ROW MERGE: DATA REDUCTION THROUGH EXPANSION INTO POSSIBLE WORLDS FOR DATA SUSTAINABILITY

by

Peter Mahhov

Supervisor: Prof. Dr. Yannis Velegrakis
Co-supervisor: Dr. Ioana Karnstedt-Hulpus

A thesis submitted in conformity with the requirements
for the degree of Master of Science in Computing Science

Department of Information and Computing Sciences
Utrecht University

Row Merge: Data reduction through expansion into possible worlds for data sustainability

Peter Mahhov
Master of Science in Computing Science
Department of Information and Computing Sciences
Utrecht University
2025

Abstract

We have created *Row Merge* — a new way to save storage space in large datasets while minimizing the loss of information. In recent years, the amount of data collected in all aspects of life has increased not only beyond our ability to process it all, but to even store it. Existing data reduction methods mainly revolve around either the calculation of summary statistics or the deletion of less important components, both of which permanently erase attribute relationships. Our approach involves the merging of similar rows by replacing differing cells with null values, which allows information to be preserved as uncertain that would have instead been lost if tuples had been deleted. A database reduced in this manner will never give false negative answers to queries compared to the original. However, Row Merge does introduce false positive query answers.

In this work, we have developed the theory behind the informational value of incomplete databases and designed several data reduction algorithms using this principle. We have created quantitative metrics to evaluate the amount of information in an incomplete table without the need to possess the original, and evaluated the quality and performance of several different novel approaches on both real and synthetic datasets. We have discovered the ideal use cases for each new algorithm, and showed that Row Merge surpasses deletion in preserving information after data reduction in all the real-world datasets we tested.

Contents

Chapter 1

Introduction

We are collecting more data than we can effectively store and use — the current situation is being called a ”data deluge”, a flood of information [DGM+23] [Mil19]. Global data collection has long surpassed the total effective storage capability. For example, in 2018, the total data created in the United States was 6.9 ZB (Zettabytes, equivalent to a billion TB), while the amount stored was just 0.9 ZB [GRR19]. The amount of data is predicted to only increase by 2025, with 3.4 ZB stored out of 30.6 ZB generated in the US [GRR19]. Worldwide, the numbers are similar: generated data is predicted to rise from 33 ZB in 2018 to 175 ZB by 2025 [GRR19], leaving storage capabilities far behind. This disparity between data generation and storage highlights the need for data sustainability — the ability to manage, store, and utilize data in a way that minimizes waste and inefficiency while maximizing its utility and long-term value [She18].

There are significant benefits from better data sustainability. First, we could effectively use larger datasets that normally would not be possible due to data storage constraints [Mil19]. Second, there could be fewer resources devoted to management and processing when the data involved is reduced, which can significantly lower the costs involved for both businesses and research institutes [Mil19] [KS17]. Third, the utility value of large data collections decreases not only with time, as the information becomes stale and loses value, but also with the amount, as more datapoints become irrelevant when seeing the same observations again and again [KS17]. Fourth, while storage for growing databases may be seen as relatively cheap through the services offered by Cloud providers, the usage of the stored data may turn more expensive over time due to input latency and retrieval costs [KS17]. Finally, there are potential environmental effects involved: by 2025, 49% of all data stored by both consumers and organizations is expected to be held within the public cloud [GRR19]. Data centers are predicted to consume up to 30% of all global energy generation by 2030 [AE15] and it is estimated that the data center industry will account for 8% of carbon emissions worldwide by then due to this massive energy consumption [CZH+22]. Therefore, making the best use of our limited storage capabilities is paramount to being more effective and sustainable, on both a global and organizational scale.

Businesses nowadays address this issue by resorting to suboptimal methods and absorbing the costs. The ease of hoarding massive amounts of data often results in much of it being stored indefinitely, never accessed after its initial collection [KS17]. When companies do attempt to manage this data, their common solutions to rising storage and management costs involve either deleting data to reduce volume or limiting data collection from the start. However, these approaches risk losing potentially valuable information [KS17]. While data can also be compressed for long-term storage, the compression renders it unintelligible, making it unusable for immediate analysis and decision-making until it is decompressed [Ahm19]. Consequently, compression

does not reduce the effort needed to process and extract knowledge from the data — it merely postpones it.

We have created a new way of data reduction that addresses some of these issues. Our solution, called *Row Merge*, is a novel approach to saving storage space in large datasets while minimizing the loss of valuable information. Instead of deleting data, which permanently erases it, Row Merge combines similar rows by replacing differing individual cells with null values. This preserves the lost information as uncertain rather than removing it outright. For example, when two rows are merged into one, no data is entirely lost: a database reduced using Row Merge will never produce false negatives in response to queries compared to the original dataset. In contrast, deletion permanently removes information, which can lead to false negatives when querying the lost data. That said, Row Merge may introduce errors in the form of false positives. These can occur when queries involve data represented by null values in the reduced database, reflecting the inherent uncertainty of the preserved information.

Developing an effective method for database reduction through row merging involves addressing several technical challenges in design and implementation. First, good Row Merge algorithms must be efficient, i.e. capable of processing data quickly enough to provide timely results. They should also scale effectively as the size of the dataset increases, ensuring practicality for large-scale applications. Second, the algorithms must not just be fast, but they should deliver good solutions as well. For that, it is essential to establish a robust framework for evaluating the quality of databases that contain null values, such as those produced by Row Merge algorithms. This means defining clear criteria to assess whether a database has been reduced well — retaining as much useful information as possible while achieving significant storage savings. Establishing these criteria is critical for guiding the development and refinement of the algorithms. These challenges must be overcome to ensure that Row Merge is both practical and reliable in real-world use cases.

To address the challenges behind developing, evaluating, and optimizing Row Merge data reduction, we created a quality metric and a range of algorithmic approaches. Our quality metric, called *relationship counting*, is grounded on the theory of incomplete databases. A relationship is a pair of column-value (attribute-value) pairs that can appear together in the same row. We classify relationships as certain, possible, or impossible, depending on the incomplete database we are assessing and the domains of its columns. By counting the relationships, we can reliably measure how much information has been preserved or lost after nulling cells to create a reduced database. On the algorithmic side, we adapted standard approaches — exhaustive, greedy, and randomized algorithms — to the specific requirements of the Row Merge problem. To mitigate the limitations of these approaches, we introduced improvements: small additions like pruning that optimized the classic algorithms themselves, and larger changes that became separate algorithms. The Sorted Order algorithm was created by having the exhaustive algorithm consider the most promising possibilities first. This sorted order was also applied to the new Merge Greedy algorithm that commits to the best available merge at each step. Finally, we developed a fundamentally novel approach that focuses on merging entire rows based on their overall similarity, rather than making decisions at the level of individual cells.

This work is organized as follows. In Section 2, we review existing methods for data reduction, providing context for our contributions. Section 3 introduces our Row Merge approach with a motivating example, illustrating its potential benefits. In Section 4, we formalize the problem with definitions and mathematical formulations and detail the quality metric we developed to evaluate reduced databases. Section 5 presents algorithms designed to solve the problem effectively: applications of the classical problem-solving methods to tree-expansion algorithms, modifications to them, and the fundamentally different pair-similarity approach to row merging. In Section 6, we assess the scalability and quality of our algorithms using both synthetic and real-world datasets. Finally, Section 7 summarizes our findings and highlights avenues for future research.

Chapter 2

Related Work

Data reduction is a field that covers the various methods that are used on datasets to reduce their size in terms of volume while reproducing specific analytical results acquired from the original dataset on the new reduced one [PdMCD24]. In this work we are studying the storage of structured data — information that is organized in a pre-defined manner with fixed fields, such as a relational table [Ahm19]. Common categories of data reduction techniques include deletion, caching, summarization, and compression. In this section, we provide an overview of the most frequently used structural data reduction techniques under each category and also explain the theory of incomplete databases, which is the underpinning of our novel data reduction approach.

2.1 Reduction through Deletion

The most intuitive way to reduce space in a dataset is to delete elements from it until the storage constraints are fulfilled. However, it is not as simple as it could seem, as one needs to determine the deletion factor out of the many different applicable data quality attributes or decision heuristics [WS96]. This decision could be based on the accuracy, relevancy, representation, or accessibility of the data, with surveys having found over a hundred differently worded criteria used by people working in industry, such as timeliness, reliability, completeness, accessibility, source traceability, and more [WS96]. It is safe to say that there is no singular standard by which deletion is performed.

A common but naive deletion approach is discarding data *upstream*: not recording all of the data that is produced. Such can be often seen in sensor arrays that are set to transmit at a deliberately slower rate, or any other logging that is done in intervals instead of continuously [KS17]. For example, if a sensor is set to measure at one-tenth of its maximal rate, then 90% of the potentially recorded data is already lost at the source. This could very likely result in possibly valuable information — such as groups of sequences with interesting patterns [Ahm19] — being lost, which makes the choice of deletion logic all the more important.

The exact best deletion method can strongly depend on the application that it is used for. For example, on some scientific measurement data that is destined to be aggregated later down the line, the periodic loss of measurements would likely have a vanishingly small effect when compared to the error already inherent in taking physical observations [KS17]. However, in some fields, such as medical imaging, the recorded information is significantly more sensitive to particular details [WKH$^+$20]. Furthermore, there may even be legal constraints restricting the data that can be deleted and mandating what must be kept [Mil19].

An *amnesia strategy* is a method by which a database management system can decide which parts of the

database are less valuable than the rest and thus can or should be forgotten [KS17]. The simplest amnesia strategy is the *First In First Out (FIFO)* algorithm. Just like a buffer, FIFO keeps a steady flow of information available without keeping any records about past data. This is typically used for streaming database applications [KS17]. There are other time-based or *temporally-biased* strategies available. For example, a probabilistic method similar to the reservoir sampling technique [Vit85] (see Section 2.3.2) exists called the *Uniform-amnesia algorithm*. In this algorithm, after each new batch of datapoints is added to a dataset an equal amount of tuples is uniformly chosen to be removed [KS17]. At any update round, every datapoint has an equal chance to be removed, but older tuples will go through the forgetting process many more times. These two temporally-biased amnesia strategies are examples of *retrograde amnesia*, in which older datapoints are more likely to be deleted in comparison to newer ones. The opposite is *anterograde amnesia*, in which newer datapoints have a higher chance to be removed [KS17]. Prioritizing recently added tuples for deletion over historical data can help ensure that only datapoints that appear often get stored in the database.

Instead of focusing on the temporality of tuples, an alternative is to take the result of past queries into account: *query-based amnesia*. Datapoints that appear often within query results could be assumed to be more important and should thus be kept over those that are never needed [KS17]. It is a commonly applied assumption that future queries will be similar to historical queries [AMP+13]. If frequency of access tables are available, it would be a simple matter to periodically delete the pieces of information that are utilized the least. Such an approach runs the risk of deleting freshly acquired information that has not yet had time to appear in any recent queries, so implementations of query-based amnesia should typically also include a period of waiting time before a tuple can be deleted [KS17].

2.2 Caching

A cache typically refers to a memory system that is very fast to use but is limited in storage capacity [PIT+18]. Caches are used to prefetch popular content into faster memory for ease of access, either locally or in a central cache server [MAN14]. If one can find correlations in how people access a database, then adding a cache can bring great benefit both in speed and stability of the system. Thus, caching frequently accessed data is a commonly used method to improve the performance and reliability of data systems [RDK03]. While caching methods are not strictly meant to save space, they do utilize similar techniques to the temporal and query-based amnesia methods, as detailed in Section 2.1, as cache storage is limited and its space needs to be used to its fullest extent. The key difference for caching when compared to deletion methods is that deleted data is forgotten, and will never show up in query results again unless a backup is explicitly recovered [KS17].

The equivalent of an amnesia strategy in caching is called a *paging rule* [ST85], *admission and eviction policy* [DMT+19], or *caching replacement policy* [FNNT14]. These are the algorithms that determine which contents are added or removed from a cache whenever information not present in the cache is requested (a *miss* or *page fault* occurs). The most frequently studied paging rules are the LRU, LFU, and TTL. The *Least-Recently Used* (LRU) algorithm is one of the simplest caching replacement policies. When a new item needs to be added to a cache, LRU dictates that the oldest item added to the cache must be deleted [ST85]. LRU is useful in applications where recency is a major factor in the quality of the requested data. Under finite adversarial sequences, LRU has been proven to achieve the optimal competitive ratio [PIT+18]. The *Least-Frequently Used* (LFU) paging rule involves replacing the page in the cache that has been requested the least number of times [ST85]. In essence, the LFU prioritizes the items that have been requested most in the past. It is often used in applications where the request sequences are *stationary* — meaning that the

request distributions of the documents do not change (the popular items stay popular) [PIT+18]. While the LFU needs a large amount of storage to function well, it has been proven to be optimal under independent request sequences [FRP18].

As mentioned above, these paging rules are conceptually very close to the amnesia strategies detailed in Section 2.1. The LRU is similar to temporally biased retrograde amnesia, and the LFU is like query-based amnesia, as it uses frequency-of-access tables. The key difference is that paging rules are deterministic and do not include probability distributions. Both LRU and LFU utilize on-demand paging — no item is removed from a cache until the storage space is needed to add a new one. As online algorithms, they require no knowledge of future document accesses to function [ST85].

Caches can also attempt to predict how long an item should stay in a cache. Such methods are called *Time-to-Live heuristics* (TTL), and they function by determining the *staleness* of stored documents [DMT+19], similarly to *data rotting* mechanisms [Ker15]. TTL policies work by assigning timers to each item added to the cache. Once the timer expires, the item is considered *stale* and is removed from the cache [DMT+19]. Whenever an item is requested, its individual TTL timer is refreshed as the content will be considered *fresh* once again, adding it to the cache if the item was not already in it. If the cache is full at that point, then the item with the timer closest to expiration will be discarded [FNNT14]. TTL heuristics operate with the assumption that recently modified documents are more likely to be changed again in the short-term future, and thus should not be kept in cache as long as those that stay unedited for a long time [BBM+97]. The general choice of assigning timer parameters to maximize the cache hit probability has been formulated as a nonlinear optimization program [FNNT14]. The main advantage of TTL policies is that in principle, the choice to evict an object from the cache is not coupled with any other object [FNNT14]. This allows different items or classes of items to be given different timers, which is useful when the content has differing levels of utility [DMT+19].

2.3 Summarization

A summary aims to maintain the features of the original with a smaller number of objects. The objects in a summary could either be a subset of the original [MBK16] [SSS07] (as is the case in sampling and submodular optimization), or different while still preserving key features (abstractions the likes of statistical aggregates or sketches of the data) [Ahm19] [RHM02]. While in deletion and forgetting one would look for the less important datapoints — those that one can afford to lose — the basic assumption in summarization is that each original point is equally valuable, as the goal is to maintain the overarching behavior of the dataset.

Thus, summarization algorithms do not necessarily work to minimize information loss but instead tend to maximize diversity and coverage [DGM+23] [MBK16]. Coverage refers to a summary containing elements and attributes from different parts of the data as opposed to focusing on just one section, and diversity means that the datapoints that do get chosen are as dissimilar from each other as possible. If the main objective were the minimization of information loss and not specifically maintaining any overall behavior, then coverage would still be important but diversity would become less of a priority. A diverse condensed dataset might generally contain more information than a less diverse set, so some information might still be preserved, but for a summarization algorithm that is not the overall goal. The importance of this distinction lies in the fact that coverage and diversity can sometimes be conflicting requirements: the more elements one chooses, the more coverage increases, but choosing too many similar elements reduces diversity [MBK16]. Thus, the objective function of a summarization problem is not monotonic [DGM+23] — the size of the optimal

summary will not grow endlessly, while the optimal dataset to minimize information loss should aim to be as close to the storage limit as possible.

2.3.1 Summary Statistics

The most intuitive summary is an aggregate statistic, such as a count or an average. Maintaining many tuples in storage is not necessary when one is more interested in performing a more general analysis of the data [KS17]. This allows summaries to be used as an alternative to data deletion or movement to cold storage with a cache — keeping a few aggregated values of all forgotten data will reduce storage requirements while allowing general queries to be made [KS17]. Statistical aggregates can allow one to estimate the distribution of numerical data that can be used to approximate patterns. Commonly used statistical aggregate functions are the mean (arithmetic, geometric, harmonic), median, standard deviation, skewness, and kurtosis.

The *arithmetic mean* is the most commonly used aggregation function [BPC$^+$07]. Given a finite number of numeric datapoints, the output is their sum divided by their total amount (see Equation 2.1). The arithmetic mean is typically used with additive values.

$$\mu(\mathbf{x}) = \frac{1}{n}(x_1 + x_2 + \cdots + x_n) = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.1}$$

The *geometric mean* shows central tendency through the product of the datapoints' values instead of the sum (Equation 2.2), which is useful with percentages, some ratios, or any other multiplicative data [BPC$^+$07].

$$\mu_g(\mathbf{x}) = \sqrt[n]{(x_1 + x_2 + \cdots + x_n)} = (\prod_{i=1}^{n} x_i)^{\frac{1}{n}} \tag{2.2}$$

The *harmonic mean* (Equation 2.3), while less common than the arithmetic or geometric mean, applies when one needs to compare the reciprocals of the measurements, such as when finding average rates.

$$\mu_h(\mathbf{x}) = \frac{n}{(\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n})} = n(\sum_{i=1}^{n} \frac{1}{x_i})^{-1} \tag{2.3}$$

The *median* of a dataset is the value that divides the data into two equal halves, helping to determine whether any given datapoint lies in the upper or lower half of the set [BPC$^+$07]. It is calculated using Equation 2.4. Compared to the mean, the median is often a better representation of a typical value, as the mean can be significantly affected by outliers.

$$f_n(\mathbf{x}) = \begin{cases} \frac{1}{2}(x_{(k)} + x_{(k+1)})), \text{ if } n = 2k \text{ is even} \\ x_{(k)}, \text{ if } n = 2k-1 \text{ is odd,} \end{cases} \tag{2.4}$$

where $x_{(k)}$ is the k-th largest (or smallest) component of $\mathbf{x}$. [BPC$^+$07]

The *standard deviation* (SD) is a representation of data dispersion. Assuming the numerical data follows a normal distribution the SD shows how closely the mean values represent the entire dataset [LIL15]. This is often a reasonable assumption to make if the sampling was conducted with a large enough sample size and a suitable probabilistic method — see *Sampling* under Section 2.3.2. The SD is calculated as the square root of the variance, which includes the differences in the values of individual datapoints from the arithmetic mean (see Equation 2.5).

$$\sigma(\mathbf{x}) = \sqrt{\frac{\sum_{i=1}^{n}(\overline{x} - x_i)^2}{n-1}} \tag{2.5}$$

If a dataset sample is not normally distributed, this can be indicated by the measurements of *skewness* and *kurtosis*, the *normality measures* [CZY17]. Skewness and kurtosis are also called the *shape parameters* since they respectively refer to the symmetry and length of the tails of the distributions [bYW12]. Skewness measures the symmetry of the distribution: the further from zero is the skewness value, the more asymmetrical the distribution [CZY17]. If the value is negative, then the distribution is said to be skewed to the left — the tail is longer to the left, and most of the datapoints are larger than the mean. If the skewness is positive, then it is said to be skewed to the right — the tail is to the right, and most datapoints are smaller than the mean. The equation for skewness is the third standardized central moment of the distribution (Equation 2.6) [bYW12].

$$\tilde{\mu}_3(x) = \frac{\sum_{i=1}^{n}(x_i - \mu)^3}{n\sigma^3} \tag{2.6}$$

Kurtosis measures the peakness of the distribution. The expectation of the kurtosis in a normal distribution is zero [CZY17]. If a dataset has positive kurtosis, then its distribution has a fatter tail than the normal distribution with the same variance — it contains more extreme values than expected. A smaller kurtosis is generally indicative of a flatter distribution. The equation for kurtosis is the fourth standardized central moment of the distribution (Equation 2.7) [bYW12].

$$\tilde{\mu}_4(x) = \frac{\sum_{i=1}^{n}(x_i - \mu)^4}{n\sigma^4} \tag{2.7}$$

There is a direct relationship between kurtosis and skewness: the former must increase as the latter increases: kurtosis $\geq$ skewness$^2 - 2$ [CZY17].

2.3.2 Sampling

Sampling methods are used to select a *sample* (smaller dataset) from the broader *population*. The population can be a static large dataset [Tah16] or a dynamic data stream [Ahm19]. Many sampling methods used for statistical inference are probability-based. Two commonly discussed methods are simple random sampling and systematic sampling [Tah16]. In *simple random sampling*, every data point in the population has an equal chance of being selected. However, this method requires access to the entire population at once to create a sampling frame, which is often impractical or infeasible [Sed14]. *Systematic sampling* is a more widely used approach, where every n-th datapoint is chosen, with n determined by the desired sample size, starting from a randomly chosen point [Tah16]. While systematic sampling is easy to implement and doesn't require a full list of the population, making it suitable for streaming data, it may introduce bias if there is an underlying pattern in the data that aligns with the sampling interval.

More complex sampling methods include stratified random sampling, cluster sampling, and multi-stage sampling, all of which involve dividing the population into subgroups [Tah16]. In *stratified random sampling*, a random sample is taken from each subgroup. This method is useful when there is a high variance between groups but a low variance within groups. However, selecting the right stratification variables is critical, and the method becomes more costly as more variables are introduced [Tah16]. *Cluster sampling* involves randomly selecting entire subgroups (clusters), and including all members of the chosen clusters in the sample, while completely excluding the unselected clusters [Sed14]. In this approach, each cluster may have an equal

chance of being selected, or probability can be weighted based on the size of the cluster, depending on the specific needs of the study. Cluster sampling is often implemented as part of *multistage sampling* — where sampling occurs in multiple stages. First, clusters are selected, and then individual datapoints are sampled from within the chosen clusters, usually with an equal amount of points from each selected cluster [Ahm19]. Multistage sampling is commonly used when the population has a nested, hierarchical structure of natural clusters [Sed14].

The sampling methods discussed so far are probabilistic, meaning that different iterations over the same population can yield different results. While the more complex probabilistic sampling methods can reduce this variability through careful selection of subgroups, it is impossible to fully control the characteristics of the selected sample. If control over the sample is more important than obtaining accurate, generalizable inferences about the broader population, then non-probabilistic sampling techniques can be used, such as quota or convenience sampling [Tah16]. These methods involve researcher selection based on predetermined characteristics (in the case of *quota sampling*) or availability (in the case of *convenience sampling*), and as a result, they can introduce significant bias into the findings.

Data streams, or dynamic datasets, are ordered datasets where new instances are rapidly generated or acquired [Ahm19]. These streams often originate from distributed sources such as environmental sensors or network traffic logs [GNHW07]. Due to the vast volume of data involved in stream-based applications, it is typically not feasible to store all the information or revisit past data at a later time [SMB16]. As a result, summarization algorithms that are independent of dataset size become essential [Ahm19].

One of the most fundamental methods for stream sampling is *reservoir sampling* [Agg06]. This technique involves sampling the data in a single pass and storing it in a dynamic sample or *reservoir*, that represents the current state of the data stream. New datapoints are inserted probabilistically, and once the reservoir reaches its predetermined size, random deletion occurs to maintain the sample size. When a random sample is required, it is drawn from the reservoir instead of the original stream [Vit85]. Since reservoir sampling preserves the original representation of the data, the sample taken from a reservoir can be processed using the same methods that would have been applied to the original dataset. It also provides an unbiased estimate of the datastream, with proven guarantees for error estimation [Ahm19]. However, in some cases, this unbiased approach may not be ideal, as certain applications may benefit from introducing some bias. For example, when recent data is considered more valuable than older historical data, biased sampling may be more appropriate [Agg06].

2.3.3 Sketching

The primary limitation of sampling is that it is not well suited to answer queries that require detailed knowledge of specific records within the dataset. This problem parallels the challenge of false negatives found in data deletion and forgetting: if one seeks to determine whether a particular item exists in the full dataset, a negative response could either be accurate or merely indicate that the item was excluded from the sample [Cor17]. In contrast, *sketching* provides a concise representation of data by summarizing specific properties, enabling the efficient approximation of specific functions [Cor23]. Common sketching techniques include Bloom filters, Count-Min sketches, and HyperLogLog sketches. These methods are optimized for answering membership, frequency, and cardinality queries, respectively [Cor17].

Bloom filters are used for answering discrete set-membership queries: determining a given datapoint's presence in the datastream at any moment in the past [Ahm19]. Bloom filters turn a set of n elements into an array of m bits by using k independent hash functions that all map to a number from 1 to m, where

$kn < m$. The array is initialized to all 0 values for every bit. For each element in the set, the element is passed through all of the independent hash functions, which will produce a series of numbers from 1 to m. The bits in the array with those numbers as indexes are then changed from 0 to 1. If a query is later made to determine whether an item is within the set, the bloom filter runs it through all the k hash functions and returns a positive answer if the respective bits in the array are all 1-s or a negative if a 0 is found [BM04].

Bloom filters can significantly reduce the representation size of a set of items when compared to keeping an explicit list, at the cost of producing errors. However, a key feature is that the errors are always false positives — these can be produced due to hash collisions. Negative answers to set-membership queries are guaranteed to be correct, as no member of the set can have a 0 in the indexes of the array that its set-membership query would check. This allows for many applications in the field of networking, such as probabilistic routing of resources or packets, where the effects of false positives can be effectively mitigated [BM04].

The *Count-Min* sketch allows to keep track of the approximate count of specific items in the larger dataset [Cor17]. Similarly to the Bloom Filter, the Count-Min sketch uses k independent hash functions. Each hash function h_1 to h_k has its own row of size m, and each possible input element to one of the hash functions is mapped to an index of its row. This forms an array of size $k * m$ [CM05]. The array is meant to keep track of the counts of the elements in the dataset in the following way: when the count of an element x is increased by a number c, then for each row i from 1 to k, the number present at the index $h_i(x)$ (a number from 1 to m) is increased by the count update value c. Answering the query of finding the count of an element x involves finding all the values present at indexes $h_i(x)$ and returning the minimum. The returned count value can never be smaller than the true count, as any errors will be due to hash collisions incorrectly adding to the count. Therefore the Count-Min sketch is best applied when handling large counts, or when tracking which items exceed a given threshold. For smaller numbers, and especially if checking for presence like one would with a Bloom filter, the uncertainty behind the Count-Min might make it less precise [Cor17].

The *HyperLogLog* (HLL) sketch is designed to keep track of unique values in a datastream or dataset. An example application is in online advertising, where it is more valuable to keep track of unique views to an advertisement, as opposed to the total views [Cor17]. In the HLL algorithm, a special hash function g is applied to every item. The distribution of the function g is designed such that it maps each item to a number j with the probability 2^{-j}. That is, any item is hashed into the number 1 with a probability of $\frac{1}{2}$, into the number 2 with a $\frac{1}{4}$ chance, and so on. This is typically achieved by taking the first zero bits in the binary expression of a uniformly random hash value [Cor17]. Only the highest j-value is preserved as the elements in the dataset get analyzed one by one. The expected count of unique items matches this highest j-value if the dataset is large enough — the more unique items get processed, the higher the chance that a low-probability j-number gets produced by the hash function g. To minimize the errors from possible variation further, a uniform hash function h can be applied at the start to sort the items out into different groups, whose largest j-values from the function g would then be tracked separately. To reduce the skewing effect of a few large estimates on the overall result, the final output of the HLL sketching function is the harmonic mean (see Section 2.3.1) of the largest stored j-values of the different groups.

2.3.4 Submodular Summarization

Submodular summarization is a method that uses specific set function properties to turn the finding of a summary subset into an optimization problem. A set function is a function $f : 2^V \rightarrow \mathbb{R}$ that assigns each subset a value $f(S)$, where V is a finite *ground set* [KG14]. It requires there to be a utility set function f that would measure the usefulness of any possible selected subset. For submodular optimization techniques to be

used, this utility function must be non-negative, monotone, and submodular (have diminishing returns). For every set A and set B in which $A \subseteq B \subseteq V$, a set function $f : 2^V \to \mathbb{R}$ is:

- Monotone if $f(B) \geq f(A)$,
- Non-negative if $f(A) \geq 0$,
- Submodular if for every element $e \in V \setminus B$, $f(A \cup \{e\}) - f(A) \geq f(B \cup \{e\}) - f(B)$ (diminishing returns)

If a problem's objective function is proven to adhere to these three properties simultaneously, then submodular optimization algorithms can be used.

As the problem of submodular optimization is NP-Hard, then a simple commonly used heuristic is the greedy approach [LKG+07]. The greedy algorithm first fully considers all feasible solutions from the subsets of cardinality 1 and 2, the best of which is named S_1 [Svi04]. The rest of the subsets are then checked in a greedy manner: for every subset of cardinality 3, the addition of every item in the dataset is iteratively checked. The item that gives the best improvement to the objective function is added to the set until the cardinality of the set grows equal to the *knapsack constraint* (the largest permitted size of the summary). The utility values of all these final sets are compared, and the best is named S_2. The set that is output as the solution is the one that gives the most utility out of S_1 or S_2 [Svi04].

While originally applied in economics, game theory, and operations research, there have recently been more applications of submodular optimization in data reduction, such as document summarization [LB11] or photo archiving [DGM+23]. Some applications [LKG+07] [DGM+23] notably use a lazy-evaluation variant of the greedy submodular optimization algorithm that makes it more scalable, at the cost of a smaller worst-case approximation guarantee.

2.4 Compression

Data compression techniques aim to represent a complete dataset with the least storage space possible, usually by identifying repeating structures within the data that can be encoded into a more compact form [Say17]. Thus, a compression algorithm is typically a syntactic method for reducing the data [Ahm19] — all data is just represented as a large string of bytes to optimize, without giving importance to the underlying meaning [JMN99]. Semantic compression algorithms also exist, which consider the meaning of the given data. These are most often implemented as pre-processing or intermediate steps alongside syntactic compression schemes [JNOT04]. The main difference between compression and a summary, amnesiac dataset, or cache, is that compressed data is not intelligible. To work with a compressed dataset, the representation needs to be decompressed (reconstructed) for analysis.

2.4.1 Lossless Syntactic Compression

There are two main approaches to data compression: lossless and lossy compression [PdMCD24]. *Lossless compression* provides an exact representation of the initial data, with no loss of information upon reconstruction [Hos12]. Thus, lossless compression is often used with medical data that must be preserved completely, such as Electroencephalogram (EEG) and Electrocardiogram (ECG) measurements [PdMCD24].

Lossless compression algorithms typically work by either comparing different sets of input data to remove redundant information [PdMCD24], or by creating a probabilistic model [Say17]. If a model is created of

the data, it is then used to turn an input string of symbols into an encoded (compressed) output [WNC87]. A decoding algorithm must then have access to the same model to be able to decompress the data. The most commonly used lossless algorithms are statistical methods such as Huffman coding [Mof19] and Arithmetic coding [Lan84] [WNC87], or dictionary methods like the Lempel-Ziv-Welch algorithm [ZL77].

Huffman coding (with its many subsequent variants) is one of the most fundamental compression approaches. Huffman coding works by assigning a set of binary prefix codewords to the symbols or letters that need to be encoded [Say17]. In a prefix code, no codeword is a prefix to another codeword, ensuring that the code will be uniquely decodable. The prefixes in Huffman coding are assigned based on three main rules: symbols that occur more frequently will be assigned shorter codewords, the two least frequent symbols must have codewords of the same length, and these must differ only in the last bit. If the estimate of the symbol probability distribution (the model) is accurate, then this tends to lead to a more compact representation of the input [MSWB94].

Huffman coding is a simple but effective method and is often preferred over other compression techniques because of its speed, both in coding and decoding [Mof19]. It functions well for lossless image and audio compression, such as in the FLAC (Free Lossless Audio Codec) scheme [Say17]. However, it might not perform as well when the underlying probability distribution is skewed (when some symbols are significantly more common than others) [MSWB94]. This is the case in English language text compression, for example, where there is a large difference between the probabilities of the most and least likely next symbol in a sequence [WNC87].

Arithmetic coding is a scheme that is theoretically superior to Huffman coding in terms of compression [WNC87], specifically in the case of skewed alphabets [MSWB94] or alphabets that contain few symbols [Say17]. However, it requires more memory [MSWB94] and takes longer to perform, because it needs additional computations [Mof19]. In arithmetic coding, any message is represented by a numerical interval between 0 and 1 [WNC87]. Each symbol is assigned a subinterval within this range. This subinterval gets a range that is proportionally sized to its frequency according to the model, and all subintervals combined span the entirety of [0, 1.0) [Lan84]. For example, an alphabet composed of three symbols A, B, and C, where A is twice as frequent as B or C, could have a subinterval table as in Table 2.1.

Symbol	**Probability**	**Range**
A	0.5	[0, 0.5)
B	0.25	[0.5, 0.75)
C	0.25	[0.75, 1.0)

Table 2.1: Example fixed model for alphabet [A,B,C] (simplified from [WNC87]).

The code for a one-symbol message is the range as seen in the third column of Table 2.1. Each successive symbol added to the message narrows the interval according to the subinterval of the new symbol. The code for A in this example would be [0,0.5). To get the code for AC from that, one would need to find the last quarter of the interval of A, since the code for C is [0.75, 1.0). Hence the result would be [0.375,0.5].

The decoding process of an arithmetic code rests on the fact that the width of the subinterval of any message is the product of the probabilities of its symbols. With the usage of a stop symbol to denote the end of the message, the decoder can precisely determine the original message from a given range and model. The range will lie completely within the space allocated to one specific symbol by the model — the first symbol in the message. The next symbols can be determined recursively by examining the subinterval that the given range falls into within the range of the previous symbol.

Dictionary compression methods focus on finding repetitive patterns in the data instead of creating probabilistic models. These most recurrent patterns are placed in a dictionary, and the frequent pattern is replaced with its dictionary index whenever it appears in the original message [Say17]. Such methods are particularly valuable when previous knowledge about the data distribution is limited and statistical testing cannot be accurately performed [ZL77]. If prior knowledge is available, then a static, application-specific dictionary can be designed to store any particular dataset [Say17]. However, without this knowledge, the dictionary must adapt as the input sequence grows.

A common adaptive dictionary-based compression scheme is the *Lempel-Ziv-Welch algorithm* (LZW) [Say17]. It is the fundamental compression technique behind the popular GIF (Graphics Interchange Format), a format for reducing image file sizes without a degradation in quality [AN08]. The LZW starts with a dictionary primed with all the letters of the alphabet: it contains every possible one-symbol message. The LZW encoding algorithm then goes through the input sequence once from the beginning. The first letter of the input sequence will have an index in the dictionary — the index is sent as the first symbol of the encoded output. The dictionary is then updated with an entry that consists of the first and second input symbols combined. Now the encoder tries to find a pattern starting from the second input symbol. The input is accumulated in a pattern as long as it is found within the dictionary. If the accumulated pattern is no longer found in the dictionary, then the index of the previous pattern is sent to output, and the new pattern is added to the dictionary as a new entry. This starts a new cycle of the algorithm, looking for a new pattern, starting from the last letter of the previous pattern.

Dictionary-based methods work best in long messages with a few long frequently occurring patterns, such as in text or inter-computer communication [Say17]. As the algorithm progresses, more sequences can be recorded into the dictionary as more of the structure of the input sequence is captured [Say17]. This does mean that shorter messages will see less compression, and the coding method will become less efficient under those circumstances. There is, however, a benefit that holds true regardless of input size: decoding an LZW-encoded message only requires the same initial alphabet dictionary as the encoder. The complete dictionary can always be reconstructed by concatenating any received index with the next received characters until a pattern is reached that can be added to the dictionary under a new index [Say17].

2.4.2 Lossy Syntactic Compression

A *lossy compression* scheme always involves losing some information after the compression process has been completed. Data that has been compressed with information loss can never be identically recovered [Say17], thus lossy algorithms are also called irreversible compression [CPM22]. These are often used in fields where the inexactitude of the decompressed data when compared to the original is less important, for example in audio or image storage, where the quality of a file might be reduced while still keeping the content comprehensible [Hos12].

Lossy algorithms mainly focus on removing specific sets of attributes that are not useful for particular applications. Widely used lossy compression algorithms include the Fourier transform and the (discrete) wavelet transform. *Transforms* are representations of frequency signals, and images and sound are spacial and temporal frequencies respectively. Transform representations allow the parts of such signals that are less important or harder to perceive to be de-emphasized, which results in high-quality (albeit still lossy) compression [Say17]. The most known application of transform coding is the JPEG picture compression standard, which uses a variant of the Fourier transform called the Discrete Cosine Transform [Wal91].

While lossy compression is not typically used for textual and structured data [Hos12], it has found a

particular use-case in sampled sensor data. When acquiring information to be used for machine-learning approaches, lossy algorithms have been found to obtain similar results for significantly fewer samples [CPM22]. Some of these lossy algorithms for structured data include Linear and Bezier curve approximation, which are used to minimize energy consumption in wireless sensor data transmission [LLR06]. With *linear approximation*, an increasing set of samples is represented by the two farthest points in the set. Since the error must be kept below a user-specified precision, then if the addition of a new sample would cause the linear approximation to go over the error, the two representative points are sent and a new dataset collection period starts [CPM22]. The method can be improved by replacing straight lines with cubic *Bezier curves*, which while requiring four points to make instead of two, can draw more complex graphs without resorting to traditional curves that are more difficult to evaluate [LLR06].

2.4.3 Semantic Compression

Semantic compression algorithms aim to derive a compressed model of a dataset by taking into account the meanings and ranges of individual attributes [BGR01]. This allows the exploitation of more complex correlations and dependencies in the data, as tables are not just viewed as large byte strings like in syntactic compression [JNOT04]. Semantic compression has been shown to provide good results when combined with syntactic compression algorithms, better than either individually [JNOT04]. We will describe two such methods, Fascicles and ItCompress.

Fascicles are groups of rows that share some similar compact attributes. More formally, a fascicle, denoted as $F(k, t)$, is a subset of tuples from the dataset that all have k compact attributes whose range or number of distinct values does not exceed the tolerance value t [JMN99]. The tolerance value t can vary between columns, as it depends on the meanings and dynamic ranges of each of the attributes — that is what makes this method semantic. In essence, fascicles try to detect row-wise patterns, where sets of tuples have approximately similar values for some attributes, but not necessarily all of them [BGR01].

Fascicles can be used for both lossless and lossy compression when combined with traditional syntactic compression methods. For the former, discovered fascicles can be used to reorder the tuples of the dataset, while the compact attributes can be outright deleted for the latter. Applying a respectively lossless or lossy syntactic compression method after such a procedure can greatly improve the size of the final compressed version of the dataset [JMN99]. As an illustrative example, consider the following three-row dataset:

col1	**col2**	**col3**	**col4**
A	B	C	D
E	F	G	H
X	Y	C	D

Table 2.2: An example dataset.

In Table 2.2, the first and third rows can compose a fascicle F(k=2, t=0), as the third and fourth attributes are equal in both these rows. To use this fascicle for compression in this example, one could replace the [C, D] section of the rows with a central reference, omitting the need to store the same pattern twice.

There are also standalone semantic compression techniques, such as *ItCompress* (ITerative Compression). Compared to syntactic compression, ItCompress has the advantage of allowing local decompression of rows. This allows the answering of queries at the attribute level without having to decompress the entire dataset, significantly speeding up the retrieval process [JNOT04]. The method works by assigning each tuple in the

dataset to a *representative row*, the tuple in the dataset that is most similar to it within a given tolerance. Then, only the representative rows need to be stored fully. For every other tuple, one would only need to store a reference to its representative row (RR), which attributes differ, and the differing attributes' values. Whether attributes differ from the RR can be seen from the stored bitmap, which records 1 for when the attribute is within tolerance when compared to the equivalent attribute in the RR, and 0 for when it is different [JNOT04]. See Table 2.3 for an example of how ItCompress could depict the three-row dataset from the Table 2.2 with representative rows.

RRID	Difference Bitmap	Differing Values
1	1111	
2	1111	
1	0011	X, Y

Table 2.3: ItCompress storage example.

The final set of representative rows is found as follows: the ItCompress algorithm chooses an initial random set of representative rows, and then iteratively improves on it using a hill-climbing heuristic, choosing new RRs and discarding old ones. Each iteration needs only a single scan over the data and is proven to monotonically improve the solution with a relatively fast rate of convergence [JNOT04].

Compared to the fascicles method, ItCompress addresses the limitation that every compact attribute in a fascicle must be matched completely — a single row in a fascicle with a different value for a compact attribute cannot be included. In ItCompress, the bitmap allows for far more granularity in storage.

2.5 Incomplete Databases

Our approach is deeply connected to the theoretical foundations of incomplete databases. The purpose of having an incomplete database is so users can obtain partial responses on the basis of incomplete information [ILJ84]. The intuitive way to define an incomplete database is as a database where some tuples may be missing [Lev96], or a database that contains labeled nulls [GMM$^+$24]. However, it can be mathematically argued that an incomplete database is, in fact, a collection of many complete databases [GMS22] (a set of instances) [AKG91] that all satisfy a certain specification. These individual databases that together compose a single incomplete database are called *possible worlds*, as they represent different possible states of the *real world* behind the incomplete representation [AKG91].

A query to an incomplete database will return a set of answers, one for each possible world. If a specific answer is always returned in response to a given query (the set of all answers from the possible worlds has a size of 1), then that result is a *certain answer* with respect to the incomplete database and the query [AKG91] [FKMP05]. If the set of answers returns with a size larger than 1 (if the answer is different in some possible worlds when compared to others), then all those answers are called *possible answers* [Len02] [GNHW07]. Any answer to a given query that is not returned in any of the possible worlds (is not present in the answer set) is thus called an *impossible answer* [LW85] [GNHW07].

The most used approaches to compactly represent incomplete databases are based on null values [AKG91]; we have decided to use *column-labeled nulls*. By letting labeled nulls appear in the tuples of a relational database, the possible worlds can be derived by replacing the nulls with appropriate constants according to their label [GMS22]. The nulls are *column-labeled* if a null could be replaced with any value in the domain of the column of the cell. For the purposes of this work, we are defining a *null* as a value that exists but is

not known. There are other possible definitions, such as inapplicable nulls, where the value does not exist, or no-information nulls, where it is not known whether there is an existing value behind it or not, but those are outside the scope of the current study.

To provide a simple example, let there be an incomplete dataset as follows:

col1	**col2**
A	B
A	*

Table 2.4: An example incomplete dataset

in which A and B are known values, and * is a column-labeled null value from the domain {A, B, C}. The set of possible worlds of the dataset in Table 2.4 would comprise all the combinations of what the nulls in the dataset could stand for — as represented in Figure 2.1:

col1	**col2**
A	B
A	A

col1	**col2**
A	B
A	B

col1	**col2**
A	B
A	C

Figure 2.1: The tables that form the set of possible worlds of Table 2.4.

If one would query for all the value pairs (col1, col2) present in the same row, then (A, B) is a certain answer, as it appears in every possible world. (A, A) and (A, C) are possible answers, as they only appear in some worlds and not in others. All other value pairs, such as (B, B) or (C, A) are impossible answers, as none of the possible worlds of the incomplete dataset contains such pairs.

Chapter 3

Motivating Example

Imagine a dataset consisting of a single table with three columns and four rows, as shown in Table 3.1. Suppose we face a storage constraint and can only retain three rows instead of the full table. This limitation forces us to make choices about which data to keep and which to discard.

col1	col2	col3
A	B	C
D	B	E
A	E	C
A	B	F

Table 3.1: An example dataset.

The typical solutions in such cases are either deleting the whole dataset and replacing it with summary statistics, or just deleting rows deemed less critical, resulting in a reduced three-row dataset like the one shown in Figure 3.1. While such approaches save storage, they come at a cost: the relationships between certain attributes can be lost. For example, the relationship (col2, E)-(col3, C), which is present in the original table, is no longer represented in the reduced dataset.

This loss is significant because relationships within a row carry meaningful context. In the relational model, data in the same tuple is treated as a coherent unit, such as a person's name, date of birth, and address. These values are semantically related, and their combination within a tuple preserves important information. When relationships between attributes are lost, so is the context and meaning that their co-occurrence provides.

col1	col2	col3
A	B	C
D	B	E
A	E	C
A	B	F

$\rightarrow$

col1	col2	col3
A	B	C
D	B	E
A	B	F

Figure 3.1: Reducing the example dataset by deleting the third row.

If someone queries whether attributes E and C appear together in the shortened dataset, the answer would be no — even though the complete dataset would show otherwise. This demonstrates the risk of false negative

answers when rows are deleted. Furthermore, deletion makes it impossible to provide fully *certain* negative answers about any relationships that might have existed in the original table. For instance, if asked whether there had been a pairing between E and F, one cannot definitively say no — it is entirely possible that the deleted row contained such a relationship and therefore the pairing is *possible* or *uncertain*. This irrevocable loss of information underscores a critical limitation of deletion-based data reduction methods.

There is, however, a potential solution — instead of saving space by deleting rows, one could replace specific cells with null values: [A B C] and [A E C] can both become [A * C], where the '*' symbol represents all the conceivable values that could be held in that cell. Now every relation in the original dataset is also present in the new *expanded* one, thus eliminating the risk of false negatives. Of course, this method opens up the possibility of false positives, as an expanded dataset could also include relationships that might not have been present before. How can expanding the dataset save storage space if we are not deleting any rows? If the cells to be replaced with nulls are chosen carefully, then similar rows can be collapsed together into one:

col1	**col2**	**col3**
A	B	C
D	B	E
A	E	C
A	B	F

→

col1	**col2**	**col3**
A	*	C
D	B	E
A	B	F

Figure 3.2: Reducing the example dataset by merging the first and third rows together.

As shown in Figure 3.2, after merging the first and third rows, the table has been reduced to three rows from four, and we have conserved storage space without outright deleting any rows entirely. Similarly, if we can only store a maximum of two rows, Figure 3.3 shows how one can further reduce it by replacing more cells with nulls. We call this novel approach to data reduction *Row Merge*, which is a way to produce *merge-reduced* datasets.

col1	**col2**	**col3**
A	*	C
D	B	E
A	B	F

→

col1	**col2**	**col3**
A	*	*
D	B	E

Figure 3.3: Reducing the example dataset by merging the first and third rows together a second time.

col1	**col2**	**col3**
A	B	C
D	B	E
A	B	F

col1	**col2**	**col3**
A	*	C
D	B	E
A	B	F

Figure 3.4: Deletion reduction (left) and Row Merge reduction (right) of the initial data from Table 3.1.

Now we can compare the two different three-row reduced versions of the dataset as shown side by side in Figure 3.4. In the table on the left, one row ([A E C]) has been deleted. In the table on the right, the two rows [A B C] and [A E C] have been merged into one, [A * C]. Table 3.2 displays some queries that one could

Query	Original	Deletion answer	Merge answer
col1=A ∧ col2=B	Yes	Yes	Yes
col1=A ∧ col2=E	Yes	No (FN)	Yes
col1=A ∧ col2=G	No	No	Yes (FP)
col1=E ∧ col2=F	No	No	No

Table 3.2: Selected queries on the two tables of Figure 3.4.

ask about these datasets, with their respective answers from each. If one asks a query in the form of col1=X ∧ col2=Y, then the dataset reduced with deletion (left on Figure 3.4), can give true positive, true negative, or false negative results, as seen on the third column of Table 3.2. No query in the above format will produce a false positive: *Yes* answers are certain, but *No* answers are uncertain, because we have lost the information of what contents the deleted rows had. If one would ask the same queries after reducing the original dataset by the method of merging similar rows (right on Figure 3.4), the results can be true positive, true negative, or false positive, as seen on the fourth column of Table 3.2. Contrary to the previous case with deletion, negative answers after a Row Merge reduction are always certain — when one receives a *No* answer to a query, one can be sure that the object of the request does not exist in the dataset. The merge-reduced table thus has the ability to conclusively rule out that specific relationships were ever present in the original dataset, completely eliminating false negatives as a query answer possibility. Furthermore, the *Yes* answers are only uncertain if one of the cells matching the queried relation is a null (*) in the dataset — compare the first and second rows of Table 3.2, where A-B is present in the reduced dataset, while A-E only exists in the form A-*.

Using Row Merge to create a reduced version of a large dataset is a task that requires a thought-out strategy for choosing the right cells to replace with a null. Replacing any random cell will more often than not just result in a loss of information for no particular gain:

col1	col2	col3
A	B	C
D	B	E
A	E	C
A	B	F

→

col1	col2	col3
A	B	C
*	B	E
A	E	C
A	B	F

Figure 3.5: An ineffectual replacement of a cell with a null.

In Figure 3.5, no rows can be collapsed together, and so the dataset on the right contains less certain information with the same amount of storage compared to the one on the left. Therefore some replacements might be unhelpful to the final objective of row reduction while minimizing information loss. However, a dataset with too many nulls might use little storage but could contain a significantly lower number of relationships than the original. For example, a table that is just [* * *] in its entirety does occupy only one row, but has barely any informational content to speak of when compared to the figures above.

col1	col2	col3
A	*	C
D	B	E
A	B	F

col1	col2	col3
A	B	*
D	B	E
A	B	F

Figure 3.6: Two different merge-reduced variants of Table 3.1.

Furthermore, the choice between different available solutions can often be non-trivial. For example, the example table could feasibly be converted into three-row forms in the two different ways shown in Figure 3.6. Both of the tables in Figure 3.6 only contain one null value. However, the second table has rendered less positive-answer queries uncertain than the first one. Merging the rows from the first table loses the yes-answer certainty from all relationships in the form of (col1-A, col2-*) and (col2-*, col3-C) except (col1-A, col2-B), which is present in the third row. The second option loses all relationships in the form of (col1-A, col3-*) and (col2-B, col3-*) except (col1-A, col2-F), (col2-B, col3-E), and (col2-B, col3-F). In the case that the domains of all columns are equal, it can be fair to say that the second reduced dataset stores more relationship information than the first, which was not immediately apparent from the outset.

Chapter 4

Problem Description

4.1 Preliminaries

Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be a collection of attributes, each associated with a finite domain Dom_{A_i}. An *atomic condition* or *attribute-value pair* is an expression of the form $A = v$ or $[v, A]$, where $A \in \mathcal{A}$ and $v \in Dom_A$. A *tuple*, *record*, or *row* is an $n-$dimensional vector $d = (v_1, \ldots, v_n)$ where the *cells* are attribute-value pairs $[v_i, A_i] | v_i \in Dom_{A_i}$. The set $\mathfrak{Constrs}(d) = \{A_1 = v_1, \ldots, A_n = v_n\}$ is referred to as the *set of conditions* of tuple d. The set of all possible tuples $\mathcal{U} = Dom_{A_1} \times \cdots \times Dom_{A_n}$ constitutes the *universe*.

A *full* or *complete dataset* is a set $F \subseteq \mathcal{U}$ (i.e., a finite collection of tuples). *Nulling* is the operation of replacing the value of one or more non-null cell $[v_i, A_i]$ in a tuple into *column-labeled nulls* $[*_i, A_i]$. A *compact* or *incomplete dataset* is a set $C \subseteq \mathcal{U}$ where at least one cell has a null value $v_i = *$, i.e $\exists [*, A]$.

Whenever two or more tuples become identical due to the nulling of a cell, then the nulling operation would end with a *merge*, which entails deleting all but one of the now duplicate rows, reducing the size of the dataset. For any compact dataset C, the set of *possible worlds* W_C comprises those full datasets F that can become the given compact dataset after a sequence of nulling operations: $F \in W_C$ if $\exists$ a sequence of nullings $F \to C$.

All atomic conditions $[v, A]$ in an incomplete dataset can be classified as certain, possible, or impossible, based on their presence in the dataset. A condition is *certain* if it exists within the set of conditions of at least one tuple in every possible world: a value-attribute pair $[v, A]$ is certain if $\forall W_C \exists [v, A] \in W_C$. An *impossible condition* is not present within the set of conditions of any tuple in any possible world: $[v, A]$ is impossible if $\nexists F \in W_C$ such that $[v, A] \in W_C$. If a condition $[v, A]$ is neither certain nor impossible, then it is *possible*, as it is present in some possible worlds, but not all.

A *relationship* R in a complete dataset is a combination of two atomic conditions $R = [[v_i, A_i], [v_j, A_j]]$ such that $i \neq j$ and both of the conditions are simultaneously present in the set of conditions of at least one tuple in the dataset: $[v_i, A_i] \cup [v_j, A_j] \in \mathfrak{Constrs}(d) | d \subset F$. In an incomplete dataset, relationships can be certain, possible, or impossible. Similarly to the atomic conditions, a *certain relationship* is present in at least one tuple of every possible world, an *impossible relationship* is present in none of the possible worlds, and a *possible relationship* is present in some but not all possible worlds.

4.2 Problem statement

Given a number k smaller than the cardinality of a full dataset F with m tuples and n attributes (Eq 4.1), find an incomplete dataset C for which F is a possible world (Eq 4.2) and that has cardinality at most k (Eq 4.3), while minimizing loss of information.

$$k < |F| = m \tag{4.1}$$

$$F \in W_C \tag{4.2}$$

$$|C| \leq k \tag{4.3}$$

4.3 Quality metric

In order to select the best incomplete dataset out of different possible solutions, there needs to be a metric by which a solution's quality can be measured. In the problem statement, we mention minimizing the loss of information. The specifics of this problem can be understood as the following: the solution should retain as much accurate data as possible while minimizing the inclusion of false information. We retain accurate data by setting the aim to conserve as many relationships as possible that are present between the attribute values of the original dataset. We also seek to minimize the extra uncertainty that is added to the incomplete dataset by the nulling operations in comparison with the original. We achieve this by counting the different types of relationships.

As defined previously, a *relationship* can be denoted by the names of the columns and their associated values in the given cells. For example, suppose there is a complete dataset of just one tuple of three attributes (col1, col2, col3) with values [A,B,C] respectively. The relationships present in the dataset are (col1, A)-(col2, B); (col1, A)-(col3, C); and (col2, B)-(col3, C), so the count is three total relationships. The order of the columns does not matter, so the relationship (col1,A)-(col2,B) is equivalent to (col2,B)-(col1,A). Any relationship consisting of three or more columns can be indicated by the presence of all possible subrelationships (combinations of components), so those need not be separately considered. Within the context of incomplete datasets, it must be specified that the above relationships are all examples of *certain relationships*. These are those relationships that are present without any nulls — they are known to exist in the dataset equally across all possible worlds. In a complete dataset, every existing relationship is certain. In an incomplete dataset, there are also *possible relationships*. To reiterate the definition, these are all of the relationships that appear in some, but not all of the possible worlds of the dataset. This distinction between certain and possible relationships mirrors the difference between certain and possible query answers, as described by Greco et al. in 2022 [GMS22]. It can still be the case that a relationship does not appear in any of the possible worlds, in which case it is called an *impossible relationship*.

To illustrate the three different types of relationships, in a single-row incomplete dataset ([A, B, *]) only the relationship (col1, A)-(col2, B) is certain. If the domain for all attributes is single capital letters, then the possible relationships include (col1, A)-(col3, C), (col1, A)-(col3,D), (col2, B)-(col3,E), and many more. All of these include at least one cell that is a null in the dataset, which means that they must vary across the possible worlds. An example of an impossible relationship would be any that includes (col1,B), as column 1

does not have a B nor a null value — the attribute has the value of A across every possible world.

One could argue that a simpler way of tracking information loss in incomplete datasets would just be to count the number of column-labeled nulls. However, that would not take into account that the total number of nulls can be increased without a change in the dataset's informational content, and the amount of information can be reduced without a corresponding increase in the number of nulls. The latter case can happen after a nulling leads to a merge: for example, reducing a two-row dataset ([A B *], [A B C]) into a single-row dataset ([A B *]) by nulling the last C (see Figure 4.1) retains the constant number of nulls (one null in the dataset), but loses the only certain C value in the dataset's column 3. The former case can also be demonstrated: while adding a column-labeled null to a dataset will increase the number of possible worlds by the cardinality of its domain, these introduced worlds might not necessarily add any new relationships. For example, the two datasets displayed in Figure 4.2 have the same number of both certain and possible relationships, despite having a different number of nulls. In this example, nulling the first row's C would not lose the certain relationships (col1, A)-(col3-C) and (col2, B)-(col3-C), because they are present in the fourth and fifth row respectively. Furthermore, this nulling would not create the possible relationships (col1, A)-(col3, *) and (col2, B)-(col3, *), as those are already present in the second and third rows respectively. Therefore, it is not enough to merely count the column-labeled nulls — that is not an appropriate indicator of quality with regard to the informational content of an incomplete dataset.

col1	**col2**	**col3**
A	B	*
A	B	C

$\rightarrow$

col1	**col2**	**col3**
A	B	*

Figure 4.1: Information can be lost without changing the number of nulls in the dataset.

col1	**col2**	**col3**
A	B	C
A	D	*
D	B	*
A	D	C
D	B	C

$\rightarrow$

col1	**col2**	**col3**
A	B	*
A	D	*
D	B	*
A	D	C
D	B	C

Figure 4.2: An extra null can be added without loss of information.

A relationship-based quality metric that seeks to minimize information loss (as set in the problem statement) should both try to maximize the number of certain relationships present and minimize the number of possible relationships in the incomplete dataset. The former stipulation comes from the following: the more certain relationships there are, the more information the dataset holds. Regarding the latter: if the combined number of both certain and possible relationships in an incomplete dataset is larger than the number of certain relationships in the original complete dataset, then some possible relationships must have been impossible in the original. Therefore, if the number of certain relationships is kept equal in an incomplete dataset, then the more possible relationships it contains, the more likely it is that any given possible relationship was impossible in the original full dataset, thus increasing overall uncertainty. An example: let there be a dataset composed of the rows [A,B,A], [A,B,B], and [B,B,A] with a domain of $\{$A,B$\}$ for all columns (see Figure 4.3 left). If we perform a nulling operation on the B of the first row [A,B,A]$\rightarrow$[A,*,A] (see Figure 4.3 right), no certain relationship is lost, as both (col1,A)-(col2,B) and (col2,B)-(col3,A) are present in the other rows.

However, the relationships (col1,A)-(col2,A) and (col2,A)-(col3,A) now become possible when they were impossible before — the number of possible relationships increases as uncertainty is added. The certain relationships remain constant; the loss of information is reflected in the increase in possible relationships. Another example is illustrated in Figure 4.4: the dataset ([A * *]) holds more information than ([* * *]), despite either having no certain relationships preserved, because the tuple [A * *] has fewer possible relationships than [* * *]. Any relationship that involves the first column having a value different to A is impossible in [A * *], but possible in [* * *], which reflects information lost.

col1	col2	col3
A	B	A
A	B	B
B	B	A

→

col1	col2	col3
A	*	A
A	B	B
B	B	A

Figure 4.3: Example of a nulling operation that keeps the certain relationships constant.

col1	col2	col3
A	*	*

→

col1	col2	col3
*	*	*

Figure 4.4: Example of a nulling operation on a dataset with no certain relationships.

col1	col2	col3
A	*	B
A	C	B
A	C	A
C	C	B

→

col1	col2	col3
A	*	B
A	C	A
C	C	B

Figure 4.5: Example of a dataset that can be merge-reduced without loss of information.

Given what we have established so far, it becomes possible to derive some properties of the quality metric that can be useful to solve the problem. We can show that dataset B $\subseteq$ dataset A if there is a sequence A $\rightarrow$ B of nulling operations. That is because after any nulling we can map the new null variable to its previous non-null constant value, and the previous null variables and the constants stay the same. Thus, a way to generalize the examples we have shown is to say that if a dataset A can be contained by an incomplete dataset B, then A will have a larger or equal number of certain relationships, and equal or fewer possible relationships than B. Furthermore, since nulling is an inherently destructive operation, then any compact dataset that fulfills the cardinality requirement will never be improved by additional nulling, even if it would allow for further merges down the line. More nulling can never provide new certain relationships nor reduce the number of possible relationships, as no extra information is acquired with the process. However, it must be noted that it is still sometimes possible to apply further nulling to an already compact dataset without losing certain relationships or adding more possible relationships — the quality metric is monotone but not strictly increasing/decreasing. We can show this with another example: let there be a compact dataset composed of the rows [A, *, B], [A, C, B], [A, C, A], [C, C, B] with a domain of $\{$A,B, C$\}$ for all columns (see Figure 4.5 left). If we perform a nulling operation on the second row [A, C, B] $\rightarrow$ [A, *, B], then it will be merged with the first (see Figure 4.5 right). The certain relationships that this process could potentially lose are (col1,A)-(col2,C) and (col2,C)-

(col3,B). However, both of them also appear in the last two rows. Furthermore, since the resulting row [A, *, B] is already present before the nulling operation, no further possible relationships are created. Therefore in this example, after one nulling operation, the size of the compact dataset was reduced without losing any relationship information.

To summarize, there are two optimization criteria that should considered when aiming to reduce information loss: maximizing certain relationships and minimizing possible relationships. Neither of these criteria is fundamentally superior to the other, as they represent different information. Maximizing certain relationships involves preserving information that is known to be true, keeping as many true positives from becoming false negatives. In contrast, minimizing possible relationships focuses on reducing uncertainty by keeping as the number of false positives as low as as possible. To simultaneously optimize for both criteria, they must be combined into a single quality metric. Although the two criteria are equally valid, establishing a priority order between them allows for both to be considered while maintaining monotonicity, which is not necessarily the case with all aggregate formulas, such as the harmonic mean of the relationship counts.

For the purposes of measuring the quality of a proposed solution, the number of certain relationships needs to take priority over the number of possible relationships. This prioritization is necessary to enable meaningful comparison between Row Merge and other data reduction methods. The distinction between these priority orders becomes apparent when comparing a dataset reduced by merging with one reduced by deletion, while prioritizing possible relationships. A dataset reduced by row deletion inherently avoids creating null values, resulting in zero possible relationships. Consequently, a merge-reduced dataset with even a single possible relationship would be evaluated as worse than any dataset reduced through deletion — even one that deletes every row — because the deletion-based method does not produce possible relationships. This outcome skews the evaluation in favor of deletion methods, regardless of their potentially severe information loss. Prioritizing certain relationships avoids this bias, enabling a more accurate comparison.

Chapter 5

Method

The quality metric described in Section 4.3 can be used to compare two or more reduced datasets, in order to determine which is the best. We have established that for this work, the best solution is one that maximizes the number of certain relationships and if those are equal, the one that minimizes the number of possible relationships. We describe our algorithms for computing these values in Section 5.1. The main challenge is finding the best *valid solutions* (defined as tables within the given size limit) from a potentially large number of alternatives. We have designed two different algorithmic approaches to solve this: *tree expansion algorithms*, and *pair similarity algorithms*, detailed in Sections 5.2 and 5.3 respectively.

5.1 Algorithmic relationship counting

Algorithm 1 describes our approach for calculating the total number of certain relationships in a dataset. The process begins by iterating over every row in the dataset, examining all possible combinations of column-value pairs in that row. For each pair of columns, the corresponding values in the row are retrieved, and if neither value is null, the algorithm creates a relationship consisting of the two column-value pairs. These relationships are then added to a global set, *CertainsSet*, which is initialized as an empty set at the start of the algorithm. The use of a set ensures that only unique relationships are retained, as duplicate entries are automatically discarded due to the properties of sets. This guarantees that even if the same relationship appears in multiple rows, it is only counted once in the final result. After processing all rows in the dataset, the algorithm calculates and returns the cardinality of the *CertainsSet*, which represents the total number of unique certain relationships in the data.

Algorithm 1 Counting certain relationships

Require: Table
  CertainsSet $\leftarrow$ Set()
  for all row R$\in$ Table **do**
    for all column pair $c_1, c_2 \in$ R **do**
      $v_1 \leftarrow$ R$[c_1]$
      $v_2 \leftarrow$ R$[c_2]$
      if $v_1 \neq * \ \wedge \ v_2 \neq *$ **then** ▷ Get all pairs with no nulls
        CertainsSet $\leftarrow$ CertainsSet $\bigcup$ Set([c_1, v_1],[c_2, v_2]) ▷ Final set keeps unique sets from each row
  return |CertainsSet|

Algorithm 2 details our method for calculating the total number of possible relationships in an incomplete dataset. This algorithm relies on the availability of a set of all certain relationships, such as the one generated by Algorithm 1, and requires the domain cardinalities of all columns to be either known or assumed. In the absence of explicit domain cardinalities, we estimate them as the number of unique values in each column across all rows of the dataset. The algorithm begins by creating the set of *unexpanded* possible relationships (*PossiblesSet*) by iterating over all rows in the dataset and identifying relationships where at least one of the two column-value pairs contains a null. These relationships are considered unexpanded because the null values are not replaced during this step and are instead retained as '*' values.

Algorithm 2 Counting possible relationships

Require: Table
Require: CertainsSet
Require: DomainCardinalities
  PossiblesSet ← Set()
  for all row R ∈ Table **do**
    for all column pairs $c_1, c_2 \in$ R **do**
      $v_1 \leftarrow$ R$[c_1]$
      $v_2 \leftarrow$ R$[c_2]$
      if $v_1 = *$ OR $v_2 = *$ **then** ▷ Get all pairs with at least one null
        PossiblesSet ← PossiblesSet $\bigcup$ $\{[c_1, v_1]$,$[c_2, v_2]\}$
  ExpCount ← 0
  GroupedPossiblesSet ← PossiblesSet.groupBy($\{c_1, c_2\}$)
  for all Set(c_1, c_2) in GroupedPossiblesSet.groups **do** ▷ Iterate over grouped unique column pairs
    CurrentValuePairs ← GroupedPossiblesSet[$\{c_1, c_2\}$)]
    if $\exists\ [*, *] \in$ CurrentValuePairs **then** ▷ Expand the nulls
      ExpCount ← ExpCount + DomainCardinalities$[c_1] *$ DomainCardinalities$[c_2]$
      ExpCount ← ExpCount $- |\{([c_1, x], [c_2, y]) \mid ([c_1, x], [c_2, y]) \in$ CertainsSet$\}|$
    else
      c1_pairs ← $\{[v_1, v_2] \mid [v_1, v_2] \in$ CurrentValuePairs $\wedge\ v_1 = *\}$
      ExpCount ← ExpCount + |c1_pairs| * DomainCardinalities$[c_1]$
      for all $[*, v_2] \in$ c1_pairs **do**
        ExpCount ← ExpCount - $|\{([c_1, x], [c_2, v_2]) \mid ([c_1, x], [c_2, v_2]) \in$ CertainsSet $\}|$
      c2_pairs ← $\{[v_1, v_2] \mid [v_1, v_2] \in$ CurrentValuePairs $\wedge\ v_2 = *\}$
      ExpCount ← ExpCount + |c2_pairs| * DomainCardinalities$[c_2]$
      for all $[v_1, *] \in$ c2_pairs **do**
        ExpCount ← ExpCount - $|\{([c_1, v_1], [c_2, y]) \mid ([c_1, v_1], [c_2, y]) \in$ CertainsSet $\}|$
      for all $[v1, v2] \mid [v1, *] \in$ c2_pairs $\wedge\ [*, v2] \in$ c1_pairs **do** ▷ Adjust for double counting
        ExpCount ← ExpCount - 1
  return ExpCount

The algorithm then proceeds to calculate the total number of possible relationships (*ExpCount*, from *expanded*), which forms the final result. To achieve this, the set of unexpanded relationships is grouped by unique column pairs, and each group is processed individually. For a column pair that contains a relationship between two nulls ([*, *]), the total number of expanded possible relationships is increased by the Cartesian product of the domain cardinalities of the two columns, and decreased by the count of certain relationships between those two columns. For column pairs where only one of the values is null ([*, v2] or [v1, *]), the algorithm accounts for the possible expansions by multiplying the domain cardinality of the column containing the null by the number of occurrences of such pairs and then subtracting the count of certain relationships that match the non-null value in the pair. Finally, the algorithm corrects for potential double counting in cases

where relationships such as [v1, *] and [*, v2] could lead to the same expanded possible relationship [v1, v2] being counted twice. Each instance of such double counting is compensated by decrementing the total count by one. The result of this process, stored in (*ExpCount*), represents the total number of possible relationships that can exist in the dataset.

5.2 Tree expansion algorithms

A *tree expansion algorithm* is a method to merge-reduce a dataset by creating a tree, starting from the initial table as the root, and then creating branches by performing one different nulling operation each (including any merges if possible). We call the branch-creating procedure *expanding* a node (see Algorithm 3), and illustrate it in Figure 5.1. Every cell in the table that does not yet contain a null value is a valid candidate for nulling, and thus for creating a branch for the tree. Thus, the maximum number of children that can come out from any given node in the tree is equal to the number of remaining cells with non-null values in the table represented by that node.

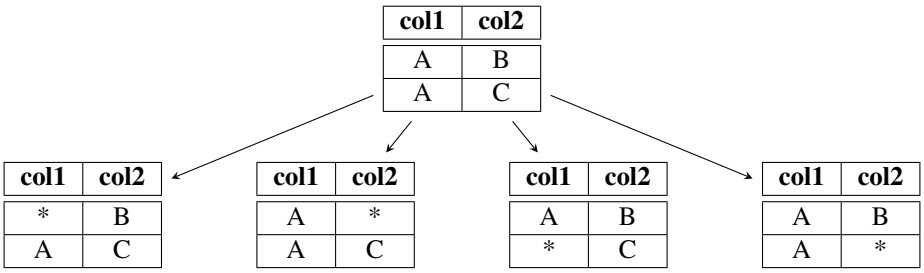

Figure 5.1: A tree formed by expanding a complete dataset (the root node) once into four incomplete datasets.

The *Exhaustive algorithm* (see Algorithm 4) produces all possible combinations of nulling operations in every order, creating every possible table that can arise from a sequence of nulling operations on the initial dataset. If all those tables can be found and compared, then that guarantees an optimal solution to the problem. Let the original dataset be of size $m * n$ (rows and columns). To form the tree, we start with that table as the root node. Then, for each of its cells, we create a branch stemming from the root, where only that cell is nulled. We call these leaves the first *layer* of the tree, and it will have $m*n$ different branches, with one table per branch. From each branch in the first layer, we stem off further branches into the second layer, by nulling all remaining cells one at a time (as mentioned, this is the Expand node function, detailed in Algorithm 3). After expanding all tables in the first layer, the second layer will contain $m * n - 1$ branches, for a total of $(m * n) * (m * n - 1)$. For each branch, we continue expanding each branch into a new layer, until there are no more cells that can be nulled. Thus, the third layer will contain $m*n*(m*n-1)*(m*n-2)$ nodes, and the pattern continues. The total number of nodes in that case would be $(m * n)!$, which speaks of a factorial complexity for the Exhaustive algorithm in the worst case scenario. Out of all the valid solutions found, the best-scoring table will be returned as the answer. As all possible combinations of nulling operations are checked, the Exhaustive algorithm will guarantee an optimal solution to the problem.

However, the worst case scenario where the maximum number of possible tables must all be considered is unlikely to come to pass. First, any time a merge takes place, then that reduces up to an entire row's worth of nullings required to conduct to check every possibility. Second, any time when a branch provides a valid solution, we know that any further nulling would produce children tables that are all contained by the parent

Algorithm 3 Expand node

Require: Table
ChildrenSet ← Set()
for all element ∈ Table **do**
    if element ≠ * **then**
        Child ← Table.makeNull(element)
        ChildrenSet ← ChildrenSet ⋃ {Child}
return ChildrenSet

Algorithm 4 Exhaustive algorithm

Require: Root
AnswerSet ← Set()
HighestScore ← [0, inf]
function LOOP(x):
    Y ← EXPAND(x)
    if $\exists y \in Y \mid y$ is valid **then**
        ValidsSet ← $\{y \mid y \in Y \wedge y$ is valid$\}$
        HighestSet ← MAX SCORING(ValidsSet)
        COMPARE SCORES($h \in$ Highest, $a \in$ AnswerSet)
        if SCORE(h) > SCORE(a) **then**
            AnswerSet ← HighestSet
            HighestScore ← Score(h)
        else if SCORE(h) = SCORE(a) **then**
            AnswerSet ← AnswerSet ⋃ HighestSet
    else if $\exists y \in Y \mid y$ is not valid **then**
        InvalidsSet ← $\{y \mid y \in Y \wedge y$ is not valid$\}$
        for all $y \in$ InvalidsSet **do**
            if SCORE(y) >= HighestScore **then**
                LOOP(y)
LOOP(Root)
return AnswerSet

table. Therefore, as additional nullings can not produce a table containing more information, once a branch reaches a valid solution its children do not need to be checked, even for a comprehensive algorithm. Finally, since the scoring function is monotonic, then any node that has too many rows but already has a score lower than the current best will never provide a better answer, and thus does not need to be searched either.

It could still be worthwhile to have ways to further restrict the search space. There are other algorithms that can be used instead in order to reduce the number of nodes needed to be checked on the tree, at the cost of losing the guarantee of finding the best solutions. Traditionally, these are the Greedy algorithm and the Random Walks algorithm.

The *Greedy algorithm* (see Algorithm 5) will only choose to expand the best-scoring child of the current node, and will return the first valid solution it encounters. In case of a tie, the child to be expanded is chosen randomly, so the output is non-deterministic. The Greedy algorithm will be faster but runs the risk of getting stuck in local optima. An example: let there be a table ([A,B,C],[A,*,*]), which needs to be reduced to one row. The Greedy algorithm must now choose which nullings to continue with on the next layer. If any of the cells in the first row is nulled, then two certain relationships are lost, but if the second [A,*,*] is turned to [*,*,*], then all certain relationships are kept (compare the left sides of Figures 5.2 and 5.3). However, the former case allows for a final solution of [A,*,*] to be found, while the latter must eventually settle for [*,*,*], an inferior solution due to the larger number of possible relationships (compare the right sides of Figures 5.2 and 5.3. Even in the worst case scenario, with no merges until the end, the complexity of the Greedy algorithm goes from $O((m*n)!)$ to $(m*n)+(m*n-1)+(m*n-2)+...+1 = 1+2+...+(m*n) = (m*n)*(m*n+1)/2 = (m*n)^2/2 + (m*n)/2 \rightarrow O((m*n)^2)$, as it only traverses the tree once.

col1	col2	col3
A	B	C
A	*	*

$\rightarrow$

col1	col2	col3
A	B	C
*	*	*

$\rightarrow \quad \ldots \quad \rightarrow$

col1	col2	col3
*	*	*

Figure 5.2: The Greedy algorithm ignores long-term consequences.

col1	col2	col3
A	B	C
A	*	*

$\rightarrow$

col1	col2	col3
A	*	C
A	*	*

$\rightarrow$

col1	col2	col3
A	*	*

Figure 5.3: A non-greedy algorithm could have found the best solution.

The *Random Walks algorithm* (see Algorithm 6) is one where the tree is traversed multiple times such that only one node is expanded per layer, which is randomly chosen until a valid solution is found. The algorithm returns the best valid solution found across all the walks. The number of walks is a constant number w; more walks increase the likelihood of finding better results, but also linearly increases the time. While the Greedy algorithm traverses the tree one time, the Random Walks algorithm traverses it w times, but as w is a constant hyperparameter, then while practically it should take w times longer to run, the big-O time complexity ends up being the same: $O(w*(m*n)^2) = O((m*n)^2)$. Similarly to the Exhaustive algorithm, the monotonicity property of the scoring function can be used to stop a random walk if the node it reaches already has a lower score than the current best valid answer.

One way to possibly improve on the performance of the Exhaustive algorithm is to expand the tree in order of decreasing score instead of following a simple breadth-first or depth-first order. We call this the *Sorted*

Algorithm 5 Greedy algorithm

Require: Root
1: **function** LOOP(x):
2: Y ← EXPAND(x)
3: **if** $\exists y \in Y \mid y$ is valid **then**
4: ValidsSet ← $\{y \mid y \in Y \wedge y$ is valid$\}$
5: HighestSet ← MAX SCORING(ValidsSet)
6: **return** HighestSet
7: **else**
8: HighestSet ← MAX SCORING(Y)
9: z ← CHOOSE RANDOM(HighestSet)
10: LOOP(z)
11: LOOP(Root)

Algorithm 6 Random Walks

Require: Root
Require: Total walks
AnswerSet ← {}
HighestScore ← [0, inf]
Walks ← 0
function LOOP(x):
Y ← EXPAND(x)
if $\exists y \in Y \mid y$ is valid **then**
ValidsSet ← $\{y \mid y \in Y \wedge y$ is valid$\}$
HighestSet ← MAX SCORING(ValidsSet)
COMPARE SCORES($h \in$ Highest, $a \in$AnswerSet)
if SCORE(h) > SCORE(a) **then**
AnswerSet ← HighestSet
HighestScore ← SCORE(h)
else if SCORE(h) = SCORE(a) **then**
AnswerSet ← AnswerSet $\bigcup$ HighestSet
else if $\exists y \in Y \mid y$ is not valid **then**
InvalidsSet ← $\{y \mid y \in Y \wedge y$ is not valid$\}$
z ← CHOOSE RANDOM(InvalidsSet)
if SCORE(z) >= HighestScore **then**
LOOP(z)
repeat
LOOP(Root)
Walks ← Walks +1
until Walks == Total walks
return AnswerSet

Order algorithm (see Algorithm 7). In this algorithm, after the expansion of a node, all children are scored and added to a priority queue of tables that is sorted by the score. While no valid answer is found, the next node to expand is chosen from this heap, taking the node with the highest score out of all of the nodes that have not been expanded yet. The first valid answer is immediately submitted as output, without continuing the search. The Sorted Order algorithm guarantees an optimal solution to be found. As the scoring metric of counting certain relationships is a monotonic function, any valid solutions beyond the first one found will necessarily be equal or worse-scoring than the first. However, the algorithm will only provide one valid solution and will ignore any potential subsequent equally good solutions. In the worst case scenario, the entire tree will need to be expanded to find a single valid answer. Therefore, the time complexity remains $(m * n)!$ like an exhaustive search.

Algorithm 7 Sorted Order algorithm

```
Require: Root
  TablesSortedByScore ← PriorityQueue()        ▷ Set of tables ordered by the Compare Scores function
  function LOOP(x):
      Y ← EXPAND(x)
      if ∃y ∈ Y | y is valid then
          ValidsSet ← {y | y ∈ Y ∧ y is valid}
          HighestSet ← MAX SCORING(ValidsSet)
          return HighestSet
      else
          for all {y | y ∈ Y} do
              TablesSortedByScore ← TablesSortedByScore ⋃ {(y, SCORE(y))}
          z ← POP HIGHEST(TablesSortedByScore)     ▷ Take highest scoring table and remove it from the
  queue
          LOOP(z)
  LOOP(Root)
```

Algorithm 8 Merge Greedy algorithm

```
Require: Root
  TablesSortedByScore ← PriorityQueue()        ▷ Set of tables ordered by the Compare Scores function
  function LOOP(x):
      Y ← EXPAND(x)
      for all y ∈ Y do
          if y is valid then
              return y
          else if SIZE(y) < SIZE(x) then
              TablesSortedByScore ← PriorityQueue()                          ▷ Reset queue
              TablesSortedByScore ← TablesSortedByScore ⋃ {(y, SCORE(y))}
          else
              TablesSortedByScore ← TablesSortedByScore ⋃ {(y, SCORE(y))}
      z ← POP HIGHEST(TablesSortedByScore)
      LOOP(z)
  LOOP(Root)
```

The last tree expansion algorithm that we introduce is a greedy variant of the Sorted Order algorithm, called the *Merge Greedy algorithm* (see Algorithm 8). Its broad strokes are similar to the Sorted Order algorithm, in that a priority queue of tables is kept to determine the next table to expand. The key difference

is that this queue is reset whenever a nulling operation leads to a merge, discarding all the yet untested tables and starting over with the new, smaller table as the root. Despite this performance improvement, the worst case scenario is that there are no merges possible until the entire table is nulled (such as a table where every cell has a unique value), giving it a worst-case complexity of $(O(m * n)!)$.

5.3 Pair similarity algorithms

An alternative to the tree expansion algorithms is to use pairwise similarity to determine which rows to merge. In the *Similarity algorithm* (see Algorithm 9), we use the Jaccard similarity to compute the similarities between the rows, by tuning a pair of rows into two sets of attribute:value pairs, and calculating their intersection over the union. This procedure is done across every possible pair of rows and pointers to the pairs are stored in a priority queue, ordered by the similarity. Then, the two most similar pairs are merged together. Merging two rows is significantly easier than merging a larger amount, as it is simple to determine the needed nulling operations to perform. Any values that are different between the same column of the two rows must be nulled, and all the values that are equal stay untouched (see Figure 5.4). The previous two rows are removed from the table and the priority queue, while the newly merged row is added to the table. New similarities are calculated between the new row and every remaining row in the table and added to the priority queue before selecting the next most similar pair. New pairs will keep getting selected and merged until the size of the table has been reduced to the requested number of rows. The similarity algorithm does not directly optimize for the preservation of relationships. However, rows that are more similar are likelier to share more relationships with each other, and so merging the most similar pairs of rows will indirectly keep information loss minimal.

c1	c2	c3	c4	c5	c6	c7
A	B	D	E	G	J	K
A	C	D	F	H	I	K

$\rightarrow$

c1	c2	c3	c4	c5	c6	c7
A	*	D	*	*	*	K

Figure 5.4: A pair of rows can be efficiently merged by nulling non-identical cells.

Timewise, the algorithm involves first calculating the Jaccard similarity between every pair of rows: $O(m^2)$, and inserting them into a Priority Queue: a multiplication by $O(1)$. The Jaccard similarity involves calculating the union: $O(2n)$ and the intersection: $O(n)$, so the preliminary steps have a time complexity of $O(m^2 * 1 * (2n + n)) = O(m^2 * 3n) = O(m^2 * n)$. Each merge will reduce the size of the table by one, so there will be a total of $n - k$ merges done. Each individual merge procedure requires taking the highest valued item in the queue: $O(1)$, going through the columns of both rows to check for equality of values: $O(2n)$, iterating through the queue to find the old similarity values to delete: adding $O(m)$, and inserting the new similarity values into the queue: adding $m * O(1) = O(m)$, so each merge takes $O(1 + 2n + m + m) = O(2n + 2m + 1) = O(m + n)$ time. In total, the similarity algorithm has a time complexity of $O(m^2 * n + (n - k) * (m + n)) = O(m^2 * n + n^2 + m * n - k * n - k * m) = O(m^2 * n + n^2 + m * n - k * (m * n) = O(m^2 * n + n^2 - (k - 1) * m * n) \approx O(m^2 * n + n^2)$.

The Jaccard similarity can also be estimated instead by calculating the Minhash of all rows, which allows for a potential performance benefit when calculating many pairs or with a dataset with many columns: the similarity between two Minhashes can be calculated in $O(1)$ instead of $O(3n)$. This is what we do in Algorithm 10, reducing the time complexity of the preliminary steps from $O(m^2 * n)$ to $O(m^2)$. In

Algorithm 9 Similarity algorithm

Require: Table
Require: Desired size
Rows ← Table.Rows
RowPairSimilarity ← PriorityQueue()
for all Row1, Row2 ∈ Rows | Row1 ≠ Row2 **do**
    Pair ← Set(Row1, Row2)
    Similarity ← JACCARDSIM(Pair)
    RowPairSimilarity ← RowPairSimilarity ⋃ {(Pair, Similarity)}
while |Rows| > Desired size **do**
    BestPair ← POP HIGHEST(RowPairSimilarity)
    Row1, Row2 ← BestPair
    NewRow ← MERGEROWS(Row1, Row2)
    Rows ← Rows \ {Row1}
    Rows ← Rows \ {Row1}
    Rows ← Rows ⋃ {NewRow}
    for all Pair ∈ RowPairSimilarity **do**
        if Row1 ∈ Pair ∨ Row2 ∈ Pair **then**
            RowPairSimilarity ← RowPairSimilarity \ {Pair}
    for all Row ∈ Rows **do**
        Pair ← {(Row, NewRow)}
        Similarity ← JACCARDSIM(Pair)
        RowPairSimilarity ← RowPairSimilarity ⋃ {(Pair, Similarity)}
Table.Rows ← Rows
return Table

total, the *Minhash Similarity* algorithm has a time complexity of $O(m^2 * 1 + (n-k) * (m+n)) = O(m^2 + n^2 + m*n - k*n - k*m) = O(m^2 + n^2 - (k-1)*m*n) \approx O(m^2 + n^2)$. As shown in Table 5.1, this is the fastest time complexity out of all the investigated algorithms in this work.

Algorithm	Time Complexity	Optimal solution?
Exhaustive	$O((m*n)!)$	Yes
Greedy	$O((m*n)^2)$	No
Random Walks	$O((m*n)^2)$	No
Sorted Order	$O((m*n)!)$	Yes
Merge Greedy	$O((m*n)!)$	No
Similarity	$O(m^2*n+n^2)$	No
Minhash Similarity	$O(m^2+n^2)$	No

Table 5.1: A summary of the time complexities and optimality guarantees of all the above algorithms.

Algorithm 10 Minhash Similarity algorithm

Require: Table
Require: Desired size
Rows ← Table.Rows
RowMinHashes ← Dict()
for all Row ∈ Rows **do**
    RowMinHashes[Row] ← MINHASH(Row)
    RowPairSimilarity ← PriorityQueue()
for all Row1, Row2 ∈ Rows | Row1 ≠ Row2 **do**
    Pair ← Set(Row1, Row2)
    Similarity ← JACCARDSIM(RowMinHashes[Row1], RowMinHashes[Row2])
    RowPairSimilarity ← RowPairSimilarity ⋃ {(Pair, Similarity)}
while |Rows| > Desired size **do**
    BestPair ← POP HIGHEST(RowPairSimilarity)
    Row1, Row2 ← BestPair
    NewRow ← MERGEROWS(Row1, Row2)
    Rows ← Rows \ {Row1}
    Rows ← Rows \ {Row2}
    Rows ← Rows ⋃ {NewRow}
    RowMinHashes[NewRow] ← MINHASH(NewRow)
    for all Pair ∈ RowPairSimilarity **do**
        if Row1 ∈ Pair ∨ Row2 ∈ Pair **then**
            RowPairSimilarity ← RowPairSimilarity \ {Pair}
    for all Row ∈ Rows **do**
        Pair ← {(Row, NewRow)}
        Similarity ← JACCARDSIM(RowMinHashes[Row], RowMinHashes[NewRow])
        RowPairSimilarity ← RowPairSimilarity ⋃ {(Pair, Similarity)}
Table.Rows ← Rows
return Table

Chapter 6

Experiments

To demonstrate the effectiveness of our algorithms, we have conducted an extensive series of experiments that show significant variations in their quality and performance under different conditions. We performed these experiments on a Ubuntu server with 125.912 GB of RAM and a 16-core Intel Xeon E5-2660 2.20 GHz CPU. No parallel processing was used to run the algorithms.

6.1 Datasets

To examine the algorithms' performance and quality across a wide range of starting conditions, we first conducted experiments using synthetic datasets, allowing for granular control over the tested variables. We then tested with real-world datasets, where the variables are fixed based on actual data.

6.1.1 Synthetic Datasets

We created the synthetic datasets for our experiments by generating tables given the following parameters:

- m - Rows - The number of rows in the table.
- n - Columns - The number of columns in the table.
- k - Desired size - The maximum number of rows needed for a solution to count as valid.
- $|D|$ - Domain cardinality - The number of unique cell values allowed in each column.

Since the maximum value of the scoring metric (relationship counting) depends on the starting conditions, then whenever we tested several tables in sequence, such as when modulating the number of rows or columns to determine the effects, we created the smallest table first and then added the necessary number of rows or columns to create the larger tables. This way, most of the table remained the same, and thus the scores across tables could be more comparable with each other. Furthermore, to mitigate the effect of the starting conditions, we performed each test three times with different generated tables and report the average results.

6.1.2 Real Datasets

One real-world dataset we chose focuses on real estate information, specifically related to houses. We will refer to this dataset as *Homes*. The Homes dataset contains information about various residences for sale in

the US, specifically in the states of Florida and Texas. Each of the 1000 rows contains information about one house or apartment, noting the state where it is located, the price for which it is offered, the number of bedrooms, the number of bathrooms, and 15 binary variables for whether the described place has certain amenities, such as a swimming pool, garage, or smoke detector. Furthermore, any of the 19 columns in a row can sometimes contain a NULL value, which we replace with a ('*') as part of our preprocessing. Since our algorithms require categorical data, we apply a process called binning, which involves grouping continuous values into discrete categories. In this case, we bin the *price* values of each row into four broader categories: less than $100,000; between $100,000 - $500,000; between $500,000 - $1,000,000; and greater than $1,000,000.

To better understand the performance of our algorithms in different kinds of real datasets, we also use a dataset containing information about cars available for sale in the United States. We will refer to this dataset as *Cars*. The Cars dataset contains information about various cars available for sale in the US. The dataset is composed of 1000 rows, containing information about cars for sale in both New York and Texas. Each row contains data about one car, noting the state where it is located, the price for which it is offered, the make, color, number of cylinders, and 32 binary variables for whether the described car has certain amenities, such as a sunroof, heated seats, or a CD player. Furthermore, whenever the value of any of the 37 columns is listed as unknown, we replace it with a null value ('*'). For this dataset we also bin the continuous *price* values of each row into four broader discrete categories, this time we divide them into: less than $10,000; between $10,000 - $20,000; between $20,000 - $30,000; and greater than $30,000.

6.2 Algorithm configuration parameter impact

6.2.1 Varying the number of random walks

Before starting with the comparative experiments between the algorithms, any necessary hyperparameters must be tuned first. The Random Walks algorithm is the only algorithm described above that has a hyperparameter to be tuned — the number of walks it does through the tree. To figure out the best number to use, we generate 3 tables each of size 3x3 and 4x4 (rows x columns) with a fixed domain cardinality of 3 for every column, and have the Random Walks algorithm find the best table of 2 rows or below that it can find with 1-10, 15, 20, 30, 40, and 50 random walks. We measure the average time it takes for the Random Walks algorithm to complete. Figure 6.1 shows the average performance of the Random Walks algorithm on the 3x3 tables (left), and on the 4x4 tables (right).

We measure the hyperparameter's impact on the performance of the Random Walks algorithm by recording the time it takes to run the program on the above-generated tables with a different number of walks. We can observe that initially, the processing time increases linearly in proportion to the number of walks in both 3x3 and 4x4 size tables, as expected. However, in the smaller 3x3 tables, the time per walk starts to decrease after 30 walks, as evidenced by the decreasing slope of Figure 6.1 (left). That is because our Random Walks algorithm implements a pruning behavior: if the current random walk reaches a non-valid table with a lower score than the best valid solution found, then the walk is stopped prematurely. This is possible only because of the monotonicity of the score function — such a walk will never provide a result that scores higher than the previous best. The performance-boosting effect of pruning does not appear to have made an impact on the performance of Random Walks in the larger tables, as we can observe that Figure 6.1 (right) maintains linearity even at a larger number of walks. The likely reason for this is that with a larger decision space for a

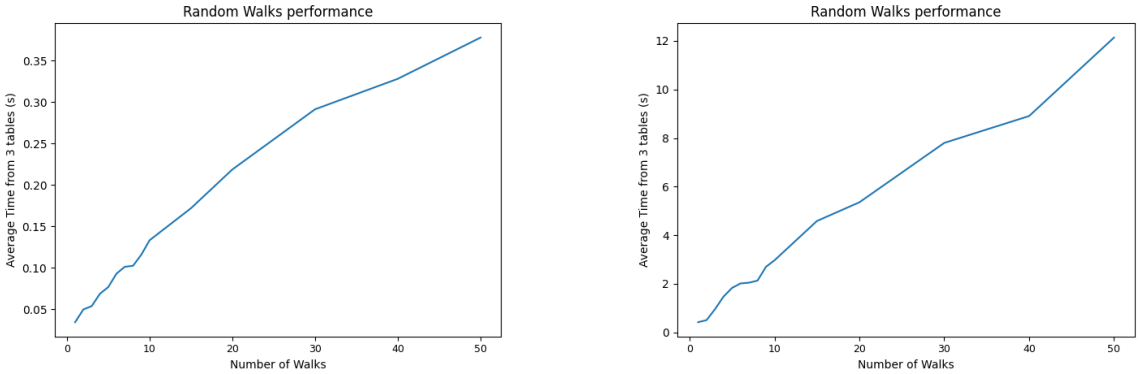

Figure 6.1: The average performance of the Random Walks algorithm on 3x3 (left) and 4x4 (right) tables as the number of walks increases.

random walk to cover, the probability of coming across one of the best solutions early on will be correspondingly lower, and thus it will be more likely for later random walks to still provide an improvement to the score. Therefore random walk pruning would happen far less frequently in the processing of larger datasets than with smaller tables.

In addition to measuring the performance, we also measure the changes in quality of the Random Walks algorithm that come from a differing number of walks by counting the number of certain relationships preserved by the output (the *score*). Figure 6.2 shows the average score of Random Walks on tables of 4 rows and 4 columns with a column domain cardinality of 3, when aiming to reduce the table into 2 rows with 1-10, 15, 20, 30, 40, and 50 random walks. We find that with a low number of walks, the score can vary quite a bit, but as the number of walks increases, so does the score overall. This is reasonable, as with a larger sample of random walks, it becomes likelier that at least one ends up with a higher-quality solution.

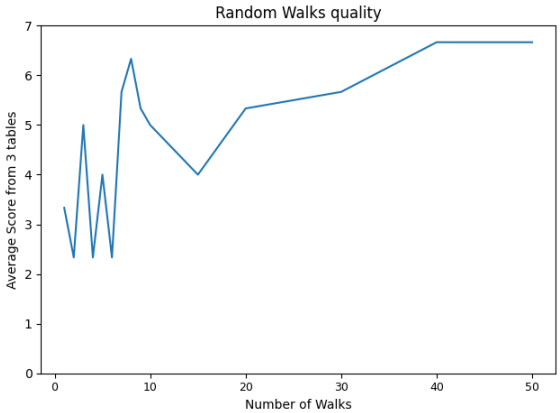

Figure 6.2: The average quality of the Random Walks algorithm on three 4x4 tables as the number of walks increases.

To compare the Random Walks algorithm with the other algorithms we investigate in this work, a fixed value should be chosen for the hyperparameter. Based on Figure 6.2, we choose it to be 10 random walks per application of the algorithm, as it is the lowest number of walks that have managed to get a consistently high score in this experiment.

6.3 Scalability

To determine the scalability, we measure the time it takes for each of the algorithms in Chapter 5 to find a solution to the problem as a function of the size of the input table. We run experiments wherein we change one variable out of the following three dimensions: the number of rows, the number of columns, and the cardinality of the column domains.

6.3.1 Performance over the varying number of rows

We create three different tables with 3 rows and 3 columns with a domain cardinality of 3. The number of columns and their domain cardinality are fixed. We gradually increase the number of rows from 3 to 20, by adding one new row at a time to the previous tables. We run our algorithms on each of those tables with the aim to bring each table down to 2 rows and measure the time elapsed until each algorithm outputs an answer. Since we have three different tables of each size, we report the average time taken per table size, per algorithm. The results are shown in Figure 6.3.

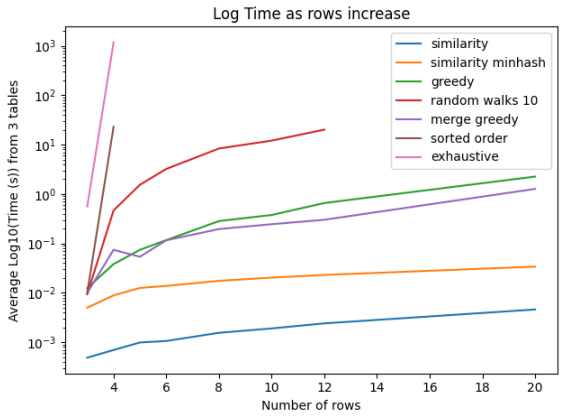

Figure 6.3: The average log performance of the algorithms on three tables as the number of rows increases, up to 20 rows.

We can observe that both the Exhaustive algorithm and the Sorted Order algorithm have a nearly vertical slope, and both run out of memory at only 5 rows. These are the only two algorithms that provide an optimal solution, while the faster algorithms are suboptimal. Out of these two, the Sorted Order algorithm provides the optimal solution around 50 times faster on average than the Exhaustive method, even after including pruning for the latter. The other five (non-optimal) algorithms show a scalability slope that is much faster than the optimal algorithms and that is very similar between themselves, but they still vary in absolute performance.

The Random Walks algorithm is reliably around 10 times slower than the Greedy algorithm for tables with 8 or more rows — this is expected, as 10 random walks should on average take 10 times longer to run than a single (greedy) walk down the tree. This gap decreases for smaller tables, which is further evidence of the benefits of pruning in our implementation of the Random Walks algorithm. As stated above in Section 6.2.1, random walk pruning stops having a noticeable impact when the chance of finding a good answer early is lower, which is the case for larger tables.

The Merge Greedy algorithm shows no reduction compared to the Greedy algorithm, and in fact, ends up being marginally faster in tables with 8 rows or more. This illustrates the time benefit of considering intermediate tables in order of score rather than by layers, in situations where merges are easy to reach due

to the low number of columns. More specifically: the Greedy algorithm needs to evaluate every child of an expanded node to find the best-scoring one to expand next, while the Merge Greedy will stop the current expansion and skip straight to expanding the next whenever it finds a child that has a smaller number of rows than its parent.

The similarity-based algorithms are consistently the most time-efficient. Similarity without minhashing outperforms both greedy algorithms by two orders of magnitude, while including minhashing reduces that gap to a single order of magnitude. The benefits can be attributed to the more efficient method of merging entire rows rather than a single nulling operation at a time as is done with the Expand method. The single magnitude time difference between Similarity with and without minhashing comes down to the added overhead that comes from needing to perform a minhash on every row. With a large number of rows, many minhashes would need to be computed. Furthermore, the Jaccard similarity scales by the number of columns ($O(3n) = O(n)$ for calculating both the union and intersection). With a low number of columns, the benefit of approximating the Jaccard similarity will not be very significant, evidently failing to offset the added cost of performing many minhashes.

We test this last comparison further by running only the Similarity and Minhash Similarity algorithms on tables with many more rows than before, more closely mimicking the size of a real dataset. We create three different tables with 10 rows and 4 columns with a domain cardinality of 3. We gradually increase the number of rows from 10 to 50, 100, 200, 300, 500, 1000, 2000, 5000, 10000, 20000, 50000, and finally 100000 rows. We run both Similarity and Minhash Similarity on each of those tables with the aim to bring each table down to 2 rows and measure the time elapsed until either algorithm outputs an answer. Since we have three different tables of each size, we report the average time taken per table size, per algorithm. The results are shown in Figure 6.4.

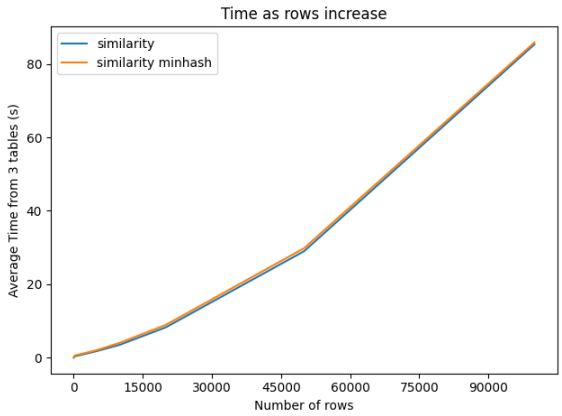

Figure 6.4: The average performance of the similarity algorithms on three tables as the number of rows increases, up to 100,000 rows.

From these results, we can see that both similarity-based algorithms can reduce datasets with even up to 100,000 rows down to 2 rows in under two minutes if the number of columns is low. Furthermore, the difference between the two algorithms created by either the incurred overhead or benefits of minhashing becomes proportionally tiny as the number of rows increases to the point of being undetectable.

6.3.2 Performance over the varying number of columns

We create three different tables with 3 rows and 2 columns with a domain cardinality of 3. The number of rows and the columns' domain cardinality are fixed. We gradually increase the number of columns from 2 to 30, by adding one new column at a time. We run our algorithms on each of those tables with the aim to bring each table down to 2 rows and measure the time elapsed until each algorithm outputs an answer. Since we have three different tables of each size, we report the average time taken per table size, per algorithm. The results are shown in Figure 6.5 (left). Then, we start adding columns up to 50 and then add more columns in multiples of 50 up to 1000 columns. For the tables with more than 30 columns, we run only the Similarity and Minhash Similarity algorithms due to memory constraints. These results are shown in Figure 6.5 (right).

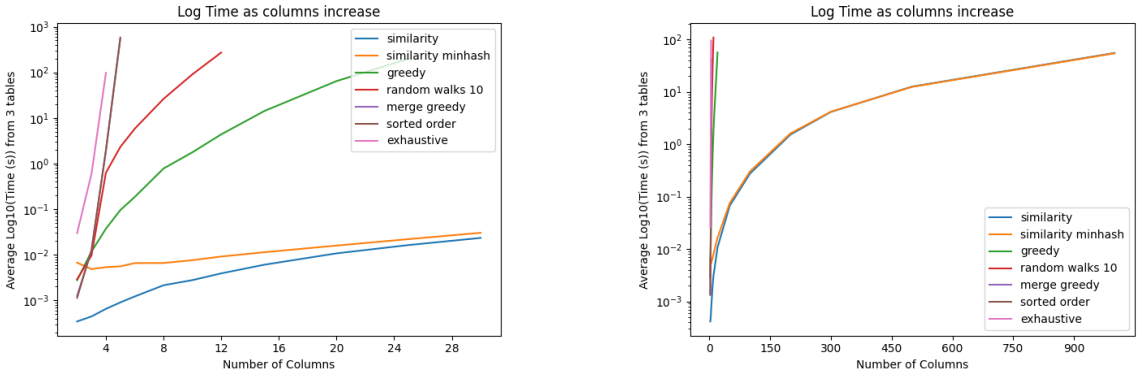

Figure 6.5: The average log performance of the algorithms on three tables as the number of columns increases, up to 30 columns (left) and up to 1000 columns for the fastest algorithms (right).

From Figure 6.5 (left), we can observe that the Exhaustive, Sorted Order, and Merge Greedy algorithms have a nearly vertical slope. The Exhaustive algorithm runs out of memory at 4 columns, while the Sorted Order and Merge Greedy algorithms run out of memory at 5 columns. Comparing the algorithms that provide an optimal solution, Sorted Order still produces an optimal answer around 50 times faster than the Exhaustive algorithm on the same tables.

When comparing an increase in rows with an increase in columns, the major difference is that the Merge Greedy algorithm performs identically to the Sorted Order algorithm, despite having no optimality guarantees. That is due to the higher number of columns present in this table, which leads to an increased difficulty in finding a merge. In the absence of easily discoverable merges, the Merge Greedy algorithm will function as the Sorted Order algorithm does, explaining the similarly high scalability curve.

The gap between Random Walks and Greedy remains the same as with an increase in rows, while both of these algorithms are once again significantly outperformed by the similarity-based algorithms. This time, however, the difference between Similarity and Greedy is vastly larger, up to 4 orders of magnitude compared to 2, with the gap being ever-increasing. With a constant low number of rows, the more efficient method of full row merging will only need to be performed a constantly low number of times, while the number of nulling operations required increases with every added column. Thus, extra columns demand much more processing from algorithms that null one cell at a time compared to those that null entire rows at once. This difference is further highlighted in Figure 6.5 (right), where only the similarity-based algorithms were able to produce valid reduction solutions to tables with many columns, given the memory constraints of our experimental setup.

Comparing Similarity with and without minhashing, we can observe from Figure 6.5 (left) that the low overhead of the former gives a concrete benefit for tables with a low number of columns. However, as the number of columns increases, the added processing time from performing union and intersection operations on ever larger sets adds a cost to the Similarity algorithm that is not present in Minhash Similarity, bringing the performance of the two algorithms closer together.

To find out whether there is an inflection point after which the addition of minhashing reduces the total processing time, we run only the similarity-based algorithms on tables with many more columns than before, more closely mimicking the size of a real dataset. We created three different tables with 4 rows, and 10 columns with a domain cardinality of 3. We gradually increase the number of columns from 10 to 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, and finally 1200 columns. We aim to bring these tables down to 2 rows and measure the time elapsed until either algorithm outputs an answer. Since we have three different tables of each size, we report the average time taken per table size, per algorithm. The results are shown in Figure 6.6.

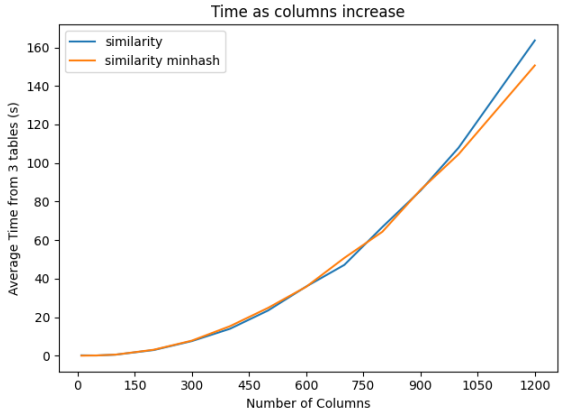

Figure 6.6: The average performance of the similarity algorithms on three tables as the number of columns increases, up to 1200 columns.

From these results, we can see that both similarity-based algorithms can reduce datasets with up to 1200 columns in under three minutes if the number of rows is low. Furthermore, we find that there is an inflection point (in this case at around 900 columns), after which it becomes faster to approximate the Jaccard similarity using minhashing, rather than repeatedly calculating the unions and intersections of large sets. The gap appears to increase proportionally to the number of columns added after this point.

6.3.3 Performance over the varying cardinality of domains

To test the cardinality of the domains, we create different tables with a fixed size, but increasingly large column domains. For each integer from 2 to 20, we create three tables of 4 rows and 3 columns, with the domain cardinality of all columns being that value. We run our algorithms on each of those tables with the aim to bring each table down to 2 rows and measure the time elapsed until each algorithm outputs an answer. Since we have three different tables of each column domain cardinality, we report the average time taken per column domain, per algorithm. The results are shown in Figure 6.7, which displays the average performance of the algorithms as the cardinality of the column domain increases from 2 to 20.

We can observe that the change in the cardinality of the column domains had no impact on the perfor-

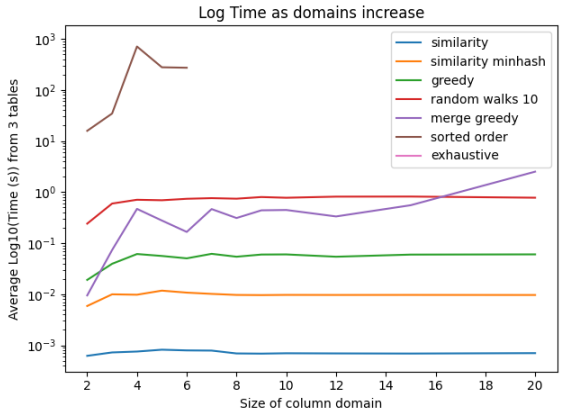

Figure 6.7: The log performance of the algorithms on three tables as the cardinality of the column domains increases.

mance of most of the algorithms, as expected. The Merge Greedy algorithm, however, suffers a decrease in performance as the column's domains increase in size. An increase in the domain results in a more diverse dataset, which causes the rows to become more dissimilar. If the rows become less similar, then merges will become more difficult, thus increasing the amount of time the Merge Greedy must exhaustively consider all options ordered by score until it can find the first possible merge.

The Sorted Order and Exhaustive algorithms were also observed to increase their processing time as the column domains became more diverse. The former ran out of memory after the cardinality of the column domain became 6, while the latter was only able to handle a column domain of cardinality 2. With only one datapoint, the Exhaustive algorithm does not appear in Figure 6.7, but the log file shows that its average time was 895 seconds to process, around 60 times longer than Sorted Order for the same tables. A fully exhaustive algorithm would consider every possible order of nulling operations, which remains constant if the table size does not change. However, our implementation includes the pruning of branches that are known not to provide an optimal solution, exploiting the monotonicity of our chosen quality metric. As the diversity of the dataset increases, it is less likely that some solutions will be significantly better than others. As such, pruning will become less frequent and the algorithm will have to consider more options, using more time and memory. The reason behind the Sorted Order algorithm's increase in time and memory is similar to what happened to the Merge Greedy algorithm: if the rows become less similar, merges become more difficult, and more of the decision space will need to be searched by the Sorted Order algorithm in order to find the optimal solution.

6.3.4 Performance on the Homes dataset

We test the performance of the similarity-based algorithms on the 1000 rows of the Homes dataset. Figure 6.8 displays the time it takes for Similarity and Minhash Similarity algorithms to process this data into merged tables of different sizes. We set the final size (budget) to go from 100 rows to 900 rows, in steps of 100 rows.

We observe that the time it takes to reduce the dataset from 1000 rows to 900, 800, 700, and 600 is identically low. The reason behind this is that the Homes dataset contains slightly over 400 homes that are identical to at least one other home in the dataset. These duplicate rows get automatically merged by the similarity-based algorithms before any nulling can take place, which explains the consistently fast performance of the

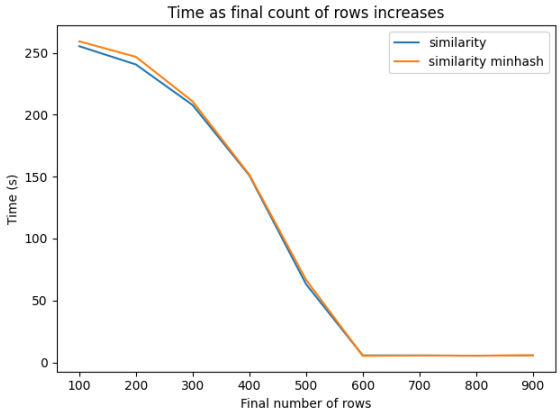

Figure 6.8: The performance of the similarity algorithms on the Homes dataset as the final number of rows increases.

algorithms on this dataset until that point.

Beyond that point, the smaller the requested final size of the reduced dataset, the longer it takes for the similarity-based algorithms to finish, as more pairs of rows would need to be merged. However, one could expect this graph to be linear, as every time the dataset needs to be reduced by another row, only one pair of rows will need to be merged. Yet we observe that the slope is not linear, but gradually changing: it is faster to reduce an extra 100 rows from 200 to 100 than it is to do so from 600 to 500.

The explanation lies in the information-management component of the similarity-based algorithms. Whenever a pair of rows is reduced, then those rows need to be removed from the heap that contains all currently available row pairs, and new Jaccard similarities (or approximations of such) must be calculated between the newly merged row, and every other row still in the heap. As the dataset is further merged and the number of rows decreases, there will be fewer row pairs available, and the heap will become ever smaller. Fewer row pairs require fewer similarity calculations, and updating the heap will also be faster.

6.3.5 Performance on the Cars dataset

We test the performance of the similarity-based algorithms on the 1000 rows of the Cars dataset. Figure 6.9 displays the time it takes for Similarity and Minhash Similarity algorithms to process this data into merged tables of different sizes. We set the final size (budget) to go from 100 rows to 900 rows, in steps of 100 rows.

We observe very similar results as above with the Homes dataset, with the exception that there is no 0-slope line on the right side of the graph. The Cars dataset contains almost double the amount of columns than the Homes dataset (37 as opposed to 19), and thus the chances of any two cars having identical attributes are significantly lower. We also see the scalability slope decreasing as the requested final size of the dataset decreases, just as with the Homes dataset. As mentioned above, this can be explained by there being fewer recorded row pair similarities to update when merging an interim dataset with fewer rows.

6.4 Quality

To determine the quality of the algorithms, we measure the number of certain relationships preserved by the outputs of the above experiments. We call this number the *score* of an algorithm on a given table. As before,

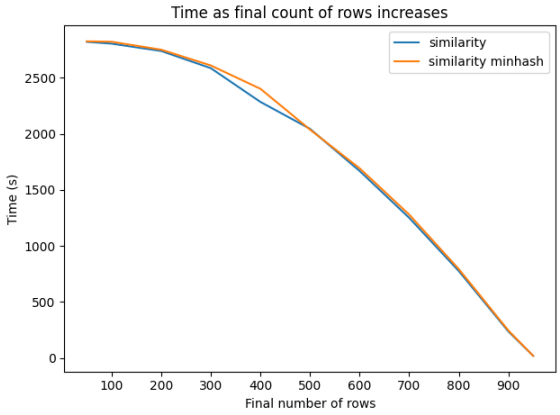

Figure 6.9: The performance of the similarity algorithms on the Cars dataset as the final number of rows increases.

we run experiments wherein we change one variable out of the following three dimensions: the number of rows, the number of columns, and the cardinality of the column domains.

6.4.1 Quality over the varying number of rows

We create three different tables with 3 rows and 3 columns with a domain cardinality of 3. The number of columns and their domain cardinality are fixed. We gradually increase the number of rows from 3 to 20, by adding one new row at a time. We run our algorithms on each of those tables with the aim to bring each table down to 2 rows and measure their quality by the number of certain relationships preserved in the valid answers output by the algorithms (the score). Since we have three different tables of each size, we report the average score achieved per table size, per algorithm. The results are shown in Figure 6.10.

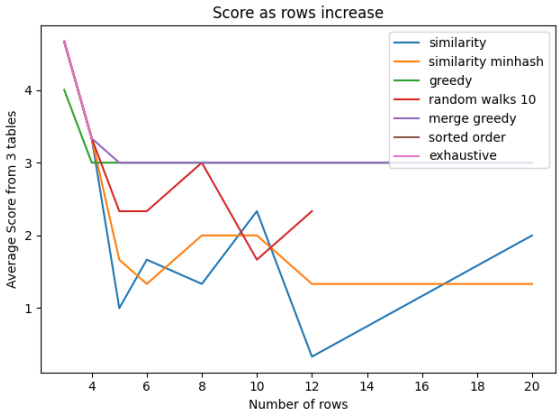

Figure 6.10: The average score of the algorithms on three tables as the number of rows increases, up to 20 rows.

We can observe that both the Exhaustive algorithm and the Sorted Order algorithm run out of memory at only 5 rows. These are the only algorithms that provide an optimal solution — both in theory, and also now in practice, as the scores of their answers match exactly, and no other tested algorithm has provided better scoring answers.

The Random Walks algorithm provides answers with scores that can be relatively high at first, but that on average decrease as the number of rows increases. With a larger space for the random walks to explore, the algorithm will struggle to stumble upon the better answers that are proportionally fewer compared to the total number of all possible walks.

The Greedy and Merge Greedy algorithms prove to maintain a consistently high quality throughout. They achieve identical scores on almost all tested tables, except on the smaller tables with 2, 3, or 4 rows, in which Merge Greedy gets a slightly better result. From these tests, one could conclude that the situations in which the Greedy algorithm gets stuck in local optima are relatively rare in larger tables, but more common in smaller tables. In the latter, the Merge Greedy algorithm did not fall into local optima like the Greedy algorithm did, as it provided optimal solutions identical to the outputs of the Exhaustive and Sorted Order algorithms.

As expected, the similarity-based algorithms had consistently worse quality than the other algorithms (except for Random Walks at 10 rows). Unlike the others, these algorithms do not directly optimize for the quality metric (the count of certain relationships), but instead, they approach it indirectly, by finding and merging similar rows, which is likely the reason behind this reduction in quality. From Figure 6.10, it is unclear whether minhashing has a positive or negative effect on the score as the number of rows increases, as it seems to depend on the particular generated tables. We explore this comparison further by running only the Similarity and Minhash Similarity algorithms on tables with many more rows than before, more closely mimicking the size of a real dataset.

We created three different tables with 10 rows and 4 columns with a domain cardinality of 3. We gradually increase the number of rows from 10 to 50, 100, 200, 300, 500, 1000, 2000, 5000, 10000, 20000, 50000, and finally 100000 rows. We run both Similarity and Minhash Similarity on each of those tables with the aim to bring each table down to 2 rows and measure the score of the output solution per table, per algorithm. The results are shown in Figure 6.11.

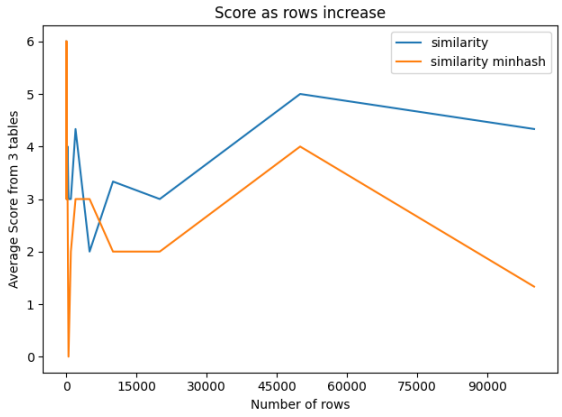

Figure 6.11: The average score of the similarity algorithms on three tables as the number of rows increases, up to 100,000 rows.

From these results, we can see that before 10,000 rows, the score of both of these algorithms can be highly variable. After 10,000 rows, however, it can clearly be seen that the Similarity algorithm without minhashing consistently outperforms Minhash Similarity. The only difference between the two is the added error in determining the most similar pair of rows at any given moment, which is caused by approximating the Jaccard similarity with minhashes instead of computing them directly. The difference only becomes ever

more pronounced as the number of rows increases from 50,000 up to 100,000 rows: more rows to merge means more approximations are needed.

6.4.2 Quality over the varying number of columns

We create three different tables with 3 rows and 3 columns with a domain cardinality of 3. The number of rows and the column domains' cardinality are fixed. We gradually increase the number of columns from 2 to 30, by adding one new column at a time. We run our algorithms on each of those tables with the aim to bring each table down to 2 rows and measure their quality by the number of certain relationships preserved in the valid answers output by the algorithms (the score). Since we have three different tables of each size, we report the average score achieved per table size, per algorithm. The results are shown in Figure 6.12.

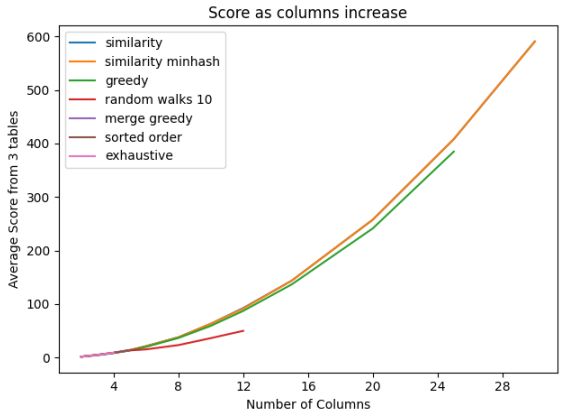

Figure 6.12: The average score of the algorithms on three tables as the number of columns increases, up to 30 columns.

We can observe that the faster algorithms still get optimal or near-optimal solutions in the small ranges where the optimal algorithms do not yet run out of memory (up to 5 columns). As the number of columns increases, the answer space increases in a similar manner as with an increase in rows, so the Random Walks algorithm suffers similarly in quality.

The main difference from the last test is that the similarity-based algorithms are now producing equal or better quality solutions than every other tested algorithm. This is most likely due to the low number of rows, a constant that is maintained throughout this experiment. The main decision point behind the similarity-based algorithms is the choice of which row pair to choose for merging. As only the number of columns increases, this decision space remains constantly small, allowing the similarity-based algorithms to maintain a consistently high quality. It is easier to choose which row pairs to merge if there are fewer row pairs to pick. As for the other, tree-expansion-based algorithms, their decision space is the number of possible nulling operations, and that amount increases with every added column.

From Figure 6.12, it is unclear whether minhashing has a positive or negative effect on the score as the number of columns increases, as they both seem to produce identical results. We explore this comparison further by running only the Similarity and Minhash Similarity algorithms on tables with many more columns than before, more closely mimicking the size of a real dataset. We create three different tables with 4 rows and 10 columns with a domain cardinality of 3. Then we gradually increase the number of columns from 10 to 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, and finally 1200 columns. We aim to bring these

tables down to 2 rows and measure the score of the output solution per table, per algorithm. The results are shown in Figure 6.13.

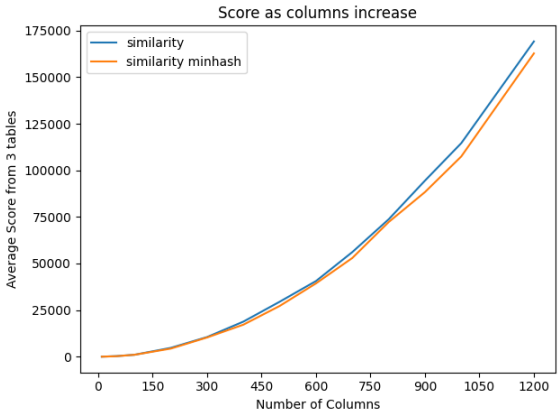

Figure 6.13: The average score of the similarity algorithms on three tables as the number of columns increases, up to 1200 columns.

From these results, we can see that the Similarity algorithm without minhashing seems to have around the same score as with minhashing up to 800 rows. After that, the Similarity algorithm without minhashing provides marginally better answers than the version that uses minhashing. As the number of columns increases, the sets whose unions and intersections need to be calculated to obtain the Jaccard similarities become larger. Thus, the minhash approximation eventually becomes less accurate, which results in a reduction in final output quality.

6.4.3 Quality over the varying cardinality of domains

To test the cardinality of the domains, we create different tables with a fixed size, but increasingly large column domains. For each integer from 2 to 20, we create three tables of 4 rows and 3 columns, with the domain cardinality of all columns being that value. We run our algorithms on each of those tables with the aim of bringing each table down to 2 rows and measuring their quality by the number of certain relationships preserved in the valid answers output by the algorithms (the score). Since we have three different tables of each size, we report the average score achieved per column domain, per algorithm. The results are shown in Figure 6.14, which displays the average score of the algorithms as the cardinality of the column domain increases from 2 to 20.

We can observe that with a very small column domain (2 and 3), all algorithms provide near-optimal outputs. That is most likely due to the fact that with a very repetitive dataset, most available choices to merge are unlikely to permanently remove any certain relationships, and as such the algorithms reach a valid solution before losing too many.

As the diversity of the dataset increases, the quality of each algorithm remains constant. As seen above, the similarity-based algorithms generally give lower quality results than the others, likely because they do not directly optimize for the quality metric. Only the Random Walks algorithm has a variable score due to its unpredictable nature.

Interestingly enough, as the size of the column domain increases beyond four times the number of rows (datapoints per column), the scores of all algorithms start converging again. This is likely a specific special

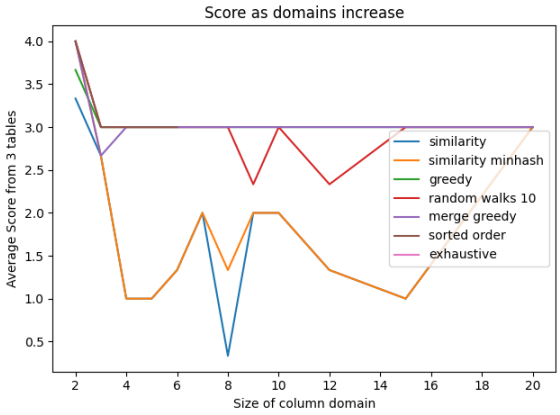

Figure 6.14: The average score of the algorithms on three tables as the cardinality of the column domains increases.

case: as the domain cardinality becomes much larger than the number of rows, then this will increase the likelihood that nearly every single datapoint is unique. In such a situation, the only way to merge two rows is to fill them with nulls in their entirety, which vastly limits the number of achievable valid solutions. This means that most algorithms would be likely to end up scoring similarly in such a situation, explaining the convergence in the achieved score.

6.4.4 Quality on the Homes dataset

We test the quality of the similarity-based algorithms on the 1000 rows of the Homes dataset. Figure 6.15 displays the score of the final results after both the Similarity and Minhash Similarity algorithms process this data into merged tables of different sizes. We set the final size (budget) to go from 100 rows to 900 rows, in steps of 100 rows. Furthermore, as a point of comparison with our similarity-based algorithms, we also measure the number of certain relationships kept after deleting enough rows from the end of the dataset such that we only keep the first 100, 200, 300,..., up to 900 rows of the Homes dataset.

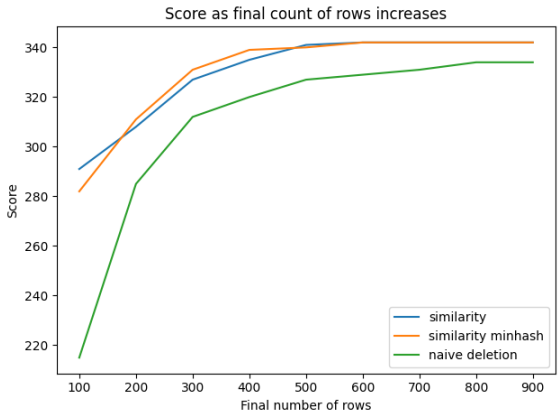

Figure 6.15: The quality of the similarity algorithms on the Homes dataset as the final number of rows increases, compared to the deletion of rows.

We observe that for up to 600 rows, the score does not change. As mentioned above, this is because there

are slightly over 400 homes in the dataset that are identical to at least one other existing row in the table. As the duplicate rows get merged before any nulling can take place, there is no reduction in the number of certain relationships.

Beyond the 600-row point, the score remains similar for a further reduction of 200 rows for the outputs of both similarity-based algorithms. This is likely because those 200 rows are highly similar (but not identical) to other existing rows preserved in the 600-row iteration of the dataset. Merging those 200 rows would thus cause the loss of only very few certain relationships. Thus, the similarity-based merging algorithms could safely reduce the Homes dataset by up to 60% of its original size, without a significant loss of information. Merging more after that causes the slope of the score to increase significantly. The remaining rows are likely more different from each other, and thus as more rows get removed, the faster the score drops — merging two dissimilar rows will require adding more nulls, which increases the chances that some certain relationships will be permanently removed from the final merged table.

In contrast, simply deleting the same number of rows as opposed to merging them is shown to provide consistently worse-scoring results. Every row deletion risks the permanent removal of certain relationships. The difference keeps increasing as more of the original dataset is reduced, and it becomes especially notable at large reduction percentages. After 90% of the dataset has been reduced, the similarity-based merge algorithms managed to keep 80% and 85% of the certain relationships of the full dataset (with and without minhash respectively), while just deleting 90% of the rows drops that number down to 60%. Thus, similarity-based row merging is shown to reduce the Homes dataset much better than deletion, for any reduction in size.

6.4.5 Quality on the Cars dataset

We test the quality of the similarity-based algorithms on the 1000 rows of the Cars dataset. Figure 6.16 displays the score of the final results after both the Similarity and Minhash Similarity algorithms process this data into merged tables of different sizes. We set the final size (budget) to go to 50 rows, and from 100 rows to 900 rows, in steps of 100 rows. Furthermore, as a point of comparison with our similarity-based algorithms, we also measure the number of certain relationships kept after deleting enough rows from the end of the dataset such that we only keep the first 50, 100, 200, 300, . . . , up to 900 rows of the Cars dataset.

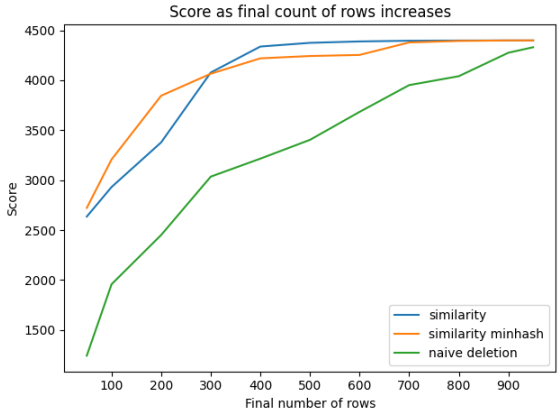

Figure 6.16: The quality of the similarity algorithms on the Cars dataset as the final number of rows increases, compared to the deletion of rows.

We observe very similar results in the Cars dataset as above with the Homes dataset. Even though in this

data there were no identical rows that would merge immediately, as was the case in the Homes dataset, the similarity-based algorithms still managed to preserve almost all of the around 4400 certain relationships of the full 1000-row dataset with only 400 rows. The row merging algorithms could thus safely reduce the Cars dataset by up to 60% of its original size. without a significant loss of information. Merging more after that did cause the slope of the score to significantly increase, for the same likely reason as above — the remaining rows are likely more different from each other.

The difference between similarity-based merging and deletion is even more pronounced in the Cars dataset. Since the Cars dataset has more columns, the rows are more dissimilar, and deleting rows causes the loss of unique certain relationships much earlier. After 60% of the dataset had been reduced, the similarity-based merge algorithms managed to keep 97% of the certain relationships, while deletion kept only 70%. At large reduction percentages, there is an even bigger difference: after 95% of the dataset had been reduced, the similarity algorithms kept 59%, while deletion kept 27% of all certain relationships of the original table. Thus, similarity-based row merging is shown to also reduce the Cars dataset much better than deletion, for any reduction in size.

6.5 Discussion

The choice of the best algorithm to use depends on the characteristics of the dataset, with different algorithms excelling under specific conditions. We can look at these conditions along two dimensions: the number of rows and the number of columns in the dataset, resulting in four broad categories. First, for small tables with a low number of rows and columns, the Sorted Order algorithm is the most reliable option. It guarantees an optimal solution to the problem and is approximately 50 times faster than the Exhaustive approach, even when pruning techniques are applied. This efficiency makes it the best choice for datasets of limited size. Second, as the number of rows increases while the number of columns remains low, the Merge Greedy algorithm becomes the preferred solution. This algorithm balances speed and quality effectively, offering significant performance advantages over Sorted Order while maintaining a higher level of solution quality than a purely Greedy algorithm. If the table contains more rows than the Merge Greedy algorithm can manage efficiently, the Similarity algorithm should be used. Specifically, the version of Similarity without minhashing is recommended, as the approximation introduced by minhashing noticeably reduces answer quality without providing any performance benefits in these cases. Third, for datasets with a large number of columns and relatively few rows, the Minhash Similarity algorithm stands out as the ideal choice. Its ability to handle tuples with many attributes with exceptional speed makes it far more efficient than any other tested algorithm under these conditions, without compromising solution quality. Finally, when dealing with large tables that contain both many rows and many columns — a common scenario in real-world applications — similarity-based algorithms prove to be the most effective. Both variants of the Similarity algorithm are highly time- and memory-efficient, making them well-suited for handling complex datasets. Additionally, they preserve significantly more information compared to the simple approach of discarding rows, ensuring that the integrity of the dataset is maintained for further use cases.

Chapter 7

Conclusion

Data reduction is a critical challenge in the field of data sustainability, particularly as the amount of stored data continues to grow. Existing solutions often rely on methods such as deleting rows or summarizing data, which result in significant loss of information and reduced utility of the datasets. In this work we have introduced Row Merge, a novel approach to data reduction that minimizes information loss by introducing controlled noise into the data. This technique allows to reduce the size of the dataset while preserving more information within the data compared to traditional methods. To quantify how much information is preserved or lost, we developed new metrics based on counting the certain and possible relationships retained in the reduced table. These metrics provide a precise way to evaluate the quality of incomplete databases and ensure that the reduction process is both measurable and meaningful.

We also presented seven algorithms designed to perform Row Merge data reduction, categorized into three distinct approaches. Initially, we explored tree expansion algorithms based on classical techniques: Exhaustive search, Greedy search, and Random Walks. Each has its strengths and limitations — Exhaustive search guarantees optimality but is computationally expensive, Greedy search sacrifices long-term performance for short-term gains, and Random Walks lacks reliability. Recognizing these limitations, we proposed two modified variants of these algorithms, Sorted Order and Merge Greedy. Sorted Order maintains optimality while being approximately 50 times faster than Exhaustive search, making it suitable for small datasets. Merge Greedy offers a practical balance between speed and quality, particularly when the dataset has many rows but relatively few columns. Finally, we developed a fundamentally novel approach based on pair-similarity algorithms: Similarity and Minhash Similarity. These algorithms focus on merging entire rows at once based on their overall similarity rather than nulling individual cells, resulting in far greater efficiency.

We validated our methods through extensive experiments, evaluating both scalability and quality. The results demonstrated that the best choice of algorithm depends on the characteristics of the dataset. Sorted Order is optimal for small datasets, Merge Greedy performs well for datasets with many rows and few columns, Minhash Similarity excels for datasets with many columns and few rows, and Similarity is best for datasets with many rows and columns — common in real-world scenarios. Our algorithms significantly outperformed the simple approach of deleting rows, preserving more information and yielding superior results for practical applications.

Future work could further enhance the Row Merge data reduction method by adding parallelization to the process. One possible approach could involve dividing the dataset into groups of similar rows, effectively creating smaller sub-tables that represent smaller instances of the same problem. Each group could then be

processed independently using slower but higher-scoring algorithms, potentially resulting in more information preservation and solutions that are closer to optimal. By leveraging parallelization, these computations can be performed simultaneously, significantly reducing the overall processing time. This approach would be particularly advantageous in cloud-based environments, where large numbers of commodity computers are readily available for distributed processing. The main challenge would lie in developing methods to group rows most efficiently. Similar rows should ideally be grouped together while dissimilar rows are separated, and the groups must be small enough to allow commodity computers to handle the computations quickly. Additionally, this parallelized approach would need to be rigorously evaluated in terms of both quality and scalability, especially when applied to large-scale datasets. A key benchmark would be its performance against our best single-computer algorithm for such cases — the pairwise Similarity algorithm. By addressing these challenges, parallelization could offer significant improvements in the practicality of the Row Merge data reduction technique for handling the massive volumes of data prevalent in the modern world.

REFERENCES

[AE15] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015.

[Agg06] Charu C Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618, 2006.

[Ahm19] Mohiuddin Ahmed. Data summarization: a survey. *Knowledge and Information Systems*, 58(2):249–273, 2019.

[AKG91] Serge Abiteboul, Paris Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theoretical computer science*, 78(1):159–187, 1991.

[AMP+13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*, pages 29–42, 2013.

[AN08] Hamza A Ali and Bashar M Ne'ma. Effective variations on opened gif format images. *IJCSNS*, 8(5):70–75, 2008.

[BBM+97] Michael Baentsch, L Baun, Georg Molter, Steffen Rothkugel, and P Sturn. World wide web caching: The application-level view of the internet. *IEEE Communications Magazine*, 35(6):170–178, 1997.

[BGR01] Shivnath Babu, Minos Garofalakis, and Rajeev Rastogi. Spartan: A model-based semantic compression system for massive data tables. *ACM SIGMOD Record*, 30(2):283–294, 2001.

[BM04] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[BPC+07] Gleb Beliakov, Ana Pradera, Tomasa Calvo, et al. *Aggregation functions: A guide for practitioners*, volume 221. Springer, 2007.

[bYW12] Sarah binti Yusoff and Yap Bee Wah. Comparison of conventional measures of skewness and kurtosis for small sample size. In *2012 International Conference on Statistics in Science, Business and Engineering (ICSSBE)*, pages 1–6. IEEE, 2012.

[CM05] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[Cor17] Graham Cormode. Data sketching. *Communications of the ACM*, 60(9):48–55, 2017.

[Cor23] Graham Cormode. Applications of sketching and pathways to impact. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 5–10, 2023.

[CPM22] Juan David Arias Correa, Alex Sandro Roschildt Pinto, and Carlos Montez. Lossy data compression for iot sensors: A review. *Internet of Things*, 19:100516, 2022.

[CZH+22] Zhiwei Cao, Xin Zhou, Han Hu, Zhi Wang, and Yonggang Wen. Toward a systematic survey for carbon neutral data centers. *IEEE Communications Surveys & Tutorials*, 24(2):895–936, 2022.

[CZY17] Meghan K Cain, Zhiyong Zhang, and Ke-Hai Yuan. Univariate and multivariate skewness and kurtosis for measuring nonnormality: Prevalence, influence and estimation. *Behavior research methods*, 49:1716–1735, 2017.

[DGM+23] Susan B Davidson, Shay Gershtein, Tova Milo, Slava Novgorodov, and May Shoshan. Efficiently archiving photos under storage constraints. In *EDBT*, pages 591–603, 2023.

[DMT+19] Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Sadoc Menasche, and Yong Chiang Tay. A utility optimization approach to network cache design. *IEEE/ACM Transactions on Networking*, 27(3):1013–1027, 2019.

[FKMP05] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[FNNT14] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical ttl-based cache networks. *Computer Networks*, 65:212–231, 2014.

[FRP18] Andres Ferragut, Ismael Rodriguez, and Fernando Paganini. Optimal timer-based caching policies for general arrival processes. *Queueing Systems*, 88(3):207–241, 2018.

[GMM+24] Boris Glavic, Giansalvatore Mecca, Renée J Miller, Paolo Papotti, Donatello Santoro, Enzo Veltri, et al. Similarity measures for incomplete database instances. In *Advances in Database Technology-EDBT*, volume 27, pages 461–473. OpenProceedings. org, 2024.

[GMS22] Sergio Greco, Cristian Molinaro, and Francesca Spezzano. *Incomplete data and data dependencies in relational databases*. Springer Nature, 2022.

[GNHW07] Alasdair JG Gray, Werner Nutt, and M Howard Williams. Answering queries over incomplete data stream histories. *International Journal of Web Information Systems*, 3(1/2):41–60, 2007.

[GRR19] John F Gantz, David Reinsel, and John Rydning. The us datasphere: Consumers flocking to cloud. *White Paper*, 2019.

[Hos12] Mohammad Hosseini. A survey of data compression algorithms and their applications. *Network Systems Laboratory, School of Computing Science, Simon Fraser University, BC, Canada*, 2012.

[ILJ84] Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. *Journal of the ACM (JACM)*, 31(4):761–791, 1984.

[JMN99] HV Jagadish, Jason Madar, and Raymond T Ng. Semantic compression and pattern extraction with fascicles. In *VLDB*, volume 99, pages 186–97, 1999.

[JNOT04] HV Jagadish, Raymond T Ng, Beng Chin Ooi, and Anthony KH Tung. Itcompress: An iterative semantic compression algorithm. In *Proceedings. 20th International Conference on Data Engineering*, pages 646–657. IEEE, 2004.

[Ker15] Martin L Kersten. Big data space fungus. In *CIDR*, 2015.

[KG14] Andreas Krause and Daniel Golovin. Submodular function maximization. *Tractability*, 3(71-104):3, 2014.

[KS17] Martin L Kersten and Lefteris Sidirourgos. A database system with amnesia. In *CIDR*, 2017.

[Lan84] Glen G Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, 1984.

[LB11] Hui Lin and Jeff Bilmes. A class of submodular functions for document summarization. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, pages 510–520, 2011.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, 2002.

[Lev96] Alon Y Levy. Obtaining complete answers from incomplete databases. In *VLDB*, volume 96, pages 402–412. Citeseer, 1996.

[LIL15] Dong Kyu Lee, Junyong In, and Sangseok Lee. Standard deviation and standard error of the mean. *Korean journal of anesthesiology*, 68(3):220–223, 2015.

[LKG+07] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429, 2007.

[LLR06] Ying Li, Seng W Loke, and MV Ramakrishna. Energy-saving data approximation for data and queries in sensor networks. In *2006 6th International Conference on ITS Telecommunications*, pages 782–785. IEEE, 2006.

[LW85] Lena Linde and Yvonne Waern. On search in an incomplete database. *International journal of man-machine studies*, 22(5):563–579, 1985.

[MAN14] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on information theory*, 60(5):2856–2867, 2014.

[MBK16] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, and Amin Karbasi. Fast constrained submodular maximization: Personalized data summarization. In *International Conference on Machine Learning*, pages 1358–1367. PMLR, 2016.

[Mil19] Tova Milo. Getting rid of data. *Journal of Data and Information Quality (JDIQ)*, 12(1):1–7, 2019.

[Mof19] Alistair Moffat. Huffman coding. *ACM Computing Surveys (CSUR)*, 52(4):1–35, 2019.

[MSWB94] Alistair Moffat, Neil Sharman, Ian H Witten, and Timothy C Bell. An empirical evaluation of coding methods for multi-symbol alphabets. *Information Processing & Management*, 30(6):791–804, 1994.

[PdMCD24] Laércio Pioli, Douglas DJ de Macedo, Daniel G Costa, and Mario AR Dantas. Intelligent edge-powered data reduction: A systematic literature review. *ACM Computing Surveys*, 56(9):1–39, 2024.

[PIT+18] Georgios S Paschos, George Iosifidis, Meixia Tao, Don Towsley, and Giuseppe Caire. The role of caching in future communication systems and networks. *IEEE Journal on Selected Areas in Communications*, 36(6):1111–1125, 2018.

[RDK03] Qun Ren, Margaret H Dunham, and Vijay Kumar. Semantic caching and query processing. *IEEE transactions on knowledge and data engineering*, 15(1):192–210, 2003.

[RHM02] Dragomir Radev, Eduard Hovy, and Kathleen McKeown. Introduction to the special issue on summarization. *Computational linguistics*, 28(4):399–408, 2002.

[Say17] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[Sed14] Philip Sedgwick. Cluster sampling. *Bmj*, 348, 2014.

[She18] Yi Shen. Data sustainability and reuse pathways of natural resources and environmental scientists. *New Review of Academic Librarianship*, 24(2):136–156, 2018.

[SMB16] Zubair Shah, Abdun Naser Mahmood, and Michael Barlow. Computing hierarchical summary of the data streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 168–179. Springer, 2016.

[SSS07] Ian Simon, Noah Snavely, and Steven M Seitz. Scene summarization for online image collections. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE, 2007.

[ST85] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[Svi04] Maxim Sviridenko. A note on maximizing a submodular set function subject to a knapsack constraint. *Operations Research Letters*, 32(1):41–43, 2004.

[Tah16] Hamed Taherdoost. Sampling methods in research methodology; how to choose a sampling technique for research. *How to choose a sampling technique for research (April 10, 2016)*, 2016.

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[Wal91] Gregory K Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[WKH+20] Martin J Willemink, Wojciech A Koszek, Cailin Hardell, Jie Wu, Dominik Fleischmann, Hugh Harvey, Les R Folio, Ronald M Summers, Daniel L Rubin, and Matthew P Lungren. Preparing medical imaging data for machine learning. *Radiology*, 295(1):4–15, 2020.

[WNC87] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[WS96] Richard Y Wang and Diane M Strong. Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, 12(4):5–33, 1996.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.