Sampling based techniques for profiling keys

Alistair Evans
Department of Computer Science
The University of Auckland

Supervisor: Associate Professor Sebastian Link

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Professional Studies in Data Science, The University of Auckland, 2014.

Abstract

Keys play a fundamental role in understanding both the structure and properties of data. Despite their importance, research into data profiling (and key mining as a sub field) has remained limited perhaps because it is an extremely challenging problem. The number of possible keys increases exponentially with the number of attributes.

Recent key discovery research has mainly looked at profiling all keys from a given relation exactly. This research is based on two assumptions. The first is that all keys must be found. The second is that the data source is consistent and has no “dirty data”. Recent trends in data collection and management have made both of these assumptions less valid.

The assumption that key mining algorithms need to find all keys exactly is potentially over ambitious given the difficulty of the problem and the growing volume of data. The second assumption does not seem to reflect modern application requirements, in particular in the context of data integration and dirty data. The veracity of data seems to indicate that a focus on approximate key profiling is a better choice.

The purpose of this research is therefore to investigate sampling for discovering both exact and approximate keys and determine its robustness in the presence of dirty data. Sampling is shown to have a large performance benefit over profiling a full dataset with a sample size of 10% needing just 1% of the running time of the algorithm on the original dataset but still providing a high level of accuracy. A preferential sampling technique is also devised to improve on random sampling.

The key mining algorithm used is then parallelised using the Apache Spark framework and its combination with sampling is investigated. Parallelisation by itself is shown to provide a significant decrease in the running time but with the trade-off of decreased accuracy. When combined with random sampling the running time is decreased again but with no further loss of accuracy. When combined with preferential sampling, the accuracy improves.

Acknowledgment

I would like to thank my supervisor, Associate-Professor Sebastian Link for his guidance with the project.

I would like to thank my partner Victoria for her unwavering support and for letting me off so many dishes in 2015. Thank you also to my family for their support and to my parents in particular for being such big advocates of higher learning.

Contents

1. INTRODUCTION

Keys play a fundamental role in understanding both the structure and properties of data. Given a collection of entities, a key is a set of attributes whose values uniquely identify an entity in the collection [SIS-06]. In a relational database, the key for a collection of rows in a table is the set of one or more columns such that no two rows have matching values in each of the key columns.

Keys have played an important part in the management of data in relational database management systems for several decades. They are core to the tasks of data modelling, query optimization, indexing, anomaly detection, and data integration [ABITE-95] [NAU-14]. Key discovery research is also important in other data profiling fields, such as functional dependency or foreign key detection [HEI-13].

Despite their importance, research into data profiling (and key mining as a sub field) has remained limited. This is perhaps for a number of reasons. The first is that it is an extremely challenging problem. The number of possible keys increases exponentially with the number of attributes [SIS-06]. [GUN-03] showed the problem of finding the number of all keys of a given relation, and alternatively, the problem of finding the number of minimal keys of a given relation to both be #P-hard. #P-hard problems are the set of counting problems associated with the decision problems in the NP set. NP problems (where instances of the problem for which the answer is yes have proofs) can be verified in polynomial time at best [WIK-NP].

The recognition of the difficulty in discovering exact keys led to some early research into approximate keys. [KIV-95] explored the inference of approximate functional dependencies. Functional dependencies are not directly addressed in this paper, but keys are subsumed by functional dependencies in the relational model. An approximate key may still be useful if it has a desired minimum *strength*. Strength is defined as the number of distinct key values in the dataset divided by the number of entities [SIS-06]. This area of research has remained limited as the primary use of key mining algorithms has been to facilitate the enforcement of constraints in database management systems and these require exact keys.

Another reason for the limited research into key mining is that most open source and proprietary database management systems have developed reasonably effective methods for discovering just the keys of most interest to a user [NAU-14]. There are a number of tools which use SQL queries or specialised algorithms to aid the task e.g. IBM's Information Analyzer [IBM-WEB], Microsoft's SQL Server Integration Services (SSIS) [MS-WEB], or Informatica's Data Explorer [INFOR-WEB]. They take advantage of the relational structures that are imposed on the data at the point of storage, and any constraints, keys or indexes that are placed on tables. They achieve acceptable performance by limiting the search space to attribute combinations with only a small number of attributes [HEI-13], and use feedback from users to target the profiling to look for semantically meaningful keys.

The field of data profiling has risen in prominence again recently however as 'big data' has opened up new opportunities in industry and research. Traditional relational databases have been the dominant method of storing and profiling data for several decades, but as the volume, velocity and variety of data has exploded in recent years they have been challenged by a number of new technologies. To allow the storage of data more cost effectively, the Hadoop file system (HDFS) emerged in 2005 [DEAN-08]. HDFS enables data to be stored with or without structure so that its use can be explored later rather than modelled up front. This is typically called building a 'data lake' or 'data reservoir'. With less built-in structure to leverage in these new repositories, automated data profiling becomes more important again.

Recent key discovery research has mainly looked at profiling all keys from a given relation exactly. This research is based on two assumptions. The first is that all keys must be found. The second is that the data source is consistent and has no “dirty data”. Recent trends in data collection and management have made both of these assumptions less valid.

The assumption that key mining algorithms need to find all keys exactly is potentially over ambitious given the difficulty of the problem and the growing volume of data. Only a small number of keys discovered in a dataset may ultimately be useful as database management systems can only enforce a limited number of constraints efficiently. It is also possible that not all keys that exist in a dataset are semantically meaningful. Some keys may exist by accident, and it requires expertise in the information domain to identify them as accidental.

Finding only exact keys is also arguably becoming less critical. Modern database systems are moving towards parallel architectures based on a cluster of commodity computers called “shared-nothing nodes” (or separate CPU, memory, and disks) connected through a high-speed interconnect [STONE-10]. These parallel databases use horizontal partitioning of relational tables, and partition the execution of SQL queries. Distribution keys are typically more important in these parallel databases than primary keys for efficient query processing. It is desirable for these distribution keys to be unique because it ensures even distribution of data over nodes for processing, but it is not strictly necessary. Approximate keys are also very effective.

“Dirty data” is any data that is inaccurate, incomplete or erroneous. In the database context, an informal definition is data that contains “mistakes such as spelling or punctuation errors, incorrect data associated with a field, incomplete or outdated data, or even data that has been duplicated in the database” [WIK-DIRT]. Key mining has traditionally been used to purge dirty data from databases by ensuring semantically meaningful keys hold exactly. Any data that conflicts with these keys is cleaned or removed. Approximate key profiling is a better choice. For example, a meaningful key may be violated due to not enforcing it, but can never be discovered by looking at all the data.

Data stored with less structure using more recent HDFS based tools (e.g. in data lakes) has the potential to have an even greater proportion of dirty data. It may be desirable to clean the data and model it relationally by using key mining algorithms to ensure exact keys can be enforced. This may not be practical given the volume and volatility of the structure of the data. In this situation, key mining algorithms allow exploratory analysis of the data, and both exact and approximate keys (keys with strength less than 1) are valuable.

2. AIM AND STRUCTURE

The aim of this dissertation is to investigate the following research question :

“When does sampling work well for key discovery in the presence of dirty data?”

Recent research has mainly looked at profiling all keys from a given relation exactly. This assumes that we want to find all keys and that the given data source is consistent. The first assumption is potentially over-ambitious given the difficulty of the problem, the growing volume of data, the available computational resources, and the potential use of all the keys. The second assumption does not seem to reflect modern application requirements, in particular in the context of data integration and dirty data. The veracity of data seems to indicate that a focus on approximate key profiling is a better choice. For example, a meaningful key may be violated due to not enforcing it, but can never be discovered by looking at all the data.

The purpose of this research is therefore to investigate sampling for discovering both exact and approximate keys and determine its robustness in the presence of dirty data.

The research structure is described as follows:

1. Keys and the hypergraph transversal algorithm for key discovery are formally defined
2. Key strength as a measure for approximate keys is defined
3. Strategies for sampling are defined including sequential selection and random selection with and without replacement. Their effectiveness and the effectiveness of a stopping parameter are explored with a set of commonly used benchmark datasets which replicate real world clean and dirty data.
4. An alternative preferential sampling technique is defined and it is compared to random sampling experimentally.
5. A parallelised version of the hypergraph transversal algorithm is defined and its use with both random and preferential sampling is tested using the Apache Spark framework

3. LITERATURE REVIEW

3.1. DATA PROFILING

Data profiling is a broad field that has been researched for several decades. It covers a range of techniques from collecting simple statistical information about attributes to more complex metadata involving multiple attributes. [NAU-14] provides a useful taxonomy of data profiling tasks for understanding the field shown in *Figure 1*. The tasks in the single source branch of the tree are the more researched areas to date, whereas the tasks in the multiple source branch of the tree have only recently emerged. This paper will focus on uniqueness and keys with multiple columns in the single source space.

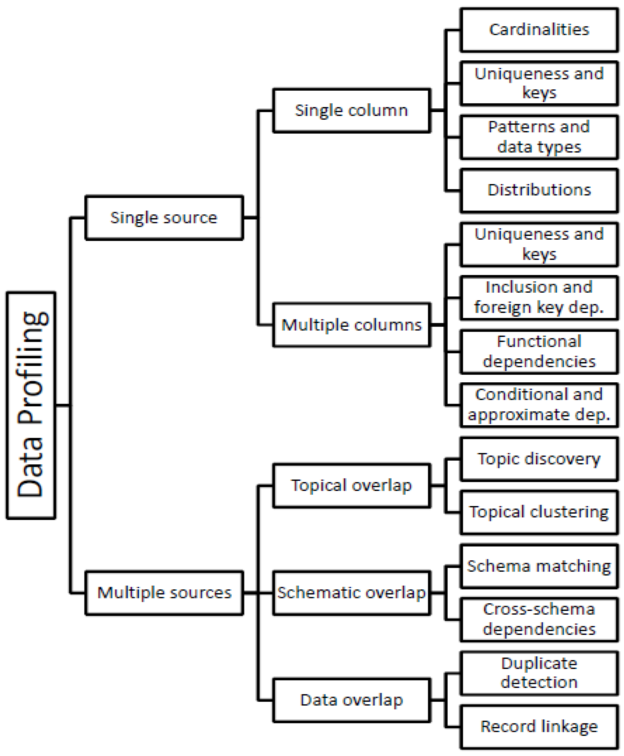


Figure 1 – Data Profiling

The most common traditional use cases for data profiling are data cleansing and query optimisation. Both of these use cases enable the more efficient use of relational databases. Data cleansing is focussed on identifying dirty data. The degree of dirtiness that is acceptable may change over time as data is applied to new use cases. The constraints on data that are implemented when it is first stored may prove to be inadequate over time requiring the data to be profiled and cleansed before being stored again typically with stricter constraints.

Keys are essential to allowing the efficient operation of database management systems. Some examples of how they are used are provided by [SIS-06]: (1) query optimisers use them to provide better selectivity estimates and speed up queries; (2) database administrators (DBA) use them to improve the efficiency of data access via physical design techniques such as data partitioning or the creation of indexes and materialised views; and (3) data integrators use them to automate the merging of data.

Keys supports query optimisation in two ways [NAU-14]. The first is by allowing the implementation of constraints and indexes to be placed on data which the database query optimiser can use. The second is where constraints can't be applied, but the database can still use the statistics gathered to optimise queries.

3.2. KEYS AND FUNCTIONAL DEPENDENCIES

An attribute set K is a key if there are no two different tuples that have matching values on all the attributes in K. A key is defined as:

For a given attribute set $X = \{A_1, A_2, \ldots, A_n\}$ of attributes, the expression kX is called a key for a relation r, if and only if:

$$\forall\, t_1, t_2 \in r : (t_1 \neq t_2) \Rightarrow (t_1[X] \neq t_2[X])$$

A key kX is minimal for r iff X does not contain a proper subset Y that is also a key kY for r.

The discovery of keys is a specific branch of data profiling research. Keys are special cases of functional dependencies (FD) in which the key attributes functionally determine all other attributes of the schema. The difference between a key and a FD is that a key dependency is a functional dependency $X \rightarrow U$, where U is the full set of attributes of the relation [ABITE-95].

Employee	**Department**	**Manager**	**Salary**
Hamilton	Racing	Lauda	50m
Newey	Engineering	Horner	12m
Rosberg	Racing	Lauda	11m

Figure 2 - Example

In *Figure 2*, there are several functional dependencies that hold:

{Employee} → {Department , Manager, Salary}
{Salary} → { Department , Manager}
{Department} → {Manager}
{Manager} → {Department}
{Department, Salary} → {Employee}

Of the functional dependencies listed, only {Employee} and {Salary} are minimal keys.

The example also illustrates that not all keys are of equal value to a user. The first key is clearly intended by design to be a key whereas the second appears to be accidental. Entirely automatic key mining algorithms cannot identify semantically meaningful keys. These may still be valuable for query optimisation however.

Data sets may also be dirty and violate semantically meaningful keys. For instance, the data in the example above may exist in a human resources (HR) system and we want to merge it with the data in a payroll system to form a data warehouse. The payroll system may also have a tuple for Hamilton but with a salary of 52m instead of 50m. Combining the two tuples in a new table that also has the key of Employee would lead to a violation of the semantically meaningful {Employee} key but not the accidental {Salary} key. This situation of dirty data is very common when integrating data from different systems and is only exacerbated when the source systems do not have any uniqueness constraints enforced due to poor database design, or data coming from unstructured sources like perhaps a Hadoop based data lake.

The task of finding all possible keys in a dataset is computationally difficult. The problem of finding the number of all keys of a given relation, and alternatively, the problem of finding the number of minimal keys of a given relation have both been shown by [GUN-03] to be #P-hard. The complexity comes from both the number of attributes and the number of tuples in a dataset. For a relation with *n* tuples, there are 2^n possible keys of length 0 to *n.* For a relation with *t* tuples, the number of comparisons that must be made to check for all keys is $(t^2 - t)/2$.

The problem of finding single keys is less computationally difficult but is still not trivial. A naïve approach would be to take each attribute, sort it and compare each tuple to its preceding tuple for the entire length of the relation. This approach of finding keys by ensuring there are no duplicates can be done with a number of sorting algorithms that have average complexity of *O(t log t)* where *t* is the number of tuples in this case.

Another challenge with finding unique keys is where the input data is too large to fit into the main memory of a single machine. As discussed above, it is increasingly common for data to be stored in distributed fashion on HDFS.

3.3. SAMPLING

An early paper of significance in the sampling space is [KIV-95]. The authors are concerned with approximate dependency inference and develop various error measures to achieve this. The principle is that given the complexity of discovering all keys with full accuracy, it may be acceptable to find a high quality set of “approximate keys”. False keys are still valuable if they have a high enough *strength* according to [SIS-06]. Strength is defined as the number of distinct key values in the dataset divided by the number of entities.

The authors of [KIV-95] are focussed on relational datasets where the number of tuples is much greater than the number of attributes. Their goal is to find an algorithm which has the running time of the form *O(n t log t)* where *t* is the number of tuples in the relation and *n* is the number of attributes. This can be achieved by the fact that a sorting algorithm can be used to

They show that in general, to ensure that all functional dependencies discovered from a sample have a strength exceeding $(1- \varepsilon)$ with probability $(1-\delta)$, the minimum sample size required is $O(t^{1/2} \varepsilon^{-1} (n + \log \delta^{-1}))$ where *t* is the number of tuples and *n* the number of attributes.

The authors only look at random sampling and how to find the smallest possible sample *S* from relation *R* where the keys completely hold in *S* but hold with a given probability of error in *R*. They use random sampling with replacement where the probability of picking any tuple follows a uniform distribution.

As far as the authors of this paper are aware, there is no specific research into key discovery using sampling in distributed environments.

3.4. KEY DISCOVERY

At most, there are $\binom{n}{n/2}$ candidate keys in a relation of n attributes [DEM-80].

TANE is an algorithm that finds functional dependencies in large datasets by partitioning the set of tuples by attribute value [HUHT-99]. It has been designed to scale up in dataset size but not scale out to run over datasets in parallel. The use of partitioning makes it particularly appropriate for finding approximate dependencies. When the size of the dependencies reach half the number of attributes in the relation, the running time becomes exponential.

A data mining approach to discovering keys is combined with using Armstrong tables to validate the results in [LINK-12]. The Armstrong tables computed provide a summarised view of the data to allow a user of the algorithm to easily check if the discovered keys are semantically meaningful.

The hypergraph transversal algorithm used in this paper was first studied for use in discovering functional dependencies in [MANN-92]. The authors first show that a naiive approach to functional dependency discovery with exponential complexity in the best case is not feasible for relations with more than a very small number of attributes.

They develop two algorithms which are better in their best case behaviour than a naiive approach. The first algorithm is the hypergraph transversal algorithm which runs in quadratic time but only for datasets with relatively few attributes. The second algorithm uses sorting and again works very well for small datasets but is still exponential in its worst case behaviour. They leave as an open question the combination of the two algorithms.

3.5. ATTRIBUTE BASED APPROACHES

The work in [MANN-94] is built on in [GIAN-99] which approaches the problem from a data mining perspective and investigates the use of a-priori pruning of attributes in a bottom-up, top-down and hybrid fashion. As in [MANN-94], their goal is to find all minimal keys in a relation using only time that is polynomial in the number of attributes and minimal keys, and sub-quadratic in the number of tuples. The level-wise algorithm they develop still only achieves exponential running time because they must generate repeated subsets.

[ABED-11] looks at a-priori pruning again and focusses on bottom-up pruning. The authors develop an algorithm called HCA which optimises the candidate key generation strategy and incorporates the maximal non-unique discovery part of Gordian.

3.6. TUPLE BASED APPROACHES

The Gordian algorithm [1] developed in 2006 addressed the problem of composite keys directly. Previous research had largely been focussed on the discovery of single keys using brute force approaches. Gordian can be considered to be a row-based technique which scans the relation to create a prefix tree and then traverses this prefix tree to discover maximal non-uniques. The drawback with this approach is that the prefix tree must be held in main memory and may grow as large as the input dataset itself. The algorithm may achieve a large part of its scalability to large datasets through using a sample of the input data. The algorithm has not been designed to be run in parallel.

3.7. HYBRID APPROACHES

The algorithm developed by [HEI-13] is a hybrid of the attribute and tuple based approaches that preceded it. Similar to the HCA algorithm, it uses depth-first pruning but combines this with a random walk strategy. To prune the number of attributes in the graph of possible keys, the algorithm traverses the graph in a depth-first manner until it finds the first unique column combination and then it starts following the boundary between uniques and non-uniques in a random walk manner. The idea is that this allows DUCC to mainly depend on the solution set size rather than on the number of columns.

Unlike previous algorithms, DUCC is also designed to scale out and run in parallel on multiple machines. The authors use Hadoop MapReduce and perform the graph traversal on multiple machines. DUCC uses tuple indexing to perform uniqueness checks faster so this checking of tuples is done with a single MapReduce job and the index structures then distributed across HDFS. The performance is claimed to be up to 631 times faster than Gordian and up to 398 times faster than HCA.

A dynamic version of DUCC called SWAN is developed in [ABED-14]. SWAN incrementally updates the set of keys. Like DUCC, SWAN uses indices to speed up the discovery process, but instead of depending on the size of the initial dataset, the running time of SWAN is dependent on the size of the incremental updates. It is claimed to be one order of magnitude faster than GORDIAN and DUCC. The motivation for developing an incremental algorithm is that database management systems are only able to validate all the user defined constraints after each inserted tuple, and must abort the insertion if it does not satisfy one of these constraints. They cannot find new uniques and non-uniques after a set of tuples has been inserted.

3.8. APACHE HADOOP AND SPARK

Apache Hadoop is an open-source cluster computing software framework for distributed storage and distributed processing of very large data sets. It was designed to be built from commodity hardware to allow more cost effective storage of data while remaining resistant to hardware faults. The framework consists of the following modules: (1) Hadoop Distributed File System (HDFS) – a distributed file system (2) YARN – a resource management platform (3) MapReduce – a programming model (4) Hadoop Common – programming libraries to support the other three modules.

The MapReduce programming model allows users to specify a computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks [DEAN-08]. The map function starts with a set of key/value pairs as input, performs a computation on them and outputs an intermediate set of key/value pairs. The key/value pairs in the intermediate set are grouped together and become the input for the reduce function which typically performs a computation that outputs a smaller set of key/value pairs.

Apache Spark is a more recent open-source cluster computing framework with the same scalability and fault tolerance of Hadoop MapReduce. It has been developed to improve on MapReduce in two key areas: it provides a more flexible programming paradigm and it is significantly quicker in several popular parallel computing applications [ZAH-10]. Spark is able to use a variety of resource management platforms including its own and YARN, and it interfaces with HDFS and other distributed file systems.

Instead of the two-stage data model of MapReduce, Spark uses a programming abstraction called Resilient Distributed Datasets (RDDs) [ZAH-12]. An RDD is a read-only collection of objects partitioned across machines that can be rebuilt if a partition is lost. They stop short of being a shared memory abstraction because they are read-only, but they allow a user's program to load data into memory over a cluster and query it multiple times.

3.9. DATA BENCHMARKING

The authors in [ALEX-13] survey the field of big data testing and benchmarking and conclude there are currently no commonly used methods or frameworks developed specifically for big data profiling. They confirm that the TPC-H framework developed in the 90's and used extensively for benchmarking in relational database research, is still the most widely used with big data research.

4. KEY MINING

4.1. ALGORITHMS

The main idea is to understand how well sampling works for key discovery on potentially inconsistent data sets in terms of the quality of the results and the time spent on computing them. For this, the simplest key discovery algorithm known from the literature is applied to random samples. That algorithm is based on hypergraph transversals, and illustrated in this section, together with the sampling strategies tested.

The Key Mining algorithm is composed of three algorithms:

4.1.1. Disagree Sets

The key mining algorithm finds the disagree sets for the data sample. A disagree set is found for a pair of tuples by comparing the values of each attribute and recording the names of the attributes where the values are different. The pseudo code for the algorithm is shown

```
procedure Disagree Sets
disagree-sets(r) <- $\emptyset$
FOR ALL $t_1 \in r$ DO
    FOR ALL $t_2 \in r$ SUCH THAT $t_1 = t_2$ DO
        dis-ag($t_1$ $t_2$) <- $\{A \in R \mid t_1(A) \neq t_2(A)\}$
        disagree-sets(r) <- disagree-sets(r) U { dis-ag($t_1$ $t_2$) }
    END DO
END DO
```

Figure 3 shows the disagree sets for an example relation:

Tuple #	Employee (E)	Department (D)	Manager (M)	Disagree Set
t1	Alonso	Engineering	Brawn	dis-ag (t1,t2) = {D}
t2	Alonso	Racing	Brawn	dis-ag (t1,t3) = {E,D,M}
t3	Button	Racing	Jordan	dis-ag (t2,t3) = {E,M}

Figure 3 – Disagree Sets

The output of the algorithm is a set of attribute sets. Each disagree set can only contain an attribute once, and the final collection of disagree sets will only contain a single instance of a disagree set even if two different pairs of tuples produce the same disagree set.

In the above example, the first tuple is chosen and then compared to the other tuples by iterating through the relation in order. In the experiments, three different strategies have been used to do comparisons. These strategies can all be used with the complete dataset as in the case of finding all minimal keys that hold on the input, or with a sample of rows:

Strategy A: as in the example above, each tuple is chosen in order and then compared with the other tuples in the order they exist in the relation. After tuple 1 has been compared to all other tuples, tuple 2 is then chosen and compared to all other tuples that come after it in the relation. Each unique pair of tuples is compared only once.

Strategy B: random selection is introduced to the algorithm. First, two tuples are chosen at random and compared. Another tuple is then chosen at random and compared to the previously chosen tuples. This is repeated so that every time a new random tuple is chosen, it is compared to all previously chosen tuples. The random selection is done without replacement so no comparisons are made twice.

Strategy C: all unique pairs of tuple comparisons are chosen then the order in which they are made are shuffled.

The number of comparisons made by the algorithm is *(t*(t-1))/2* where t is the number of tuples.

4.1.2. Minimise Disagree Sets

The Key Mining algorithm produces a collection of disagree sets with no duplicates but in order to reduce the computations of the minimal transversal algorithm into which it feeds, the output can still be reduced. This is done by minimising the disagree sets so that if a disagree set X is a superset of another disagree set (Y) then X is removed.

In the example above, the disagree sets were computed to be {{D},{E,D,M},{E,M}}. While {E,M} is a minimal disagree set, {E,M,D} is not since it contains the proper subset {E,M}. Removing {E,D,M} therefore leaves {{E},{E,M}} as the minimal disagree sets. The pseudo code for the algorithm is shown:

```
procedure Minimise Disagree Sets
min-disagree-sets(r) <- { X ∈ disagree-sets(r) | ¬ ∃ Y ∈ disagree-sets (Y ⊂ X) }
```

4.1.3. Minimal Transversals

The algorithm used is derived from [MANN-92] which used this form of the minimal hypergraph transversal algorithm to discover functional dependencies. The authors in that paper developed the first exact algorithm which computes minimal transversals in hypergraphs in a time which is provably lower than $O(2^n)$. Hypergraphs are a generalization of graphs, where the edges (called hyper-edges) may consist of one or more vertices. A hypergraph is of rank k if the maximum number of vertices in any of the hyper-edges is k. A transversal is a subset $T \subseteq V$ of the vertices that includes at least one vertex from every hyper-edge.

The algorithm takes the minimal disagree sets as its input and outputs a set of minimal transversals which are essentially the minimal keys of the relation. The attribute schema of the relation (V) and the set of minimal disagree sets (H) form a hypergraph, and a vertex set $X \subseteq V$ is called a transversal of H if X intersects every hyperedge of H. A transversal is said to be minimal if none of its subsets is a transversal.

```
procedure MIN-TRANSVERSALS($\mathcal{H} = (V, H)$)
    $Tr(\mathcal{H}) \leftarrow \{\emptyset\}$;
    for all $E \in H$ do
        $Tr(\mathcal{H}) \leftarrow \{X \cup \{A\} \mid X \in Tr(\mathcal{H}) \wedge A \in E\}$;
        $Tr(\mathcal{H}) \leftarrow \{X \in Tr(\mathcal{H}) \mid \neg\exists Y \in Tr(\mathcal{H})(Y \subset X)\}$;
    end for;
    return $Tr(\mathcal{H})$;
end procedure
```

In *Figure 3* above, the minimal disagree sets were computed to be {{D},{ E,M}}. There are no attributes that are in all sets, so all the minimal transversals must be composite keys. The minimal transversals or the minimal sets of keys are {D,E} and {D,M}. If an attribute is in all disagree sets, it must be a single key.

4.2. STRENGTH

There are three key factors that determine the effectiveness of the implementation of a key mining algorithm (1) the number of tuples (2) the number of attributes (3) and the strength of the sets of attributes that form keys.

The strength of a set of attributes is defined as the number of distinct key values in the dataset divided by the number of entities [SIS-06]. It is essentially a proportion of uniqueness of an attribute set as it is the ratio of the number of unique values to the total number of values. For example, if a relation has 100 rows and there are 2 duplicates in a given attribute, then the attribute has a strength of 98%. A key must by definition have a strength of 100%.

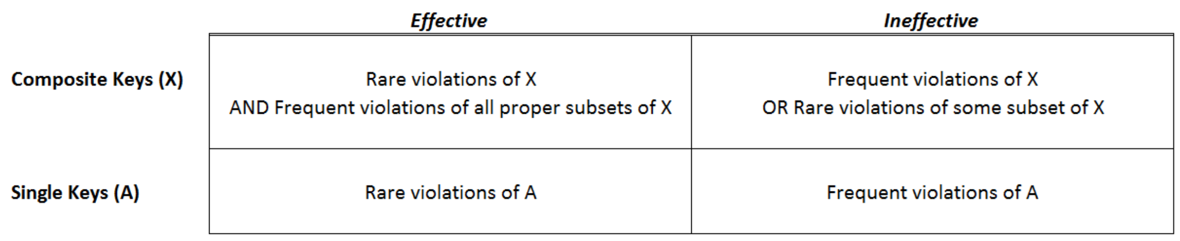
The concept of strength is important in data profiling where the characteristics of the data are unknown initially. In the case of a dataset where an attribute would appear to be a key but there is no enforcement of the uniqueness, it is possible "dirty data" has crept into the relation making the attribute not quite 100% unique. Attributes with close to (but not equal to) 100% strength will have a large effect on the accuracy of any key profiling algorithm.

4.3. COMPOSITE KEYS

A composite key is a key made up of more than one attribute. Each proper subset of a composite key cannot be a key by itself but together the attributes are unique across all the tuples for at least one of the attributes.

Composite keys will be shown to be harder to accurately determine with a random sample than a single key because they require the disagree sets to satisfy more conditions. A single key for attribute X only requires all disagree sets to include the attribute X. A composite key is found if none of its proper subsets is a composite key and for all pairs of different tuples there is some attribute of the composite key on which they have different values.

To show that no proper subsets of a composite key are keys we must find for each proper subset some tuple pair that agrees on the subset, as otherwise the subset itself is a better candidate for a composite key. Finding such tuple pairs becomes more difficult when the attributes involved in the composite key have high strength. These high strength attributes are typical of "dirty data". Some simple rules for when key profiling is effective are outlined in *Figure 4*.



	Effective	*Ineffective*
Composite Keys (X)	Rare violations of X AND Frequent violations of all proper subsets of X	Frequent violations of X OR Rare violations of some subset of X
Single Keys (A)	Rare violations of A	Frequent violations of A

Figure 4 – Effectiveness of key profiling

Even if data is dirty it makes sense to assume that violations of meaningful constraints are the exception. Therefore, single keys that are semantically meaningful are rarely violated. That means the sampling approach is effective in the presence of dirty data. For composite keys, the situation is more difficult: while samples are unlikely to contain rare violations of a meaningful composite key X, not identifying a proper subset as a meaningful key may require larger sample sizes to ensure that a violation of this proper subset is covered in the sample. In other words, the higher the frequency of violations of proper subsets of a composite key, the better the sampling will work for discovering the composite key.

4.4. UPPER BOUNDS ON NUMBER OF MINIMAL DISAGREE SETS

In order to accurately find all the keys in a relation, it is necessary to have good coverage of the minimal disagree sets within the relation. It will be demonstrated that the number of possible minimal disagree sets can be very large, but only a relatively small number of tuples is required to have a sample large enough to include all possible minimal disagree sets.

The maximum number of minimal disagree sets possible for a relation is bounded by both the number of attributes and the number of tuples. A bound is given by the minimum of:
(a) the number of unique combinations possible for the given attributes
(b) the number of comparisons possible given the number of tuples.

The first lower bound (a) for attributes is its power set. A power set is the set of all subsets of a set including the empty set and the set itself. If S is a finite set with n elements then the power set of S is 2^n. For example, the set S = {x,y,z} will produce the power set {{}, {x}, {y}, {z}, {x, y}, {x, z}, {y, z}, {x, y, z}} which has 2^3 = 8 elements.

The diagram on the left below shows the maximum possible number of disagree sets for a given number of attributes. The exponential relationship of the power set is clear and even at a modest number of attributes (20) the number of possible minimal disagree sets is over 1 million.

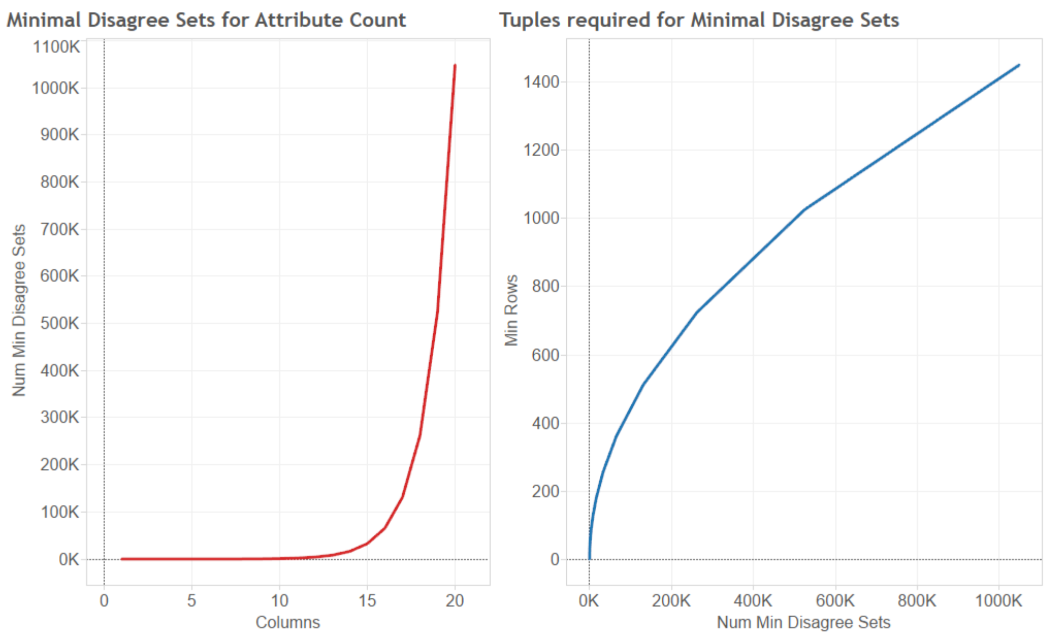


Figure 5 – Minimal Disagree Set Bounds

The second lower bound (b) for tuples is given by the function: *$(t^2 – t)/2$* where *t* is the number of tuples. The sorts of real world datasets we are concerned with in this study typically have a much higher number of tuples than attributes (at least 1000x more tuples than attributes) so the number of tuples is very unlikely to determine the maximum number of disagree sets. This is illustrated by the chart on the right in *Figure 5*. The first chart showed that for 20 attributes, there could be over 1 million minimal disagree sets. The second chart shows that even at 1 million minimal disagree sets, less than 1500 tuples are required to ensure the number of tuples doesn't restrict the maximum number of minimal disagree sets.

5. EXPERIMENTS

5.1. RANDOM SAMPLING EXPERIMENTS

The experiments were performed using data that is synthesized, but is typical of a transactional database. Three datasets from the TPC-H benchmark have been used. Each dataset has 15,000 tuples.

The three datasets are:
(1) Orders - 10 columns with 1 single key and 11 composite keys
(2) Customers - 9 columns with 4 single keys and 3 composite keys
(3) Line Item – 16 columns with no single keys and 181 composite keys

The running time for each of the three algorithms that form the Key Mining algorithm were recorded. An additional modification was made to the disagree sets algorithm to return results at every 10% (or less) of the number of comparisons. This was done by taking the disagree sets at that point, minimising them and computing the minimal transversals. The result from this could then be compared to a "model answer" which had been computed by running the Key Mining algorithm in its entirety earlier. This approach of comparing the output of the key mining algorithm to a model answer also allows the true and false positive rates to be calculated

5.1.1. SAMPLING STRATEGY COMPARISON

Two different strategies for random sampling and one iterative strategy were initially explored. Each of the three strategies is described in more detail above. In summary, Strategy A involves selecting tuples in the order they occur in the dataset, Strategy B involves random sampling without replacement, and Strategy C involves random sampling with replacement. The three strategies were tested with all three TPC-H datasets.

Strategy C was immediately excluded as being viable as it failed to find the exact set of keys even with 100% sampling (i.e. the complete dataset). The strategy uses sampling with replacement so there is no guarantee that all tuples will be compared and duplicate comparisons may be performed. The strategy was considered because it had the implementation benefit of not requiring the previously compared tuple indices to be kept in memory – useful for very large datasets or when exploring different parallelisation strategies. The trade-off with accuracy proved to be unacceptable however. The true positives percent attained by this strategy starts low and remains low for both datasets.

The result for the remaining two strategies are shown in *Figure 6*. The charts show the accuracy of the algorithm in determining the correct keys over increasing numbers of tuple comparisons.

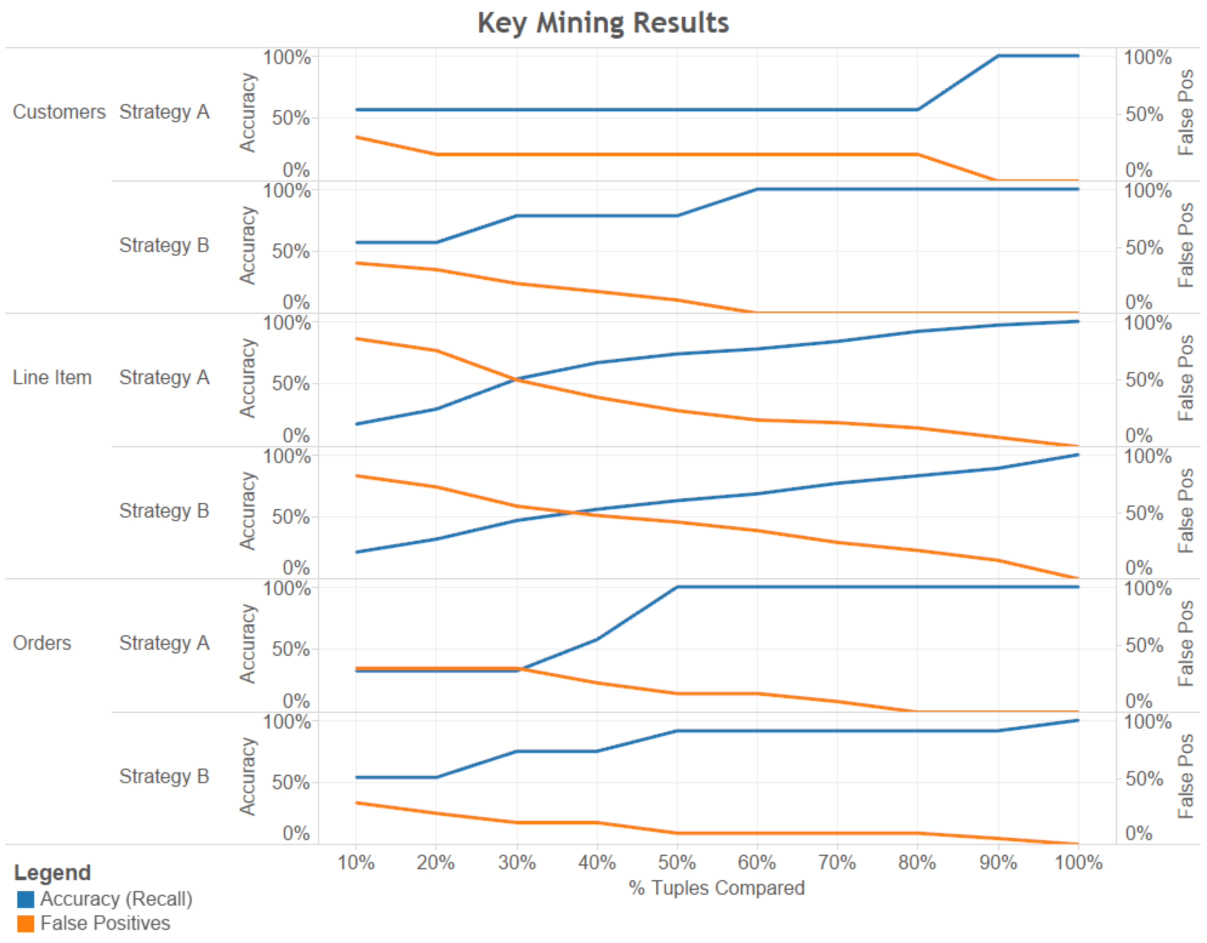


Figure 6 – cumulative accuracy (recall) and false positive percentages at increasing sample rates

Strategy B is more effective than Strategy A for the *Customer* and *Orders* datasets. The accuracy as measured using recall improves earlier and more steadily with the random sampling of Strategy B than it does with the sequential sampling of Strategy A. Strategy B also has a higher initial accuracy than Strategy A for the *Orders* dataset. It achieves over 50% accuracy immediately where Strategy A starts at around 40% and doesn't reach 50% until a sample size of 35% is reached. The patterns for false positives are similar for both strategies for the *Orders* dataset but the number of false positives hits zero earlier for the *Customer* dataset with Strategy B.

The results for the *Lineitem* dataset are similar for both strategies with both starting below 20% accuracy for a sample size of 10%. Strategy A reaches 50% accuracy slightly earlier.

5.1.2. ACCURACY

For the *Customer* and *Orders* datasets for Strategy B, over 50% of the true positives are found at just 10% of the tuple comparisons. Closer inspection of these results shows the initial high accuracy for the *Customer* dataset is from the discovery of the single keys. The discovery of the genuine composite keys takes additional tuple comparisons. The *Customer* and *Orders* datasets contain both single and composite keys whereas the *Lineitem* dataset only contains composite keys. The lower initial accuracy of the *Lineitem* dataset is explained by the fact that it has significantly more keys than the other two datasets and all 181 of them are composite keys.

The results show that small random samples are very effective for finding single keys but larger sample sizes are required with composite keys. This is because a very high proportion of the tuples must be compared to find all the minimal disagree sets that both prove the attributes in the composite key are not single keys by themselves, and that the combination of the attributes leads to unique values across the dataset. This appears to be true regardless of the number of attributes in the composite key. The results from the three datasets showed no correlation between the number of attributes in the composite key and the number of tuple comparisons required to find them. In other words, higher numbers of attributes in the composite key do not make them more difficult to find.

5.1.3. PERFORMANCE

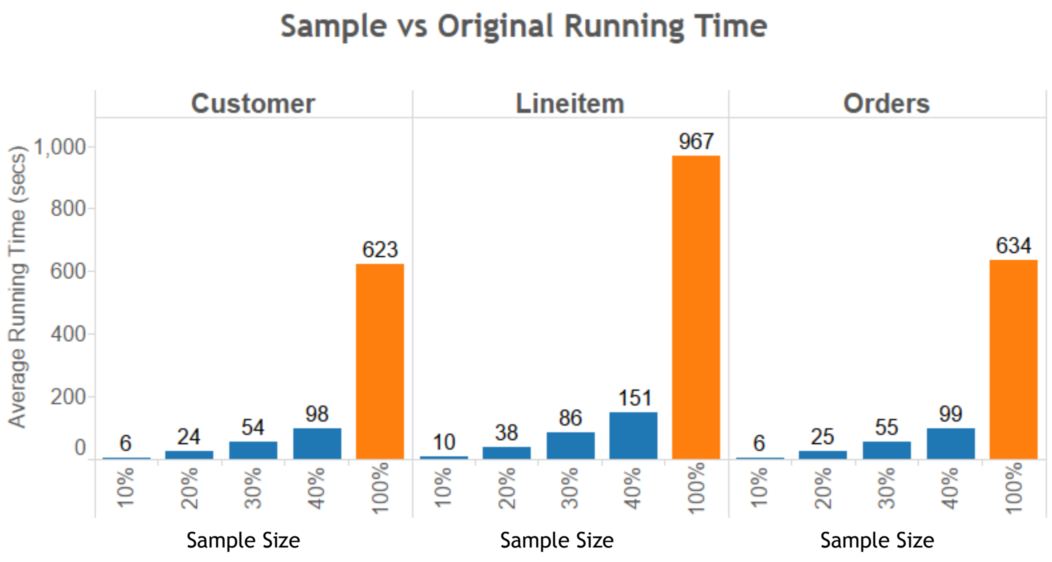


Figure 7 – Total running time for sample sizes 10-40% compared to full dataset (shown as 100%).

The total running time for each sample are shown in *Figure 7*. The proportion of the original running time taken by the algorithm at each sample size is constant across all three datasets. The proportions are shown in *Figure 8*.

Data Sample Proportion	**Running Time Proportion**
10%	1%
20%	4%
30%	9%
40%	16%

Figure 8 – Data sample proportion vs proportion of total running time of original dataset

The performance benefits sampling brings are immediately obvious from the two charts. Even with a relatively large sample size of 40%, the total running time for the sample is only 16% of the running time of the original dataset. With a 10% sample size which has still been shown to given good accuracy, the total running time is just 1% of the original.

The running times for all three parts of the algorithm were recorded independently as well as the total running time. The running time for any administrative tasks performed outside of these algorithms (e.g. data loading and transformation) were also recorded. The disagree sets part of the algorithm accounted for at least 99% of the total running time of the algorithm with datasets of 15,000 tuples or more. The combination of the other two algorithm parts plus all other administrative overhead was less than 1% of the running time so has not been reported separately.

5.1.4. STOPPING PARAMETER

The use of a stopping parameter in the algorithm was investigated. The hypothesis was that the proportion of the number of comparisons (out of the total possible comparisons for the full dataset) made since the last minimal disagree set was discovered would correlate with the accuracy of the discovered keys. If this was true, it would allow us to stop the running of the algorithm when the last discovery count reached a threshold which indicated a desired accuracy of the algorithm. The sample size could then be dynamically determined at run time rather than set to a predetermined value.

The algorithm was run on the three TPC-H datasets while a count was kept of the number of tuple comparisons since the last discovery of a minimal disagree set. The progress of this last discovery count at each level of overall tuple comparisons is shown in *Figure 9* below as a proportion of the total number of comparisons for the dataset i.e. a dataset with 15,000 tuples requires over 110 million comparisons so the last discovery count is shown as a proportion of this.

It can be seen that the proportion of tuple comparisons since the last minimal disagree set discovery is not an effective indicator of accuracy. For instance, for the *Customer* dataset for Strategy A, the comparison proportion climbs to around 60% while the accuracy (measured as recall) remains near 50%. For Strategy B for the same dataset, the counter remains under 20% comparisons for 50% accuracy. The situation is worse for the *Lineitem* dataset. Due to the high number of disagree sets (and keys) in the dataset, the counter is continually discovering new minimal disagree sets so the comparison counter remains near zero.

Accuracy Versus Comparisons Since Last Min DS Discovery

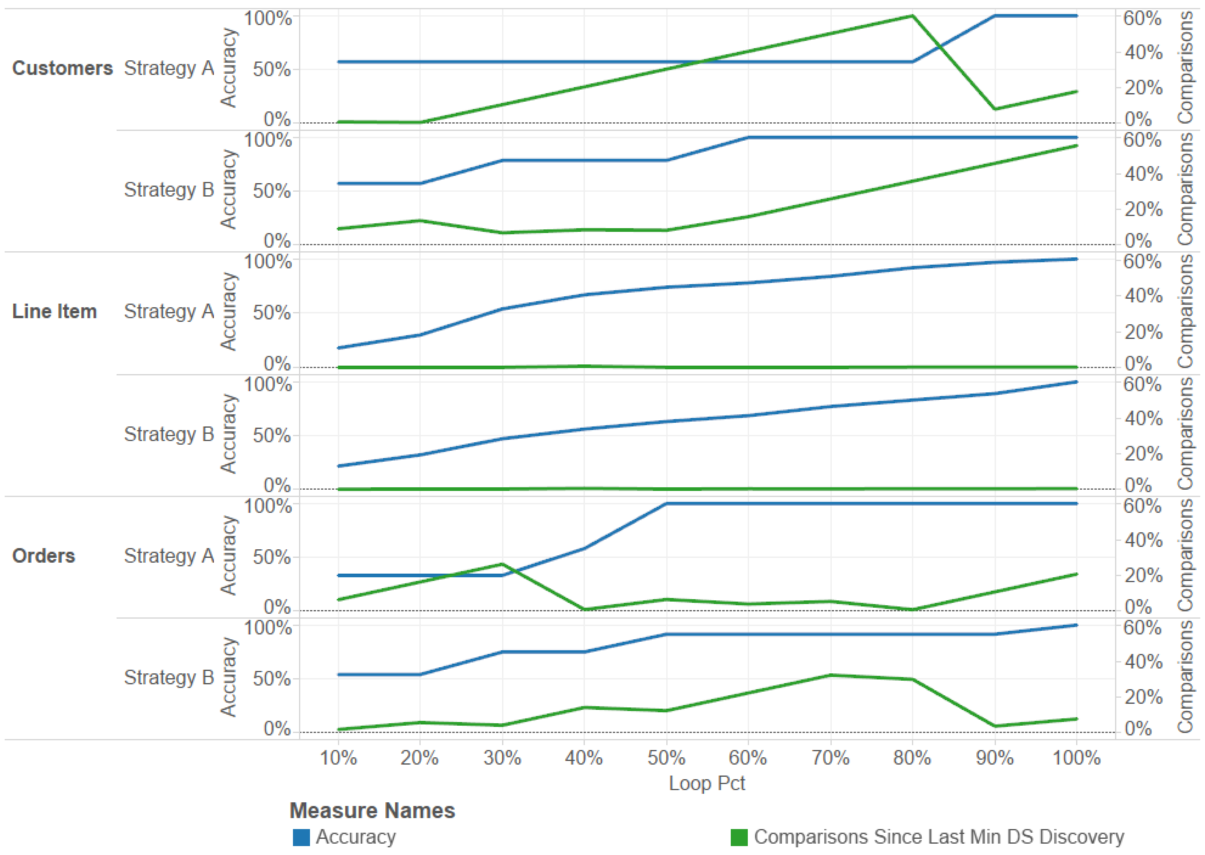


Figure 9 – Stopping parameter experiment

5.2. PREFERENTIAL SAMPLING

Attributes sets with a high strength (but less than 100%) are more likely to form composite keys and they are also more likely to throw false positives when sampling is used. A strategy to improve the performance of the algorithm in the presence of high strength attribute sets using preferential sampling has been tested.

The strategy involves first finding the strength for each attribute in the dataset by sorting on the attribute and comparing every tuple with the immediate next tuple. For the attributes with high strength and therefore a low number of duplicates, the index positions of duplicate tuples are gathered. For attributes with low strength and therefore a low number of unique tuples, the index positions of the unique tuples are gathered. The stored indices are then used to retrieve a sample of tuples from the source dataset. The preferential sample may be supplemented with a random selection of other tuples to ensure a minimum sample proportion (e.g. 10%) is created.

The main determinant of the complexity of the algorithm is the sorting operation it performs. The use of an efficient sorting algorithm will give the preferential sampling algorithm an average complexity of *O(n t log t)* where *n* is the number of attributes and *t* is the number of tuples.

```
Algorithm: Preferential Sampling
Input: source dataset
Output: sample dataset

indices ← Ø

// Calculate strength for each attribute
for each attribute A
    sort attribute
    calculate strengt by pairwise comparison

// Collect duplicates for high strength attributes and uniques for low strength attributes
for each attribute
    sort attribute
    for all tuples where t1= t2 and attribute has high strength (low duplicates)
        add t1 and t2 index to indices
    for all tuples where t1≠ t2 and attribute has low strength (high duplicates)
        add t1 and t2 index to indices

// Ensure minimum number of random samples is picked
while length of indices < min sample size
    add random index to indices
```

The algorithm uses a threshold variable for the minimum attribute strength for collecting duplicates from high strength attributes. A single threshold has been used for both collecting duplicates and uniques, so the maximum attribute strength for collecting uniques from high strength attributes is derived by subtracting the threshold from 1. This is illustrated with an example relation with three attributes: (1) *Employee* with 0.98 strength, *Manager* with 0.52 strength, and *Department* with 0.03 strength. If we used a threshold of 0.95 we would collect duplicates from those attributes with strength greater than or equal to 0.95 (*Employee*), and we would collect uniques from the attributes with strength less than or equal to 0.05 (*Manager*).

There are two aspects to the preferential sampling that have been measured. The first is the simple measure of whether the right keys have been discovered. The second measure is of the strength of the keys discovered against the original dataset. All of the discovered keys will have the maximum strength of 1 in the sample dataset, but to see how representative the preferential sample is, we test their strength against the original dataset. Keys are described below as genuine if they have a strength of 1 in the original dataset.

5.2.1. ACCURACY

	Threshold	*Sample Ratio*	*Min Precision*	*Min Recall*
Customer	0.85	10%	100%	100%
	0.90	10%	100%	100%
	0.95	10%	100%	100%
Lineitem	0.85	11%	41%	33%
	0.90	11%	41%	33%
	0.95	11%	41%	33%
Orders	0.85	27%	92%	100%
	0.90	27%	92%	100%
	0.95	10%	83%	83%

Table 10 – Results for preferential samples at different thresholds

Figure 10 shows a selection of results for each of the TPC-H datasets tested. Three different thresholds between 0.85 and 0.95 are shown for each dataset. The sample ratio (out of the original 15,000 tuples) is shown for each threshold value. Thresholds lower than 0.85 would lead to sample ratios over 75% for some of the datasets so were excluded. A minimum sample ratio of 10% was used and randomly chosen tuples were added to the sample if the threshold didn't lead to enough tuples being selected.

It can be seen that each dataset produces a different sample ratio for the same threshold. This is determined by the distribution of attribute strengths in the relation. Relations with more attributes with strength closer to 0.5 (but still within the threshold) will produce a higher sample ratio. The attribute strengths for the source datasets (all with 15,000 tuples) are shown.

Customers

Index	*Strength*
0	1.0000
1	1.0000
2	1.0000
3	0.0017
4	1.0000
5	0.9928
6	0.0003
7	0.9999
8	0.0001

Lineitem

Index	*Strength*
0	0.2493
1	0.9635
2	0.5169
3	0.0005
4	0.0033
5	0.9907
6	0.0007
7	0.0006
8	0.0002
9	0.0001
10	0.1663
11	0.1635
12	0.1662
13	0.0003
14	0.0005
15	0.9919
16	0.0001

Orders

Index	*Strength*
0	1.0000
1	0.9203
2	0.0002
3	0.9999
4	0.1601
5	0.0003
6	0.0667
7	0.0001
8	0.9997
9	0.0001

Figure 11 – Attribute strengths for source datasets

Using the preferential samples, the key mining algorithm was able to identify genuine single keys immediately, regardless of the sample size or threshold used. As such, any inaccuracy in the following results relate either to a composite key or a false positive on a single key (rather than a missed genuine single key).

The *Customer* dataset produced the best performance of the three with preferential sampling. At each of the three threshold levels it produced a sample ratio below the minimum, so randomly selected tuples were added to bring it up to the minimum. It achieved maximum precision and recall by finding all 7 genuine keys at all three of the threshold values. With a 10% random sample, the same dataset was shown to achieve just over 50% precision.

The *Lineitem* dataset was the worst performing of the three with preferential sampling. The precision (41%) and recall (33%) remained the same at all three threshold values. It can be seen from *Figure 11* that the source *Lineitem* dataset has a high proportion of attributes with either very high or very low strength leading to a large number of composite keys (181) and no single keys.

The *Orders* dataset has performance in the middle of the other two datasets. It achieved its best accuracy of 92% precision and 100% recall at the lowest threshold of 0.85 while searching for the 12 genuine keys. This threshold produced a large sample dataset however with a ratio of 27%. The 0.90 threshold produced the same result but the higher 0.95 threshold dropped the accuracy slightly to 83% for both precision and recall.

The accuracy results for the *Customer* and *Orders* datasets show the biggest improvement at small sample sizes. *Customer* improves from 57% recall at a 10% random sample size to 100% for a 10% preferential sample. *Orders* improves less at the 10% sample size but it is able to achieve 100% recall at a 27% preferential sample size where it requires a random sample size over 50% to achieve the same.

5.2.2. KEY STRENGTH

The number of genuine keys discovered is one way of assessing the accuracy of preferential sampling but it doesn't indicate the quality of the keys discovered. The strength of the keys discovered from the sample as measured against the original dataset is a better indicator of how representative the sample is. A given source dataset may have a number of keys which are accidental or which might not exist with just small variations to the data.

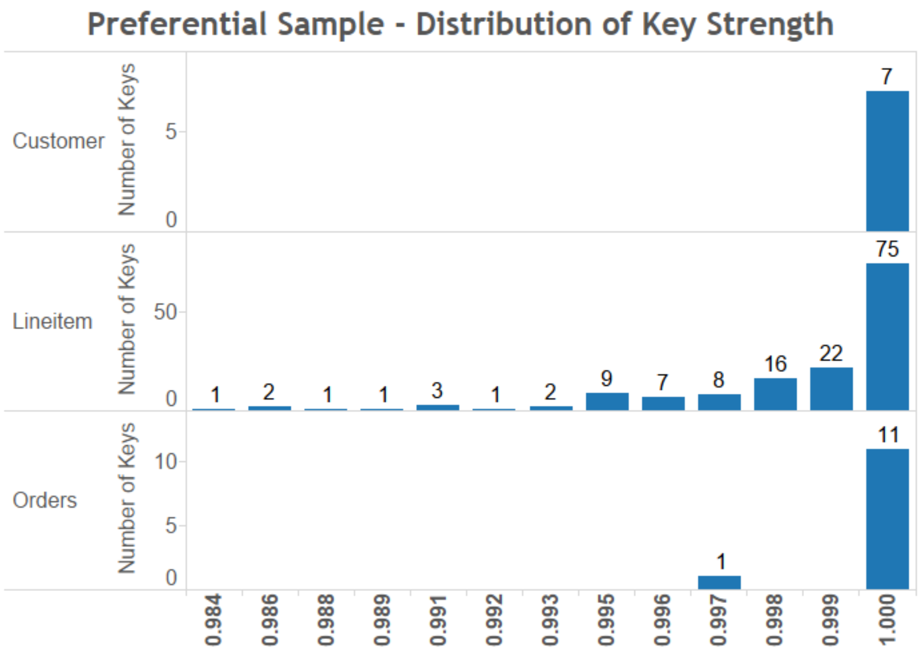


Figure 12 – Distribution of strength of keys found from preferential samples at 0.95 threshold

The *Lineitem* source dataset has 181 genuine keys. Using a preferential sample of *Lineitem* at threshold 0.95, the key mining algorithm discovers 148 keys with a distribution shown in *Figure 12*. It can be seen that the minimum strength of the discovered keys is 0.984 but the majority of keys discovered (75) are genuine keys in the source dataset as they have strength equal to 1. The distribution for the *Orders* dataset is also shown. Of the 12 keys discovered, 11 of them are genuine keys in the source dataset and only one has strength below this at 0.997. The two distributions shown represent the worst case scenarios for the preferential samples tested. The Customers dataset is not shown because all keys discovered from the preferential sample have full strength.

To compare the effectiveness of preferential sampling to random sampling, the strengths of keys from random samples with equivalent sample ratios to the 0.95 threshold datasets are shown in *Figure 13*. Overall, the preferential sample results are better for an equivalent sample size. The random sample for *Customer* misses a genuine key and discovers a key of strength 0.998 where the preferential sample gets 100% precision and accuracy. The *Lineitem* random sample discovers 134 keys with the minimum strength only slightly lower than the preferential sample which discovered 148 keys (the original dataset has 181 keys). The big improvement for the preferential sample is that 75 of the keys it discovered were genuine versus 33 for the random sample. The *Orders* performance is similar between the samples but the preferential sample discovers an extra 2 genuine keys (out of a possible 12) that the random sample doesn't.

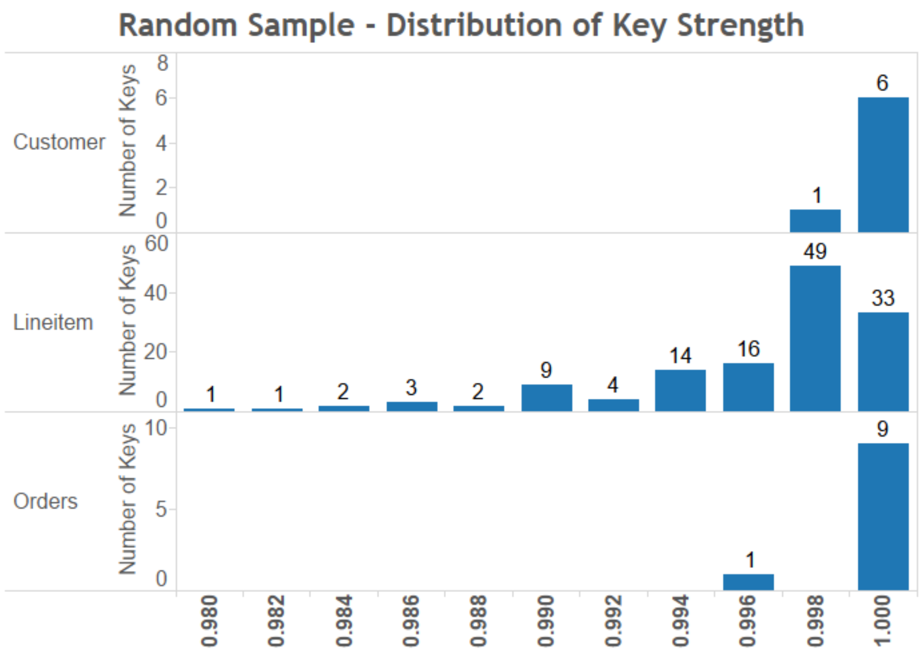


Figure 13 – Distribution of strength of keys found from random samples

5.2.3. PERFORMANCE

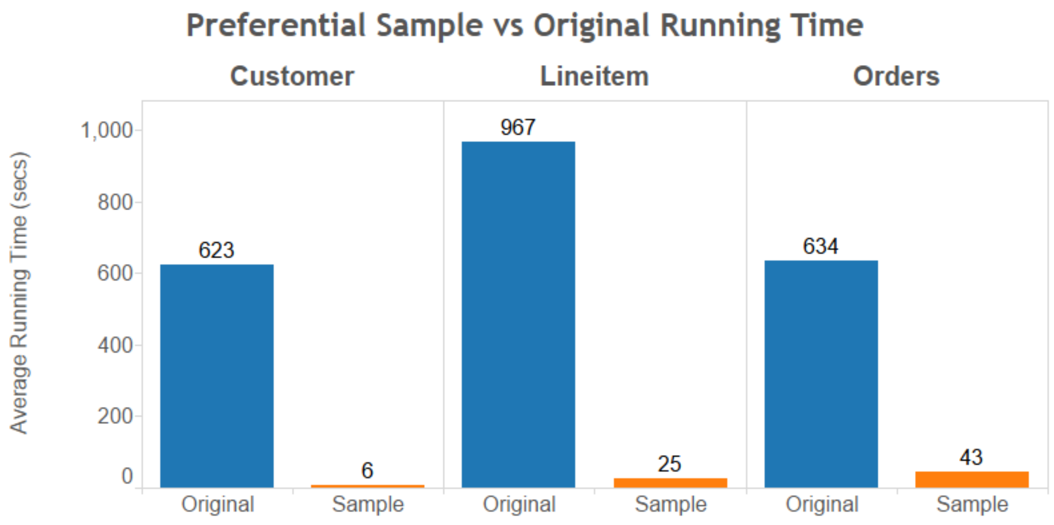


Figure 14 – Average running time for original datasets versus worst case preferential sample

The average total running time for the original dataset and its preferential sample are shown in *Figure 14*. The running times for the preferential sample with threshold of 0.85 have been used as they have the best accuracy of the three preferential samples described above, and also have the longest running time. The *Customer* preferential sample has the best performance improvement with the sample taking just 1% of the running time of the original. The *Lineitem* dataset has the next best performance with the sample taking 3% of the original running time. *Orders* has the least improvement with the sample taking 7% of the original running time. This is because it has a sample ratio of 27% where the other two datasets have sampling ratios of 10-11%.

6. PARALLELISATION

Different methods of parallelisation of the key mining algorithm have been tested using the Apache Spark framework. Spark was chosen over MapReduce because it performs better and because it is also considerably more flexible in its programming paradigm. This has allowed experimentation with different parallelisation approaches. Partitioning in the Spark context means splitting the data between separate nodes with a "shared nothing" architecture. An algorithm must be run on each partition of data entirely independently of the other nodes with no communication between them. The results are then collected and joined at the end.

There are a number of ways to parallelise the key mining algorithm. The three algorithms that form the whole algorithm could each be parallelised separately or in combinations of their parts. The input to each of the algorithms could also be partitioned in different ways. For example, the initial input to the disagree sets algorithm could be partitioned by tuple or by attribute.

One parallelisation approach tried was running all three algorithm parts within a partition and collecting the keys produced by each partition at the end. The accuracy of the results from this approach proved to be poor so it was discarded. All other parallelisation approaches focussed on parallelising the disagree sets part of the algorithm only.

The main input partitioning approach investigated was partitioning by tuple. Partitioning by attribute was briefly explored but initial accuracy was poor so the approach was left for future study. Partitioning by tuple requires just the disagree sets algorithm to be partitioned. The disagree sets algorithm is applied to each partition of the dataset and the disagree sets produced from each partition are combined before being minimised and run through the minimal transversal algorithm on a single node.

The following experiments first explore the performance benefits parallelisation offer by itself, then the combination of parallelisation and sampling are investigated. We introduce the additional variable of the number of partitions and look at its effect by repeating a number of previous experiments, so to keep the analysis simple we focus on just the *Customer* dataset.

6.1. PARALLELISATION EXPERIMENTS

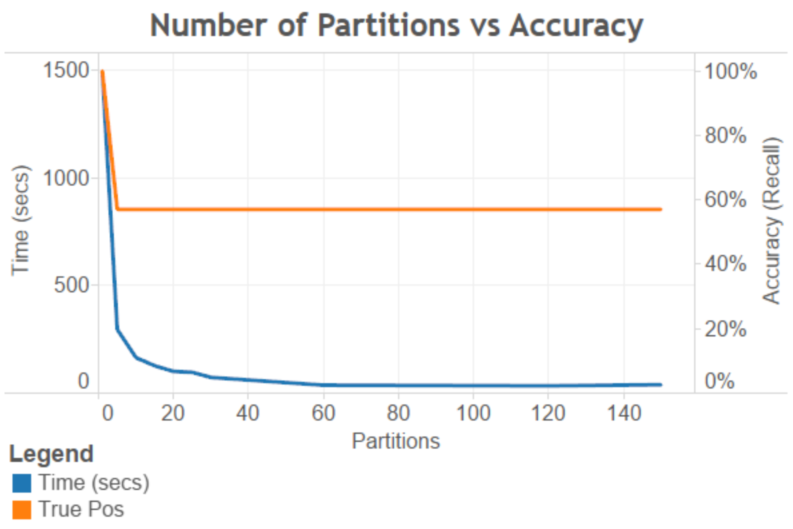


Figure 15 – total running time vs accuracy with increasing numbers of partitions

The results are shown in *Figure 15* for the parallel version of the algorithm running on the full *Customers* dataset. The Customer dataset has 7 keys – 4 single keys and 3 composite keys. The change in accuracy between no parallelisation (1 partition) and parallelisation with increasing numbers of partitions is clear with the accuracy (recall) immediately dropping to 57% as all the 4 single keys are still correctly identified but none of the correct composite keys are found. The accuracy does not change as the number of partitions increase however, as the algorithm continues to find the correct single keys but none of the composite keys at higher numbers of partitions. The running time for the algorithm improves substantially however right up to 120 partitions at which point the running time is only 2% of the non-parallelised running time.

The strength of the keys found measured over the full dataset were also recorded. The minimum strength of the discovered keys was still high at 0.993.

6.2. PARALLELISATION AND SAMPLING EXPERIMENTS

The results from running the algorithm on the full dataset show that parallelisation offers large benefits in terms of decreasing the running time, but with a trade-off in accuracy. The parallel algorithm has been run on a selection of sample datasets to investigate whether we can further improve performance without sacrificing too much accuracy.

Figure 16 shows the results of running the parallel algorithm over a 10% random sample from the *Customer* dataset, and a preferential sample with threshold 0.85 (which also has a sample ratio of approximately 10%).

Dataset	Partitions	Recall	False Positives	Min Key Strength
Random	1	57%	43%	0.998
	5	57%	33%	0.993
	10	57%	33%	0.993
	15	57%	33%	0.993
	20	57%	33%	0.993
	25	57%	33%	0.993
Preferential	1	100%	0%	1
	5	100%	0%	1
	10	100%	0%	1
	15	67%	33%	0.999
	20	67%	20%	0.999
	25	71%	17%	0.997

Figure 16 – Random and preferential samples of the Customer dataset run using the parallel algorithm

The running time for the samples is not shown in the table but both samples ran in approximately 8 seconds with no parallelisation (1 partition). This is approximately 2 seconds slower than the ordinary sequential algorithm because of the additional overhead Spark adds. With parallelisation (number of partitions > 1) the algorithm ran in 2 seconds regardless of the number of partitions. This is because it had hit its minimum running time as determined by the overhead of distributing the data amongst the nodes to run the computation. With a larger sample size, there is no reason to expect we wouldn't see a speed-up (number of partitions vs running time) relationship as we saw with the full dataset. The parallelised sample therefore achieves a running time which is 0.3% that of the original full dataset.

Looking at the accuracy of the two samples, the random sample appears to have similar accuracy (recall) to the parallel algorithm run over the original dataset. It finds the single keys but is unable to find the composite keys either with or without parallelisation. The preferential sample is able to find all the keys immediately with no parallelisation, and continues to find all the keys at 10 partitions before dropping in accuracy at 15 partitions. The strength of the keys it finds remains higher than either the parallelisation on the full dataset or the random sample.

7. CONCLUSION

Previous key mining research has mainly been focussed on profiling all keys from a given relation exactly. This is based on the assumptions that all keys must be found and that the data source is consistent and has no “dirty data”. Recent trends in data collection and management have both changed the way keys are used and increased the potential for dirty data. Given these changes, it is becomes more desirable to find both exact and approximate keys with a measurable strength.

In this paper, sequential and random sampling strategies were investigated and random sampling without replacement was determined to be the most effective of them. The results showed that small random samples (around 10%) are very effective for finding single keys but larger sample sizes are required with composite keys.

Sampling was shown to have a large performance benefit over profiling a full dataset with a sample size of 10% needing just 1% of the running time of the algorithm on the original dataset. A sample size of 40% was shown to still only require 16% of the running time.

Attributes sets with high strength (but less than 100% strength) were found to be more likely to form composite keys and throw false positives when key profiling. These high strength attributes are typical of “dirty data”. A preferential sampling strategy to improve the performance of the algorithm in the presence of high strength attribute sets was defined and tested experimentally. The strategy has a favourable running time complexity of $O(n\ t\ log\ t)$ for building a sample.

The preferential sample was shown to improve the accuracy (measured as recall) of the number of keys found for a sample of a given size compared to random sampling. Accuracy of 100% was achieved with smaller sample sizes using preferential sampling compared to random sampling. The strength of approximate keys found with preferential sampling was also improved compared to random sampling. The minimum key strength was higher and the number of keys with close to 100% strength was greater.

Finally, the parallelisation of the algorithm in the Apache Spark framework was investigated. Parallelisation by itself was shown to provide a significant decrease in the running time but with the trade-off of decreased accuracy. When combined with random sampling the running time was decreased again but with no further loss of accuracy. When parallelisation was combined with preferential sampling, it also shared the same benefits of decreased running time with random sampling but the accuracy improved. Using parallelisation and preferential sampling it was possible to discover all keys accurately but with a running time that was less than 0.5% that of running the sequential algorithm running on the full dataset.

8. FUTURE WORK

The combination of preferential sampling and parallelisation has shown promise as an area for further research. The potential performance improvement from the combination of the two could be tested on much larger datasets using HDFS and Spark.

Another area of potential research is the relationship between the minimum strength of discovered keys and the threshold used by the preferential sampling algorithm. It would be very useful to be able to guarantee a minimum key strength by the choice of threshold. It would allow a user of the algorithm to make an informed choice about the tradeoff between higher guaranteed minimum strength with a lower threshold versus the larger sample size and longer running time it would require.

9. REFERENCES

[AGRA-08] Agrawal, Rakesh, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, AnHai Doan et al. "The Claremont report on database research." ACM Sigmod Record 37, no. 3 (2008): 9-19.

[ALEX-13] Alexandrov, Alexander, Christoph Brücke, and Volker Markl. "Issues in big data testing and benchmarking." In Proceedings of the Sixth International Workshop on Testing Database Systems, p. 1. ACM, 2013.

[ARM-15] Armbrust, Michael, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng et al. "Spark SQL: Relational Data Processing in Spark." SIGMOD, 2015.
[DEAN-08] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. Commun. ACM, 51(1):107–113, 2008.

[DEM-80] Demetrovics, Janos. "On the equivalence of candidate keys with Sperner systems." Acta Cybernetica 4, no. 3 (1980): 247-252.

[FLORA-14] Floratou, Avrilia, Umar Farooq Minhas, and Fatma Ozcan. "Sql-on-hadoop: Full circle back to shared-nothing database architectures." Proceedings of the VLDB Endowment 7, no. 12 (2014).

[GIAN-99] Giannella, C., and C. Wyss. "Finding minimal keys in a relation instance." (1999).

[GUN-03] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. ACM Trans. Database Syst., 28(2):140–174, 2003.

[HUHT-99] Huhtala, Ykä, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. "TANE: An efficient algorithm for discovering functional and approximate dependencies."The computer journal 42, no. 2 (1999): 100-111.

[IBM-WEB] http://www-03.ibm.com/software/products/en/ibminfoinfoanal

[IMP-WEB] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[INFOR-WEB] https://www.informatica.com/data-profiling.html#fbid=QtXv_UvBwsh

[KIV-95] J. Kivinen and H. Mannila. Approximate dependency inference from relations. Theoretical Computer Science, 149:129–149, 1995.

[LINK-12] Link, Sebastian, and Mozhgan Memari. "Schema-and data-driven discovery of SQL keys." Journal of Computing Science and Engineering 6, no. 3 (2012): 193-206.

[MANN-92] Mannila, Heikki, and Kari-Jouko Räihä. The design of relational databases. Addison-Wesley Longman Publishing Co., Inc., 1992.

[MANN-94] Mannila, Heikki, and Kari-Jouko Räihä. "Algorithms for inferring functional dependencies from relations." Data & Knowledge Engineering 12, no. 1 (1994): 83-99.

[MS-WEB] https://msdn.microsoft.com/en-us/ms141026.aspx

[NAU-14] Naumann, Felix. "Data profiling revisited." ACM SIGMOD Record 42, no. 4 (2014): 40-49.

[SIS-06] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: Efficient and scalable discovery of composite keys. In Proceedings of the International Conference on Very Large Databases (VLDB), pages 691–702, Seoul, Korea, 2006.

[STONE-10] Stonebraker, Michael, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. "MapReduce and parallel DBMSs: friends or foes?." Communications of the ACM 53, no. 1 (2010): 64-71

[THU-09] A. Thusoo, J. S.Sarma, N. Jain, Z. Shao, P.Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy: Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB 2(2), pp. 1626-1629 (2009)

[WAHL-04] Wahlström, Magnus. "Exact algorithms for finding minimum transversals in rank-3 hypergraphs." Journal of Algorithms 51, no. 2 (2004): 107-121.

[WIK-DIRT] http://en.wikipedia.org/wiki/Dirty_data

[WIK-NP] http://en.wikipedia.org/wiki/Sharp-P

[ZAH-10] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010, June). Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (pp. 10-10).

[ZAH-12] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012, April). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association.