Fast Foreign-Key Detection in Microsoft SQL Server PowerPivot for Excel

Zhimin Chen
Microsoft Research
zmchen@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

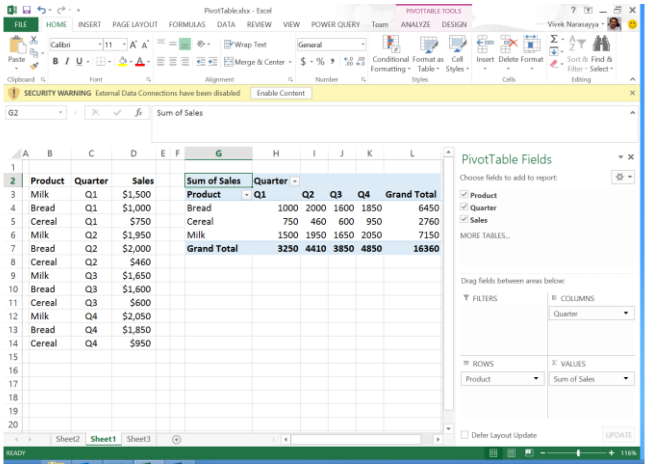
Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Microsoft SQL Server PowerPivot for Excel, or PowerPivot for short, is an in-memory business intelligence (BI) engine that enables Excel users to interactively create pivot tables over large data sets imported from sources such as relational databases, text files and web data feeds. Unlike traditional pivot tables in Excel that are defined on a single table, PowerPivot allows analysis over multiple tables connected via foreign-key joins. In many cases however, these foreign-key relationships are not known a priori, and information workers are often not be sophisticated enough to define these relationships. Therefore, the ability to automatically discover foreign-key relationships in PowerPivot is valuable, if not essential. The key challenge is to perform this detection *interactively* and with *high precision* even when data sets scale to hundreds of millions of rows and the schema contains tens of tables and hundreds of columns. In this paper, we describe techniques for fast foreign-key detection in PowerPivot and experimentally evaluate its accuracy, performance and scale on both synthetic benchmarks and real-world data sets. These techniques have been incorporated into PowerPivot for Excel.

1. INTRODUCTION

Pivot tables in Excel are a powerful tool for multi-dimensional data summarization and analysis, and have been extremely popular with information workers for many years. Among other functions, a pivot table can automatically sort, filter, group-by and compute common aggregate functions such as count, sum, etc. over columns in a single table or tabular range of data values in Excel. For example, Figure 1 shows a screenshot of a pivot table in Excel. Using a pivot table makes it simple for the user to analyze *Sales* – one of the columns in the raw data table on left – by different attributes *Product* and *Quarter*. The pivot table shows the total *Sales* grouped by *Product* along the rows and by *Quarter* along the column, and thereby provides a multi-dimensional view of the data to the user. A pivot table in Excel can be viewed as an instance of OLAP over a single table [1]. One of the common sources of data used by information workers for analysis via pivot tables is data stored in a relational database, which they can import into Excel. Other sources of data are text files, web data feeds or in general any tabular data range imported into Excel.



Sum of Sales

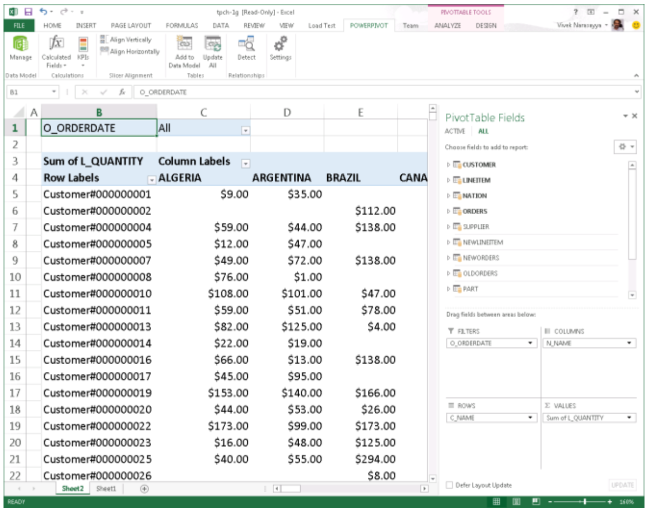
Product	Quarter	Sales
Milk	Q1	$1,500
Bread	Q1	$1,000
Cereal	Q1	$750
Milk	Q2	$1,950
Bread	Q2	$2,000
Cereal	Q2	$460
Milk	Q3	$1,650
Bread	Q3	$1,600
Cereal	Q3	$600
Milk	Q4	$2,050
Bread	Q4	$1,850
Cereal	Q4	$950

Sum of Sales	Quarter				
Product	Q1	Q2	Q3	Q4	Grand Total
Bread	1000	2000	1600	1850	6450
Cereal	750	460	600	950	2760
Milk	1500	1950	1650	2050	7150
Grand Total	**3250**	**4410**	**3850**	**4850**	**16360**

Figure 1. Example of pivot table in Excel. It enables multi-dimensional analysis over a single table.

Microsoft SQL Server PowerPivot for Excel [2] (or PowerPivot for short) is an in-memory, self-service business intelligence (BI) product first released in Microsoft SQL Server 2008 R2 and is an Excel Add-In. It enhances the traditional pivot table functionality in Excel in two very significant ways. First, unlike traditional pivot tables in Excel, which only operate on a *single* table of data in Excel, PowerPivot extends pivot table functionality by allowing it to be specified over *foreign-key joins of multiple tables*. Logically, the semantics of a pivot table in PowerPivot is equivalent to a traditional pivot table, but over a de-normalized relation obtained by joining the tables using foreign-key joins. Second, PowerPivot is designed to scale to very large data sets (e.g. hundreds of millions of rows) while retaining interactivity. Figure 2 shows a screenshot of a pivot table in PowerPivot over the TPC-H benchmark [17] database. Observe that in this case the user has selected (in the right hand pane) the *lineitem.l_quantity* field to aggregate, the *customer.c_name* and *nation.n_name* fields to group-by (that appear in rows and columns respectively of the pivot table), and the *orders.o_orderdate* field to filter on. This pivot table is generated over the foreign-key join of *lineitem*, *orders*, *customer*, *supplier* and *nation* tables. Observe that even though the *supplier* table is not explicitly selected by the user as one of the pivot table fields, this table is necessary in the joins to connect the *nation* table with the *lineitem* table.

Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08



B1 O_ORDERDATE

	A	B	C	D	E
1		O_ORDERDATE	All		
2					
3		Sum of L_QUANTITY	Column Labels		
4		Row Labels	ALGERIA	ARGENTINA	BRAZIL	CANA
5		Customer#000000001	$9.00	$35.00	
6		Customer#000000002			$112.00
7		Customer#000000004	$59.00	$44.00	$138.00
8		Customer#000000005	$12.00	$47.00	
9		Customer#000000007	$49.00	$72.00	$138.00
10		Customer#000000008	$76.00	$1.00	
11		Customer#000000010	$108.00	$101.00	$47.00
12		Customer#000000011	$59.00	$51.00	$78.00
13		Customer#000000013	$82.00	$125.00	$4.00
14		Customer#000000014	$22.00	$19.00	
15		Customer#000000016	$66.00	$13.00	$138.00
16		Customer#000000017	$45.00	$95.00	
17		Customer#000000019	$153.00	$140.00	$166.00
18		Customer#000000020	$44.00	$53.00	$26.00
19		Customer#000000022	$173.00	$99.00	$173.00
20		Customer#000000023	$16.00	$48.00	$125.00
21		Customer#000000025	$40.00	$55.00	$294.00
22		Customer#000000026			$8.00

PivotTable Fields
ACTIVE ALL
Choose fields to add to report:
- CUSTOMER
- LINEITEM
- NATION
- ORDERS
- SUPPLIER
- NEWLINEITEM
- NEWORDERS
- OLDORDERS
- PART

Drag fields between areas below:

FILTERS: O_ORDERDATE
COLUMNS: N_NAME
ROWS: C_NAME
VALUES: Sum of L_QUANTITY

□ Defer Layout Update UPDATE

Figure 2. Example of pivot table in PowerPivot for Excel. It enables OLAP like analysis over *multiple tables* joined via foreign-keys.

Data that is analyzed in PowerPivot is organized as a collection of tables with foreign-key relationships between the tables. Common examples of schemas used are *star schemas* and *snowflake schemas* [1]. In a star schema there is a Fact table that contains details (e.g. sales transactions) and is connected to other Dimension tables (e.g. Customer, Country, Year, Store) via foreign-key relationships. A snowflake schema is similar to a star schema except that the Dimension tables are organized in hierarchies. For example, the Time dimension could have multiple levels in the hierarchy such as Date, Month and Year. Figure 3 shows an example of a snowflake schema.

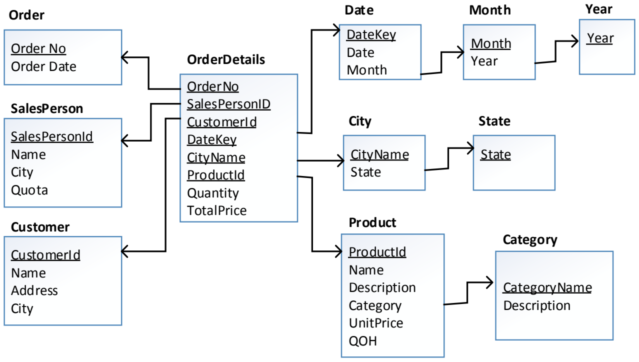


Figure 3. Example of snowflake schema. The arrows represent foreign-key relationships.

Similar to the case of traditional pivot tables, users often load data from relational databases into PowerPivot for analysis. In some cases when the user loads data into Excel, the foreign-key relationships among the loaded tables is already known e.g. the tables were loaded from a database in which the foreign-key relationships were declared. However, in other situations, these relationships are not known at data load time. A common reason is that foreign-keys may not declared in the database, typically driven by performance considerations, and referential integrity is enforced in the application layer instead. If a user attempts to create a pivot table involving multiple tables with missing foreign-key relationships, PowerPivot alerts the user about the need to create relationships and provides the option of detecting relationships *automatically*. If the user chooses this option, PowerPivot attempts to automatically discover the relationships with a *high degree of precision* and creates them automatically on behalf of the user. This functionality is highly desirable (if not crucial) since pivot table users in Excel are typically not well-versed in defining relationships across tables. Observe that if the user proceeds to create the pivot table *without* having defined the relationships, the aggregates would be computed on the cross-product of the tables, which would most likely produce unintended results. More sophisticated users who understand foreign-key relationships have the option of reviewing the automatically created relationships and changing them if needed.

In this paper, we describe techniques for automatically discovering foreign-key relationships in PowerPivot. The PowerPivot setting brings new requirements and associated challenges when compared to the foreign-key detection problem in relational databases, which we illustrate with the following example. Consider a typical use case where the user attempts to create a pivot table by selecting a few measures from one or more "fact" tables to aggregate e.g. *Quantity* and *TotalPrice* columns from the *OrderDetails* tables in Figure 3, and a few columns from the "dimension" tables, e.g. *Name* column in the *Customer* table and the *CategoryName* column in the Category table. Suppose there are no declared relationships in the database. Then PowerPivot will attempt to detect and automatically create relationships between the tables selected for analysis (*OrderDetails*, *Customer*, *Category* in this example). There are two major challenges: accuracy and performance that we discuss next.

First, since analysis in PowerPivot is designed to be interactive, relationship detection needs to be performed at *interactive speeds* as well. This can be challenging since both data volumes and schema (i.e. number of tables and columns) can be large. Observe also that the foreign key relationships necessary to perform the join may involve tables not selected by the user (e.g. the *Product* table in Figure 3); so the search space potentially includes candidates over all tables/columns even when the user selects only a few columns in the pivot table. Furthermore, the need for high precision requires that we do not output a foreign-key (e.g. $R.a \rightarrow S.b$) unless it passes the *necessary conditions*: *S.b* must be unique (*uniqueness check*), and *R.a* must be contained in *S.b (containment check)*. The most expensive necessary condition that needs to be verified is the containment check since it requires a full semi-join between *R.a* and *S.b*. Thus, any approach for automatic foreign-key detection in PowerPivot must attempt to minimize the number of such full semi-joins invocations. We note that the performance overhead of detecting a relationship is incurred once, and is amortized over all successive analysis involving that relationship.

Second, the high precision bar in the PowerPivot setting is crucial since (a) an incorrect foreign-key could result in a wrong join (and hence answer) in the pivot table, and (b) a typical pivot table user in Excel is likely not capable of identifying or correcting errors in the discovered foreign-keys. Note that testing for the necessary conditions described above is often insufficient to eliminate all the spurious alternatives. An example of this is the "id problem", where a number of different id columns in different dimension tables appear as plausible targets for a given id column in a fact table. For example, the values of the column *OrderDetails.ProductId* might be contained in each of the following key columns: *SalesPerson.SalesPersonId*, *Customer.CustomerId*, *Product.ProductId*. Hence each combination of *OrderDetails.ProductId* with one of the key columns, may appear as a potential foreign-key but only one of those relationships is a true foreign-key. Hence other signals (beyond uniqueness and containment properties) such as table and column names are crucial in differentiating among the candidates.

We use two key ideas to address these challenges. First, we combine different signals including column data characteristics, string similarity between table and column names and relative strength of peer candidate foreign-keys to achieve high precision. Our combination scoring function is effective in distinguishing true foreign-key candidates from spurious candidates. To achieve the interactive speed requirement we explore the join paths between the tables selected by the user for creating the pivot table by exploiting the ranked ordering of candidates, and develop lightweight tests based on *random sample based probing (*i.e. using a random sample of *R.a* to probe *S.b*) and *min-hashing* [5] to efficiently prune candidates that are unlikely to pass the containment check. Using the above ideas, in most cases we are able to terminate the search for foreign-keys without exploring many irrelevant candidates and we typically execute very few unnecessary full semi-joins.

The techniques we describe in this paper have been incorporated in PowerPivot for Excel Add-In and is available in all releases starting in 2008. We include a set of experiments run on PowerPivot for Excel 2013. We report results on both synthetic data sets (such as the TPC-H benchmark [17], TPC-E benchmark [19], and the AdventureWorksDW database [4]) as well as several real-world data sets that we used to evaluate the quality (i.e. accuracy), performance and scale of the techniques. The results show that our techniques are able to achieve both requirements: good performance and high precision on data sets with large schemas (tens of tables and hundreds of columns, e.g. in the AdventureWorksDW database), and large data volumes (hundreds of millions of rows, e.g. as in the TPC-H 10GB database).

We present a formal problem statement and overview of solution in Section 2, describe the techniques in Section 3 and report the results of experiments in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. PROBLEM STATEMENT AND SOLUTION OVERVIEW

We first provide a brief overview of Microsoft SQL Server PowerPivot for Excel (Section 2.1). We then give a formal statement of the foreign-key detection problem (Section 2.2) and present an overview of the solution (Section 2.3).

2.1 Microsoft SQL Server PowerPivot for Excel

Microsoft SQL Server PowerPivot for Excel [2] is a self-service business intelligence (BI) product from Microsoft SQL Server. It includes an Add-In to Microsoft Excel that allows users to create sophisticated and high performance BI solutions. Users can perform ad-hoc modeling and analysis over large amounts of data using familiar Excel operations such as filtering, sorting, aggregation and in particular using pivot tables etc. Unlike traditional pivot tables in Excel, which only operate on a *single* table of data in Excel, PowerPivot extends traditional pivot table functionality by allowing it to be specified over *foreign-key joins* of multiple tables. Data processing in PowerPivot is supported through the Microsoft SQL Server Analysis Services xVelocity in-memory analytics engine (earlier named VertiPaq) that runs *in-process* inside Excel. This engine is designed to scale to large data sets by taking advantage of large main memories and multiple cores available on modern 64-bit architectures. It performs several optimizations such as column-oriented representation in memory, aggressive compression, and efficient operations on compressed data to enable interactive analysis on very large data sets. Finally, since PowerPivot is designed to be self-service, it is intended to allow users (information workers) with no specialized BI or analytics training to develop data models and perform analysis.

2.2 Formal Problem Statement

The problem formulation mimics the user interaction model in PowerPivot (shown in Figure 2). To create a pivot table, the user needs to select one or more measures: numeric columns over which aggregation is to be performed. The selected measures define a set F of "fact" tables for the particular pivot table. Likewise, the user can also select a set of columns to group-by or filter in the pivot table – these columns define a set D of dimension tables. In this paper we use the notation $R.a \rightarrow S.b$ to denote a candidate foreign-key, where $S.b$ is the parent (or key-side) and $R.a$ is the child (or foreign-key side). We formally define the problem as follows:

Input: (i) A set of tables V and optionally a set of edges E that are *declared* foreign-keys. (ii) A set of fact tables $F \subseteq V$, a set of tables $D \subseteq V$ as dimension tables. F and D are identified by the user by attempting to create the pivot table. (iii) For each column, the metadata about the columns: table name, column name, number of distinct values.

Output: A set of foreign-keys E' (if any) such that for each pair tables of (f,d) where $f \in F$ and $d \in D$, there is a path connecting f to d using edges from $E \cup E'$.

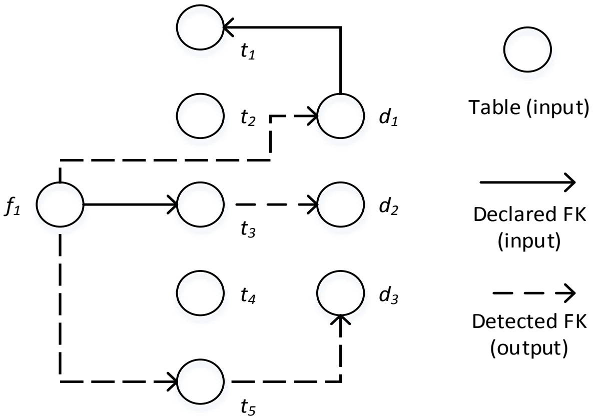


Figure 4. $F = \{f_1\}$ is a set of fact tables, and $D = \{d_1, d_2, d_3\}$ is a set of dimension tables selected by the user attempting to create a pivot table. The solid edges represent E, the declared foreign-key relationships. The dashed lines represent the foreign key relationships detected (output). Observe that the join paths between F and D may include other tables not in F or D, t_3 and t_5 in this example; and that some of the declared foreign-keys might be irrelevant for this pivot table (e.g. edge from d_1 to t_1).

We note that PowerPivot imposes some restrictions on the foreign-key relationships allowed [18]. For example, in Microsoft SQL Server 2008 and 2008 R2 version, only single-column foreign-keys are allowed, only one FK relationship can exist between a given pair of tables etc. Although the above problem formulation is general, in our solution we exploit such restrictions to reduce the search space. Finally, we note that PowerPivot does allow the join even if a small number of values (this number is configurable) in *R.a* do *not* occur in *S.b*. Therefore, any solution must allow for some slack in the containment requirement (based on configured value) since otherwise some true foreign-keys may go undetected.

2.3 Solution Overview

As explained above, the goal of foreign-key detection in PowerPivot is to find a set of foreign-key relationships that connects the given set of fact (*F*) and dimension tables (*D*) in the pivot table that the user is creating. The two *necessary* conditions for a candidate *R.a* → *S.b* to be a foreign-key are: (i) *S.b* is unique and (ii) $R.a \subseteq S.b$, i.e. *R.a* is contained in *S.b*. A naïve approach that relies exclusively on checking these necessary conditions for each possible candidate falls short on two accounts. First, it does not scale for large data sets or schemas since: (a) The containment check requires a semi-join of *R.a* and *S.b* and can potentially be very expensive. (b) The number of candidates grows quadratically with the number of columns in the database. Second, there are often many candidates that satisfy the necessary conditions but are spurious, i.e. false positives. Thus high precision demands additional techniques for distinguishing among the candidates.

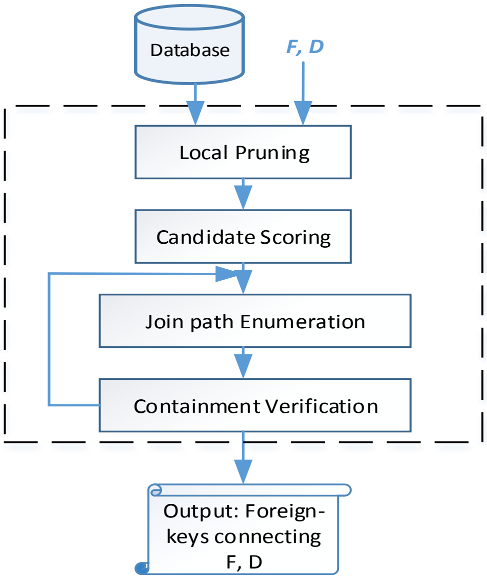


Figure 5. Foreign-key detection components in PowerPivot.

In order to meet the *high precision* and *interactive speeds* requirements for foreign-key detection in PowerPivot, we adopt the following approach (Figure 5 shows the components). First, we develop a set of pruning techniques that allows us to eliminate a large fraction of the candidates using very lightweight methods – in particular these techniques only rely on table or column metadata available in PowerPivot's catalogs (accessible at negligible performance cost). We refer to this as *Local Pruning* (Section 3.1). For each candidate that survives this step, we compute a score that attempts to identify how likely this candidate is to be a true foreign-key. In subsequent steps, we examine candidates in ranked order of scores. Intuitively, an ideal scoring function would allow the search to end by evaluating the full-semi join check for containment only on the true foreign keys. We refer to this step as *Candidate Scoring* (Section 3.2). The goal of the *Join Path Enumeration* step (Section 3.3) is to identify a minimal graph consisting of high ranking candidate foreign-keys that connects the tables in *F* and *D*. Intuitively, if the candidates in this minimal graph pass the subsequent containment verification, then we would output these edges as our recommended foreign-keys. The final step, which we refer to as *Containment Verification* (Section 3.4) is responsible for determining if the edges in the graph identified during join path enumeration meet the containment check or not. This is typically the most expensive check and we therefore develop additional optimizations based on random sampling and min-hashing that can eliminate some candidates and thereby avoid the full semi-join. If one edge in the graph fails this step, then we return to join path enumeration and continue with additional candidates in ranked order of scores; and repeat the process until either all candidates in the graph pass the verification step or no more candidates exist.

3. SOLUTION

3.1 Local Pruning

Initially, the set of candidates includes all pairs of columns (*R.a*, *S.b*) where *R* and *S* are any two distinct tables in the database and *a* and *b* are two columns from *R* and *S* respectively. Observe that a foreign-key is directional so each pair of columns defines two candidates (i.e. *R.a*→*S.b* and *S.b* →*R.a*). Local pruning techniques need to be very lightweight since the initial set of candidates can be very large – e.g., it is not unusual to have tens of thousands of candidates in the initial set. Therefore, we restrict local pruning logic to only rely on *metadata* properties of the pair of columns (*R.a* and *S.b*) such as data type and cardinality that are already available in the system catalogs. Below we list the local pruning rules and provide a brief justification for each. Each rule specifies a condition under which a candidate *R.a*→*S.b* is *pruned*.

$$\boldsymbol{LP_1}\text{:}\quad \frac{|S.b|}{|S|} < 1$$

Prune candidate if the ratio of *S.b*'s cardinality to table *S*'s cardinality is less than 1. This rule directly follows from the uniqueness requirement on *S.b*.

$$\boldsymbol{LP_2}\text{:}\quad \frac{|R.a|}{|S.b|} > 1+\varepsilon$$

Prune candidate if the ratio of cardinality of *R.a* to cardinality of *S.b* is greater than a pre-defined threshold (1 + ε). If we require that containment strictly holds, then we can set ε=0. This rule follows from the fact that if *R.a*→ *S.b* is a true foreign-key, then each distinct value in *R.a* must also occur in *S.b* and hence the cardinality of *R.a* cannot exceed *S.b*. However, PowerPivot does allow the join even if a small number of values (this number is configurable) in *R.a* do not occur in *S.b*; hence we allow for ε > 0.

***LP*₃**: Prune if either *R.a* or *S.b* belong to one of the following data types: *floating point*, *Boolean*.

The rationale for this heuristic is that in practice it is very unlikely that keys or foreign-keys are defined on these data types. While this rule could, in principle, miss true foreign-keys, we did not encounter such a case in the data sets that we evaluated on (Section 4).

3.2 Candidate Scoring

We develop a scoring function to quantify how likely it is for *R.a*→ *S.b* to be a foreign-key. All candidates surviving the local pruning are scored and then sorted by score. Similar to local pruning, the scoring function is designed to be very lightweight and only relies on metadata (table, column names) and cardinality information of the columns that define the candidate. As discussed earlier, the scoring determines the order in which candidates are explored during join path enumeration. With a well-designed scoring function, the most expensive computation namely the full semi-join to check for containment of *R.a* in *S.b* can be avoided for most of spurious candidates altogether. Hence the precision and speed of foreign-key detection heavily depends on a good scoring function.

When there are multiple candidates between a pair of tables (as is typically the case), the candidate scores allow staging the detection by focusing on the candidates that have high scores first. For a given pair of tables it is often the case that there are many candidates *R.a*→*S.b* that pass the local pruning test, but *R.a*→*S.b* is not a true foreign-key. A common scenario is when *R.a* and *S.b* are "id" columns in the table. For example, in Figure 3, several columns may appear as potential foreign-keys to *Product.ProductId*: *OrderDetails.SalesPersonId, OrderDetails.CustomerId, OrderDetails.ProductId* since each candidate passes the local pruning step described in Section 3.1. One observation is that in such cases the table and column names can be a useful signal to distinguish among the candidates. Hence the *name similarity* is an important component in the scoring function (similar ideas are leveraged in schema matching techniques, e.g. see [12] for a survey). We first describe a similarity function we use to compute name similarity and then describe the overall scoring function next.

String similarity function: We exploit a string similarity function based on ideas similar to those used in fuzzy record matching (e.g. [13]). Given two strings *x* and *y* (where *x* and *y* could be column names or table names), we tokenize *x* and *y* into two sets of tokens $\{x_1, x_2, \ldots, x_m\}$ and $\{y_1, y_2, \ldots, y_n\}$ respectively. The tokenization procedure uses typical delimiters: whitespaces or symbols mark the boundary of tokens ('_','-', etc.). We also consider the change from lower case to upper case as a boundary of tokens. For instance, in string "ProductId", the case change from 't' to 'I' is a boundary that results in tokens: "Product" and "Id". Then, for each token y_j, we find the unmapped token x_i that has the largest similarity according to a given string distance function, and map x_i to y_j. We denote this maximum similarity as sim_j. In our implementation we use the Jaro-Winkler distance function; in principle other functions such as Levenshtein or Smith–Waterman could be used instead (see [14] for a comparison of strengths of string distance matching functions). We also compute f_j as follows: if y is a column name, f_j is the fraction of columns whose name contains token y_j. If *y* is a table name f_j is the fraction of tables whose name contains token y_j. We define:

$$similarity\ (x, y) = \frac{\sum_{j=1}^{n}(\mathrm{sim_j} \times \ln(\frac{1}{f_j}))}{\sum_{j=1}^{n}\ln(\frac{1}{f_j})}$$

Intuitively, the similarity function returns a higher value when tokens have lower string distances and gives higher weight to rarer tokens [14] via the f_j measure.

Overall scoring function: The scoring function for a candidate *R.a*→*S.b* is a weighted sum of the following scores:

- s_1 = *similarity*(*ColumnName(R.a)*, *ColumnName*(S.*b*)), i.e. similarity between column names of *R.a* and *S.b*.
- s_2 = *similarity*(*ColumnName(R.a)*, *TableName*(S.*b*)), i.e. similarity between column name of *R.a* and table name of *S.b*
- $s_3 = \frac{|R.a|}{|S.b|}$, i.e. the ratio of the cardinality of *R.a* to cardinality of *S.b*.
- s_4 = 1 if *R.a* is a key and 0 if not.

$$SF\ (R.a \rightarrow S.b) = \sum_{i=1}^{4} w_i \times s_i$$

While the importance of s_1 is easy to see, the importance of s_2 arises from the fact that in many cases the name of the foreign-key column in *R* is very similar to the table name of *S*. For example, consider *Sales.ProductId* → *Product.Id*. In this case, s_1 has a low similarity, but s_2 will return a high similarity. We observe such patterns to be quite common in practice. As described in Section 3.1 (in pruning rule ***LP₂***), the ratio of the cardinality is an important indicator of foreign-key. In a true foreign-key we typically see this ratio ≈ 1. Thus consider a case where we have two possible candidates: *Sales.ProductId* → *P.Id* and *Sales.ProductId* → *C. Id.* Suppose the table *P* is a dimension table of products and *C* is a dimension table of customers, and suppose there are 100 distinct products and 100,000 distinct customers. Then s_3 for *Sales.ProductId* → *P.Id* will likely be much closer to 1 than s_3 for *Sales.ProductId* → *C. Id*. The motivation for s_4 is to de-prioritize key-key joins, since such joins are typically used to model specialization (e.g. is-a relationships) and is unlikely to be a true foreign-key. We do this by providing a *negative* weight w_4. Note that if *R.a* is not a key, then no penalty is incurred since s_4 = 0. In our current implementation, these weights are manually tuned and appear to perform well when evaluated on several real and synthetic databases. It is potentially interesting future work to consider if a machine learning based approach could be used to identify appropriate weights; however we do not have a rich enough and suitably labeled training set to attempt this approach.

One important class of candidates that require special consideration is when both *R.a* and *S.b* are "complex" types: which we define as (a) long strings or large integer numbers *and* (b) large number of distinct values (e.g. a column contain Social Security Number). Interesting cases are when the entity name itself serves as the key and foreign-key or when some large numeric identifiers are used as the key for a dimension table. Although many such candidates are false positives, these false positives are amenable to efficient detection via the random sample probing technique or min-hashing technique (described in Section 3.4). Thus, to ensure that we don't miss out on good candidates, we artificially boost the scores of such candidates so that they are ranked highest. These candidates therefore have a high chance of being considered in join-path enumeration.

3.3 Join-Path Enumeration

The join path enumeration finds the join paths to connect the selected fact tables *F* to the selected dimension tables *D* using the currently highest scoring candidate foreign-keys. If such join paths are found, they go through the containment check process (described in Section 3.4); any candidate *R.a*→*S.b* that passes the containment check is then output by the algorithm as a foreign-key.

The key ideas underlying join-path enumeration are: (a) Only explore those join paths that can potentially help connect the user-selected fact tables *F* and dimension tables *D*. (b) Exploit the candidate scoring function (described in Section 3.2) to guide the process of exploring join paths – in particular, we aim to create a minimal graph connecting *F* and *D* using the highest ranking edges, i.e. candidates that have not yet been pruned. (c) Since join path enumeration and containment verification steps are run in a loop, at any point in time, we leverage the partial solution obtained thus far (i.e. candidates that have passed containment verification and are therefore known to be in the output) to perform *conflict pruning* (described below) and thereby eliminate other candidates. We next describe conflict pruning, and then present the overall algorithm.

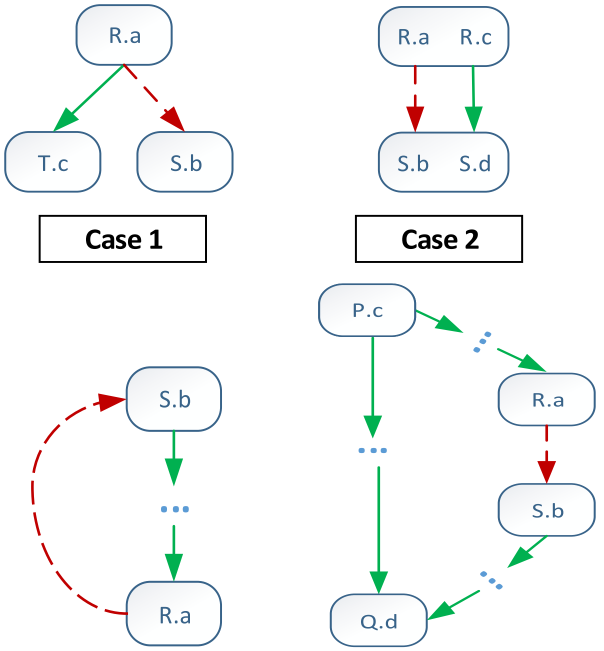


Figure 6. Conflict pruning. In each case a candidate *R.a* → *S.b* denoted by a red dashed arrow is in conflict with other candidates denoted by green solid arrow. If each green solid edge has a higher score than *R.a* → *S.b*, then *R.a* → *S.b* is removed from consideration as foreign-key.

3.3.1 *Conflict Pruning*

Conflict pruning occurs when a candidate *R.a*→ *S.b* is in conflict with other declared, output, or higher ranked candidate foreign-key(s). Note that since join-path enumeration and containment verification happen in a loop, when a candidate passes containment verification, it may result in additional conflict pruning during the next iteration through the loop.

Conflicts arise due to constraints on foreign-keys, some of which are specific to the PowerPivot data model, whereas others are more general and would also occur in other systems such as relational databases. Figure 6 illustrates some examples of constraints. The solid green edges denote either declared foreign-keys or candidate foreign-keys with higher score than the red edge. The dashed red edge (shown as *R.a* → *S.b* in the figure) denotes the candidate that is in conflict with the green edges. We refer to the set of green edges as the conflict set of *R.a* → *S.b*. ***Case 1***: A particular column (say *R.a*) can be the child of only one foreign-key. ***Case 2:*** There can exist at most one foreign-key between any pair of tables. ***Case 3***: There cannot be any cycles in the foreign-key graph. ***Case 4***: there can exist at most one foreign-key *path* between any pair of tables.

```
Input: Set of fact tables F, set of dimension tables D, and a list
of candidates L sorted by score
Output: a set of foreign-keys connecting F and D
result = ∅
For each pair of tables (f, d), where f ∈F and d ∈ D do
   result = result ∪ FindForeignKeys(f, d, L);
Return result;
```

Figure 7. Overall algorithm for detection foreign-keys between *F* and *D*, given a sorted list of candidates *L*.

```
FindForeignKeys (f, d, L)
Loop
   Let SG = FindConnectingSubgraph (f, d, L)
   AllTrue = VerifyAllTrue (SG)
Until AllTrue or SG is empty
Return SG
```

Figure 8. Detect foreign-keys between a pair of tables *f* and *d*, given a sorted list of candidates *L*.

```
FindConnectingSubgraph (f, d, L)
Initialize an empty graph G
For each candidate c in descending order of scores in L
     If c's state is Declared, VerifiedTrue or None,
        add c as an edge to G
     If G contains both f and d as nodes, break loop
For each edge e in G, if there is a path from some R to S,
and e is in that path, then mark e
Return the set of edges marked in step c.
```

Figure 9. Finding the best connecting subgraph between *f* and *d*, given a sorted list of candidates *L*.

```
VerifyAllTrue (SG) // SG is a connecting subgraph
For each candidate c in SG
   Let CS = conflict set of c
   If CS is empty then
      If VerifyContainment (c) is true
          Then Mark c's state as VerifiedTrue
      Else Mark c's state as VerifiedFalse; Return false
   Else // non-empty conflict set
      If VerifyAllTrue (CS) is true then
         Mark c's state as Pruned; Return false
      Else Return false;
Return true
```

Figure 10. Verifying edges in a connecting subgraph. Note that conflict pruning may prevent an edge from being verified.

We observe that constraint-based pruning leverages the fact that while a candidate may individually have a high score as a foreign-key, the existence of other foreign-keys (perhaps even on other tables) with higher score may result in this candidate being pruned. Although our implementation of constraint-based pruning comes from the restrictions of a particular version of PowerPivot, this type of pruning can still be used in relational databases, e.g. ***Case 1*** and ***Case 3*** above likely also hold in most relational database engines.

3.3.2 Algorithm

The pseudo-code for the foreign-key detection algorithm is presented in Figures 7 to 10. The outer loop of the algorithm (shown in Figure 7) invokes the function *FindForeignKeys* on each pair of tables (f, d) $\in F \times D$. In turn, *FindForeignKeys* relies on the function *FindConnectingSubgraph* to identify a set of edges that connect f and d, and *VerifyAllTrue* to verify whether all edges thus identified pass containment verification or not. If not, *FindForeignKey* repeats the loop until either all edges pass containment verification or no connecting subgraph can be found.

We maintain a *state* for each candidate in L, which can take on one of the following values: (i) *Declared*, if it is a declared foreign-key. (ii) *VerifiedTrue*, if it has been verified to be true in the containment verification step. (iii) *VerifiedFalse*, if it has been verified to be false in the containment verification step, (iv) *Pruned*, if it has been pruned due to conflict pruning, (v) *None*. Each candidate's state is initialized as *Declared* if it is a declared foreign-key, otherwise *None*.

Figure 9 shows how the algorithm detects a connecting subgraph taking into account the ordering of candidates by score, and Figure 10 describes the algorithm for verifying edges in a given connected subgraph. We note that conflict pruning and containment verification (denoted by the function *VerifyContainment*) are invoked as part of *VerifyAllTrue*. Prior to deciding if containment checking should be invoked for a candidate c, we check if the candidate has a conflict set (defined in Section 3.3.1). If the conflict set is empty, then we invoke containment verification on the candidate. If not, we recursively invoke *VerifyAllTrue* on the conflict set, which if it passes results in pruning of the candidate c. If the loop 1-10 executes without returning, it implies all edges in the connecting subgraph have passed containment verification, so we return true.

3.4 Containment Verification

As described earlier, verifying that *R.a* is contained in *S.b* is necessary for *R.a*→*S.b* to be a true foreign-key. It requires a full semi-join and is the most expensive operation in foreign-key detection and therefore should be deferred until other (cheaper) tests have been first attempted. We introduce two lower cost tests for containment verification, which allow us to reject a candidate much more inexpensively. These two techniques are based on the same intuition – quickly detect if one or more values in *R.a* do not occur in *S.b*. However, they have different performance trade-offs, and hence a decision needs to be made as to whether to apply only one or the other. These tradeoffs are similar the decision of whether to use an *index nested loop* join or a *hash* join in relational database query processing.

Random sample based probing: The basic idea is to obtain a random sample of k values in *R.a* and probe *S.b* for each distinct value in the sample. If more than a pre-specified fraction (θ_p) of values probed do not exist, then we rule out candidate *R.a*→*S.b*. If more than the pre-specified fraction of values *do* exist, we proceed to the next test (either *min-hashing* or full semi-join). If we require full containment, then we can set the pre-specified fraction to 1, i.e., even a single probe failure would terminate the test. The key reasons that makes this test very efficient is that PowerPivot provides low overhead support for the following two operations: (a) Obtain a random sample of the distinct values in a column. This efficiency is enabled by the fact in PowerPivot all data is memory resident, and the API allows accessing any index of the dictionary of distinct values of a column. (b) Probing a column for a small set of given values. Note also that spurious candidates are likely to fail in this test even for a small k such as 50. In Section 4, we show the effectiveness of this technique in reducing spurious full semi-joins.

We observe that the technique of random sample based probing is particularly effective for in-memory BI engines such as PowerPivot where the appropriate access methods for random sampling and probing exist. However, this technique may not always be efficient in other contexts, e.g. relational database engines, where: (a) not all data is memory resident (b) no index may exist on the column being probed. In such cases, other techniques such as min-hashing (described next) are likely to be more scalable.

Min-hash based comparison: This test is based the idea of *min-hashing* [5], which is a technique for quickly estimating how similar two sets are, in particular it estimates the Jaccard similarity between the two sets. In our context, we use this technique to develop a test to identify if *R.a* is unlikely to be contained in *S.b*. If the test reveals that it is unlikely that *R.a* is contained in *S.b*, we reject the candidate *R.a*→*S.b*.

We apply n different hash functions $H_1,\ldots H_n$ to each value in both *R.a* and *S.b*. Then, if M_i (X) is the smallest hash value when applying function H_i to set X, we check if $M_i(R.a) < M_i(S.b)$. Observe that any occurrence where the min-hash of *R.a* is less than min-hash of *S.b* is an indication that there exists a value in *R.a* that is not contained in *S.b*. The strict version of this check eliminates the candidate if the above inequality holds for even a single hash function. However, since PowerPivot does allow joins even when a few values are not contained, a more general check is described below.

We define an indicator function as follows:

$$P_i = \begin{cases} 1 & if\ M_i(R.a) < M_i(S.b) \\ 0 & otherwise \end{cases}$$

Then, if $\frac{(P_1 + \ldots + P_n)}{n} > \theta_h$, we reject the candidate. Note that this test requires a full scan of a column to compute the min-hashes. In practice, we use n=100 hash functions. Although the cost of computing min-hashes is not negligible, it is a *one-time cost* (per column) and can therefore be amortized across tests for different candidates that the column is a part of.

Finally, in our experiments, we find that if a candidate passes the random sample based probing test, it is very likely that it also passes the min-hashing test. Given that the random sample based probing is much less expensive in PowerPivot compared to min-hashing, the trade-offs strongly favor using only random sample based probing. However, it is important to note that in a relational database setting, obtaining a random sample as well as probing may incur a high cost (comparable to a full scan), particularly when appropriate physical structures such as indexes do not already exist in the database. In such cases, min-hashing may in fact be the preferred technique for containment verification.

Full semi-join: The last, and most expensive, option for checking containment is to perform the full semi-join between *R.a* and *S.b*. Although this join is performed very efficiently in PowerPivot (exploiting columnar storage and ability to process on compressed data), the processing time is a function of the data sizes of *R.a* and *S.b*. Thus, the time for a full semi-join grows with data size. Therefore, for very large data sets, we invoke this only for the highest ranking candidates that have passed all other tests.

4. EXPERIMENTS

The techniques described in this paper have been incorporated into PowerPivot for Excel [2]. We report the results of our experiments on PowerPivot for Excel 2013.

Goals: The goals of the experiments are to evaluate: (a) *Quality*: i.e. the accuracy of the foreign-keys output by the algorithm. As discussed in the introduction the property of very few false positives is very important since these discovered foreign-keys are auto-created by PowerPivot. (b) *Performance* and *Scalability*: (i) We want to verify whether foreign-key detection remains within the realm of interactive response time even for large data sizes and schemas. (ii) Drill-down into the effectiveness of various pruning techniques described in Section 3.

Table 1. Databases used in experimental evaluation.

Database	*#Tables*	*#Columns*	*Raw Size (MB)*	*Comp. Size (MB)*
TPC-H 10GB	**8**	**61**	**10,000**	**5,730**
TPC-H 1GB	**8**	**61**	**1,000**	**470**
TPC-E	**33**	**191**	**1,200**	**600**
AW-DW	**24**	**313**	**210**	**11**
REAL-1	**3**	**21**	**1.5**	**0.3**
REAL-2	**7**	**58**	**30**	**10**
REAL-3	**12**	**132**	**75**	**25**
REAL-4	**3**	**26**	**0.9**	**0.3**
REAL-5	**2**	**21**	**50**	**14**
REAL-6	**19**	**421**	**93,000**	**4,300**

Databases: The databases that we used in our evaluation are shown in Table 1. We use 4 synthetic databases: TPC-H [3] (with 1GB and 10GB scaling factors respectively), TPC-E [19] (scale factor 500) and AdventureWorks-DW [4] database (abbreviated as AW-DW in the table/charts below). TPC-H 1GB and 10GB versions allow us to observe how the performance changes when data size is increased. In TPC-H 10GB database, the *lineitem* table has around 60 million rows and the *orders* table has around 15 million rows. TPC-E and AW-DW both have both large schemas as well as relatively large databases, thereby presenting good test of scalability of the techniques. For example, AW-DW has over 300 columns and hence over 90K candidate foreign-keys, and TPC-E has around 200 columns from 33 tables, and 1GB raw data size. We also evaluated our techniques on 6 real-world customer databases of Microsoft SQL Server labeled REAL-1 through REAL-6. Of these, REAL-6 which is the database of a large book retailer, offers the strongest test of scalability with its large database size (93GB raw data size) and schema (19 tables, 400+ columns). We note that PowerPivot obtains very significant data compression – Table 1 shows the compressed data sizes; we observed compression ratios ranging between ~2x to ~20x in the data sets we evaluated.

Methodology: For each data set, we already know true foreign-keys since these were Microsoft SQL Server databases where the foreign-keys were explicitly declared. These declared foreign-keys are the "ground truth" we use for the quality evaluation. Of course, it is possible that not *all* semantic foreign-keys are explicitly defined in the database. As we observe in practice on occasion (and reported in our results below), there are *plausible* foreign-keys that are not actually defined, i.e. a manual inspection of the foreign-key shows it is meaningful: a foreign-key join would not cause incorrect results. For example TPC-E has two such plausible foreign keys that we discuss below (Section 4.1). Thus, we report two metrics of *false positives*. The strict version flags any detected foreign-key that is not declared in the database as a false positive. The second version is identical to the strict version except that it does not flag plausible foreign-keys as false positives. A *false negative* is reported if one of the declared foreign-keys is not discovered. As we noted earlier, PowerPivot does not allow some foreign-keys that are typically allowed in relational databases. For example, PowerPivot only allows single-column foreign-keys, only one foreign-key between any pair of tables, and it does not allow a foreign-key from a table to itself. Since some of the databases we experimented with did contain such foreign-keys, some of the false negatives we report are due to limitation in PowerPivot. In the results reported here, we relax one of these limitations in PowerPivot: in particular, we are capable of recommending more than one foreign-key between a given pair of tables.

When we load the database into PowerPivot, we drop all declared foreign-key relationships. PowerPivot compresses the data during loading and prepares it for analysis. Then, for each pair of tables in PowerPivot we invoke the foreign-key detection code and measure the running time and evaluate the quality of the results. This all-pairs invocation represents the *worst case* running time for foreign-key detection in PowerPivot. In practice, most user created pivot tables do not reference all tables, and thus the running time for foreign-key detection are typically much smaller than those reported below. All numbers were obtained on a machine with an Intel Xeon CPU with 2 processors (6 cores each) and 96GB RAM.

4.1 Accuracy of Foreign-Key Detection

Table 2 shows the number of declared foreign-keys in each database and the number of foreign-keys discovered using our techniques in PowerPivot.

First, we observe that in 7 out of 10 databases, there were no false positives (strict or otherwise). Of the 9 cases of strict false positives found across all databases, 6 of those were plausible. Therefore, out of the 143 foreign-keys detected by our technique only 3 could definitely be said to false positive (i.e. 98% precision). Similarly, in 8 out of 10 databases, there were no false negatives either. Thus, the overall accuracy of our techniques is high. We included REAL-5 since this is an example where we know there are no declared foreign-keys between the two tables, so the goal was to check for false positives only. We next discuss each database where we reported either a false positive or false negative.

TPC-E: Out of the three foreign-keys that we flag as FPs (strict) in TPC-E, two of those are plausible: (1) [*Holding*].[*H_CA_ID*] → [*Customer*].[*CA_ID*] (2) [*Holding*].[*H_S_SYMB*] → [*Security*].[*S_SYMB*] but are not declared in the benchmark schema. One of the false negatives is the multi-column foreign key between the tables *Holding* → *Holding_Summary*. We are unable to detect four other foreign-keys because the destination tables are empty. In our current implementation we choose not to recommend such foreign-keys since we get no signals from the data. Similarly, we do not discover five other foreign-keys because the source columns are either empty or contain only one value. Once again, in the interest of erring on the side of precision, our implementation does not recommend such foreign-keys. It is worth noting that in other contexts where a different trade-off between precision and recall needs to be made, our techniques could be adapted to support such trade-offs.

Table 2. Summary of quality evaluation showing number of actual foreign keys, foreign keys discovered, number of false positives and false negatives.

Database	*#FKs actual*	*#FKs disc.*	*#FPs (strict)*	*#FPs*	*#FNs*
TPC-H 10GB	8	8	0	0	0
TPC-H 1GB	8	8	0	0	0
TPC-E	50	43	3	1	10
AW-DW	42	39	2	0	5
REAL-1	2	2	0	0	0
REAL-2	9	9	0	0	0
REAL-3	17	17	0	0	0
REAL-4	2	2	0	0	0
REAL-5	0	0	0	0	0
REAL-6	19	23	4	2	0

AW-DW: Both false positives (strict) in AW-DW look plausible, i.e. in addition to meeting all syntactic requirements, they also appear to be semantically viable. An example is a foreign-key from column [*FactSurvey*].[*ProductCategory*] → [*DimProductCategory*].[*ProductCategoryKey*]. The false negatives in AW-DW arise due to the fact that PowerPivot is more restrictive that a relational database in the kinds of foreign-keys it allows. We miss 4 foreign-keys which were "self-links" in the relational database (e.g. *T.ParentEmployeeId* → *T.EmployeeId*), which are valid in a relational database but disallowed in PowerPivot. We also miss one multi-column foreign-key.

REAL-6: Two out of the four false positives (strict) are plausible. The two remaining false positives arise due to an issue with how our algorithm deals with string columns. In this case, the application uses a string column to encode integer values (this is because the application also uses the column to store a single string value called "Unknown" in addition to the integer values). Recall from Section 3.2 that we boost the scores of candidate foreign-keys involving string columns. This causes our algorithm to give an unnecessarily high score to this candidate and since it meets all other syntactic criteria including containment, the foreign-key is recommended. We note that this false positive would not have occurred if the application had used an integer column and, for example, encoded the "Unknown" value as NULL.

4.2 Performance and Scalability

Figure 11 shows the running time for identifying foreign-keys between *all pairs* of tables in the database. As mentioned above, this is the worst case time for foreign-key detection. Note that we use a logarithmic scale on the y-axis since there is a relatively large variability across databases in running time depending on data and schema size. First, we see from the figure that in REAL-6 with a raw data size of 93 GB (compressed size of 4.3GB) and a large schema size, all foreign-keys can be detected in around 200 seconds. Similarly, for TPC-H 10GB, where the compressed data size is more than 5.7GB, we are able to detect all foreign-keys in the schema in about 1.5 minutes. In this case, almost all the time was spent in containment verification – in fact in the full semi-join evaluation for the 8 FKs that we output. Thus, it is not possible to reduce the foreign-key detection time much further since we are required to do a full semi-join to verify containment for the output FKs. Note also that foreign-key determination only needs to be done once – so the cost is amortized over all subsequent pivot table analysis.

We also report that in practice, if a user were to select a smaller subset of tables in the pivot table, analysis is even faster. For example, if we only select the *lineitem*, *nation*, and *region* tables, we can detect the 3 relationships (between *lineitem* and *supplier*, *supplier and nation*, *nation and region*) in ~25 seconds. For the TPC-H 1GB database we are able to detect the same 8 foreign-keys in 14 seconds when invoked on all table pairs. This ~6x speedup compared to TPC-H 10GB is entirely due to the reduction in the cost of full semi-joins.

Finally, we observe that in practice for most pivot tables on real-world databases, foreign-key detection speeds are truly interactive (a few seconds in most cases).

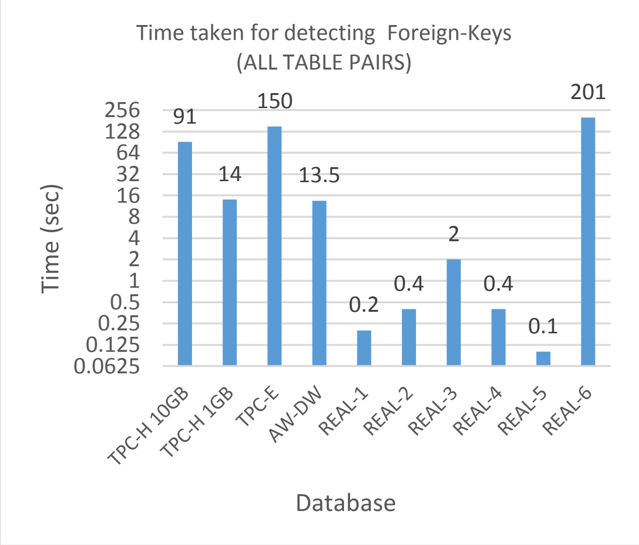


Figure 11. Time taken for FK detection for each database.

Drill-down: Next we drill-down into specific datasets to examine the effectiveness of various pruning techniques and report the results in Figure 12, 13 and 14. Note that all the conflict pruning rules in Figure 6 are applied in this experiment and hence the final numbers of pairs in the figures are slightly lower than those reported in Table 2. An interesting case is AW-DW in which the time taken for foreign-key detection is mostly due to the complexity of the schema. To understand this case better, we show a drill-down of various pruning techniques for AW-DW in Figure 12. The x-axis shows different pruning techniques, the y-axis on the left shows the number of candidate foreign-keys on a log scale, and the y-axis on the right shows the percentage of candidates pruned by the pruning technique. The total number of candidates that the algorithm needs to consider exceeds 91K. We observe that the local pruning techniques (Section 3.1) are able to eliminate about 97% of those candidates. However, this still leaves over 3K candidates that still need to considered. The name similarity and conflict pruning together are able to further cut this number to around 1K. Notice that the random sample based probing (Section 3.4) is very effective in cutting down all remaining spurious candidates which can be seen from the fact all remaining candidates for which we need to invoke the full semi-join in fact pass the test.

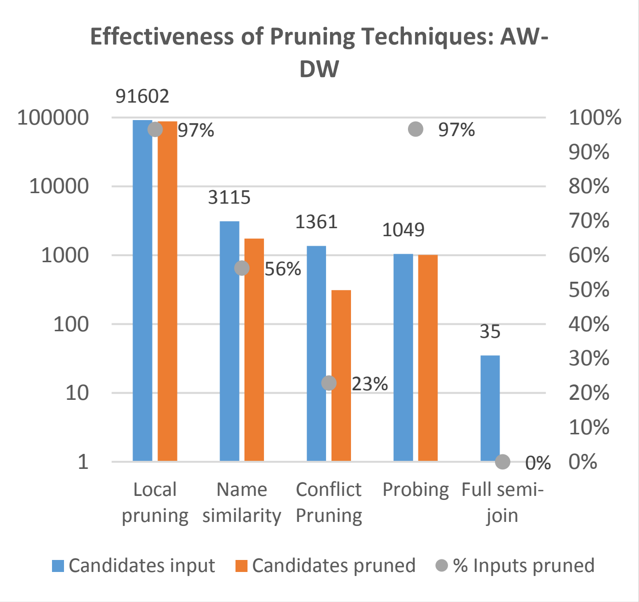


Figure 12. Drill-down for AdventureWorksDW database.

Figures 13 and 14 show similar drill-downs for REAL-3 and REAL-6 databases. The overall trends are similar to AW-DW. We observe that local pruning and random sampling based probing are very effective in eliminating most of their inputs across all three data sets. The importance of conflict pruning however varies significantly across different data sets, from 23% in AW-DW to less than 1% in REAL-6. We also observe that in REAL-6 the full semi-join is necessary to prune 15 out of 38 candidates whereas in other cases all false positives were eliminated by other techniques.

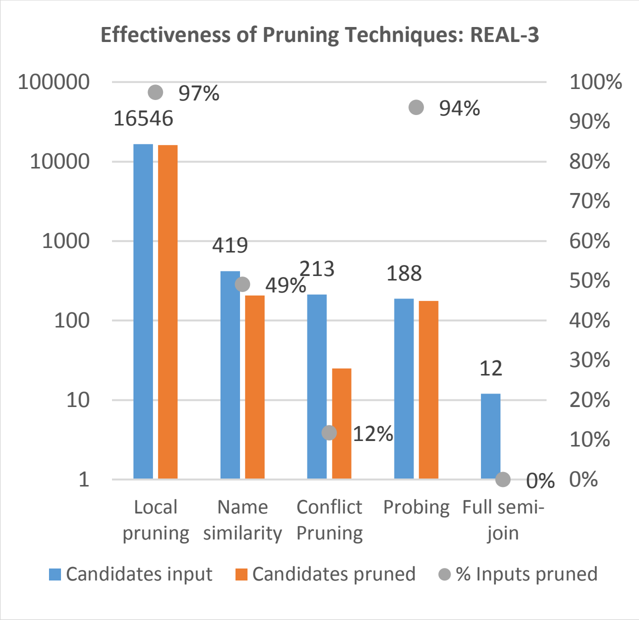


Figure 13. Drill-down for REAL-3 database.

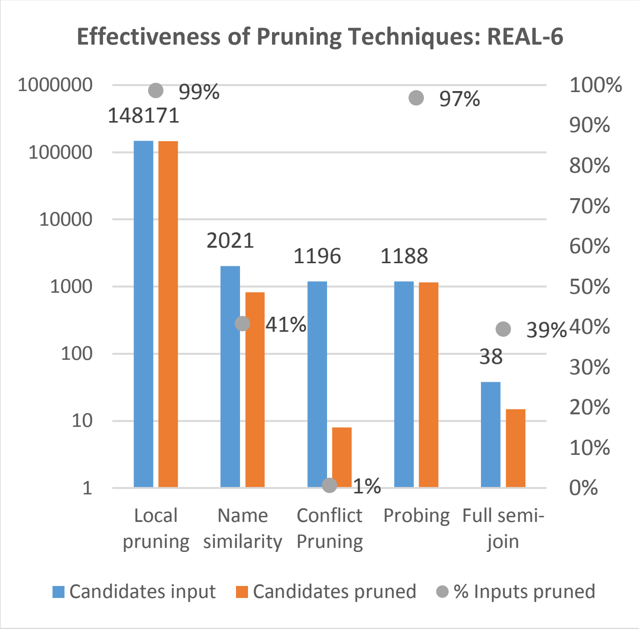


Figure 14. Drill-down for REAL-6 database.

5. RELATED WORK

There is a large body of work on foreign-key detection and detection of inclusion dependencies in relational databases. Examples of such work include [6]-[11]. In most of this work, the goal is to identify all suitable foreign-key candidates (i.e., focus is typically on recall) in an offline batch processing setting for advanced database users such as database administrators or data stewards. In contrast, our work is substantially different in two major aspects. First, we are required to find high precision foreign-keys only – since PowerPivot automatically creates relationships thus discovered and the users (typically information workers) may not be sophisticated enough to understand and undo erroneous relationships. Second, there is a requirement that the detection be performed at interactive speeds. These requirements leads to several pruning techniques described in this paper. We are aided by the fact that data is memory resident in PowerPivot and has statistics and data structures optimized for the operations we rely on. For instance, PowerPivot has in-memory dictionaries of each column that allows us to very efficiently lookup the cardinality of each column and execute random sample based probing. Such techniques may not always be possible in relational databases when suitable index structures do not exist.

We discuss here two representative examples of related work on foreign-key detection ([7] and [10]) to highlight the differences in goals and consequently the accuracy and performance/scale. In [7], the authors conjecture that in most cases the values in a foreign-key column form a nearly uniform random sample of values in the key column. They devise a test for this randomness property and demonstrate its effectiveness as a heuristic to identify foreign keys. For TPC-E they report 0.57 precision and 0.82 recall, and about 0.8 precision and 0.8 recall when evaluating against their augmented set of foreign-keys. While such a trade-off between precision and recall may be acceptable in other scenarios, it would not be appropriate for ours. From our experience with the datasets we have worked with it appears unlikely that any single heuristic or rule will achieve high precision by itself. However, combining multiple signals together during detection does appear to greatly reduce false positives. Another important difference is that the approach of [7] is not designed to be interactive, but rather operate in an offline batch mode. For example, they report a running time of approximately 2.5 hours for TPC-H 10GB database – contrast this with about 1.5 minutes for the same database using our techniques, and in seconds when the user choose to detect foreign-keys for a small number of tables. The work of [6] is closely related to [7] in which the authors use a measure of distance between column value distributions, and cluster columns based on that distance metric. However the application is to find similar columns rather than detecting foreign-keys.

In [10], a learning based approach is used for detecting foreign-keys from a given set of previously identified inclusion dependencies (using the algorithm of [19]). They define ten features for a candidate foreign-key including cardinality, column names, difference in value length between the two columns etc. They train several classifiers (NaiveBayes, SVM, decision tree based, etc.) and then compare these different classifiers on several real-world databases. They report F-measures that vary between 0.71 and 1.0, the average F-measure of classifiers range from 0.78 to 0.93 (no precision numbers are reported). They also report performance numbers of around 4 hours to identify the inclusion dependencies on a database of size 32GB, although the classification itself is very fast. Besides the differences in emphasis on precision and interactivity, our techniques leverage similarity between table and column name similarity more deeply (including fuzzy string matching) as well as in the use of the global topology of the derived join graph to improve precision of detection. It is an interesting direction of future work for us to consider if a learning based approach could be leveraged in our setting (e.g. to learn weights for different similarity measures in candidate scoring). A key challenge is the generalizability of the model from a few specific training datasets to the diversity of datasets used by information workers in enterprises.

We adapt the ideas and techniques for string similarity used previously in fuzzy record matching, de-duplication and schema matching scenarios (e.g. [12][13]). Not surprisingly, these techniques are crucial for those cases where it is insufficient to rely purely on data characteristics (such as indicators of containment). The most common example of this case is the “id problem” where a large number of different id columns in different dimension tables appear plausible targets for a given id column in a fact table.

In [15], the authors briefly allude to techniques for identifying join paths in Tableau. They require exact match of column names and data types, and if they are date/time fields, they require that both columns represent the same granularity in the time dimension hierarchy. In contrast, our techniques allow a much larger class of foreign-keys to be identified. To the best of our knowledge in most other in-memory business intelligence engines, a data steward is expected to build the data model (i.e. identify foreign-keys etc.) and the information workers then consume this model. Our foreign-key detection technology in PowerPivot enables information workers “self-service” analysis of their data since they are not required to explicitly create the data model.

We note that several commercial data profiling and analysis tools exist that detect inclusion dependencies (e.g. [21][22]). Such tools can aid advanced users such as data stewards to explore foreign keys by presenting them a list of column pairs that are highly overlapped and allowing them to drill down to the orphan values on both sides. These tools do not try to predict foreign keys and do not target the information worker audience that we target in our work. Finally, there are tools that are related to but do not support automatic foreign key detection. For example, [9] supports linking entities as structured rows in database and as free text in documents. The linking considered is at the instance level, not at the schema level like foreign key. Similarly [23] [24] support interactive data cleansing and exploration. However these systems do not support automatic foreign-key detection.

6. CONCLUSION AND FUTURE WORK

High precision automatic foreign-key detection is an enabling technology for self-service business intelligence (BI). In this paper, we describe techniques for automatically identifying appropriate foreign-keys in a commercial, in-memory BI engine: PowerPivot for Excel. While some of our techniques leverage specific constraints of PowerPivot, many of the techniques we develop are applicable in other contexts such as a relational database engine. One interesting area of future work is generalizing in an orthogonal dimension: allowing fuzziness in the foreign-keys. This is particularly interesting for mashups where the user may have imported external data sources (e.g. web tables [16]) and wishes to join this data with existing fact/dimension table from other sources within the enterprise.

7. ACKNOWLEDGMENTS

We acknowledge Amir Netz, Ashvini Sharma, Julie Strauss and Yuzheng Ying from the Microsoft SQL Server PowerPivot team, who interacted closely with us on this effort. They helped define requirements, provided valuable feedback during this work, provided access to real-world data sets, and assisted in integrating our techniques into PowerPivot for Excel. We also thank Christian König for his insightful comments on this paper.

8. REFERENCES

[1] Chaudhuri, S. Dayal, U., and Narasayya V. *An Overview of Business Intelligence Technology*. Communications of the ACM, Vol 54 No. 8, Pages 88-98.

[2] *PowerPivot for Excel*. http://msdn.microsoft.com/en-us/library/ee210644.aspx

[3] Chaudhuri, S., Narasayya, V. *Program for TPC-D Data Generation with skew*. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/

[4] AdventureWorks database. http://msftdbprodsamples.codeplex.com/releases/view/55926

[5] Broder, A., Charikar, M., Frieze, A., Mitzenmacher, M. *Min-wise independent permutations*. In Proc. of ACM Symposium of Theory of Computation (STOC '98).

[6] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, Divesh Srivastava: *Automatic discovery of attributes in relational databases*. ACM SIGMOD Conference 2011: 109-120

[7] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, Divesh Srivastava: *On Multi-Column Foreign Key Discovery*. PVLDB 3(1): 805-814 (2010)

[8] Divesh Srivastava: *Schema extraction*. CIKM 2010: 3-4

[9] Rakesh Agrawal, Ariel Fuxman, Anith Kannan, Qi Lu, John Shafer. *Composing Structured and Text Databases*. Microsoft Research Technical Report 2012.

[10] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. *A machine learning approach to foreign key discovery*. In WebDB, 2009.

[11] S. Lopes, J.-M. Petit, and F. Toumani. *Discovering interesting inclusion dependencies: application to logical database tuning*. Information System- s, 27(1):1–19, 2002.

[12] Rahm, E. and Bernstein, P. *A survey of approaches to automatic schema matching*. VLDB Journal, Vol. 10, Issue 4, pp 334-350.

[13] Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R. *Robust and Efficient Fuzzy Match for Online Data Cleaning*. ACM SIGMOD 2003.

[14] Cohen, W.W. Ravikumar, P., Fienberg, S.E. *A Comparison of String Distance Metrics for Name-Matching tasks.* In: Workshop on Information Integration on the Web. IIWeb, 2003.

[15] Morton K., Bunker, R., Mackinlay, J., Morton, R., Stolte, C. *Dynamic workload driven data integration in Tableau*. ACM SIGMOD 2012.

[16] Microsoft Power Query for Excel. http://office.microsoft.com/en-us/excel/download-microsoft-power-query-for-excel-FX104018616.aspx

[17] TPC-H benchmark. http://www.tpch.org/tpch

[18] Relationships in PowerPivot. http://technet.microsoft.com/en-us/library/gg399148.aspx

[19] TPC-E benchmark. http://www.tpch.org/tpce

[20] Bauckmann, J., et al. *Efficiently Detecting Inclusion Dependencies*. In International Conference on Data Engineering, 2007, Istanbul, Turkey.

[21] IBM Infosphere. Data profiling and analysis. http://www.ibm.com/software/data/infosphere/

[22] Data Profiling Task in Microsoft SQL Server Integration Services. http://technet.microsoft.com/en-us/library/bb895263.aspx

[23] Trifacta. http://www.trifacta.com/

[24] OpenRefine (formerly Google Refine). http://openrefine.org/