Undocumented Data: A System for Explaining and Refining the Results of Format Inference

Zach DeVito

January 9, 2007

1 Background

While recent years have seen the development and use of standard text-based data formats like XML, large amounts of data still exist in ad hoc formats especially in fields such as bioinformatics or telecommunications [1]. As such, log files, legacy data records, business transactions, and document formats for specific programs may each have their own specific data layout. While their format is understandable to the programs that created them, they pose a substantial problem when one wants to use them for other purposes—requiring the creation of non-trivial tools like parsers to read them, or engines to query them for specific data. If such files were used only by a few tools, or if their creator never expected that anyone else would need to read them, then documentation for the format may be outdated, incomplete or even non-existent.

The Processing Ad Hoc Data Sources (PADS) project [2] begins to address these concerns by specifying a simple language for describing ad hoc text data [3]. It then provides tools to compile this description into parsers, statistical profilers, and query engines. The PADS library allows someone interested in the data to focus on the contents of the data rather than the tedious details of creating custom non-reusable tools to deal with formatting the data. However, PADS still requires the initial format to be fully specified in order to generate its tools. In cases where this data is not fully documented, coming up with an accurate format may be difficult for the programmer. In many instances all that is available is custom parsing code, which may rely on one's knowledge of the data's original use, and would not be useful in characterizing the structure of the data.

Unpublished work by Kathleen Fisher at AT&T Research, Professor David Walker at Princeton, and David Burke at Galois Connections, seeks to remedy this lack of knowledge of the format by inferring the format from sample data using a histogram-based approach that could potentially automate much of the tedious work in understanding the details of a format. While this approach may yield a workable format, the automatically generated format may not be optimal for human comprehension and may not capture important features of the data format that go beyond generating a method of parsing it.

2 Project Overview

This project attempts to address the lack of knowledge about the data format by using the automatically generated format to analyze sample data and present it in a more comprehensible format. To that end, it can be viewed as the second stage of format inference, analogous to the code optimizations performed after the initial code generation in a compiler. From this analysis, features of the data that were missed during the first stage (i.e, the histogram-based approach) could be discovered and reported. The results of the algorithm can then direct the refinement of the original format into a form that is hopefully simpler for someone to understand and use. In addition, it can report other constraints on the data that do not fit directly in the format itself but may be useful for the user of the data. Thus, the scope of the project fits in the middle of the overall problem of dealing with ad hoc data and readying the format for use in a system like PADS.

To address these issues the problem is first divided into finding additional constraints on the data format and then using these constraints to rewrite the format. Finding constraints on the data requires parsing sample data and searching for constraints such as the range of values for a particular integer, or the length of a string. Additionally, we are interested in pieces of data that may guide the structure of data, such as a tag like "NAME_ARRAY" that indicates a particular data structure may follow. A major part of this analysis requires the automatic discovery of functional dependencies in the sample data. For this purpose we implemented the TANE algorithm [4], which finds functional dependencies in table-based datasets, and adapts it to work with the types of data structures seen in these files. Knowing the functional dependencies is not only useful for finding constraints but is useful in higher-level analysis used in data-mining applications, and database design [5, 6, 7, 8]. Knowing the functional dependencies can help guide later steps in creating faster structures for querying or storing the data once it is parsed.

The system was coded in SML/NJ since it lends itself to the creation of compiler-like tools, allowing for the easy specification and use of the initial format. As it is the second step in inferring the structure of data, it requires the input of an intermediate representation used as a starting point of analysis. For this project, the intermediate representation was hand created for the particular examples that were examined since it is not yet possible to automatically generate this representation. Even without the automatic format inference, the system can still assist in simplifying a potentially under-constrained format made by a person who does not fully understand the format. In addition, we limit the type of data we are examining to records-based data that is not infinitely recursive. The output from the system is a list of constraints on the data and a simplified intermediate representation based on the one originally given to the process. The following sections will explain how the program analyzes the data from

interpreting it to reporting the constraint results. The results from running the algorithm on a few sample data formats, followed by an analysis of its performance properties are then presented.

3 Design

3.1 The Intermediate Representation (IR)

The system works with a defined input format that is suited to describing records-based files. It is composed of several base types: an IntBase that represents integers, a LettersBase that represents a string of letters, a ConstBase that represents a particular non-changing string of characters, and an REBase that represents a regular expression. While more base types would be useful in describing real world data, these were chosen because they could describe a wide variety of formats while remaining easy to parse and manipulate. These base types are then combined in one of four abstract types to create a structure that can describe a whole file.

- Tuples: each tuple contains a list of children IRs, each of which are parsed sequentially.
- Sums: a sum describes a data structure that can vary between a list of acceptable choices. Each sum contains a list of children IRs that describe these choices and the actual data must match one of them.
- Arrays: each array contains a single IR, which will be parsed until it is no longer possible to parse the format from the file.
- Base: each base holds a single base type that must be parsed from the file.

These types are then nested in each other to fully describe a data format. A fifth type—Label—has no semantic purpose except to give the IR it contains a name so that future steps can refer to it. It should be noted that although suited to records-based data, this format is limited in that it cannot describe infinitely recursive structures like nested tags in XML because there is not way to place a structure inside itself.

3.2 The Bank Account Example

To illustrate how each step of the constraining and simplifying process works a simple example will be provided. Consider a file that holds bank account data for many different people in a format that would facilitate parsing. The first record lists the number of people in the file followed by each person's information—title, name, gender—the number of bank accounts, a constant labeled "Accounts," and the various bank accounts that a person owns; so there is one record per person. A complete example file is

given in the Appendix but a few important features of the format should be noted. The string "Accounts" always appear in the same place for each record, unaltered. The example includes two different types of bank accounts: a checking account (type 1) and a savings account (type 2). Savings accounts have the additional property of having interest, which is listed in their information record. The number preceding accounts will indicate the number of accounts that are listed on a particular line.

Furthermore, the title of Mr. will always indicate a male and the title of Ms. or Mrs. will always indicate a female (other titles have been removed to simplify the presentation). These properties will help highlight the various features of the analysis system.

A possible intermediate representation for this example format is shown in Figure 1. Although it retains the general structure of the file to a point where an interpreter can read it, it remains unconstrained and will be used as the input format in subsequent examples. The letters in parenthesis to the right of each listing is the label that will be used to refer to the node in future examples.

```
Tuple (r1)
   int (r2)
   '\n' (r3)
   Array (r4)
       Tuple (r5)
           letters (r6)
           '. ' (r7)
           letters (r8)
           ' ' (r9)
           letters (r10)
           ' ' (r11)
           int (r12)
           ' ' (r13)
           letters (r14)
           ' ' (r15)
           Array (r16)
               Tuple (r17)
                   '{' (r18)
                   int (r19)
                   ' ' (r20)
                   int (r21)
                   ' ' (r22)
                   int (r23)
                   ' ' (r24)
                   Sum (r25)
                       int (r26)
                       '' (r27)
                   End Sum
                   '}' (r28)
               End Tuple
               ...
           End Array
           '\n' (r29)
       End Tuple
       ...
   End Array
End Tuple
```

Figure 1: The initial IR for the banking records example

3.3 Overall Process

The process of simplifying and constraining data contains four distinct steps. First, the initial intermediate representation given to the algorithm is simplified using a rule-based simplification. Since no constraint data is available at this stage the simplifications applied are only those that do not require constraints. Next, the new IR is used to parse example data from an input source so that it can be used to find constraints. This data is then passed to the constraint finding system that analyzes it and returns a labeled version of the IR and a map that links the labels to the various

constraints that were found. Finally, the constraints and the labeled IR are again passed to the simplification system, this time using simplification rules that require some constraints to operate. The overall result of this operation is a simplified IR that better represents the data it parses, as well as a list of constraints that could be used in further stages to illuminate the data format.

3.4 The Interpreter

To simplify the parsing of data from the original files, a simple recursive decent parser was constructed, which uses the given IR as a guide to parsing the data. The interpreter acts recursively on each node in the IR and either returns the data that the particular IR represents or, if it fails to parse what it expected, raises a ParseError. The data it returns is in a format parallel to the original IR except it includes the parsed data associated with each node in the description along with auxiliary meta-data like the length of an array or the branch of a Sum. For instance, parsing a Sum will result in a parallel SumD type that holds the index of the particular IR choice it parsed and the data that it parsed using that IR. Similarly, arrays will return the list of data they parsed as well as the number of entries, and Tuples will return the list of the data parsed from their children. The base types are returned as a BaseData type that similarly parallels the Base types discussed previously.

Each abstract type in the IR parses the file differently. Tuples will parse each of their children sequentially and raise an error if any of the children cannot be parsed. Sums will try to parse each of their options in order and, on error, they will proceed to the next option, returning the first successful parse. Arrays will try to parse their IR until an error is raised at which point they return the list of successful parses. When a base type is reached in the recursive decent, the interpreter tries to read the type off of the input stream and will raise an error if it cannot. Each level of recursion has a pointer to the point at which it began parsing, allowing it to continue from that point on failure.

While simple, this method will not be able to parse every IR as it does not backtrack on errors. For instance, if a Sum lists two options: an int or a tuple of two sequential ints separated with a space, then the interpreter will mistakenly parse the first option when the data is actually in the second form and will likely fail at a later stage even though the IR represents this data. Since the focus of this project was constraining and simplifying of data, the IRs were written in a way that avoided this problem instead of spending more time on the interpreter. The fact that Sums stop searching on a successful parse means that if a Sum includes a choice that is an Array then the parse of the Array will always be successful as a 0 member array. To resolve this situation, Arrays will raise an error if they cannot find any items. If a data format needs to represent the presence of a possible zero-length array, it can do so by creating the Sum of

the Array type or the nil constant—a constant base type with the value “” that will always be successfully parsed. The result of the top-level call to the interpreter will result in a full data tree that parallels the IR and can be used in constraint analysis. A resulting data structure for our example file is given in the Appendix.

3.5 Constraint Analysis

Since finding constraints requires looking at repeated instances of the same IR, it is useful to group data parsed from the same IR together in such a way that it allows for easy comparison. Implicit information in the data—such as the length of an array or which choice was taken in a sum—should also be analyzed as it can suggest possible simplifications. To accomplish these goals the data is first transformed from its IR interpretation into a group of tables that each represents the repeating contents of a particular Array in the data structure. This table layout is also needed since the algorithms that will be used to find functional dependencies operate on table-based data.

Converting an IR to a table depends on the structure of the IR and works recursively. Every node in the IR is given a label, which will be used to refer to that node as it exists in the tables and as it exists in the constraints. These labels will also become the headers for the data in the tables. The expansion algorithm carefully ensures that each row of a table will be the same length regardless of what branches the data represents so that each row of data has the same form and lines up with the appropriate label headers. Each non-constant Base type becomes a single entry in the array; the constant base types are not represented in the tables because they do not change and will not reveal any useful information. Tuples are expanded sequentially by recursively expanding the Tuple's child types. Sums are expanded in the same way as Tuples except that a column of integers that reflect what branch was used to parse a particular entry precedes each Sum. When data for a Sum is added to the table, the value of NONE is placed into every column within the Sum where no data was parsed. Since each column represents an atomic value, this normally results in many adjacent NONE values and a larger row size. This method was chosen over others such as leaving enough room for the longest option in the Sum and padding the rest with NONEs since it allows us to label the columns with the name of the node and ensures that each column has the same type (IntBase, LettersBase, or REBase).

Since Arrays form the basis for determining constraints using repeating instances of data, they are handled differently than the other types. Every time an Array is reached in the table expansion its length is inserted as a column in the current table. The table creation algorithm is then called on the IR that the array repeats. Instead of putting these values in the current table, a new table is created, labeled by the

name of the Array. When data is inserted into the tables, each parsed instance of data for this particular IR is inserted as an entry into this new table. In this way, the repeated instances of the same IR are grouped together for analysis. Since the top level IR does not have to start with an Array, the data parsed from a structure outside any Array node is placed in a table called "BASE," which will only have a single entry. Although at the current time the system only parses/analyzes a single file represented by its IR, it would be trivial to allow it to parse multiple files, which would provide multiple entries in the base table and facilitate its analysis.

Figure 2 shows the tables resulting from the complete conversion of the banking records example data to tables. Running the table creation process on our example banking data results in the creation of three tables: the BASE table containing a single entry (the first int in the file and the length of the person array), the table containing the records of each person, and a the table that contains each bank account record. The last table includes NONE entries because the last integer, representing the interest in a savings account, is missing from the checking account entries. The constant elements like spaces and new lines are not represented in the table.

Figure 2: The data from the banking records example in table form.

Table: BASE

r2	r4
20	20

Table: r16

r19	r21	r23	r25	r26
1	5052	50	r27	NONE
2	6052	3000	r26	5
1	5999	200	r27	NONE
1	5777	100	r27	NONE
2	6777	2000	r26	3
2	6598	3025	r26	3
1	5623	500	r27	NONE
2	6666	1000	r26	3

...

Table: r4

r6	r8	r10	r12	r14	r16
Mr	Doe	M	2	Account	2
Mr	Smith	M	1	Account	1
Mr	Jones	M	3	Account	3
Mr	Gross	M	1	Account	1
Mrs	Green	F	1	Account	1

...

3.6 Single Column Constraints

With the data now in tables, the individual base values can be examined to see if they meet certain constraints. The current system searches for a few simple single-column constraints by simply iterating each level in the table to determine if a particular constraint holds. The constraints that are found this way include Range (the minimum and maximum values of an integer), Length (every instance of this value has the same length), Uniqueness (every instance shares the same value for this column, which this constraint records), Ascending/Descending (the items are in sorted order), and Enum (there is only a limited number of options for this value). To maintain the Enum constraint, a set of the previously seen values is recorded. Currently when this set becomes larger than a certain number the Enum constraint is

abandoned. This could potentially be modified to base its decision to keep an Enum constraint on some more intelligent metric such as its fraction of the array size. Except for the Enum constraint, this portion will run in time linear to the number of entries in the table. Since the Enum must maintain a set of previous values and insert into the set at each step, it requires $O(nlgn)$ time (sets are implemented as red-black-trees, and hashing could potentially restore this procedure to linear time). During the table creation process, the labels that are encountered while putting a data entry into a table are recorded. Labels that are not recorded during this process are considered unused—they exist on a branch of a Sum that was never parsed. This constraint is useful for the later simplification algorithm since it allows unused branches to be removed. While simple, this step finds many pieces of useful information in our example. It discovers that the letters preceding the listing of bank accounts are always "Account," that the letters following a persons name are limited to the characters 'M' or 'F,' that titles are always Mr., Ms., or Mrs., and that bank accounts always start with a 1 or a 2.

3.7 Functional Dependencies

For sets of columns A and a column $r1$ in a table a functional dependency $A \rightarrow r1$ exists when the values in column a can always uniquely determine the value in column $r1$. Put another way, for any two rows of the table that share the same values in every column in A, they must also share the same values in $r1$. $r1$ is called functionally dependent on A if $A \rightarrow r1$. In Figure 3, $\{r1, r3\} \rightarrow r2$ is a valid dependency, but $\{r1, r2\} \rightarrow r3$ is not because rows 7 and 8 have the same values for $r1$ and $r2$ but a different value for $r3$. Furthermore, a dependency $A \rightarrow r1$ is minimal if $r1$ is not functionally dependent on any proper subset of A. For instance, in Figure 3, $\{r1, r3\} \rightarrow r2$ is minimal but $\{r1, r3, r4\} \rightarrow r2$ is not. A dependency $A \rightarrow r1$ is trivial if $r1$ is a member of A. By finding the non-trivial minimal functional dependencies in the tables that we have created from the data, we can find important relationships that may help in simplifying the overall data structure, or better explain its structure. For instance, in our banking records example, functional dependencies will reveal that the title Mr. will always coincide with a M, and that Mrs. and Ms. will coincide with an F. Although obvious in this example, in cases where this fact is not obvious, knowing it will help explain where the data is coming from or help direct a computer to use that data more efficiently (e.g., there is no need to store the 'M' or 'F' when it can be determined by the title alone). Functional dependencies will also reveal that the number preceding the bank accounts for each person is, in fact, the length of the following Array.

3.8 Finding Functional Dependencies

The problem of finding functional dependencies has been studied in database design and data mining [8], and many algorithms exist for finding them [4, 7, 9]. All the known algorithms are by nature exponential in row size [10], but each tries to improve its run time by eliminating extra work. For this system, the TANE algorithm was implemented using the procedures that Huhtala et al describe [4]. Several reasons went into choosing this algorithm. Most importantly, it works levelwise, examining dependencies determined by a single column, then pairs of columns, triples, and so on, finding every dependency at a certain level before moving on. This allows the algorithm to be cut short after a certain level even if it has not found all the possible dependencies, which is useful when the row size grows beyond what the algorithm can feasibly handle. Also, TANE operates very independently from the actual values in each column, allowing the functional dependency part of the code to work independently from the rest of the system. Finally, TANE did not require complicated data structures beyond the sets and maps already available in SML/NJ libraries.

3.9 The TANE Algorithm

Huhtala et al [4] provide a complete description of the algorithm, including proofs for why each part of the TANE algorithm accomplishes its goal. Outlined here is the general strategy behind TANE, including how it uses the concept of partitions to easily check if a dependency exists. Consider a single column in the table; if we number each row of the table, then we can group the rows into equivalence classes that have the same value in a particular column. For instance for column $r3$, $\{1, 3, 4, 6\}$ is one such equivalence class since they each share the value '$.' This concept can be extended to multiple columns such that two rows are in the same class only if they agree in all the columns being considered. In the TANE algorithm, the entire set of equivalence classes for a particular set of columns is the partition for those columns $r3$, denoted $\pi_{\{r3\}}$. Figure 3 shows the partitions for each column in the figure. Huhtala et al [4] shows that a functional dependency $A \rightarrow r1$ holds if and only if $|\pi_A| = |\pi_{A \cup \{r1\}}|$, where $|\pi_A|$ denotes the number of equivalence

Figure 3: Example partitions (table from Huhtala et al [4])

Tuple ID	r1	r2	r3	r4
1	1	a	$	Flower
2	1	A		Tulip
3	2	A	$	Daffodil
4	2	A	$	Flower
5	2	b		Lily
6	3	b	$	Orchid
7	3	C		Flower
8	3	C	#	Rose

Partitions of attributes:

$\pi_{\{r1\}} = \{\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}\}$
$\pi_{\{r2\}} = \{\{1\}, \{2, 3, 4\}, \{5, 6\}, \{7, 8\}\}$
$\pi_{\{r3\}} = \{\{1, 3, 4, 6\}, \{2, 5, 7\}, \{8\}\}$
$\pi_{\{r4\}} = \{\{1, 4, 7\}, \{2\}, \{3\}, \{5\}, \{6\}, \{8\}\}$

classes in the partition A. Intuitively, this is the case because if the set of columns A determines $r1$ then the addition of column $r1$ to the partition cannot break any of the equivalence classes. Furthermore, the information retained in two partitions can be used to compute the partition for the union of their columns, allowing partitions for the present step to be calculated using the partitions of the previous step. Huhtala et al [4] describe an algorithm linear in proportion to time the number of rows for this process.

Using the partitions to test whether or not a dependency exists, TANE works its way through the combinations of columns level-wise, finding all dependencies of the form $\{r1\} \rightarrow r2$ before $\{r1, r2\} \rightarrow r3$. At each level, the algorithm starts with a list of left-hand-side column sets that it will examine for dependencies. These sets for level $(l + 1)$ are generated from the sets in the previous level (l) by finding all sets of size $(l + 1)$ whose subsets of size (l) are all in the previous level. For instance if level 2 contains the sets $\{r1, r2\}, \{r1, r3\}, \{r2, r3\}, \{r2, r4\}$, and $\{r3, r4\}$ then level 3 will contain $\{r1, r2, r3\}$ and $\{r2, r3, r4\}$. By using the previous level to generate the current level, TANE can prune out sets that are no longer needed and the next stages of the algorithm will not consider them. Two separate methods prune the search space. One eliminates searching for dependencies that would not be minimal and the other method finds keys and prunes them from the search.

Since we are only interested in minimal dependencies, once a dependency $A \rightarrow r1$ is found, it is no longer necessary to test for the existence of functional dependencies whose left-hand-size are simply supersets of the dependency A. For instance, once $\{r1\} \rightarrow r2$ is found to be a dependency, any dependency of the form $A \cup \{r1\} \rightarrow r2$ does not have to be tested, since it is non-minimal. To avoid checking these relations, TANE maintains a set of right-hand-side candidates for each left-hand-side A in a level. These candidates contain all the columns $r3$ such that no functional dependency $B \rightarrow r3$ has been found where B is a subset of A—essentially it maintains the list of columns where a minimal dependency may still exist. It then uses this information to reduce the number of dependencies it has to check for as dependencies are found. When checking a set A for dependencies, it will not consider a column if it is not in its candidate set for A. The candidate sets for a new level $(l + 1)$ can be calculated from the previous level (l) by taking the intersection of all size (l) subsets of each set of columns A in the new level. If a dependency $A \rightarrow r1$ is found, it is reported and $r1$ is removed from the candidate set of A. When a candidate set for a set of columns A becomes empty, A can be pruned from the current level since any superset of it in subsequent levels will also be empty.

In addition to pruning sets that contain no non-minimal dependencies, TANE searches for superkeys, removes them from the level-wise search and reports their dependencies. A superkey is a set of columns for which no two rows have the same value in every column. In other words, it uniquely identifies a row. Since a superkey will have a partition with singleton equivalence classes, it is easy to identify (the

number of equivalence classes are equal to the number or rows). A key is a superkey in which no proper subset is a superkey (i.e., if $\{r1, r2\}$ is key, $\{r1, r2, r3\}$ is a superkey and neither $\{r1\}$ nor $\{r2\}$ is a superkey). In the pruning stage, TANE will identify the superkeys. If the superkey is a key, it will report the remaining dependencies it determines. It can determine what these are using the candidate sets it had already computed. Finally, the superkey is pruned from the current level. In both cases of pruning, Huhtala et al [4] demonstrate that no dependencies are lost.

Overall, the TANE algorithm operates iteratively using the properties described above. Using initial single member partitions calculated from the actual data, and candidate sets initialized to the entire set of columns it begins its first iteration. For each iteration, the algorithm first computes the dependencies for the level using the partitions and only checking on those dependencies indicated as possible by the candidate sets. Next, it prunes the level by finding empty candidate sets and superkeys. Using the pruned level, it then generates the sets of columns for the next level.
This stage also pre-computes the
partitions for the next level by taking the union of two distinct subsets of the partition present in the current level. The algorithm continues until a set of columns used in the next level becomes empty. Huhtala et al [4] report the worst-case time complexity of TANE is $O((R + C^{2.5})2^C)$ and the space complexity is $O((R + C)2^C/\sqrt{(}C))$ where C is the number of columns and R is the number of rows, but notes that in practical analysis the algorithm will likely run much faster. For this system TANE was implemented directly in SML/NJ using the built-in set and map structures. In addition to reporting all the functional dependencies, the implementation also returns a list of keys, which although unused at this point, could be useful in understanding the data structure.

Figure 4: Some of the dependencies found by running TANE on the banking records example.

```
{} -> r14
{r6,} -> r10
{r16,} -> r12
{r12,} -> r16
{r8,r10,} -> r6
{r8,r12,} -> r6
{r6,r8,} -> r12
{r8,r16,} -> r6
{r6,r8,} -> r16
{r8,r12,} -> r10
{r8,r10,} -> r12
{r8,r16,} -> r10
{r8,r10,} -> r16
{r21,} -> r19
{r23,} -> r19
{r25,} -> r19
{r19,} -> r25
{r26,} -> r19
{r21,} -> r23
{r21,} -> r25
{r21,} -> r26
{r23,} -> r25
{r26,} -> r25
{} -> r2
{} -> r4
```

3.10 Pruning 'Useless' Dependencies

Figure 4 shows some of the dependencies generated for the banking records example. At this point the system has identified that certain relationships exist between records in the example. For instance $\{r6\} \rightarrow r10$ describes the relationship between a person's title and his/her sex, and $\{r12\} \rightarrow r16$ (with reciprocal dependency $\{r16\} \rightarrow r12$) show the relationship between the integer preceding the bank listings

and the number of bank account listings present. Quite a few other dependencies are also listed for this small example. Some of these like $\{r19\} \rightarrow r2$ (the amount of money in the account determines the type of account) probably have little meaning and would not exist if more data was provided for analysis. Others like $\{r26\} \rightarrow r25$ have been artificially formed by the creation of tables from non-tabular data. In this specific case, a value within a Sum is determining which IR the Sum parsed. This dependency is redundant—it is simply telling us that the presence of data in the space allocated for that branch of the Sum indicates that that Sum was parsed. This kind of constraint will appear in general when a Sum exists because the presence of data in some of its columns combined with the absence in others is always going to indicate which branch was taken. Furthermore, in cases where data inside a Sum is always the same value, the branch of the Sum taken will always determine the value of this data. See Figure 5 for specific examples of these two occurrences.

In practical tests, it became clear that these constraints were obscuring more useful information and a system was designed to eliminate them before they reached further steps. In general the system applies a set of rules to each dependency that test to see if they provide redundant information. Currently two rules exist: a dependency is considered useless if the value of a Sum alone determines something inside the Sum or if a value inside of the Sum is involved in determining the value of the Sum (i.e., which branch was taken). These two rules remove the majority of the dependencies that results from anomalies due to the way the data was converted from tree to table form.

Figure 5: Examples of dependencies that are pruned for a small IR.

```
Tuple
    int (r1)
    Sum (r2)
        int (r3)
        letters (r4)
    End Sum
End Tuple
```

Some "Useless" Dependencies:

- $\{r2\} \rightarrow r4$
- $\{r2\} \rightarrow r3$
- $\{r3, r4\} \rightarrow r2$
- $\{r1, r4\} \rightarrow r2$

3.11 Determining Dependency-Based Constraints

Although the dependencies alone are certainly helpful in understanding the data, it is necessary to determine what some of these dependencies are in order to convert them into well-defined constraints on the data. The problem of finding relationships between functionally related data is a major problem in artificial intelligence and other fields, and many methods have been developed to try to create approximate versions of the function that originally determined the data. Classical neural networks attempted to tackle

this problem in general [11], and linear classifiers like Support Vector Machines [12] also tackle this classification problem. Since in this context we are more interested in finding human-understandable functions than in finding as many of the functions as possible, we limited the scope of finding a functional relationship to finding a linear relationship between integer values and finding reasonably sized mappings between functionally related columns. In addition to being readily understandable, these methods will also prove useful in understanding many records-based file formats. For instance, the linear equation constraint will help reveal when a number in a record may indicate the number of bytes needed to hold an array of a particular data-type in the file, while a mapping could reveal how certain structures are always paired together such as open and close brackets in an HTML file. The system for determining constraints applies a list of constraining rules to each constraint found with the TANE algorithm. Each rule is given the columns of data that correspond to the constraint so that it can search for its specific pattern in the data.

3.12 Linear Equation Constraint

The first of these rules attempts to find linear equations that describe constraints between numbers. Although only IntBases are considered, this constraining can also operate on Array lengths, which are represented internally as integers. Since the linear equations may include non-integer constants, we have provided a rational numbers structure that can determine the exact linear equation between integers even if the constants themselves are not integer. Furthermore, we have utilized the SML Dense Matrix libraries from the PSciCo project [13]. These libraries were adapted slightly to work with rational numbers. To find a possible linear equation that matches up with a functional dependency, the system first appends a column of ones to the input data, which, given a dependency $A \rightarrow r1$ will allow for equations of the form $r1 = c + \sum_{i=1}^{|A|} c_i * a_i, a_i \in A$ where c_x and c are constants. Then $|A| + 1$ sample rows are selected from the data such that each row is linearly independent of the others. If there are not enough rows then the problem is under-constrained and the search is ended. Although constants can be chosen to fit this under-constrained data, it is unlikely that those constants would be correct if more data were present. Otherwise, the sample rows are put in a matrix, the matrix is inverted (since each row is linearly independent it will be invertible), and used to determine the coefficients for a possible linear dependency. This equation is then verified by checking that it holds for all the rows and an Equation constraint is generated. In the banking records example, an Equation constraint will reveal that the integer preceding the Array of account listings is the same as the number of accounts listed.

3.13 Switched Constraint

If the linear equation does not hold, the system attempts to construct a Switched constraint that simply enumerates the functional mapping between the columns. This mapping may be very complicated, for instance, for keys in the table it will include an entry for every row. Thus when a mapping becomes too large (defined as being larger than one fourth of the number of rows, or greater than 50 branches overall), the algorithm searches the right hand side of the mappings to find the most frequent value and makes it a default case for the switch. If this reduces the number of entries to an acceptable level, the Switched constraint is reported otherwise it is abandoned. In the banking records example, the mapping between title and gender as well as the mapping between the type of account and the presence of an integer representing interest are both reported as a Switched constraint.

After determining the dependency-based constraints, the system aggregates all constraints into a constraint map that describes all the constraints found for each label, and returns this mapping along with the labeled IR and a list of the labels that are actually used.

3.14 IR Simplification

In addition to helping explain the data, the constraints suggest ways to rewrite the original IR in a way that can be simpler to understand, more concise and efficient. The simplification occurs in two steps. The first step occurs before the IR is constrained by the data, allowing rewrite rules to make obvious structural changes in the IR before labeling makes it difficult to restructure the data in a way that retains the same label-based constraints. After the constraints are found, the simplification system is run again using the constraint information to make further revisions. Since a method is needed to determine whether one IR is 'simpler' than another, a cost function was devised to give an integer cost to an IR based on the heuristic principle that the more nodes and constraints in an IR, the more difficult it is to understand. Each constant base type has a cost of 1, other base types have a cost of 2. More complicated structures—Sums, Tuples, and Arrays—have a cost of 3. In addition, each single column constraint has a cost of 1, while each dependency-determined constraint has a cost of 5. While somewhat arbitrarily determined, in practice these cost values lead to the simplification that is expected for the example datasets.

The simplification process works with a series of rewrite rules, implemented as functions that take the IR and constraints and possibly return a new IR based on a particular rule for rewriting the structure. Working bottom-up, the algorithm first applies itself recursively to all present child-nodes in an IR structure. Using the new child nodes, it applies each rewrite rule, obtaining a list of possible rewrites for each rule. It then selects the IR with minimum cost from the rewrites and the original IR. Since one rewrite

Figure 6: Example rewrite rules.

Rule	Before Simplification	After Simplification
Prefix/Postfix of Sums Every branch of the sum has a common prefix of int and letters and a common postfix of letters and int. These are lifted out of the Sum to simplify it.	Sum Tuple int letters int letters int End Tuple Tuple int letters letters letters int End Tuple Tuple int letters '#' letters int End Tuple End Sum	Tuple int letters Sum int letters '#' End Sum letters int End Tuple
Constant Replacement The base type is replaced with a constant because it was found to be unique.	IR: letters (lab1) Constraints: lab1: Unique 'Foo'	IR 'Foo' Constraints: lab1: none
Enum converted to Sum Converting Enum constraints into Sums sometimes reduces the cost of the IR.	IR: letters (lab2) Constraints: lab2: Enum 'Male' 'Female'	IR: Sum 'Male' 'Female' End Sum Constraints lab2: none

rule may make it possible for another rule to be applied, this process of running the rules and selecting the minimum IR is repeated until the cost of the IR no longer decreases. This iteration will achieve a local minimum in the cost of the IR, but could potentially overlook slightly better global minima. To reduce the number of times this occurs, rules that knowingly increase the complexity of the IR may call some subset of the simplification rules recursively before they return their IR to increase the chance it is considered.

The simplification system includes eleven rules. Simple rules include removing unnecessary nesting of Tuples within Tuples or Sums within Sums, compressing adjacent ConstBase types into a single node, removing degenerate Tuples and Sums that contain only zero or one item, removing empty ConstBase types from inside Tuples, and removing Labels nested in other Labels. In addition to these, the children of Sums are checked to see if they all share a common prefix or postfix, and if present these are lifted out of the Sum. Using the constraints generated in the previous step, a few more rewrite rules become possible. Values found to be unique are replaced with their constant equivalents, unused branches are removed from

Take a starting IR:

```
Tuple
    Sum (s1)
        '1 ' (r1)
        '2 ' (r2)
    End Choice
    Sum (s2)
        '10 ' (r10)
        '11 ' (r11)
        '12 ' (r12)
        '13 ' (r13)
    End Choice
    Sum (s3)
        '101' (r101)
        '102' (r102)
        '103' (r103)
        '104' (r104)
        '105' (r105)
        '106' (r106)
    End Choice
End Tuple
```

And a data a file containing only these strings:

- '1 10 101'
- '1 11 102'
- '1 11 103'
- '2 12 104'
- '2 13 105'
- '2 13 106'

These dependencies will be generated with corresponding switched constraints:

- $\{s3\} \rightarrow s2$
- $\{s3\} \rightarrow s1$
- $\{s2\} \rightarrow s1$

Since $s3$ appears most often on the left hand side, it will be chosen to guide the rewriting and adjacent constants will be combined, resulting in:

```
Tuple
    Sum
        '1 10 101'
        '1 11 102'
        '1 11 103'
        '2 12 104'
        '2 13 105'
        '2 13 106'
    End Sum
End Tuple
```

Figure 7: Example of how Switched Sums can be rewritten.

Sums, and Enum constraints are used to replace base values with Sums containing a list of constant options for the base type. Although this increases the number of nodes in the structure, changing Enums into Sums eliminates the Enum constraint and may prove less costly. Examples of the more complicated constraints are listed in Figure 6.

3.15 Rewriting Switched Sums

The last rule considers the cases when the branching decision in one Sum determines the decision in another Sum. This may occur when the initial IR description has mismatched the organization of a data format and failed to recognize larger overall patterns. An example of the whole process of Sum transformation is described in Figure 7. To perform this transformation using the constraints from the previous step, a Tuple's children are examined and the Sums are extracted. Any Switched constraints are aggregated during the process, and if any one of the Sums found never appears on the left or the right of a Switched constraint, it is eliminated.

Next, the Sum that determines the path of the most Sums (i.e., the one that appears most often on the left-hand side) is selected to guide redirection. Sums not determined by the guiding Sum are

removed from consideration. Each branch of the guiding Sum is then used to determine the values of the other Sums, and a Tuple is created for each branch of the guiding Sum with the Sums replaced with their appropriate branches.

After performing this rewrite, the overall Sum is checked for common prefixes and postfixes since the Sums may have been in the middle of the original Tuple. In many cases, this rewrite rule makes the structure more complex, but in a few cases, the rewrite allows for significant further changes that would not have been possible otherwise.

After applying the simplification system, the entirety of the analysis is reported, including the constraints that are still in use after rewriting, and a new simplified IR.

4 Results

In this section, we examine the results of three different file formats: 1) the banking records example used in previous sections, 2) a portion of Apple's plist format [14], and 3) the Apache web server's access log format [15].

The output from running our banking records example data (available in the Appendix) is a very simple example but is useful in highlighting various aspects of the system. The simplified IR, seen in Figure 8, shows how values marked as enumerations have been incorporated into the original format, and the list of constraints points out the integer (r12) determines the length of the array of bank accounts (r12), and the integer (r19) determines the type of bank account. The Switched constraint between title and gender

Figure 8: The simplified banking records example IR.

```
Tuple (r1)
    '20\n'
    Array (r4)
        Tuple (r5)
            Sum (r6)
                'Ms'
                'Mrs'
                'Mr'
            End Sum
            '. '
            letters (r8)
            ' '
            Sum (r10)
                'M'
                'F'
            End Sum
            ' '
            Sum (r12)
                '3'
                '2'
                '1'
            End Sum
            ' Account '
            Array (r16)
                Tuple (r17)
                    '{'
                    Sum (r19)
                        '2'
                        '1'
                    End Sum
                    ' '
                    int (r21)
                    ' '
                    int (r23)
                    ' '
                    Sum (r25)
                        Sum (r26)
                            '7'
                            '6'
                            '5'
                            '4'
                            '3'
                        End Sum
                        '' (r27)
                    End Sum
                    '}'
                End Tuple
                ...
            End Array
            '\n'
        End Tuple
        ...
    End Array
End Tuple
```

is also present. Since our example data contained
relatively few values, it makes some additional
changes that would not normally happen if more data records were available. For instance, many values are listed as Enums simply because there are not enough instances of them to prove they are less constrained. Furthermore, since it was only run on one copy of the file the initial integer that indicates how may records the file contains has been converted into a constant. These dependencies highlight the need to give the system sample data that covers all features of the format in order to prevent the generation of useless constraints in addition to the useful ones.

The second format was used to test the code on data formats used in real applications. Apple's plist text format [14], used to described the preferences of an application was selected as its XML based format would allow the system to find relationships between opening and closing tags, as well as demonstrate how the system can work on files that contain multi-line records. Since the initial IR description cannot fully explain XML in general, the format was restricted to describing the layout of a particular plist file, and nested tags were entered explicitly. The initial IR was constructed by hand out of its constituent parts directly in SML. At every place where a tag occurred, only a general description of a tag ('<' or '</' followed by some text and '>') was given, and the various types of tags in the format were enumerated. The initial IR as well as the sample file is available in the appendix.

Running the sample file through the system results in a much simpler description of the file (see Appendix for results). The system has identified that all opening tags are matched with closing tags, and that opening tags begin with '<' and not '</'. In addition, it has pruned the options in the original IR that were never used. An example of this pruning is shown in Figure 9. Several stages occurred to make this simplification possible. First, it was identified that in this part of the file only two tags—'string' and 'key'—occur, and that only one of the regular expressions matching the data inside occurs (label r54). Using this information, the other options are eliminated. The whole Tuple representing an entry is then rewritten using the Switched constraint that matches opening and closing tags, and then adjacent constant values are combined. The resulting structure is more compact and better describes the format of the data found in this particular part of the file. A similar chain of rewriting occurs for the other instance of opening and closing tags in Tuple r5, but this instance creates a much more redundant data structure since more tags are used in this part of the file, and the system chooses to use the original structure and simply report the Switched constraint between Sum r10 and Sum r28. Using the cost metric in the previous sections, the post-constraint reduction process reduces the cost of the plist IR from 304 to 142. Like the banking records example, since the test was run only on a small file, some false constraints are detected such as REBase r51 becoming a constant because this particular branch of a Sum was parsed only once.

Before

```
Tuple (r38)
    '\t\t' (r39)
    Sum (r40)
        '</' (r41)
        '<' (r42)
    End Sum
    Sum (r43)
        'key' (r44)
        'string' (r45)
        'real' (r46)
        'data' (r47)
        'integer' (r48)
    End Sum
    '>' (r49)
    Sum (r50)
         /0\\.[0-9\\.]+/ (r51)
         int (r52)
         /\\n\\t[A-Z0-9a-z\\n\\t=]+/ (r53)
         /[A-Z0-9a-z:;~\\[\\$\\^\\/\\- ]+/ (r54)
         '' (r55)
    End Sum
    Sum (r56)
        '</' (r57)
        '<' (r58)
    End Sum
    Sum (r59)
        'key' (r60)
        'string' (r61)
        'real' (r62)
        'data' (r63)
         'integer' (r64)
    End Sum
    '>\n' (r65)
End Tuple
```

After:

```
Sum (r38)
    Tuple
       '\t\t<key>'
       /[A-Z0-9a-z:;~\\[\\$\\^\\/\\- ]+/ (r54)
       '</key>\n'
    End Tuple
    Tuple
        '\t\t<string>'
        /[A-Z0-9a-z:;~\\[\\$\\^\\/\\- ]+/ (r54)
        '</string>\n'
    End Tuple
End Sum
```

Figure 9: Simplification of a portion of the plist file.

The third example was used to better understand what the system would produce on a real data format with relatively few obvious dependencies. Tests were run using lines from an Apache web server's access log file [15]. The log file lists access to a web server over a period of time, one per line, in a determined format. The results of running the test on 10,000 lines of the file are given in the Appendix. While less informative than the other examples, the constraints listed demonstrate that the prevalence of sample data will lower the number of trivial Enum constraints that resulted from having too little data. The remaining constraints show either simple properties such as the limited number of commands (e.g., GET, POST) the web server receives, or are Switched constraints that show when non-routine calls were made (e.g. using a POST, or protocol version 1.1).

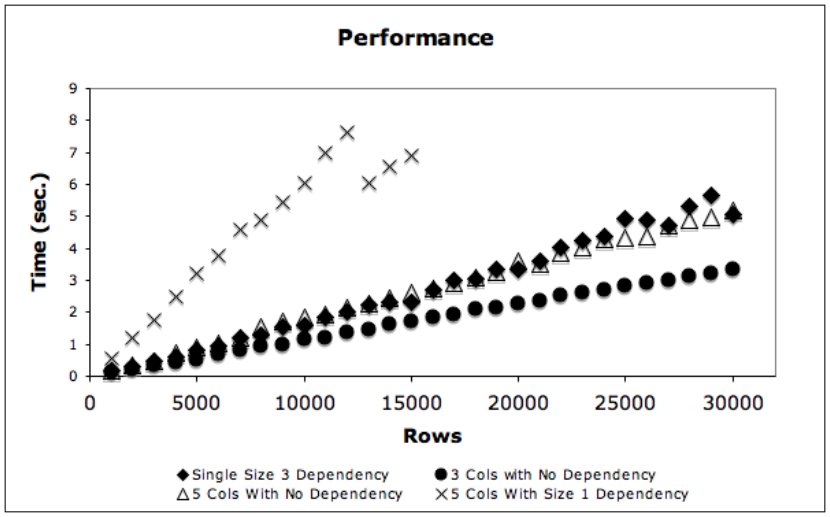
5 Performance

This system's runtime is bounded by the runtime of the TANE algorithm, which is exponential in proportion to the number of columns (i.e., fields per record), but linear in proportion to the number of rows (i.e., number of records) given that the number of dependencies it finds remains constant as row size increases [4]. Although some aspects of constraining involve $O(nlg(n))$ operations to perform set insertions, in practice the overhead of running the TANE algorithm obscures this non-linear behavior in other parts of

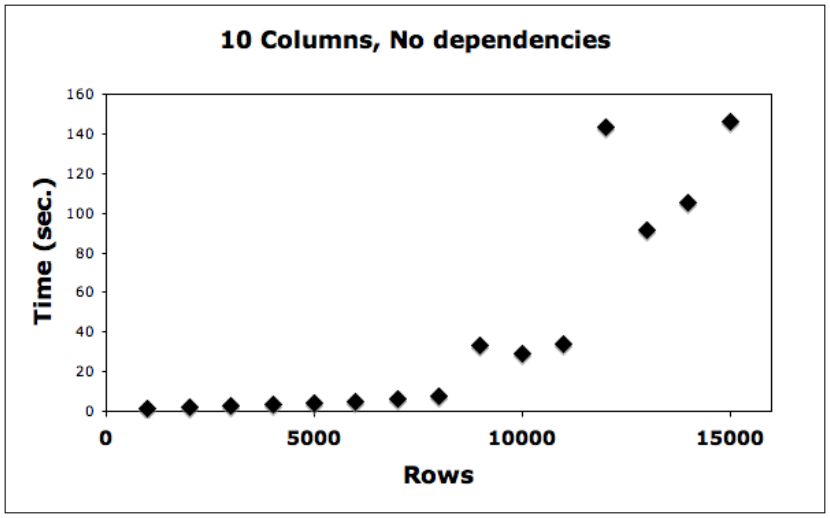
the constraint analysis. To test the performance of the system trials were run on two different formats: a randomly generated format that contains a specific number of dependencies and the Apache server log data described in the last section. All tests were performed on a MacBook Pro, with a Core 2 Duo processor, 2 GB of RAM, and SML/NJ 110.60.

When dependencies are found at early levels of the TANE algorithm, it prunes the search space of later levels. Thus, if fewer dependencies are found, the TANE algorithm takes longer to complete. Real-world data like the Apache server log file may have dependencies that hold for a few hundred lines that do not hold when more lines are added, making it difficult to assess the runtime performance purely as a function of row size. Thus, an automatically generated format was used, consisting of a table of randomly generated integers that was used to precisely control all aspects of the data. Constraints were added to this test format by creating a linear equation between n columns in the file using the general form $x = \sum_{i=1}^{n} i * c_i \mod R$, where R is the range of random numbers chosen for the columns. If R is too large, then it is likely that dependencies not explicitly crafted when generating a file would exist anyway. For instance in a table with 100 rows, if $R = 1000$ then it would be likely that a column would become a key (would all unique values) and determine every other column. In a table of M rows, C columns, and numbers ranging from 0 to $R - 1$, consider the chance that no two numbers are the same in a single column (i.e., it is a key). This is equivalent to the birthday paradox where the probability of having M people with completely unique birthdays (with R days in a year) can be approximated by $Pr[\text{every number unique}] \approx e^{\frac{-M(M-1)}{2R}}$ [16]. This can be generalized to the probability that the whole set of columns is a key by noting that the number of possible entries for a column is R^C, or $Pr[\text{row is unique}] \approx e^{\frac{-M(M-1)}{2R^C}}$. We want to choose an R that has a low probability of becoming a key and creating more dependencies than explicitly entered into the file, so we set the probability to .05 and solve for R, giving $R \approx e^{\frac{-ln(1/(M(M-1))-1.790335}{C}}$. For each trial of the algorithm R was chosen to be the nearest integer using this relation and the number of rows (M) and columns (C) in the trial.

Tests were run on files ranging in size from 1,000 rows to 30,000 rows across five different patterns of dependency: a single $\{r1, r2, r3\} \rightarrow r4$ dependency, 3 columns with no dependencies, five dependencies of the form $\{r1\} \rightarrow r2$, 5 columns with no dependencies, and 10 columns with no dependencies. The results are shown in Figure 10(a) and 10(b). The first four tests show a clearly linear pattern, while the last test is linear up to 8000 rows and then becomes non-linear. A profiling of the data points near this change in form reveals that during the linear section, approximately 29% of the time is spent doing garbage collection; after the 8000 rows, over 90% of the time is in the garbage collector. SML does not allow for very fine tuned memory management, and as a memory usage grows with column size, it reaches a point where it fills the available memory often requiring cleanup that degrades performance.



(a) Results from the randomly generated format as row size increases.

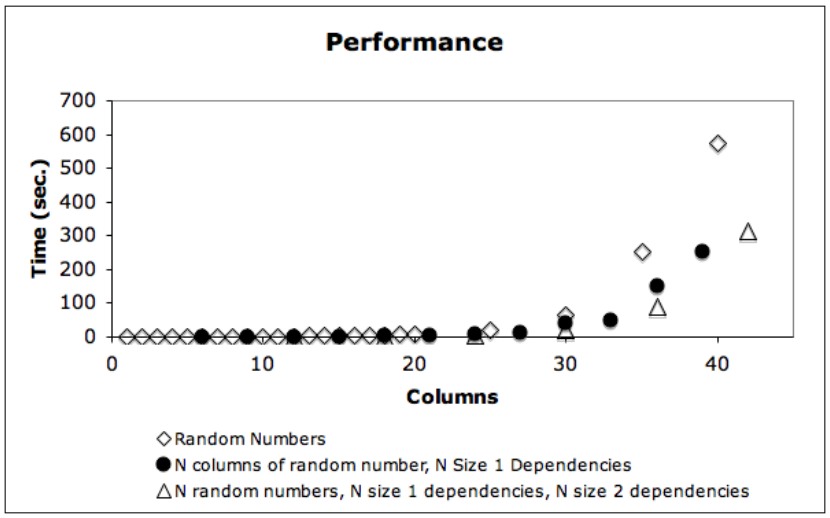


(b) An example of the non-linear behavior due to memory usage.

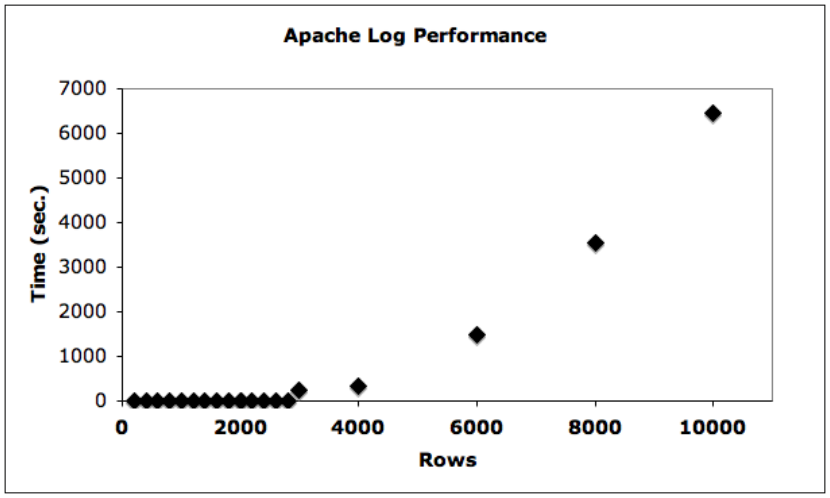
Figure 10: Row tests.

To test the performance of the overall system as the number of columns increased, the same format was used except R was set to a fixed value (1000) for every test since using the other metric would result in a very constrained choice of values (often only 0 or 1). Three variations were run, one with purely random data, another with half purely random data, and half $\{r1\} \rightarrow r2$ dependencies, and one with part random data, part $\{r1\} \rightarrow r2$ dependencies, and part $\{r1, r2\} \rightarrow r3$ dependencies. All test used a constant row size of 1000. Results are summarized in Figure 11(b), showing exponential time increase as the column size increases. As expected, the presence of dependencies decreases the overall running time of the system as pruning occurs.

To get an idea of performance on a real data set, the algorithm was run on the Apache log data. In



(a) Results from the randomly generated format as column size increases.



(b) Results from the apache log file as column size increases.

Figure 11: Column Tests and Apache Test.

table form the log data has a constant 19 columns; the number of rows was varied using the contents from a large sample log file between 100 and 10,000 rows. The results are presented in Figure 11(a). Up to approximately 2,800 rows, the performance remains linear; at 3,000 rows the performance begins to deteriorate as the limits of the memory are reached during functional dependency analysis. In this particular log file, the first time a user logs into the server instead of using it anonymously is recorded, which causes the dash that is normally a placeholder to be replace with their login name. This eliminates the $\{\} \rightarrow r1$ dependency that previously existed for the column, forcing a larger set of items to be searched at later levels, which may cause an increase in memory use leading to performance degradation. Nonetheless, the system can still complete a 10,000-line file in less than two hours of computer time.

6 Discussion

This paper has presented a new system using functional dependencies, constraint analysis, and rewriting rules to simplify and explain an intermediate representation of a data format potentially generated from an automatic inference generation algorithm. Testing on example formats has demonstrated its ability to both explain and rewrite formats in a way that can potentially help in further data analysis. While performance testing revealed certain limitations in the run-time of the system, a reimplementation of the TANE section of the code in C, using more careful memory allocation could significantly improve performance as the size of the sample file increases, likely making large data-sets feasible for analysis. Additionally, if the user does not need to know all the functional dependencies or if a portion of the format is significantly more complicated than the Apache format, the TANE part of the algorithm can be stopped at level 2, generating only dependencies of the form $\{r1\} \rightarrow r2$, which will still allow all stages of the simplification system to run, but may miss some Equation or Switched constraints.

Much application data remains in ad hoc data formats, and the success of services like Google has demonstrated the potential in understanding and using data in ways that were not originally intended. Without access to the original description of the data, researchers will benefit from the current work format inference and constraint generation that will automate the tedious stages of parsing and understanding a format. While further work can be done to improve and expand the methods of determining constraints from functional dependencies and using them to explain the data, the information obtained through this analysis will help refine and explain data formats whose authors have left incomplete or little documentation. As work on automatic format inference progresses, there will be an increasing need to understand and manipulate the machine-generated formats. This system has demonstrated that such an explanation is possible and could feasibly be run on fairly large datasets. Using the information generated through this analysis will not only ease—or even automate—the development of parsers, profilers, and query engines, it will also provide key information like functional dependencies to later stages of analysis that can use them for their own purposes in data-mining, database design, or query optimizations without wasting time discovering them independently.

References

[1] Daly, M., Mandelbaum, Y., Walker, D., Fernandez, M., and Fisher, K. (2006) Pads: An end-to-end system for processing ad hoc data. *SIGMOD*, ACM.

[2] AT&T Corporation (2006) *PADS Manual Version 1.03*.

[3] Mandelbaum, Y., Fisher, K., Walker, D., Fernandez, M., and Gleyzer, A. (2007) Pads/ml: A functional data description language. *POPL*, ACM.

[4] Huhtal, Y., Karkkainene, J., Porkka, P., and Toivonen, H. (1999) Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, **42**, 100–111.

[5] King, R. S. and Legendre, J. J. (2003) Discovery of functional and approximate functional dependencies in relational databases. *Journal of Applied Mathematics and Decision Sciences*, **7**, 49–59.

[6] Marchi, F. D., Lopes, S., Petit, J.-M., and Toumani, F. (2003) Analysis of existing databases at the logical level: the dba companion project. *SIGMOD Rec.*, **32**, 47–52.

[7] Wyss, C., Giannella, C., Robertson, E., Kambayashi, Y., Winiwarter, W., and Arikawa, M. (2001) Fastfds: A heuristic driven, depth-first algorithm for mining functional dependencies from relation instances: Extended abstract. *Lecture notes in computer science*, **2114**, 101–110.

[8] Novelli, N. and Cicchetti, R. (2001) Functional and embedded dependency inference: a data mining point of view. *Information Systems*, **26**, 477–506.

[9] Matos, V. and Grasser, B. (2004) Sql-based discovery of exact and approximate functional dependencies. *ACM SIGCSE Bulletin*, **36**, 58–63.

[10] Mannila, H. and Raiha, K.-J. (1992) On the complexity of interring functional dependencies. *Combinatorial problems in databases*, **40**, 237–243.

[11] Funahashi, K. (1989) On the approximate realization of continuous mappings by neural networks. *Neural Netw.*, **2**, 183–192.

[12] Burges, C. (1998) A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, **2**, 100–111.

[13] Blelloch, G. *The PSciCo Project*.

[14] Apple Computer (2006) *Property List Programming Topics for Core Foundation*. 3 edn.

[15] The Apache Software Foundation (2006) *Log Files - Apache HTTP Server*. 2.2 edn.

[16] Lehman, E. and Leighton, T. (2004) Mathematics for computer science, discrete Mathematics Textbook.

A Example Banking Records File

```
20
Mr. Doe M 2 Account {1 5052 50 }{2 6052 3000 5}
Mr. Smith M 1 Account {1 5999 200 }
Mr. Jones M 3 Account {1 5777 100 }{2 6777 2000 3}{2 6598 3025 3}
Mr. Gross M 1 Account {1 5623 500 }
Mrs. Green F 1 Account {2 6666 1000 3}
Ms. Porter F 2 Account {1 5888 200 }{2 6888 3000 3}
Mrs. White F 2 Account {1 5452 150 }{2 6452 4500 4}
Ms. Black F 1 Account {1 5234 1200 }
Ms. Plum F 3 Account {1 5233 200 }{2 6233 2500 4}{2 6234 15000 5}
Mr. Miller M 2 Account {2 6441 13042 4}{1 5441 201 }
Mrs. King F 3 Account {1 5775 101 }{2 6775 2022 3}{2 6770 10044 6}
Ms. Mills F 2 Account {2 6876 132440 7}{1 5440 203 }
Mr. Simon M 1 Account {1 5444 425 }
Mr. Barn M 2 Account {2 6789 13234 5}{1 5144 293 }
Mr. Leonard M 2 Account {2 6444 9876 4}{1 5254 444 }
Mrs. Freed F 3 Account {2 6001 13700 4}{1 5095 331 }{2 6004 2023 3}
Ms. Doe F 2 Account {2 6009 8020 4}{1 5005 766 }
Mr. Brown M 2 Account {1 5267 432 }{2 6290 13000 4}
Mr. Dole M 2 Account {2 6076 155000 7}{1 5074 187 }
Mrs. Robinson F 1 Account {2 6215 130000 7}
```

B Example Banking Records Parsed Data

Note: this includes only the first 2 lines of the file.

```
TupleD
    Base: '20'
    Base: '\n'
    ArrayD
    Num Entries: 20
        TupleD
            Base: 'Mr'
            Base: '. '
            Base: 'Doe'
            Base: ' '
            Base: 'M'
            Base: ' '
            Base: '2'
            Base: ' '
            Base: 'Account'
            Base: ' '
            ArrayD
            Num Entries: 2
                TupleD
                    Base: '{'
                    Base: '1'
                    Base: ' '
                    Base: '5052'
                    Base: ' '
                    Base: '50'
                    Base: ' '
                    SumD
                    Branch: 1
                        Base: ''
                    End SumD
                    Base: '}'
```

```
            End TupleD
            TupleD
                Base: '{'
                Base: '2'
                Base: ' '
                Base: '6052'
                Base: ' '
                Base: '3000'
                Base: ' '
                SumD
                Branch: 0
                    Base: '5'
                End SumD
                Base: '}'
            End TupleD
        End ArrayD
        Base: '\n'
    End TupleD
```

C Example Banking Records Result

```
Dependencies:
{} -> r14
{r6,} -> r10
{r8,} -> r12
{r8,} -> r16
{r16,} -> r12
{r12,} -> r16
{r8,r10,} -> r6
Key {r8,r10}
{r21,} -> r19
{r21,} -> r23
{r21,} -> r25
{r21,} -> r26
{r23,} -> r19
{r25,} -> r19
{r19,} -> r25
{r26,} -> r19
{r23,} -> r25
Key {r21}
{} -> r2
{} -> r4
Report

IR:
Tuple (r1)
    '20\n'
    Array (r4)
        Tuple (r5)
            Sum (r6)
                'Ms'
                'Mrs'
                'Mr'
            End Sum
            '. '
            letters (r8)
            ' '
            Sum (r10)
                'M'
                'F'
            End Sum
            ' '
            Sum (r12)
                '3'
                '2'
                '1'
            End Sum
            ' Account '
            Array (r16)
                Tuple (r17)
                    '{'
                    Sum (r19)
                        '2'
                        '1'
                    End Sum
                    ' '
                    int (r21)
                    ' '
                    int (r23)
                    ' '
                    Sum (r25)
                        Sum (r26)
                            '7'
                            '6'
                            '5'
                            '4'
                            '3'
                        End Sum
                        '' (r27)
                    End Sum
                    '}'
                End Tuple
                ...
            End Array
            '\n'
                    End Tuple
                    ...
                End Array
            End Tuple
            Constraints:
            r10:
            Switched
            (r6)
            (Mr         ) -> M
            (Mrs        ) -> F
            (Ms         ) -> F

            Length 1
            Enum:
            F
            M

            r12:
            Equation 1r16 + 0
            Range [1,3]
            Length 1
            Enum:
            1
            2
            3

            r16:
            Equation 1r12 + 0
            Range [1,3]
            Length 1
            Enum:
            1
            2
            3

            r19:
            Switched
```

```
(r25)
(r26        ) -> 2
(r27        ) -> 1

Switched
(r26)
(NONE      ) -> 1
(3          ) -> 2
(4          ) -> 2
(5          ) -> 2
(6          ) -> 2
(7          ) -> 2

Range [1,2]
Length 1
Enum:
1
2

r21:
Range [5005,6888]
Length 4

r23:
Range [50,155000]

r25:
Switched
(r19)
(1          ) -> r27
(2          ) -> r26

Length 3
Enum:
r26
r27

r26:
Range [3,7]
Length 1
Enum:
3
4
5
6
7

r4:
Range [20,20]
Length 2
Ascending
Decending
Unique: 20
Enum:


r6:
Enum:
Mr
Mrs
Ms

r8:
```

D Plist IR

```
Tuple (r1)
    '<plist version=\"1.0\">\n<dict>\n' (r2)
    Array (r3)
        Sum (r4)
            Tuple (r5)
                '\t' (r6)
                Sum (r7)
                    '</' (r8)
                    '<' (r9)
                End Sum
                Sum (r10)
                    'key' (r11)
                    'string' (r12)
                    'real' (r13)
                    'data' (r14)
                    'integer' (r15)
                End Sum
                '>' (r16)
                Sum (r17)
                    Array (r18)
                        /[0-9\\.]+ / (r19)
                        ...
                    End Array
                    /0\\.[0-9\\.]+/ (r20)
                    int (r21)
                    /\\n\\t[A-Z0-9a-z\\n\\t=]+/ (r22)
                    /[A-Z0-9a-z:;\\~\\[\\$\\^\\/\\- ]+/ (r23)
                    '' (r24)
                End Sum
                Sum (r25)
                    '</' (r26)
                    '<' (r27)
                End Sum
                Sum (r28)
```

```
                ’key’ (r29)
                ’string’ (r30)
                ’real’ (r31)
                ’data’ (r32)
                ’integer’ (r33)
            End Sum
            ’>\n’ (r34)
        End Tuple
        Tuple (r35)
            ’\t<dict>\n’ (r36)
            Array (r37)
                Tuple (r38)
                    ’\t\t’ (r39)
                    Sum (r40)
                        ’</’ (r41)
                        ’<’ (r42)
                    End Sum
                    Sum (r43)
                        ’key’ (r44)
                        ’string’ (r45)
                        ’real’ (r46)
                        ’data’ (r47)
                        ’integer’ (r48)
                    End Sum
                    ’>’ (r49)
                    Sum (r50)
                        /0\\.[0-9\\.]+/ (r51)
                        int (r52)
                        /\\n\\t[A-Z0-9a-z\\n\\t=]+/ (r53)
                        /[A-Z0-9a-z:;ˆ\\[\\$\\ˆ\\/\\- ]+/ (r54)
                        ’’ (r55)
                    End Sum
                    Sum (r56)
                        ’</’ (r57)
                        ’<’ (r58)
                    End Sum
                    Sum (r59)
                        ’key’ (r60)
                        ’string’ (r61)
                        ’real’ (r62)
                        ’data’ (r63)
                        ’integer’ (r64)
                    End Sum
                    ’>\n’ (r65)
                End Tuple
                ...
            End Array
            ’\t</dict>\n’ (r66)
        End Tuple
    End Sum
    ...
End Array
’</dict>\n</plist>\n’ (r67)
End Tuple
```

E Plist Example File

```
<plist version="1.0">
```

```
<dict>
    <key>AutoFocus</key>
    <string>YES</string>
    <key>Autowrap</key>
    <string>YES</string>
    <key>BackgroundImagePath</key>
    <string></string>
    <key>FontAntialiasing</key>
    <string>NO</string>
    <key>FontHeightSpacing</key>
    <string>1</string>
    ...
    <key>FontWidthSpacing</key>
    <string>1</string>
    <key>IsMiniaturized</key>
    <string>NO</string>
    <key>KeyBindings</key>
    <dict>
        <key>$F708</key>
        <string>[25~</string>
        <key>$F709</key>
        <string>[26~</string>
        <key>$F70A</key>
        <string>[28~</string>
        ....
            <string>scrollToBeginningOfDocument:</string>
        <key>F72B</key>
        <string>scrollToEndOfDocument:</string>
        <key>F72C</key>
        <string>scrollPageUp:</string>
        <key>F72D</key>
        <string>scrollPageDown:</string>
        ...
        <key>~F712</key>
        <string>[34~</string>
    </dict>
    <key>Meta</key>
    <string>-1</string>
    <key>NSColorPanelMode</key>
    <string>6</string>
    <key>NSColorPanelVisibleSwatchRows</key>
    <integer>1</integer>
    <key>NSFixedPitchFont</key>
    <string>Monaco</string>
    <key>NSFixedPitchFontSize</key>
    <real>10</real>
    <key>NSNavLastRootDirectory</key>
    <string>~/Library/Application Support/Terminal</string>
    <key>NSWindow Frame Inspector</key>
    <string>764 330 268 435 0 0 1440 878 </string>
    <key>NSWindow Frame NSColorPanel</key>
    <string>491 340 201 309 0 0 1440 878 </string>
    <key>OptionClickToMoveCursor</key>
    ...
    <string></string>
    <key>StrictEmulation</key>
    <string>NO</string>
    <key>StringEncoding</key>
    <string>4</string>
```

```
    <key>TermCapString</key>
    <string>xterm-color</string>
    <key>TerminalOpaqueness</key>
    <real>0.63055551052093506</real>
    <key>TextColors</key>
    <string>0.814 0.814 0.814 0.000 0.000 0.000 1.000 1.000 1.000 1.000 1.000 1.000 0.000 0.000 0.000 0.814 0.814 0
    <key>TitleBits</key>
    <string>76</string>
    <key>Translate</key>
    <string>YES</string>
    <key>UseCtrlVEscapes</key>
    <string>YES</string>
    <key>VisualBell</key>
    <string>NO</string>
    <key>WinLocULY</key>
    <string>832</string>
    <key>WinLocX</key>
    <string>25</string>
    <key>WinLocY</key>
    <string>0</string>
    <key>WindowCloseAction</key>
    <string>1</string>
</dict>
</plist>
```

F Plist Simplified IR

```
Tuple (r1)
    '<plist version=\"1.0\">\n<dict>\n'
    Array (r3)
        Sum (r4)
            Tuple (r5)
                '\t<'
                Sum (r10)
                    'key' (r11)
                    'string' (r12)
                    'real' (r13)
                    'integer' (r15)
                End Sum
                '>'
                Sum (r17)
                    Array (r18)
                        /[0-9\\.]+ / (r19)
                        ...
                    End Array
                    '0.63055551052093506'
                    int (r21)
                    /[A-Z0-9a-z:;\\~\\[\\$\\^\\/\\- ]+/ (r23)
                    '' (r24)
                End Sum
                '</'
                Sum (r28)
                    'key' (r29)
                    'string' (r30)
                    'real' (r31)
                    'integer' (r33)
                End Sum
                '>\n'
```

```
            End Tuple
            Tuple (r35)
                '\t<dict>\n'
                Array (r37)
                    Sum (r38)
                        Tuple
                            '\t\t<key>'
                            /[A-Z0-9a-z:;~\\[\\$\\^\\/\\- ]+/ (r54)
                            '</key>\n'
                        End Tuple
                        Tuple
                            '\t\t<string>'
                            /[A-Z0-9a-z:;~\\[\\$\\^\\/\\- ]+/ (r54)
                            '</string>\n'
                        End Tuple
                    End Sum
                    ...
                End Array
                '\t</dict>\n'
            End Tuple
        End Sum
        ...
    End Array
    '</dict>\n</plist>\n'
End Tuple
```

G Apache Log IR

```
Array (r1)
    Tuple (r2)
        Sum (r3)
            /[0-9]+\\.[0-9]+\\.[0-9]+\\.[0-9]+/ (r4)
            /[a-zA-Z0-9\\.\\-]+/ (r5)
        End Sum
        ' - ' (r6)
        Sum (r7)
            /[a-z]+\\@[a-z]+\\.[a-z]+/ (r8)
            letters (r9)
            '-' (r10)
            '\"\"' (r11)
        End Sum
        ' [' (r12)
        int (r13)
        '/' (r14)
        letters (r15)
        '/' (r16)
        int (r17)
        ':' (r18)
        int (r19)
        ':' (r20)
        int (r21)
        ':' (r22)
        int (r23)
        ' -' (r24)
        int (r25)
        '] \"' (r26)
        letters (r27)
        ' ' (r28)
```

```
        /[a-zA-Z0-9/\\.@_\\-&\\*%=';:\\\\]+/ (r29)
        ' HTTP/1.' (r30)
        int (r31)
        '\" ' (r32)
        int (r33)
        ' ' (r34)
        Sum (r35)
            int (r36)
            '-' (r37)
        End Sum
        '\n' (r38)
    End Tuple
    ...
End Array
```

H Apache Example File

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /turkey/amnty1.gif HTTP/1.0" 200 3013
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /turkey/clear.gif HTTP/1.0" 200 76
207.136.97.49 - - [15/Oct/1997:18:46:52 -0700] "GET /turkey/back.gif HTTP/1.0" 200 224
207.136.97.49 - - [15/Oct/1997:18:46:52 -0700] "GET /turkey/women.html HTTP/1.0" 200 17534
208.196.124.26 - - [15/Oct/1997:18:46:55 -0700] "GET /candatop.html HTTP/1.0" 200 10629
208.196.124.26 - - [15/Oct/1997:18:46:57 -0700] "GET /images/done.gif HTTP/1.0" 200 4785
208.196.124.26 - - [15/Oct/1997:18:47:01 -0700] "GET /images/reddash2.gif HTTP/1.0" 200 237
208.196.124.26 - - [15/Oct/1997:18:47:02 -0700] "GET /images/refrun1.gif HTTP/1.0" 200 836
208.196.124.26 - - [15/Oct/1997:18:47:05 -0700] "GET /images/hasene2.gif HTTP/1.0" 200 8833
208.196.124.26 - - [15/Oct/1997:18:47:08 -0700] "GET /images/candalog.gif HTTP/1.0" 200 8638
```

I Apache Results — 10,000 lines

```
Dependencies:
{} -> r8
{} -> r15
{} -> r17
{} -> r25
{r29,} -> r27
{r3,r36,} -> r27
{r4,r36,} -> r27
{r5,r36,} -> r27
{r23,r29,} -> r9
{r29,r33,} -> r9
{r29,r36,} -> r9
{r23,r36,} -> r27
{r29,r36,} -> r33
{r4,r5,r21,} -> r31
{r5,r23,r29,} -> r7
{r5,r23,r36,} -> r7
{r5,r21,r36,} -> r9
{r5,r23,r36,} -> r9
{r5,r19,r21,} -> r31
{r19,r23,r29,} -> r7
{r21,r23,r29,} -> r7
{r21,r23,r36,} -> r7
{r19,r21,r36,} -> r9
{r19,r23,r36,} -> r9
{r21,r23,r36,} -> r9
{r19,r21,r36,} -> r27
{r19,r33,r36,} -> r27
{r4,r5,r19,r23,} -> r13
{r4,r5,r21,r23,} -> r13
{r4,r5,r23,r29,} -> r13
{r4,r5,r21,r29,} -> r19
{r4,r5,r23,r29,} -> r31
{r4,r5,r23,r36,} -> r31
{r5,r19,r23,r29,} -> r31
{r5,r19,r23,r36,} -> r31
{r19,r21,r23,r29,} -> r13
{r19,r21,r23,r36,} -> r13
{r19,r21,r23,r36,} -> r31
{r19,r23,r29,r33,} -> r35
{r21,r23,r29,r33,} -> r35
{r3,r19,r21,r23,r29,} -> r4
{r3,r19,r21,r29,r33,} -> r31
{r3,r19,r21,r29,r35,} -> r31
{r3,r19,r21,r29,r36,} -> r31
{r5,r19,r21,r23,r29,} -> r4
{r4,r5,r21,r23,r36,} -> r19
{r4,r5,r21,r23,r33,} -> r35
{r4,r5,r23,r29,r33,} -> r35
{r4,r19,r21,r23,r33,} -> r13
{r4,r19,r21,r29,r33,} -> r31
{r4,r19,r21,r29,r35,} -> r31
{r4,r19,r21,r29,r36,} -> r31
{r4,r21,r23,r29,r36,} -> r31
{r5,r13,r21,r23,r33,} -> r35
{r5,r19,r21,r23,r33,} -> r35
{r5,r21,r23,r29,r35,} -> r33
{r5,r21,r23,r29,r33,} -> r36
{r5,r21,r23,r29,r35,} -> r36
{r19,r21,r23,r29,r33,} -> r31
{r19,r21,r23,r29,r35,} -> r31
{r19,r21,r23,r29,r35,} -> r33
{r19,r21,r23,r29,r33,} -> r36
{r19,r21,r23,r29,r35,} -> r36
{r3,r19,r21,r23,r29,r33,} -> r5
{r3,r19,r21,r23,r29,r35,} -> r5
{r3,r19,r21,r23,r29,r36,} -> r5
{r3,r21,r23,r29,r31,r33,} -> r36
{r4,r19,r21,r23,r29,r33,} -> r5
{r4,r19,r21,r23,r29,r35,} -> r5
{r4,r19,r21,r23,r29,r36,} -> r5
{r4,r21,r23,r29,r31,r33,} -> r36
{r13,r21,r23,r29,r31,r33,} -> r36

Report

IR:
Array (r1)
    Tuple (r2)
        Sum (r3)
            /[0-9]+\\.[0-9]+\\.[0-9]+\\.[0-9]+/ (r4)
            /[a-zA-Z0-9\\.\\-]+/ (r5)
        End Sum
        ' - '
        Sum (r7)
            'amnesty'
            '-' (r10)
            '\"\"' (r11)
        End Sum
        ' ['
        Sum (r13)
            '17'
            '16'
            '15'
        End Sum
        '/Oct/1997:'
        int (r19)
        ':'
        int (r21)
        ':'
```

```
        int (r23)
        ' -0700] \"'
        Sum (r27)
            'POST'
            'GET'
        End Sum
        ' '
        /[a-zA-Z0-9/\\.@_\\-&\\*%=';:\\\\]+/ (r29)
        ' HTTP/1.'
        Sum (r31)
            '1'
            '0'
        End Sum
        '\" '
        int (r33)
        ' '
        Sum (r35)
            int (r36)
            '-' (r37)
        End Sum
        '\n'
    End Tuple
    ...
End Array

Constraints

r1:
Range [10000,10000]
Length 5
Ascending
Decending
Unique: 10000
Enum:


r13:
Range [15,17]
Length 2
Ascending
Enum:
15
16
17

r19:
Range [0,23]
Length 2

r21:
Range [0,59]
Length 2

r23:
Range [0,59]
Length 2

r27:
Switched
(r29)
(*          ) -> GET
(/_vti_bin/shtml.exe/_vti_rpc) -> POST
(/scripts/mailform/aimember@aiusa.org/confirm) -> POST
(/scripts/mailform/edorsey@aiusa.org/confirm) -> POST
(/scripts/mailform/sharrison@igc.apc.org/confirm) -> POST
(/whatsnew.html/_vti_bin/shtml.exe/_vti_rpc) -> POST

Switched
(r3 r36)
(*          ) -> GET
(r4         553        ) -> POST
(r5         180        ) -> POST
(r5         194        ) -> POST
(r5         941        ) -> POST
(r5         2390       ) -> POST

Switched
(r4 r36)
(*          ) -> GET
(NONE       180        ) -> POST
(NONE       194        ) -> POST
(NONE       941        ) -> POST
(NONE       2390       ) -> POST
(148.4.61.168553        ) -> POST

Switched
(r5 r36)
(*          ) -> GET
(NONE       553        ) -> POST
(grommit.jj.rhno.columbia.edu2390       ) -> POST
(piweba1y-ext.prodigy.com941        ) -> POST
(tej109.market1.com180        ) -> POST
(tej109.market1.com194        ) -> POST
(ww-tj62.proxy.aol.com941        ) -> POST

Switched
(r23 r36)
(*          ) -> GET
(00         180        ) -> POST
(02         180        ) -> POST
(03         180        ) -> POST
(17         2390       ) -> POST
(22         941        ) -> POST
(45         941        ) -> POST
(48         553        ) -> POST
(51         194        ) -> POST

Switched
(r19 r21 r36)
(*          ) -> GET
(14         32         941        ) -> POST
(16         23         194        ) -> POST
(16         24         180        ) -> POST
(16         49         2390       ) -> POST
(18         51         553        ) -> POST
(21         39         941        ) -> POST

Switched
(r19 r33 r36)
(*          ) -> GET
(14         200        941        ) -> POST
(16         200        2390       ) -> POST
(16         404        180        ) -> POST
(16         404        194        ) -> POST
(18         200        553        ) -> POST
(21         200        941        ) -> POST

Enum:
GET
POST

r29:

r3:
Length 2
Enum:
r4
r5

r31:
Switched
(r4 r5 r21)
(*          ) -> 0
(NONE       1cust40.max6.santa-clara.ca.ms.uu.net07         ) -> 1
(NONE       1cust40.max6.santa-clara.ca.ms.uu.net08         ) -> 1
(NONE       62.new-york-18.ny.dial-access.att.net41         ) -> 1
(NONE       62.new-york-18.ny.dial-access.att.net42         ) -> 1
(NONE       62.new-york-18.ny.dial-access.att.net43         ) -> 1
(NONE       62.new-york-18.ny.dial-access.att.net44         ) -> 1
(NONE       62.new-york-18.ny.dial-access.att.net45         ) -> 1
(NONE       client901.shizuokanet.or.jp29         ) -> 1
(NONE       col-oh24-18.ix.netcom.com21         ) -> 1
(NONE       cyber1c22d142.cal.shaw.wave.ca19         ) -> 1
(NONE       ns3-08.viptx.net32         ) -> 1
(NONE       ns3-08.viptx.net33         ) -> 1
(NONE       ns3-08.viptx.net35         ) -> 1
(NONE       ns3-08.viptx.net36         ) -> 1
(NONE       ppp-pts43.nss.udel.edu29         ) -> 1
(NONE       ppp-pts43.nss.udel.edu30         ) -> 1
(NONE       ras5wb24.cfw.com48         ) -> 1
(NONE       tej109.market1.com19         ) -> 1
(NONE       tej109.market1.com27         ) -> 1
(NONE       tej109.market1.com28         ) -> 1
(NONE       tej109.market1.com29         ) -> 1
(NONE       tej109.market1.com31         ) -> 1
(NONE       tej109.market1.com32         ) -> 1
(204.192.112.178NONE       00         ) -> 1
(204.192.112.178NONE       01         ) -> 1
(204.192.112.178NONE       02         ) -> 1
(204.192.112.178NONE       03         ) -> 1
(204.192.112.178NONE       45         ) -> 1
(204.192.112.178NONE       58         ) -> 1
(204.192.112.178NONE       59         ) -> 1
(206.102.3.81NONE       02         ) -> 1

Switched
(r5 r19 r21)
(*          ) -> 0
(NONE       17         02         ) -> 1
(NONE       17         45         ) -> 1
(NONE       22         58         ) -> 1
(NONE       22         59         ) -> 1
(NONE       23         00         ) -> 1
(NONE       23         01         ) -> 1
(NONE       23         02         ) -> 1
(NONE       23         03         ) -> 1
(1cust40.max6.santa-clara.ca.ms.uu.net02         07         ) -> 1
(1cust40.max6.santa-clara.ca.ms.uu.net02         08         ) -> 1
(62.new-york-18.ny.dial-access.att.net19         41         ) -> 1
(62.new-york-18.ny.dial-access.att.net19         42         ) -> 1
(62.new-york-18.ny.dial-access.att.net19         43         ) -> 1
(62.new-york-18.ny.dial-access.att.net19         44         ) -> 1
(62.new-york-18.ny.dial-access.att.net19         45         ) -> 1
(client901.shizuokanet.or.jp09         29         ) -> 1
(col-oh24-18.ix.netcom.com20         21         ) -> 1
(cyber1c22d142.cal.shaw.wave.ca10         19         ) -> 1
(ns3-08.viptx.net12         32         ) -> 1
(ns3-08.viptx.net12         33         ) -> 1
(ns3-08.viptx.net12         35         ) -> 1
(ns3-08.viptx.net12         36         ) -> 1
(ppp-pts43.nss.udel.edu16         29         ) -> 1
(ppp-pts43.nss.udel.edu16         30         ) -> 1
(ras5wb24.cfw.com18         48         ) -> 1
(tej109.market1.com16         19         ) -> 1
(tej109.market1.com16         27         ) -> 1
(tej109.market1.com16         28         ) -> 1
(tej109.market1.com16         29         ) -> 1
(tej109.market1.com16         31         ) -> 1
(tej109.market1.com16         32         ) -> 1

Range [0,1]
Length 1
Enum:
0
```

```
1

r33:
Range [200,404]
Length 3
Enum:
200
206
301
304
401
404

r35:
Length 3
Enum:
r36
r37

r36:
Range [35,104301]

r4:

r5:

r7:
Enum:
r10
r11
r9
```