (t,t')
            ag(t,t') := ∅
    for all π̂_A ∈ r̂ do
        for all equivalence classes c ∈ π̂_A do
            for all (t,t') ∈ couples do
                if t ∈ c and t' ∈ c then
                    ag(t,t') := ag(t,t') ∪ A
    for all couple (t,t') ∈ couples do
        ag(r) := ∪ ag(t,t')
                                        ▷ node X compares count with parent Y
```

Algorithm 29 Computes agree sets from stripped partition databases

```
procedure Agree_Set 2
    Input: the stripped partition database r̂ of a relation r
    Output: the agree sets of r: ag(r)
    ag(r) := ∅
    for all π̂_A ∈ r̂ do
        for all equivalence class π̂_{A,i} ∈ π̂_A do
            for all tuple t ∈ π̂_{A,i} do
                ec(t) := ec(t) ∪ (A,i)
    MC:= Max ⊆ {c ∈ π̂_A/π̂_A ∈ r̂}
    for all maximal equivalence classes c ∈ MC do
        for all couple (t,t') ∈ c do
            ag(r) := ag(r) ∪{A ∈ R/∃j s.t. (A,j) ∈ ec(t) ∩ ec(t')}
```

Algorithm 30 Computes complement of maximal sets

```
procedure CMAX_SET( )
    Input: the agree sets over r: ag(r)
    Output: complements of maximal sets: CMAX(dep(r))
    for all attributes A ∈ R do
        max(dep(r), A) := Max ⊆ {X ∈ ag(r)/A ∉ X}
        for all attributes A ∈ R do
            cmax(dep(r), A) := ∅
            for all X ∈ max(dep(r), A) do
                cmax(dep(r), A) := cmax(dep(r),A)∪(R\X)
```

Algorithm 31 Computes lhs of minimal functional dependencies

procedure LEFT_HAND_SIDE( )
  Input: complements of maximal sets: CMAX(dep(r))
  Output: the lhs of minimal functional dependencies: lhs(dep(r))
  for all attributes A $\in$ R **do**
    i := 1
    $L_i := \{B/B \in X, X \in cmax(dep(r), A)\}$
    while $L_i \neq \emptyset$ **do**
      $LHS_i[A] := \{l \in L_i/l \cap X \neq \emptyset, \forall X \in cmax(dep(r), A)\}$
      $L_i := L_i \backslash LHS_i[A]$
      $L_{i+1} := \{l'/|l'| = i + 1$ and $\forall l \subset l'/|l| = i, l \in L_i\}$
      i := i + 1
    $lhs(dep(r), A) := \cup_i LHS_i[A]$

Algorithm 32 Pruning

procedure PUREPRUNE($inoutL_k, inL_{k-1}$)
  for all l $\in L_k$ **do**
    for all s **do** $\in$ l.candidate and s $\in L_k - 1$
      if l.count = s.count **then** Delete l from L_k

Time complexity

The time complexity of a naive algorithm is O(Rr^2) where r is the number of tuples and R is the number of attributes. Lopes et al. [2000] With the number of attributes is large, algorithms usually becomes impractical because of number of couples and the overhead because of the cost ag(t_i, t_j). This algorithm proposes an approach to decrease the number of candidate and this can be achieved by making use of stripped partitions. Stripped partitions reduce the number of couples.

Algorithm 33 Fast count

procedure FASTCOUNT($inoutL_{k-1}, inL_k$)
  if l.candidate $\in L_k$ return l.count **then**
  return $Max(l'.count|l'.candidate \cup l.candidate, l'.candidate \in L_{k-1})$

CHAPTER VIII

KLIPFIND

FastFD which is a depth-first search is much more space efficient than Dep_Miner which is a breadth-first search. We propose to apply a depth-first search to candidate-generate techniques such as FUN (TANE, FDMine). So our algorithm combines the pruning rules of FUN and heuristic of FastFDs.

In this chapter we suggest a depth-first, heuristic-driven search strategy KlipFind which determines the functional dependencies that hold over an instance r of the relation R. FUN does the search in a breadth-first manner.Experiments show that space savings can be achieved if level-wise strategy that is a typical approach for all the functional dependency discovery algorithms, is replaced by the depth-first,heuristic-driven strategy. We also saw FastFDs is a depth-first, heuristic-driven search strategy. This gave us the motivation for KlipFind.

Heuristic of KlipFind

Our algorithm suggests that the lattice of attributes be generated and explored in a depth first fashion. The attributes are sorted in decreasing order of the cardinality of the candidates at each level.They are then searched in this order.FastFD has a similar heuristic. Due to this strategy the nodes are usually heavier on the left hand side. The idea behind this is that the shortest branch is visited first. Once that branch is pruned there is a substantial saving in space that can be done.

for all candidate X , for each attribute a $\in$ R - X **do**

count(X $\cup \{a\}$)

sort R - X according to these counts in decreasing order, $\prec$

search supersets of X according to new order, $\prec$

Starting from level zero, candidates are generated at each level. Level zero has only one node which is $\emptyset$. The level zero ensures that the dependencies of the form $\emptyset \rightarrow$ X are checked, where X is a set of attributes of the relation R. This node has further children which are singleton attributes of the relation.Next, a functional dependency is found by comparing the cardinality of the candidates at each level with their maximal subsets. If the cardinality is the same then

it is considered as a dependency. It is yet to be identified whether the dependency found is minimal or not because the search is depth first search. In order to do this, there is another new concept introduced in our algorithm which of FD_list. FD_list is a list that contains all the functional dependencies that have been found so far. This list for the later nodes has dependencies been transferred from the previous nodes that are to the left of that node in the tree.Once a dependency is found, it is not issued right away because we only want minimal dependencies to be generated by our algorithm. Hence, it is transferred to the next node according to the current order which is the next possible minimal left hand side of the current functional dependency. Since the nodes are arranged based on a weighted lexicographic order, the placement of a node in a tree in respect to other nodes in the tree is relevant.

Each node in the tree has a set which contains a quadruple. The quadruple can be given as: (closure, quasi-closure,FD_list, free_marker)

1. The closure can be given as below:

   $X_r^+ = \text{X} \cup \{A \in R - X \ / \ |X|_r = |X \cup A|_r\}$

   In our algorithm we represent it as cl(X).

2. The quasi-closure can be given as: $X_r^\circ = \text{X} \cup \cup_{A \in X} \ (X - A)_r^+$

   The quasi-closure is represented as qcl(X) in this algorithm.

3. FD_list: FD_list has all the dependencies that have been either discovered on that node or have been passed on this node from other nodes which are to its left in the tree. These dependencies have not yet been issued and might be passed from here to other nodes to its right. Here, it is represented as fdlist(W).

4. Free_marker :free_marker keeps a record if the present candidate is free or not. If a dependency is found to have the same cardinality as its subset then it is marked as not free. As a result, there are no further descendant nodes generated from this node.

In order to check a dependency X $\rightarrow$ A .There are three possible cases:

1. We will count $|X \cup A|$ and $|X \cup A| = |X|$

   In this case A is added to the closure of X and is also added to visited(X).

2. We will count $|X \cup A|$ and $|X \cup A| > |X|$

   In this case A is not added to the closure of X but is added to visited(X).

3. $|X \cup A|$ may have to be counted later on demand as it might be the child of a key or non-free set but is required later to check a functional dependency.

   According to lemma 2 given in Novelli and Cicchetti [2001] we have:

   (a) Any subset of a free set is a free set itself : $\forall$ X $\in \mathcal{FS}_r$, $\forall$ X' $\subset$ X , X' $\in \mathcal{FS}_r$

   (b) Any superset of a non-free set is non-free : : $\forall$ X $\notin \mathcal{FS}_r$, $\forall$ X $\subset$ Y , Y $\notin \mathcal{FS}_r$

   Based on this we can say that if the candidate is a superset of a non-free node or it is a superset of a key then it that case $|X \cup A|$ is not processed.

Conjecture

There are three possible values for the total number of functional dependencies to be examined :

1. 2F : Functional dependencies that were examined but were thrown because they were not minimal.

2. F + o(F) : The actual minimal dependencies and extremely small fraction of other dependencies.

3. O(F) : There is a non-minimal functional dependency for each functional dependency so F times R. Hand simulations say that this is the worst case.

Algorithm

Algorithm 34 KlipFind

CREATE($\emptyset$)
EXPLORE($\emptyset$)
CHECK-IN(X, Y)
GENERATE-CHILDREN(node X)
PROCESS-FD-LIST(node W)
ISSUE-FDS(node X)

Algorithm 35 Explore a node

```
 1: procedure EXPLORE(node X)
 2:     for all parents Y ⊂ X do
 3:         if Y does not exist as a node, CREATE(Y)
 4:         CHECK-IN(X, Y)                      ▷ node X compares count with parent Y
 5:     if X is not free then
 6:         delete node X and exit              ▷ sets not free will have internal dependencies
 7:     if X is a key then
 8:         add R to the closure of X
 9:         issue key X                         ▷ X is free
10:         skip ahead to FD issuance           ▷ a key will have empty fdlist
11:     GENERATE-CHILDREN(X)                    ▷ this calls EXPLORE recursively on children
12:     PROCESS-FD-LIST(X)
13:     ISSUE-FDS(X)
14:     delete node X
```

Algorithm 36 Initialize a node

```
 1: procedure CREATE(set Y)
 2:     create node with attributes Y
 3:     perform count of Y and set count        ▷ scan the whole relation!
 4:     initialize closure, visited, quasi-closure, fdlist to null
 5:     mark Y as free
```

Algorithm 37 Look at all subsets Y, compare counts, determine freeness

```
 1: assume: X ⊃ Y
 2: procedure CHECK-IN(node X, node Y)
 3:     if Y not-free then
 4:         mark X as not-free and exit
 5:     if count(X) = count(Y) then
 6:         mark X as not-free
 7:         add X − Y to closure(Y)
 8:         exit
 9:     add X − Y to visited(Y)
10:     add closure(Y) to quasiclosure(X)
```

Algorithm 38 Create and explore children of X, sort and explore in decreasing order, heaviest first

Require: X is free and not a key

1: **procedure** Generate-Children(node X)
2: let Y be parent of X, denote its lex order as $\prec_Y$
3: **for all** supsets $X' \supset X$ with $X \prec_Y X'$ **do**
4: **if** X' not a node **then**
5: Create(X')
6: **if** count(X') = count(X) **then**
7: mark X' as not-free
8: add $X' - X$ to closure(X)
9: sort all children X' by count(X') in **decreasing** order
10: create lex order $\prec_X$ on $X' - X$ determined by the sort ▷ THE heuristic
11: **for all** children $X' \supset X$ in order $\prec_X$ **do**
12: Explore(X')

Algorithm 39 Look at all FDs in list of current node W

1: **procedure** Process-FD-List(node W)
2: **for all** $(X, A, k) \in$ fdlist(W) **do** ▷ $X \supset W$
3: **if** $A \in \text{cl}(W) \cup \text{qcl}(W)$ **then** ▷ $W \to A$
4: discard $X \to A$ ▷ so $X \to A$ not minimal
5: let Y be node k levels above W
6: **if** $Y = \emptyset$ **then**
7: issue fd $X \to A$ ▷ now known to be minimal
8: let Z be the parent of Y
9: assign $(X, A, k+1)$ to fdlist of $(X - Y) \cup Z \equiv X - (Y - Z)$

Algorithm 40 Create potential minimal FDs

1: **procedure** Issue-FDs(node X)
2: let node Y be parent of X
3: **if** $Y = \emptyset$ **then**
4: issue fd $X \to A$ ▷ minimal FD
5: let node Z be parent of Y ▷ grandparent of X
6: let $W = (X - Y) \cup Z \equiv X - (Y - Z)$ ▷ next node in lex-tree order
7: **for all** $A \in \text{cl}(X) - \text{qcl}(X)$ **do** ▷ as with *min-dep* from Fun
8: assign $(X, A, 1)$ to fdlist(W) ▷ ... except here *qcl* is approximate
9: **if** there is an $A \in R - (X \cup \text{visited}(X))$ **then**
10: then $X \cup \{A\}$ needs to be handled separately ▷ $X \cup \{A\}$ not generated as node
11: compute count($X \cup \{A\}$) ▷ ... so its count unknown
12: compare to count(X)
13: if new FD assign to approriate fdlist

Below are the procedures and functions that will be used by the algorithm are mentioned below:

Comments:

- A node Y will have a lexicographic ordering on some attributes of R. This ordering $\prec_Y$ can naturally be generalized to its supersets: if $A \prec_Y B$ then $Y \prec_Y Y \cup \{A\} \prec_Y Y \cup \{B\}$.
- In all proper subset inclusions of the form $Y \subset X$ or $X \supset Y$ above, the sets differ on a single attribute. That is, $|X| = |Y| + 1$ so there is a single $A \in X - Y$.
- The count of node $\emptyset$ should be 1, supporting the FD $\emptyset \rightarrow A$ in the case that A is single-valued. Note that if A were constant, then $|\pi_A| = 1$.
- Trying to justify the heuristic: for any $V, W \subseteq R$, $\max(|\pi_V|, |\pi_W|) \leq |\pi_{V \cup W}| \leq |\pi_V| \cdot |\pi_W|$
- For line 2.9, we need to justify that X is a minimal key. The Check-In procedure will ensure that X is free. A possible worry is that there may be a $Y' \subset X$ which is not free, but it may not yet be known that Y' is not free since its branch may not have been explored. Lemma 2 of Fun ensures that this cannot happen.

Time complexity

For our algorithm, the time complexity can be given as N (r + R + R · log(R) + FR) where R is the number of attributes in the relation, r is the number of rows. N is the total number of nodes throughout the system. r is for each node and because each parent and each child has to be checked we get R in the time complexity . The R.log(R) part comes from the sorting given by the heuristic that we have. For transferring each dependency to other node it will take FR of time unit. In most of the cases N is small but in worst case, this value is large, F is the number of functional dependencies, F = $(2^R/\sqrt{R}) \cdot (R/2)$

Worst case scenarios

1. Only keys and no functional dependencies : If there is only one key of size R in the entire relation then there are no functional dependencies that means. In this case, F = 0

   So, N (r + R + R · log(R) + FR) = N (r + R + R · log(R) + 0)

2. Keys are all subsets of R/2 size of attributes. In other words, if there are X attributes, then the left hand side of the dependency has X/2 attributes and the right hand side has the other X/2 attributes. Here in this case the tree goes till R/2 levels and we have ${}^{R}C_{R/2}$ keys and ${}^{R}C.R/2$. R functional dependencies.

Advantages and Disadvantages

Advantages

1. More space efficient than any of the other solutions.
2. Less number of nodes have to maintained at any point of time.
3. Keys are found faster.

Disadvantages

1. Time taken may be more but it cannot be commented for sure as of now
2. Delay in issuing of the dependencies.

CHAPTER IX

COMPARISON OF FD EXTRACTION METHODS

The comparison of various algorithms under study for this paper can be categorised on 2 grounds:

1. Based on the extraction method
2. Based on the search method

Categorization

Category 1: Based on method of extraction

Both the methods are based on partitioning the set of tuples with according to their attribute values. Table 4 shows the categorization.

1. Candidate generate-and-test approach: The algorithms that fall in this category are : TANE, FUN and FD-Mine. This approach uses level-wise search to explore the search space.These are pruning based algorithms, i.e. it reduces the search space by eliminating candidates using pruning rules. They begin by testing FDs with small left-hand sides and prune the search space as soon as possible. Using partitions,they can test the validity of FDs efciently even for large numbers of tuples. TANE, FastFDs and FUN both search the set containment lattice in a level-wise manner. By computing closure of candidates in level k, the FDs in this level are discovered and results from level k are used to generate candidates in level k + 1. The difference between TANE and FUN algorithms is that they use different pruning rules to eliminate candidates.

   The major difference between TANE,FastFD and FUN is the slightly different pruning rules each of these use.

2. Minimal cover approach: The algorithms that fall in this category are: dep_miner and FastFDs

   The minimal cover approach discovers FDs by considering pairs of tuples(agree sets). From the relation, a stripped partition database is extracted. Then, using such partitions, agree

Search strategy	Candidate generate test	Minimal cover
Breadth first	TANE,FUN,FD_Mine	dep_miner
Depth first	KlipFind	FastFDs

TABLE 4. Categorization table

sets are computed and maximal sets are generated. That is how a minimum FD cover is found according to these maximal sets is found.

Category 2: Based on the search method

In a tree, starting from the root node and traversing through the tree to intermediate nodes and leaf nodes can be done through two ways mainly, either in a breadth first manner or a depth-first manner. In DFS, a single path is followed until one cannot go any further from that node.(For example in KlipFind, we proceed until a non-free set is explored) and then backtrack to the previous path and then try the next branch in the tree. In BFS, the graph is traversed level by level, for any node all its neighbours are approached and then move further. Out of the two approaches depth first search seems to be better.The reason being the space complexity of DFS is less than that of BFS. The time taken is the same for both, O ($|V + E|$) where V is the number of vertices(attributes) and E is the number of edges.

On the basis of this , the algorithms covered in this study can be represented as in Table 1.

Differences among Candidate Generate-and-test Strategies

TANE,FUN,FD_Mine

The main difference among these algorithms are the pruning rules that they use and when these rules are applied. In addition to this, FD_Mine proves based on equivalences.

KlipFind

KlipFind is most similar to FUN as it has the same pruning rules as FUN and same approach for candidate generation. However the search strategy and heuristic is very much like FastFDs. This chapter illustrates Bernoulli example(Table 5) and a random example(Table 6).

Partition versus counting

TANE maintains partitions whereas FUN has the counts of partition sizes. FD_Mine seem to use both. In some sense it does not matter (may affect speed).KlipFind uses count but could maintain partitions.

Minimum cover strategy

Minimum cover does partitioning to set difference sets but not after that.

Time complexity

This section of the chapter provides the list of the time complexity of the algorithms:

For TANE, its exponential with respect to the number of attributes. If with addition of data, the set of dependencies remains the same then its linear.

For FDMine, for m attributes, $O(n.2^m)$ is the theoretical complexity where n is the number of tuples.

FastFDs takes $O(n(1 + w(n))K)$ amount of time in average case as mentioned by the authors of FastFDs.

For our algorithm, it should be $N\ (r + R + R \cdot \log(R) + FR)$.

Illustration

In order to clearly describe the difference in each of these algorithms,the following section covers an illustration of the Bernoulli example over the algorithms covered in the study, i.e. TANE, FUN, Dep-Miner, FastFDs and KlipFind. (Since FD_Mine is similar to TANE it is not illustrated separately.) Figure 1-23 shows the tree for the Bernoulli example on TANE, FUN, FastFDs and KlipFind and figure 24-26 illustrate KlipFind on the random example.

Φ

A	B	C	D	E	F
2	2	2	2	2	2

FIGURE 1. Bernoulli Example on TANE: Stage 1. Level 0 has only one element, the empty set and level 1 has six candidates which are all the singleton attributes of the relation. The numbers show the cardinality of the candidates.

A	B	C	D	E	F
2	2	2	2	2	2

AB	AC	AD	AE	AF	BC	BD	BE	BF	CD	CE	CF	DE	DF	EF
3	3	3	3	3	4	4	3	3	4	3	3	3	3	3

FIGURE 2. Bernoulli Example on TANE: Stage 2. Level 2 candidates are generated by combining all the possible sets of candidates at level 1. At each level the number of attributes in a candidate increase by one in count.

AB	AC	AD	AE	AF	BC	BD	BE	BF	CD	CE	CF	DE	DF	EF
3	3	3	3	3	4	4	3	3	4	3	3	3	3	3

ABC	ABD	ABE	ABF	ACD	ACE	ACF	ADE	AEF	BCD	BCE	BCF	BDE	BDF	BEF	CDE	CDF	CEF	DEF
5	5	4	4	4	4	4	4	4	6	5	4	4	5	4	5	5	4	4

FIGURE 3. Bernoulli Example on TANE: Stage 3 At every level the cardinality of each candidate is compared with the cardinality of its parents in the previous level. If the cardinality is the same for any of the cases this indicates that there is a functional dependency existing there.

ABC	ABD	ABE	ABF	ACD	ACE	ACF	ADE	AEF	BCD	BCE	BCF	BDE	BDF	BEF	CDE	CDF	CEF	DEF
5	5	4	4	4	4	4	4	4	6	5	4	4	5	4	5	5	4	4

ABCE	ABCF	ABDE	ABDF	ACDE	ACDF	ACEF	ADEF
5	5	5	5	5	5	5	5

FIGURE 4. Bernoulli Example on TANE: Stage 4 The same step as in previous stage is followed to get the complete lattice

ABCE	ABCF	ABDE	ABDF	ACDE	ACDF	ACEF	ADEF
5	5	5	5	5	5	5	5

ABCDE	ABCDF	BCDEF
6	6	6

FIGURE 5. Bernoulli Example on TANE: Stage 5 At this stage keys are identified and this becomes the final stage of the lattice.

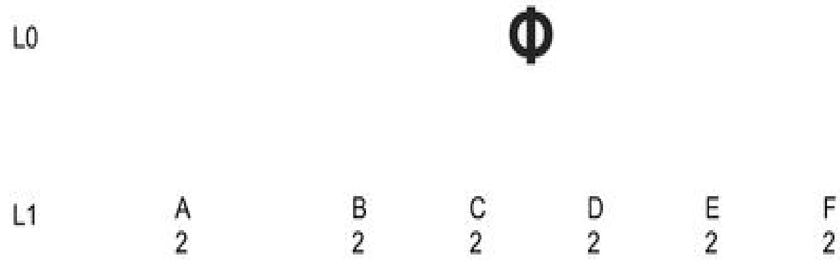

FIGURE 6. Bernoulli Example on FUN: Stage 1 Here also in level 0 there is only the empty set and level 1 has all the singleton elements of the relation.

FUN

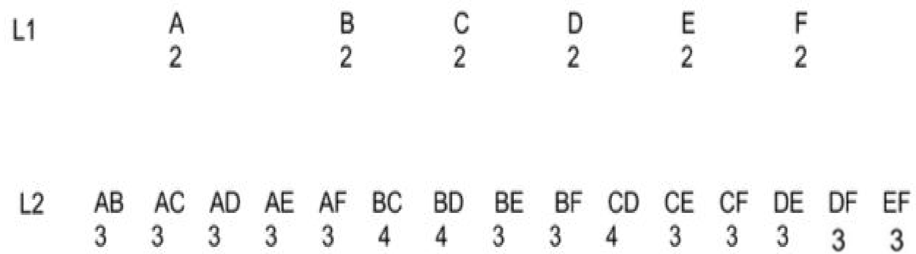

FIGURE 7. Bernoulli Example on FUN Stage 2 At every level the candidates are generated by combining candidates from the previous level and the cardinality is checked to look for non-free sets and keys.

L2

AB	AC	AD	AE	AF	BC	BD	BE	BF	CD	CE	CF	DE	DF	EF
3	3	3	3	3	4	4	3	3	4	3	3	3	3	3

L3

ABC	ABD	ABE	ABF	ACD	ACE	ACF	ADE	AEF	BCD	BCE	BCF	BDE	BDF	BEF	CDE	CDF	CEF	DEF
5	5	4	4	4	4	4	4	4	6	5	4	4	4	5	5	5	4	4
				NF					KEY			NF	NF					

FIGURE 8. Bernoulli Example on FUN: Stage 3 If a key or a non-free set is obtained then it is not included in the next level as a subset of any candidate.

L3

ABC	ABD	ABE	ABF	ACD	ACE	ACF	ADE	AEF	BCD	BCE	BCF	BDE	BDF	BEF	CDE	CDF	CEF	DEF
5	5	4	4	4	4	4	4	4	6	5	4	4	4	5	5	5	4	4
				------- NF					------- KEY			------- NF	------- NF					

L4

ABCE	ABDF	ABEF	ACEF	ADEF	CDEF
5	5	4	5	5	5

FIGURE 9. Bernoulli Example on FUN: Stage 4

L3

ABC	ABD	ABE	ABF	ACE	ACF	ADE	AEF	BCD	BCE	BDF	BEF	CDE	CDF	CEF	DEF
5	5	4	4	4	4	4	4	6	4	5	4	5	5	4	4
								-------- KEY							

L4

ABCE	ABDF	ABEF	ACEF	ADEF	CDEF
5	5	4	5	5	5

FIGURE 10. Bernoulli Example on FUN: Stage 4(Cleaner version)

$\mathfrak{D}_r$ = { CD,BF,BCDE,D,BCDF,BE,C,CDEF,BDF,BCE,ABC,ABD,ACF,ADE,ABCD }

$\mathfrak{D}_r^A$	$\mathfrak{D}_r^B$	$\mathfrak{D}_r^C$	$\mathfrak{D}_r^D$	$\mathfrak{D}_r^E$	$\mathfrak{D}_r^F$
CF DE BD BCD BC	F CDE CDF E DF CE AD AC ACD	D BDE BDF DEF BE AB ABD $\emptyset$	C BCE BCF CEF BF AB AE ABC $\emptyset$	BCD D CDF BC AD $\emptyset$	B BCD CDE BD AC

FIGURE 11. FastFDs: Stage 1 The difference set shown in the upper portion of the figure above is found by comparing each tuple with every other tuple in the the relation and combined on the basis of the same value of the attributes, Then difference set for each of the attributes(shown in the table in the above figure) is found by combining those values that contain that attribute and remove that attribute from that set of candidates. If there is a singleton attribute like C for example then the difference set of C would contain an empty set.

$\mathfrak{D}_r^A$	$\mathfrak{D}_r^B$	$\mathfrak{D}_r^C$	$\mathfrak{D}_r^D$	$\mathfrak{D}_r^E$	$\mathfrak{D}_r^F$
CF DE BD BC	F E AD AC	$\emptyset$	$\emptyset$	D BC $\emptyset$	B CDE AC

FIGURE 12. FastFDs: Stage 2 This stage is obtained from the previous stage by reducing it to a set of minimal candidates , by removing supersets of the candidates present in the respective difference sets.

FastFDs and Dep_Miner is the same till stage 2 of FastFDs. After this stage FastFDs applies DFS and Dep_Miner applies BFS.

min cover of $\mathfrak{D}_r^A$	min cover of $\mathfrak{D}_r^B$	min cover of $\mathfrak{D}_r^C$	min cover of $\mathfrak{D}_r^D$	min cover of $\mathfrak{D}_r^E$	min cover of $\mathfrak{D}_r^F$
BDF BEF CD BCE	AEF CDEF	NONE	NONE	BD CD	AC ABD ABE BC

BDF -> A , BEF -> A, CD -> A,
BCE -> A, AEF -> B, CDEF -> B,BD-> E,
CD-> E, AC-> F, ABD -> F, ABE -> F, BC -> F

FIGURE 13. FastFDs: Stage 3 The table shows the minimum cover of the difference set of all the attributes, which can be obtained from combining those attributes that alone can represent the entire set of attributes shown in the previous figure. The minimal cover of each of the attribute gives us the functional dependencies as shown in the figure above.

KlipFind

Example 1: Bernoulli Example (Refer Table 5)

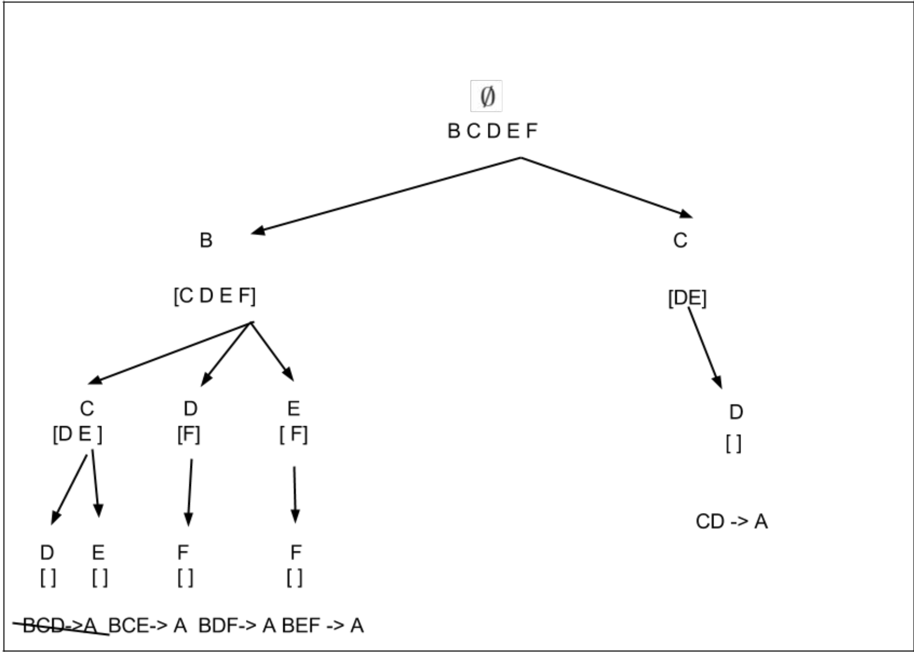

FIGURE 14. FastFDs: Stage 4 The lattice for FastFDs follows the lexicographic order at every level.

	A	B	C	D	E	F
t_0	1	1	1	1	1	1
t_1	0	0	0	1	1	1
t_2	0	0	1	0	1	1
t_3	0	1	0	1	1	0
t_4	0	1	1	0	0	1
t_5	0	0	0	0	1	1

TABLE 5. Bernoulli Example

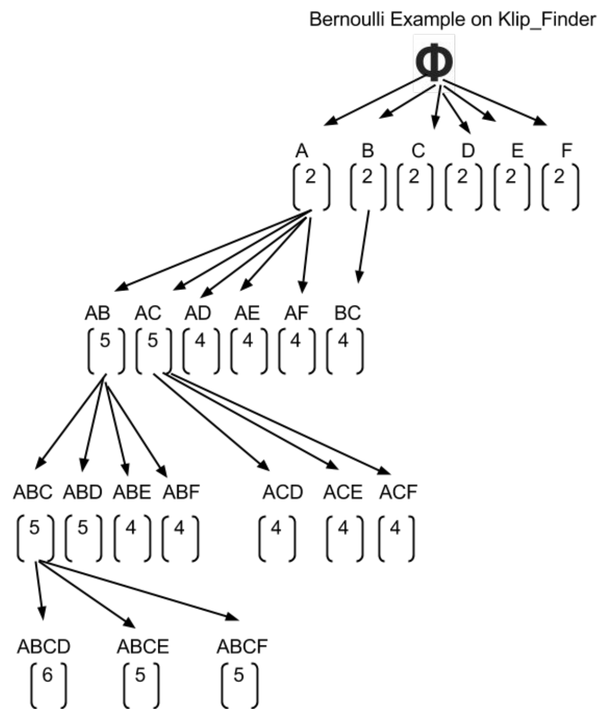

FIGURE 15. Bernoulli Example on KlipFind: KlipFind also starts off the same way as FUN, just that it follows a depth first approach.The above figure shows a stage where the leftmost candidate at level 2 is been explored and it goes down until it comes to the leaf nodes of the tree. After this stage pruning starts.

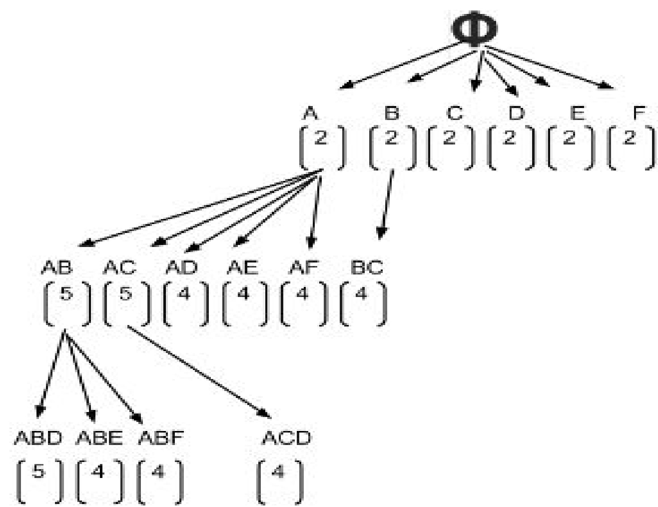

FIGURE 16. Bernoulli Example on KlipFind: This stage shows the lattice before the first prune i.e. AB and its children: Stage 2

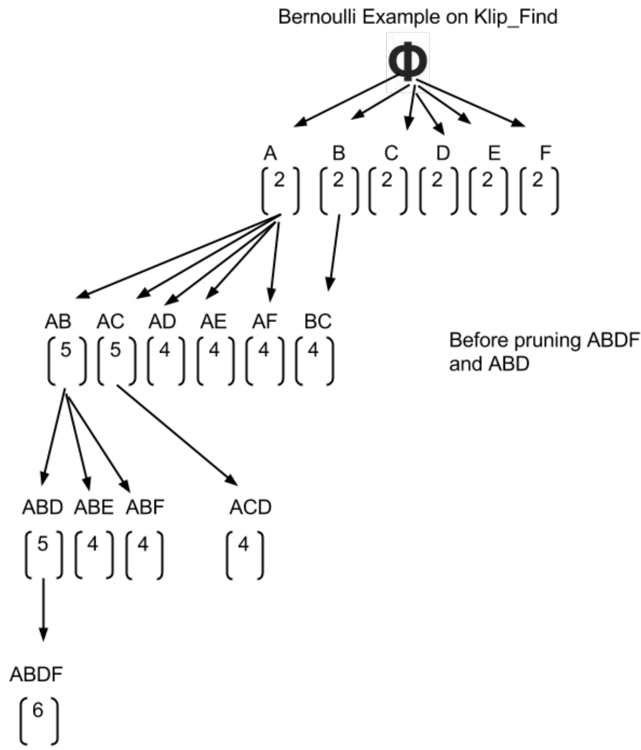

FIGURE 17. Bernoulli Example on KlipFind: Stage 3 Before pruning ABDF and ABD candidates from the lattice.

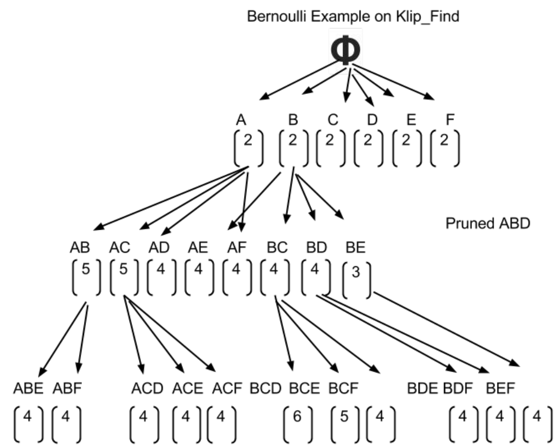

FIGURE 18. Bernoulli Example on KlipFind: Stage 4 After pruning ABD from the lattice.

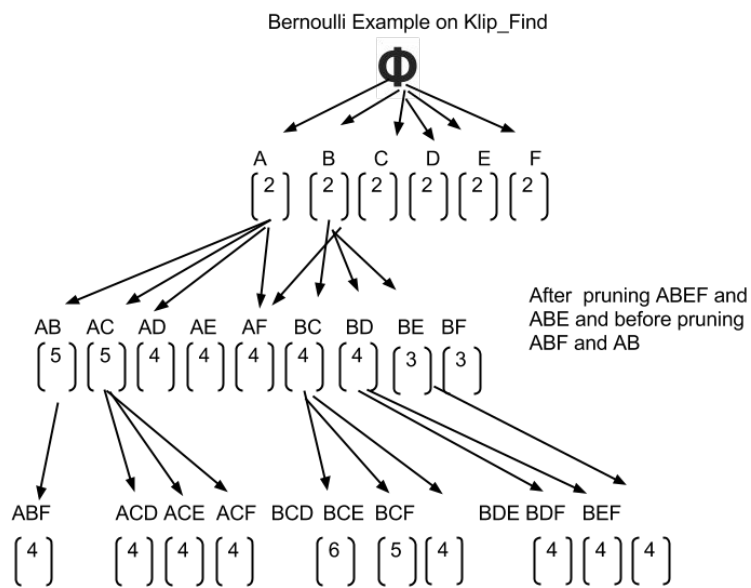

FIGURE 19. Bernoulli Example on KlipFind: Stage 5 After pruning ABEF and ABE and befor epruning ABF and AB.

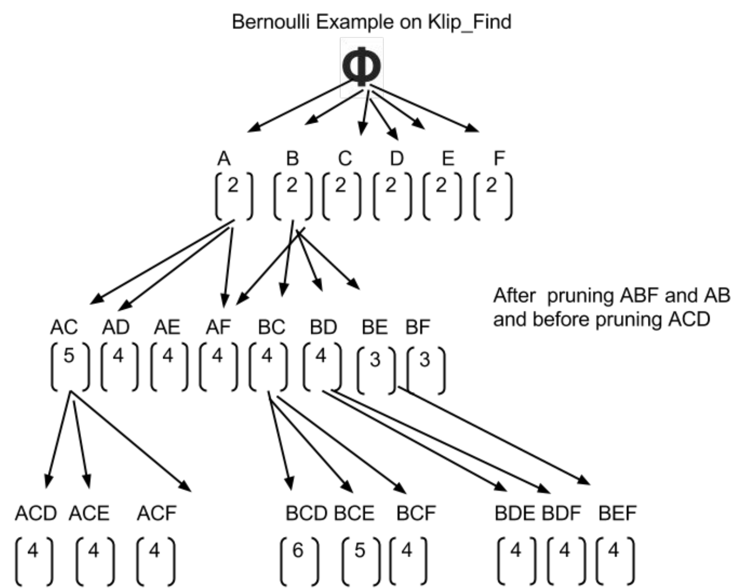

FIGURE 20. Bernoulli Example on KlipFind: Stage 6 After pruning ABF and AB and before pruning ACD.

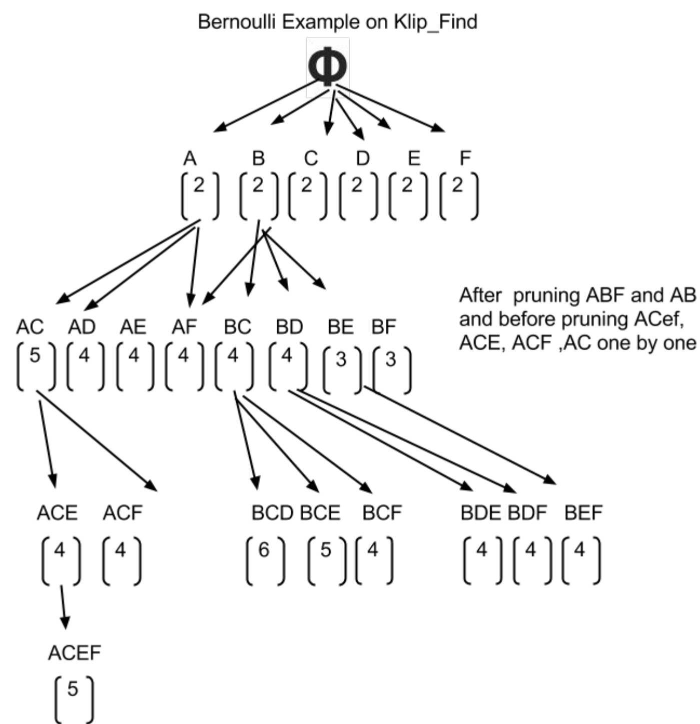

FIGURE 21. Bernoulli Example on KlipFind: Stage 7 After pruning ABF and AB and before pruning ACEF , ACE, AC one by one.

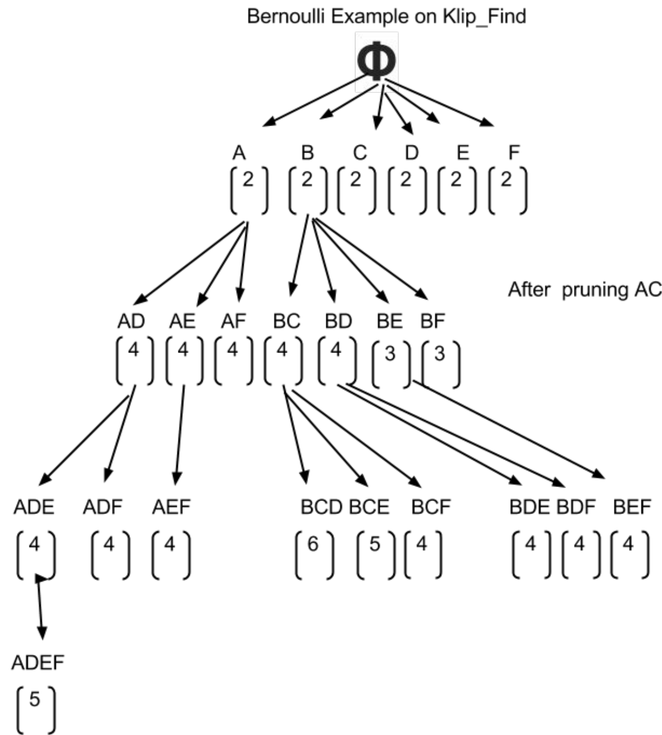

FIGURE 22. Bernoulli Example on KlipFind: Stage 8 : The stage obtained after pruning AC.

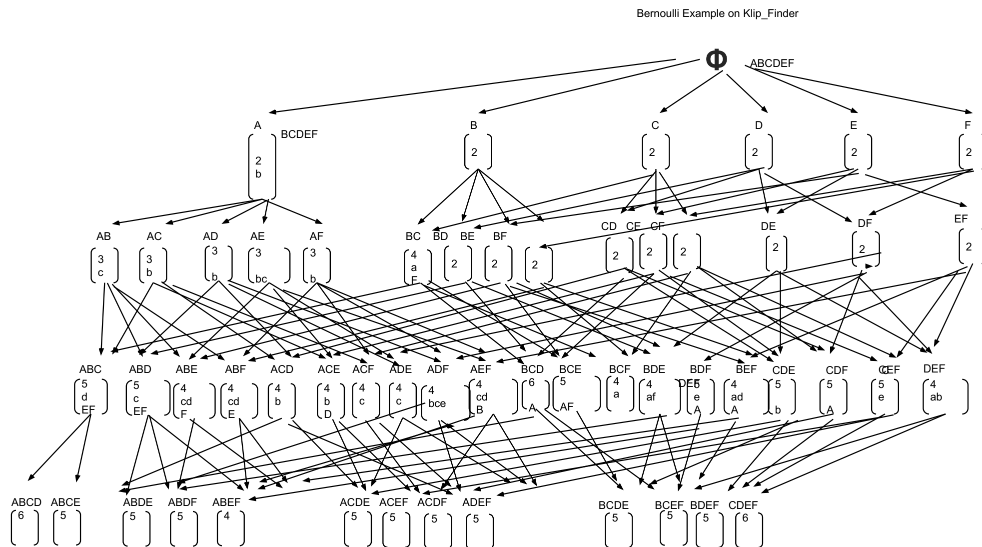

FIGURE 23. KlipFind In a similar fashion it is done for all the singleton attributes with the expansion as given in Figure 19. Not all the nodes shown in this figure are in the memory all the time.It is just showing the full expansion of the tree. The algorithm halts once all the attributes are done.

	A	B	C	D
t1	1	a	X	8
t2	1	b	Y	8
t3	1	a	X	9
t4	2	b	Y	10
t5	2	a	Z	10
t6	2	b	Y	11
t7	3	a	W	8
t8	3	a	W	9

TABLE 6. Random Example

Example 2: Random Example

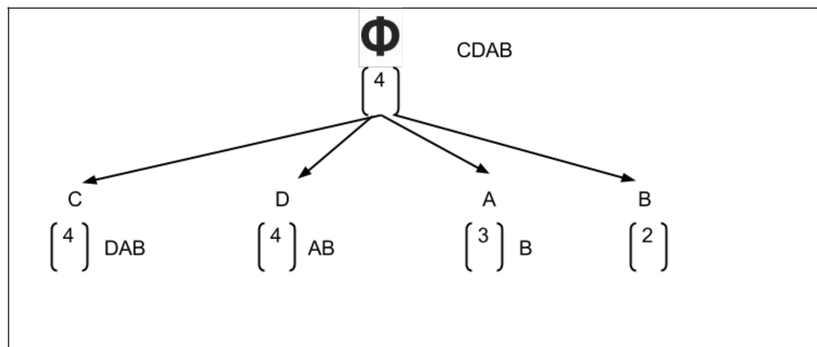

FIGURE 24. KlipFind Example 2: Stage 1

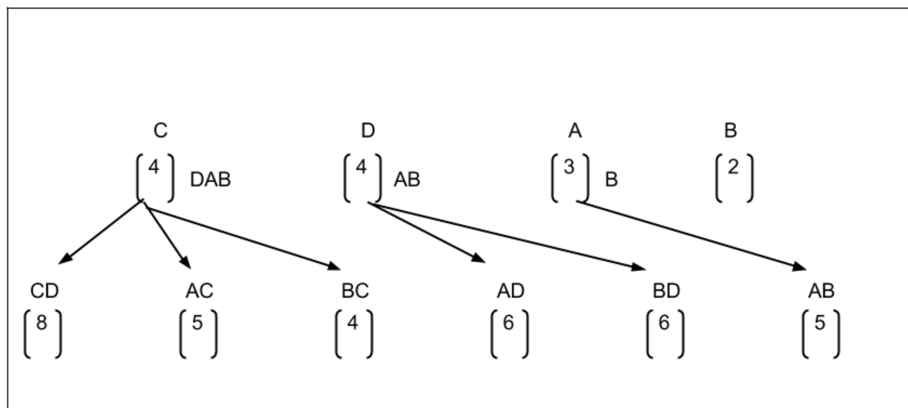

FIGURE 25. KlipFind Example 2: Stage 2

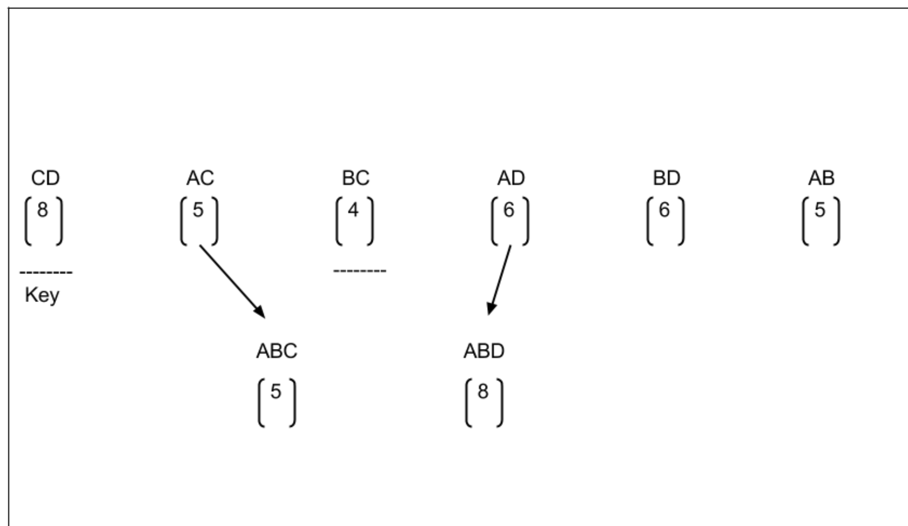

FIGURE 26. KlipFind Example 2: Stage 3

Result of the Bernoulli Example

All the algorithms mentioned in this study give the same set of functional dependencies and keys as a result.

The functional dependencies found are:

1. BDF $\rightarrow$ A
2. AEF $\rightarrow$ B
3. BEF $\rightarrow$ A
4. BCE $\rightarrow$ A
5. CDEF $\rightarrow$ B
6. CD $\rightarrow$ A
7. BC $\rightarrow$ F
8. ABC $\rightarrow$ E
9. BD $\rightarrow$ E
10. ABD $\rightarrow$ F
11. ABE $\rightarrow$ F
12. ABF $\rightarrow$ E

The keys found are:

1. BCD
2. CDEF

Time complexity

TANE

Worst case: Exponential with respect to the number of attributes but this is inevitable since the number of minimal dependencies can be exponential in the number of attributes. If the

set of dependencies do not change with increase in the number of tuples, then time complexity is linear.

FD_Mine

For m attributes, $O(n.2^m)$ is the theoretical complexity where n is the number of tuples.

FastFDs

FastFDs takes time $O(n(1 + w(n))K)$.

KlipFind

For our algorithm, the time complexity can be given as $N\ (r + R + R \cdot \log(R) + FR)$ where R is the number of attributes in the relation, r is the number of rows. N is the total number of nodes throughout the system.

CHAPTER X

CONCLUSION AND FUTURE WORK

In this study we surveyed the current techniques for extracting the minimal functional dependencies from relations.We also focussed on what are a few problems related with some of these algorithms.In the later part of this study we also did a comparison between these algorithms and provided an illustration of how they work by running the same example over some of these algorithms. We tried to compare the run times in a conservative manner.

We also introduced a new algorithm KlipFind which is more or similar to FUN, uses the same pruning rules, just that it combines this with a heuristic as that in FastFDs.In other words, we tried to combine the best features of different techniques to save space at the trade-off of time slightly being increased.

Future work includes two directions of possible extension of this research.To begin with, it is of interest to code and test the suggested algorithm and check its efficiency against the previous approaches. To imply random database results to measure average-case time and space complexity.

Another promising direction consists of simplifying the algorithm. The approach we follow includes passing the dependencies over to other nodes in the tree, travelling from left to right direction.This makes it a little complex. Although it is hard to find an alternative for that to achieve minimal dependencies, it might be feasible to achieve the same.

REFERENCES CITED

Huhtala Ykä, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2): 100–111, 1999.

Lopes Stéphane, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *Advances in Database TechnologyEDBT 2000*, pages 350–364. Springer, 2000.

Novelli Noel and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Database TheoryICDT 2001*, pages 189–203. Springer, 2001.

Wyss Catharine, Chris Giannella, and Edward Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *Data Warehousing and Knowledge Discovery*, pages 101–110. Springer, 2001.

Yao Hong, Howard J Hamilton, and Cory J Butz. Fdmine: discovering functional dependencies in a database using equivalences. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 729–732. IEEE, 2002.