Class: CS 520

Literature Review

A Hybrid Approach to Functional Dependency Discovery

Hrishika Shelar
Supriya Hinge
Jay Patel

Introduction

A functional dependency written as X → A represents all pairs of records with the same values. It implies that the values in A are functionally dependent on the values in X. For example, a person's birth date determines the age, zip code determines the city, etc. Functional dependencies are used for schema normalization, data integration, data cleansing and other data management tasks. However, despite their importance, functional dependencies are usually unknown and almost impossible to discover manually. Quite a few algorithms were proposed for identifying functional dependencies in a dataset, although none of them could process datasets of typical real world size.

Database research has offered many discovery algorithms to find functional dependencies of a dataset which are unable to work on real world size datasets which contain more than 50 attributes and millions of records. Discovery Algorithms are needed to reveal all the functional dependencies in a dataset and should be able to optimize both records and attributes.

Related Work

This Paper aims to discover all minimal functional dependencies without any restrictions or relaxations. Many algorithms use efficient pruning to cope up with the discovery problem. This Paper doesn't use efficient pruning and relies on massive parallelization. Following are 3 state of art algorithms summarized in non distributed FD discovery.

- Lattice traversal algorithms -
    - All possible FD candidates in a powerset lattice of attribute combinations are arranged Theoretically by these algorithms Tane, DFD, Fun, Fd-Mine and then traverse through this lattice.
    - Lattice algorithm has used pruning rules intensively and their candidate validation is based on position list indices.
    - But these algorithms use candidate driven strategies which do not work with the number of columns in the dataset.
    - But for this paper they have adopted the pruning rules and the position list index data structure for the validation of functional dependencies.

- Difference and agree set algorithm
    - The algorithms Dep-Miner and FastFDs analyze the datasets to form agree sets which transform into difference-sets from which all valid FDs can be derived.
    - These FDs candidates are generated from Concrete observations which make this strategy scales better than the lattice traversal strategies.
    - DEP-MINER and FASTFDs are worse when it comes to the number of records datasets contain because they need to compare all pairs of records.
    - Compared to this algorithm, the HYFD algorithm compares records pair-wise but chooses these comparisons carefully.

- Dependency induction algorithms:
    - To find all invalid functional dependencies, like HYFD this algorithm also compares all records pair wise.
    - This set is called a Negative Cover and it is stored in a prefix tree. FDEP translates this negative cover into the set of valid functional dependencies that is positive cover by successive specialization.
    - This discovery has proven that pairwise comparisons do not scale with the number of records in the input dataset and it scales with the number of attributes.

- This paper has followed a similar approach during induction of functional dependency candidates.

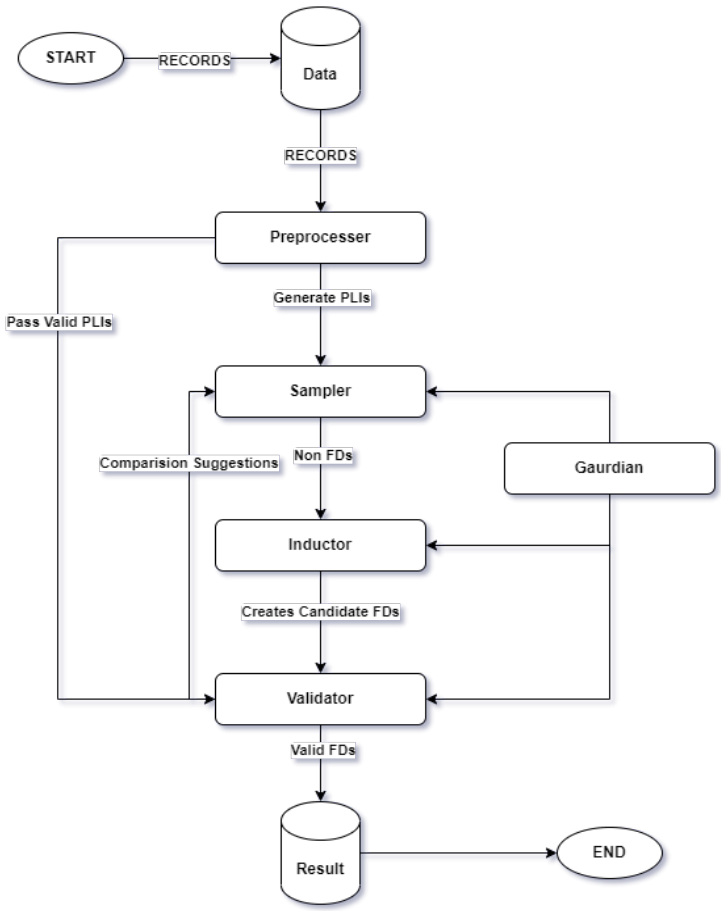
Hybrid FD Algorithm

Database research has offered many discovery algorithms to find functional dependencies of a dataset which are unable to work on real world size datasets which contain more than 50 attributes and millions of records.
Discovery Algorithms are needed to reveal all the functional dependencies in a dataset and should be able to optimize both records and attributes. Finding functional dependencies is quadratic to the number of records (rows) and exponential to the number of attributes (columns) A hybrid discovery algorithm called HYFD, which is a combination of approximation techniques and efficient validation techniques to find all the minimal functional dependencies in a given dataset. HYFD outperforms all the existing approaches while operating on compact data, and also scales to much larger datasets.

Algorithm -

- HYFD algorithm is a combination of row and column efficient discovery techniques.
- It works in two phases, in the first phase, HYFD extracts a small subset of records from the input dataset and finds functional dependencies in this sample. The result of this phase is a set of FDs that are either valid or almost valid with respect to the complete dataset.
- In the second phase, HYFD validates the discovered FDs on the entire dataset and removes FDs that are not efficiently holding on to the entire dataset.
- If the validation is inefficient, i.eThe resulting FDs do not apply to the entire dataset, HYFD is able to switch back to the first phase and continue the process again keeping in mind the previous FDs which were not applicable to the entire dataset.
- To conclude, HYFD is basically an alternating two phase discovery strategy which discovers minimal FDs in a given dataset.
- The HYFD algorithm is divided into five main parts, namely -
    - Preprocessor
    - Sampler
    - Inductor
    - Validator
    - Guardian

Flowchart of HYFD algorithm



Explanation -

1. Preprocessor transforms records into position list indexes (PLI). The records of the datasets are compressed using PLIs.

   Algorithm used for preprocessing -

   Data: records
   Result: plis, invertedPlis, pliRecords

   1 numRecs ← |records|;

```
2 numAttrs ← |records[0]|
3 array plis size numAttrs as Pli;
4 plis ← buildPlis (records);
5 plis ← sort (plis, DESCENDING);
6 array pliRecords size numRecs × numAttrs as Integer;
7 pliRecords ← createRecords (invertedP)
8 return plis, invertedPlis, pliRecords;
```

2. Sampler Implements column efficient FD induction technique and also Checks compressed records for FD violations. For example - In schema R(A,B,C) records r1(1,2,3) and r2(1,4,5) exist. A values match, B and C differ which makes r1 and r2 non- FDs. Here it uses Cluster sampling technique.

Algorithm for Sampling-

```
Data: plis, pliRecords, comparisonSuggestions
Result: nonFds
1  if efficiencyQueue = ∅ then
2      for pli ∈ plis do
3          for cluster ∈ pli do
4              cluster ← sort(cluster, ATTR_LEFT_RIGHT);
5      nonFds ← ∅;
6      efficiencyThreshold ← 0.01;
7      efficiencyQueue ← new PriorityQueue;
8      for attr ∈ [0, numAttributes[ do
9          efficiency ← new Efficiency;
10         efficiency.attribute ← attr;
11         efficiency.window ← 2;
12         efficiency.comps ← 0;
13         efficiency.results ← 0;
14         runWindow(efficiency, plis[attr], nonFds);
15         efficiencyQueue.append(efficiency);
16 else
17     efficiencyThreshold ← efficiencyThreshold / 2;
18     for sug ∈ comparisonSuggestions do
19         nonFds ← nonFds ∪ match(sug[0], sug[1]);
20 while true do
21     bestEff ← efficiencyQueue.peek();
22     if bestEff.eval() < efficiencyThreshold then
23         break;
24     bestEff.window ← bestEff.window + 1;
25     runWindow(bestEff, plis[bestEff.attribute], nonFds);
26 return newFDsIn(nonFds);
   function runWindow(efficiency, pli, nonFds)
27 prevNumNonFds ← |nonFds|;
28 for cluster ∈ pli do
29     for i ∈ [0, |cluster| − efficiency.window [ do
30         pivot ← pliRecords[cluster[i]];
31         partner ← pliRecords[cluster[i + window − 1]];
32         nonFds ← nonFds ∪ match(pivot, partner);
33         efficiency.comps ← efficiency.comps + 1;
34 newResults ← |nonFds| − prevNumNonFds;
35 efficiency.results ← efficiency.results + newResults;
```

3. After sampling Inductor Converts non-FDs to FDs passed by Sampler Process and then the Validator Implements row efficient technique on the processed data which is FDs from Inductor that are taken as input and validated against the entire dataset. If FDs are invalid, the process switches back to Sampler.

Algorithm for Functional dependency induction -

```
Data: nonFds
Result: fds
nonFds ← sort(nonFds, CARDINALITY_DESCENDING);
if fds = null then
    fds ← new FDTree;
    fds.add(∅ → {0, 1, ..., numAttributes});
for lhs ∈ nonFds do
    rhss ← lhs.clone().flip();
    for rhs ∈ rhss do
        specialize(fds, lhs, rhs);
return fds;
function specialize(fds, lhs, rhs)
invalidLhss ← fds.getFdAndGenerals(lhs, rhs);
for invalidLhs ∈ invalidLhss do
    fds.remove(invalidLhs, rhs);
    for attr ∈ [0, numAttributes[ do
        if invalidLhs.get(attr) ∨
          rhs = attr then
            continue;
        newLhs ← invalidLhs ∪ attr;
        if fds.findFdOrGeneral(newLhs, rhs) then
            continue;
        fds.add(newLhs, rhs);
```

4. Another component in this process is Guardian that Prunes the FD tree if necessary i.e., when the FD tree grows and more memory is consumed.

Algorithm for functional dependency validation -

```
Data: fds, plis, pliRecords
Result: fds, comparisonSuggestions
if currentLevel = null then
    currentLevelNumber ← 0;
currentLevel ← fds.getLevel(currentLevelNumber);
comparisonSuggestions ← ∅;
while currentLevel ≠ ∅ do
    /* Validate all FDs on the current level        */
    invalidFds ← ∅;
    numValidFds ← 0;
    for node ∈ currentLevel do
        lhs ← node.getLhs();
        rhss ← node.getRhss();
        validRhss ← refines(lhs, rhss, plis, pliRecords,
                            comparisonSuggestions);
        numValidFds ← numValidFds + |validRhss|;
        invalidRhss ← rhss.andNot(validRhss);
        node.setFds(validRhss);
        for invalidRhs ∈ invalidRhss do
            invalidFds ← invalidFds ∪ (lhs, invalidRhs);
    /* Add all children to the next level           */
    nextLevel ← ∅;
    for node ∈ currentLevel do
        for child ∈ node.getChildren() do
            nextLevel ← nextLevel ∪ child;
    /* Specialize all invalid FDs                   */
    for invalidFd ∈ invalidFds do
        lhs, rhs ← invalidFd;
        for attr ∈ [0, numAttributes[ do
            if lhs.get(attr) ∨ rhs = attr ∨
               fds.findFdOrGeneral(lhs, attr) ∨
               fds.findFd(attr, rhs) then
                continue;
            newLhs ← lhs ∪ attr;
            if fds.findFdOrGeneral(newLhs, rhs) then
                continue;
            child ← fds.addAndGetIfNew(newLhs, rhs);
            if child ≠ null then
                nextLevel ← nextLevel ∪ child;
    currentLevel ← nextLevel;
    currentLevelNumber ← currentLevelNumber + 1;
    /* Judge efficiency of validation process       */
    if |invalidFds| > 0.01 * numValidFds then
        return fds, comparisonSuggestions;
return fds, ∅;
```

Experiment Setup

Metanome, the data profiling framework is used for common task like input parsing, result formatting, and performance measurement are standardized by the framework and decoupled from the algorithms for null semantics, experiment use null = null, because this is how related work treats null values

Hardware. We run all our experiments on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB RAM. The server runs on CentOS 6.4 and uses OpenJDK 64-Bit Server VM 1.7.0 25 as a Java environment.

Varying the number of rows - Measures the runtime on different row numbers using the ncvoter and uniprot dataset. The reason why HyFD performs so much better than current lattice traversal algorithms is the fact that the number of FD-candidates that need to be validated against the many rows is greatly reduced by the Sampler component.

Varying the number of Columns - It Measures the runtime on different Column numbers using the Plista and uniprot dataset. In this experiment 2- HyFD outperforms all existing algorithms.

Varying the datasets - This experiment evaluates the algorithm on many different datasets to show that HyFD is not sensitive to any dataset peculiarity. It sets a time limit (TL) of 4 hours and a memory limit (ML) of 100 GB. Table 1 summarizes the runtimes of the different algorithms. The measurements show that HyFD was able to process all datasets and that it usually performed best.

Dataset	Cols [#]	Rows [#]	Size [KB]	FDs [#]	TANE [12]	FUN [18]	FD_MINE [25]	DFD [1]	DEP-MINER [16]	FASTFDS [24]	FDEP [9]	HYFD
iris	5	150	5	4	1.1	**0.1**	0.2	0.2	0.2	0.2	**0.1**	**0.1**
balance-scale	5	625	7	1	1.2	**0.1**	0.2	0.3	0.3	0.3	0.2	**0.1**
chess	7	28,056	519	1	2.9	1.1	3.8	1.0	174.6	164.2	125.5	**0.2**
abalone	9	4,177	187	137	2.1	0.6	1.8	1.1	3.0	2.9	3.8	**0.2**
nursery	9	12,960	1,024	1	4.1	1.8	7.1	0.9	121.2	118.9	46.8	**0.5**
breast-cancer	11	699	20	46	2.3	0.6	2.2	0.8	1.1	1.1	0.5	**0.2**
bridges	13	108	6	142	2.2	0.6	4.2	0.9	0.5	0.6	0.2	**0.1**
echocardiogram	13	132	6	527	1.6	0.4	69.9	1.2	0.5	0.5	0.2	**0.1**
adult	14	48,842	3,528	78	67.4	111.6	531.5	5.9	6039.2	6033.8	860.2	**1.1**
letter	17	20,000	695	61	260.0	529.0	7204.8	6.0	1090.0	1015.5	291.3	**3.4**
ncvoter	19	1,000	151	758	4.3	4.0	ML	5.1	11.4	1.9	1.1	**0.4**
hepatitis	20	155	8	8,250	12.2	175.9	ML	326.7	5576.5	9.5	0.8	**0.6**
horse	27	368	25	128,727	457.0	TL	ML	TL	TL	385.8	7.2	**7.1**
fd-reduced-30	30	250,000	69,581	89,571	**41.1**	77.7	ML	TL	377.2	382.4	TL	513.0
plista	63	1,000	568	178,152	ML	ML	ML	TL	TL	TL	26.9	**21.8**
flight	109	1,000	575	982,631	ML	ML	ML	TL	TL	TL	216.5	**53.4**
uniprot	223	1,000	2,439	>2,437,556	ML	ML	ML	TL	TL	TL	ML	**>5254.7**

Results larger than 1,000 FDs are only counted **TL**: time limit of 4 hours exceeded **ML**: memory limit of 100 GB exceeded

Table 1: Runtimes in seconds for several real-world datasets (extended from [20])

The datasets in Table 1 brought all state-of-the-art algorithms to their limits. These are too small as compared to real world datasets. HyFD has evaluated on much larger datasets than these seven state of art datasets.. In Table 1, HyFD's runtimes are shown only because all other algorithms cannot process these datasets in finite time and memory limits.

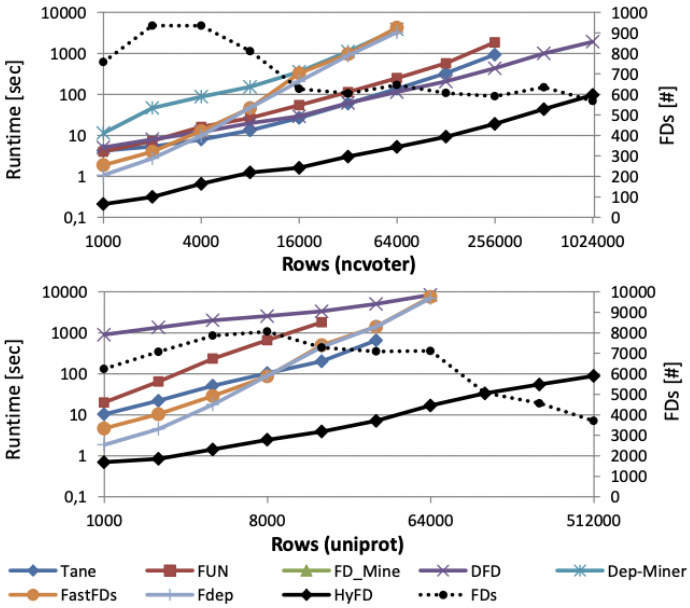


Figure 6: Row scalability on *ncvoter* and *uniprot*.

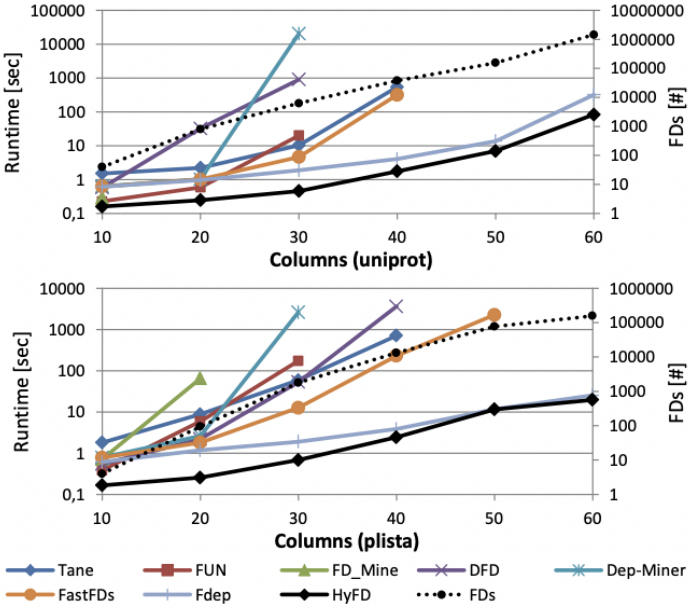


Figure 7: Column scalability on *uniprot* and *plista*.

HYFD algorithm could process the 19 column version of the ncvoter dataset in 97 seconds and the 30 column version of the uniprot dataset in 89 seconds for the largest row size. HYFD is 20 times faster on ncvoter and416 times faster on uniprot than the best state-of-the-art algorithm respectively.

Conclusion

In this paper, proposed HYFD algorithm discovers all minimal, non trivial FDs in relational database because it combines both row and column efficient discovery techniques.
HyFD is the first algorithm that can process relevant real-world size datasets which contain more than 50 attributes and a million records. HYFD offers the fastest runtimes and the smallest memory footprints for small datasets.

A task for future work is to develop use-case specific algorithms that leverage FD result sets for schema normalization, query optimization, data integration, data cleansing, and many other tasks.