APPROXIMATE JOINS FOR RELATIONAL DATA

Η

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής

Εξεταστική Επιτροπή

από τον

Ιωάννη Κρομμύδα

ως μέρος των Υπ[redacted]σεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούνιος 2008

DEDICATION

This thesis is dedicated to my family for supporting me all the way since the beginning of my studies.

ACKNOWLEGMENTS

I am thankful to my supervisor Dr. Panos Vassiliadis for guiding, encouraging and motivating me throughout this research work. I would also like to express my gratitude to Dr. Evangelia Pitoura and Dr. Apostolos Zarras for their valuable remarks during the course of my research.

At the end of my thesis I would like to thank all those people who made this thesis possible and an enjoyable experience for me; especially my colleagues and friends, Eytychia, Fotini, Panos, Tasos, Tzeni and Hara for their help and encouragement throughout this work.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ΕΚΤΕΝΗΣ ΠΕΡΙΛΗΨΗ ΣΤΑ ΕΛΛΗΝΙΚΑ

Ιωάννης Κρομμύδας του Ευαγγέλου και της Γεωργίας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος, 2008. Approximate Joins for Relational Data. Επιβλέποντας: Παναγιώτης Βασιλειάδης.

Σε μία σχεσιακή βάση δεδομένων συχνά παρατηρείται η ύπαρξη μεγάλου πλήθους εγγραφών, οι οποίες αναφέρονται στην ίδια οντότητα, αλλά αναπαρίστανται με διαφορετικό τρόπο. Το συγκεκριμένο φαινόμενο, μπορεί να οφείλεται σε τυπογραφικά λάθη, είτε στη χρήση ποικίλων τρόπων για την καταχώρηση κάποιας αλφαριθμητικής τιμής. Συνεπώς, η εύρεση των όμοιων εγγραφών θεωρείται επιβεβλημένη, ιδιαίτερα όταν εφαρμόζεται σε μία βάση δεδομένων που διατηρεί μεγάλο όγκο δεδομένων.

Στη συγκεκριμένη εργασία παρουσιάζουμε μία διαδικασία, η οποία αποτελεί επέκταση μίας από τις κυρίαρχες τεχνικές προσεγγιστικής εύρεσης διπλότυπων εγγραφών. Δοθείσης μίας βάσης δεδομένων που αποτελείται από έγκυρα δεδομένα, με τη χρήση της συγκεκριμένης τεχνικής, κάθε εισερχόμενη εγγραφή είτε αντιστοιχίζεται σε κάποια υπάρχουσα έγκυρη εγγραφή, είτε τη χαρακτηρίζεται ως νέα εγγραφή. Η προτεινόμενη τεχνική χρησιμοποιεί έναν αποτελεσματικό αλγόριθμο εύρεσης υποψήφιων εγγραφών, για τις οποίες υπολογίζεται το ποσοστό ομοιότητας με την εισερχόμενη εγγραφή βάσει συγκεκριμένων συναρτήσεων ομοιότητας. Η συνολική διαδικασία επιταχύνεται με τη χρήση δομών δεδομένων, οι οποίες διατηρούνται στη μνήμη και περιέχουν τις εγγραφές που χαρακτηρίζονται συχνά ως υποψήφιες όμοιες εγγραφές. Τέλος, παρατίθενται πειραματικά αποτελέσματα από την εφαρμογή της προτεινόμενης τεχνικής μας και παρουσιάζεται μία συγκριτική μελέτη των αποτελεσμάτων με υπάρχουσες τεχνικές.

ABSTRACT

Krommydas, Ioannis, Evagelos, Georgia. MSc, Computer Science Department, University of Ioannina, Greece. June, 2008. Approximate Joins for Relational Data. Thesis Supervisor: Vassiliadis Panos.

Relational databases often contain duplicate data entries. This may occur due to a variety of reasons, such as typographical errors, multiple conventions for recording database fields or other noise sources. Duplicate detection is a crucial procedure, especially for large databases.

In this thesis, we present a method that extends the state-of-the-art method for duplicate detection. Given a database holding valid data information, we classify each input tuple as a new tuple, or as an existing tuple. The proposed method uses an effective algorithm for determining a set of candidate reference tuples. For each candidate reference tuple, we use appropriate similarity metrics in order to decide whether the input tuple matches a reference tuple. The whole procedure is accelerated via trie data structures for caching the frequent input tuples. Finally, we present a number of experiments evaluating the effectiveness of our method and state a comparative study with the state-of-the-art method.

CHAPTER 1. INTRODUCTION

The efficiency of every information processing infrastructure is greatly affected by the quality of the data residing in its databases. Poor data quality is the result of a variety of reasons, including data entry errors (e.g., typing mistakes), poor integrity constraints and multiple conventions for recording database fields (e.g., company names, addresses). As a result, data cleaning has been at the center of research interest in recent years [Kon04].

Data cleaning is critical for many industries over a wide variety of applications, including marketing communications, commercial householding, customer matching, merging information systems, medical records etc. It is often studied in association with data warehousing, data mining and database integration. Especially, data warehousing requires and provides extensive support for data cleaning.

One of the most important tasks in data cleaning is to de-duplicate records. Duplicate detection is the process of identifying different or multiple records that refer to one unique real-world entity or object. Given a dirty database, the standard method to detect exact duplicates is to sort the database and then check if the neighbouring records are identical. In order to detect inexact duplicates, the most reliable way is to compare every record with every other record, which takes $N(N-1)/2$ comparisons, where N is the number of records in the database. However, this is infeasible when N is large [Lee99].

To ensure high data quality, data warehouses must validate and clean incoming data records from external sources. All tables that are maintained within such data

CHAPTER 1. INTRODUCTION

The efficiency of every information processing infrastructure is greatly affected by the quality of the data residing in its databases. Poor data quality is the result of a variety of reasons, including data entry errors (e.g., typing mistakes), poor integrity constraints and multiple conventions for recording database fields (e.g., company names, addresses). As a result, data cleaning has been at the center of research interest in recent years [KoMS04].

Data cleaning is critical for many industries over a wide variety of applications, including marketing communications, commercial householding, customer matching, merging information systems, medical records etc. It is often studied in association with data warehousing, data mining and database integration. Especially, data warehousing requires and provides extensive support for data cleaning.

One of the most important tasks in data cleaning is to de-duplicate records. Duplicate detection is the process of identifying different or multiple records that refer to one unique real-world entity or object. Given a dirty database, the standard method to detect exact duplicates is to sort the database and then check if the neighboring records are identical. In order to detect inexact duplicates, the most reliable way is to compare every record with every other record, which takes $N(N-1)/2$ comparisons, where N is the number of records in the database. However, this is infeasible when N is large [SuLS02].

To ensure high data quality, data warehouses must validate and clean incoming data records from external sources. All tables that are maintained within such data

warehouses and which contain clean records are called *reference tables*. In many situations, clean records must match acceptable records in reference tables. For example, product name and description fields in a sales record from a distributor must match the pre-recorded name and description fields in a product reference relation.

A significant challenge in such a scenario is to implement an efficient and accurate fuzzy match operation that can effectively clean an incoming record if it fails to match exactly with any record in the reference relation [CGGM03]. More specifically, it is crucial to implement a data cleaning method based on similarities in order to identify similar reference records. The similarity between input and reference records can be evaluated using a variety of distance functions. As a result, it is critical to choose the distance function that best suits the domain and the application.

The problem is straightforward for numerical values, but still remains very hard for string values and combinations of them in an attribute, such as names (first-, middle-, last- name), addresses, etc. One of the most common sources of mismatches in database entries is the typographical variations of string data. For example, considering company names, it is common to see "Microsoft", "Micorsoft", "Microsoft Inc." and "Microsoft Corporation" being used in different records to represent the same entity.

Duplicate detection typically relies on string comparison techniques to deal with typographical variations. In such a scenario, a simple equality or even substring comparison, for example, on names or addresses will not properly identify them as being the same entity, leading to a variety of potential problems. Consequently, approximate matching for detecting inexact duplicates presents a challenge between accuracy, efficiency and storage overheads as well.

Multiple methods have been developed for this task and each method works well for particular types of errors. Those methods define a distance metric (edit distance, affine gap distance, qgram distance, jaro distance metric etc.) and an appropriate matching threshold in order to match similar records.

Therefore, the problem we focus on is to clean a stream of incoming records, before their insertion to a reference table. Our approach is associated with the implementation of an effective method for on-line detecting similarity between input and reference records. Specifically, we check each attribute of the input record separately, using appropriate structures for accomplishing effective cleaning.

We take advantage of a structure called *Word Index,* which is a table holding information about the attribute values stored in the reference table. This structure is used for the retrieval of reference tuples that probably match input tuples according to qgram similarity. In parallel, we maintain in main memory a trie structure called *Qgram Trie*, which caches the retrieved attribute values. More specifically, this trie holds all the candidate attribute values that are similar to the input value. According to a matching procedure, matching scores between the input tuple and reference tuples are stored in a score table. The set of reference tuples whose similarity with the input word is above a similarity threshold is returned.

Additionally, we apply the LRU algorithm as a replacement policy in case the size of trie exceeds a specific percentage of main memory. More particularly, updating trie by inserting new attribute values to it, leads to the pruning of attribute values that were not recently accessed during the matching procedure. Using this replacement policy we assure that the size of trie is kept fixed and contains all the recent accessed attribute values.

The main contributions of this thesis could be summarized as follows:

- Introduction of an effective approximate matching method
- Development of algorithms using appropriate structures for handling streams of incoming records
- Implementation of experiments using variant parameters of the datasets

The remaining part of this thesis is organized in five chapters. The second chapter contains the related work that is associated with the problem we deal. In the third chapter, firstly, we describe in detail the duplicate detection problem. Then, we

represent the state-of-the-art method that is employed to the specific problem. In the third chapter, we state our approach including a detailed description of the used structures, the matching algorithm and replacement policy we adopt when the used structures need to be updated. In the fourth chapter, we present a number of experiments in order to evaluate the efficiency of our approach and compare it with the state-of-the-art method. Finally, we conclude our results and present topics for future research in fifth chapter.

CHAPTER 2. RELATED WORK

2.1. Frequent Itemsets

Frequent itemsets play an essential role in many data mining tasks that involve techniques associated with the finding of interesting patterns from databases.

The set of that kind of patterns includes association rules, correlations, sequences, episodes, classifiers and many others. The problems of (a) mining frequent itemsets or (b) association rules are considered as some of the most popular and challenging tasks. A great deal of attention is given to both of those problems due to the fact that they are encountered in real world problems such as market analysis.

Many algorithms based on different techniques are proposed for the solution of both problems. Those algorithms are evaluated according to their performance. In the following, we will employ the definitions of frequent itemsets, the notion of closed

and maximal frequent itemsets and the definition of frequent pattern trees (FP-trees). Finally, we refer to some of the frequent itemset mining methods.

2.1.1. Frequent Itemsets Definition

According to [Goet03], given a set of *items* I, every subset X of I is called an *itemset* or a *k-itemset* if it contains k items. A *transaction* T over I contains a *transaction identifier* tid and an itemset I and is said to *support* an itemset $X \subseteq I$, if $X \subseteq I$. A *transaction database* $\mathcal{D}$ over I is a set of transactions over I.

The itemsets can be described by measures such as their *cover*, *support* or *frequency*. The *cover* of an itemset X is a set that includes all the identifiers of transactions in $\mathcal{D}$ that support X and the measure of *support* is used for the counting of the transactions that belong in the cover of the itemset. Finally, the *frequency* of an itemset represents its probability of occurrence in a transaction existing in $\mathcal{D}$. Given the measure of frequency, one itemset is called *frequent* if its support is greater than a given absolute *minimal support threshold* σ_{abs}.

Taking into account the definitions described above, the *Itemset Mining problem* can be clearly defined as follows:

"Given a set of items I, a transaction database $\mathcal{D}$ over I and a minimal support threshold σ, find the collection of frequent itemsets."

Considering the following transaction database, which is shown in Fig. 2.1, the total number of frequent itemsets that can extracted is depicted in Fig 2.2.

$\mathcal{I}$ = {beer, chips, pizza, wine}

tid	*X*
100	{beer, chips, wine}
200	{beer, chips}
300	{pizza, wine}
400	{chips, pizza}

Fig. 2.1. An example transaction database *D* [Goet03]

Itemset	Cover	Support	Frequency
{}	{100, 200, 300, 400}	4	100%
{beer}	{100,200}	2	50%
{chips}	{100,200,400}	3	75%
{pizza}	{300,400}	2	50%
{wine}	{100,300}	2	50%
{beer, chips}	{100,200}	2	50%
{beer, wine}	{100}	1	25%
{chips, pizza}	{400}	1	25%
{chips, wine}	{100}	1	25%
{pizza, wine}	{300}	1	25%
{beer, chips, wine}	{100}	1	25%

Fig. 2.2. Frequent Itemsets and their support in *D* (σ_{abs} = 1) [Goet03]

2.1.2. Closed and Maximal Frequent Itemsets

In practice, the set of frequent itemsets produced from a transaction database can be very large. Consequently, it is necessary to find a way to replace the full set of all frequent itemsets with a small representative subset of itemsets from which all other frequent itemsets can be produced. Maximal and Closed frequent itemsets are used for such a representation.

A *maximal frequent itemset* is defined as a frequent itemset for which none of its immediate supersets is frequent, whereas an itemset *X* is called *closed frequent itemset* if none of its immediate supersets has the same support count as *X* and its support is greater or equal to the minimal support threshold.

According to the definitions stated above some of the frequent itemsets in Fig. 2.2 can be characterised either as maximal or closed. More specifically, the *set of maximal frequent itemsets* is {{chips, pizza}, {pizza, wine}, {beer, chips, wine}} and the *set of*

closed frequent itemsets is {{chips}, {pizza}, {wine}, {beer, chips}, {chips, pizza}, {pizza, wine}, {beer, chips, wine}}. It is obvious that all maximal frequent itemsets are closed as well.

2.1.3. Frequent Pattern Tree (FP-tree)

The notion of frequent-pattern trees (FP-trees) is associated with the construction of a compact data structure, which is in fact an extended prefix-tree structure used for storing compressed, crucial information about frequent patterns. Many frequent pattern mining methods are based upon such structures, implementing efficient frequent pattern mining techniques.

Observe the transaction database of Fig. 2.3, assuming that the minimal support threshold is set to be 3. The following observations can be made:

TID	Items bought	(Ordered) frequent items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

Fig. 2.3. Transaction Database [HPYM01]

1. Since only the frequent items play a role in the frequent-pattern mining procedure, it is necessary to perform one scan of the transaction database in order to identify the set of frequent items, in terms of the obtained frequency count.
2. If the set of frequent items of each transaction can be stored in some compact structure, it may be possible to avoid repeatedly scanning the original transaction database.

3. If multiple transactions share a set of frequent items, it may be possible to merge the shared sets with the number of occurrences registered as count. It is easy to check whether two sets are identical if the frequent items in all of the transactions are listed according to a fixed order.
4. If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the count is registered properly. If the frequent items are sorted in their frequency descending order, there are better chances that more prefix strings can be shared.

An FP-tree can be defined as follows [HPYM01]:

1. An FP-tree consists of one root labeled as "null", a set of item-prefix subtrees as the children of the root, and a frequent-item-header table.
2. Each node in the item-prefix subtree consists of three fields: (a) item-name, (b) count, and (c) node-link, where item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path reaching this node, and node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none.

Each entry in the frequent-item-header table consists of two fields: (a) the item-name and (b) the head of node-link, which is a pointer pointing to the first node in the FP-tree carrying the item-name.

Based on the observations listed previously, the construction of the FP-tree that corresponds with the example transaction database (Fig. 2.3) can be implemented as follows:

First, a scan of the database derives a list of frequent items, < *(f:4), (c:4), (a:3), (b:3), (m:3), (p:3)* >, where the number after ":" indicates the item support, in which items are ordered in frequency descending order. This ordering is important since each path of a tree will follow this order.

Second, the root of a tree is created and labeled with "null". The FP-tree is constructed as follows by scanning one more time the transaction database:

1. The scan of the first transaction leads to the construction of the first branch of the tree: < *(f :1), (c:1), (a:1), (m:1), (p:1)* >, in which the frequent items are listed according to the their order in the list of frequent items.
2. For the second transaction, since its ordered frequent item list < *f, c, a, b, m* > shares a common prefix < *f, c, a* > with the existing path < *f, c, a, m, p* >, the count of each node along the prefix is incremented by 1, and one new node *(b:1)* is created and linked as a child of *(a:2)* and another new node *(m:1)* is created and linked as the child of *(b:1)*.
3. For the third transaction, since its frequent item list < *f, b* > shares only the node < *f* > with the f -prefix subtree, f 's count is incremented by 1, and a new node *(b:1)* is created and linked as a child of *(f :3)*.
4. The scan of the fourth transaction leads to the construction of the second branch of the tree, < *(c:1), (b:1), (p:1)* >.
5. For the last transaction, since its frequent item list < *f, c, a, m, p* > is identical to the first one, the path is shared with the count of each node along the path incremented by 1.

To facilitate tree traversal, an item header table is built in which each item points to its first occurrence in the tree via a node-link. Nodes with the same item-name are linked in sequence via such node-links. After scanning all the transactions, the tree, together with the associated node-links, is depicted in Fig. 2.4.

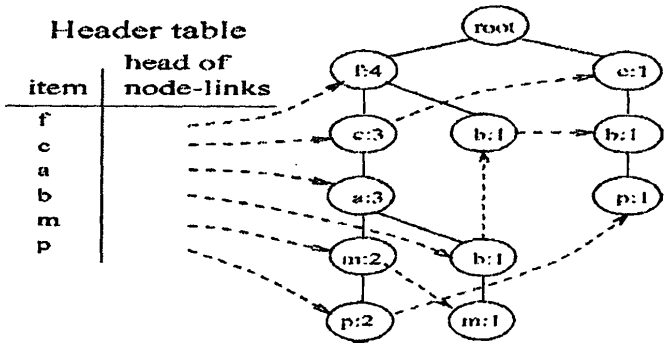


Fig. 2.4. FP-tree Structure [HPYM01]

2.1.4. Frequent Itemset Mining Methods

Frequent itemset mining methods can be categorized in two individual categories, Apriori-based methods and Frequent-pattern tree based methods. The methods that have been developed in both categories are listed in the following sections.

2.1.4.1. Apriori-based methods

Apriori-based methods take advantage of the anti-monotone Apriori principle which can be expressed as follows:

"if any pattern of length k is not frequent in the database, its super-pattern of length (k+1) can never be frequent". [HPYM04]

The essential idea is to iteratively generate the set of candidate patterns of length (k+1) from the set of frequent-patterns of length k (for $k \geq 1$) and check their corresponding occurrence frequencies in the database.

Agrawal et al. in [AgSr94] proposed the Apriori algorithm, which exploits the monotonicity property of the support of itemsets. Together with the proposal of the Apriori algorithm, Agrawal et al. in [AgSr94] proposed two other algorithms, AprioriTid and AprioriHybrid. The AprioriTid algorithm reduces the time needed for the support counting procedure by iteratively replacing every transaction in the database by the set of candidate itemsets that occur in that transaction.

Shortly after the proposal of the Apriori algorithms described before, Park et al. proposed in [PaCY95] another optimization, called DHP (Direct Hashing and Pruning) to reduce the number of candidate itemsets. During the kth iteration, when the supports of all candidate k-itemsets are counted by scanning the database, DHP already gathers information about candidate itemsets of size k + 1 in such a way that all (k + 1)-subsets of each transaction after some pruning are hashed to a hash table.

The DIC algorithm, proposed by Brin et al. in [BMUT97], tries to reduce the number of passes over the database by dividing the database into intervals of a specific size. First, all candidate patterns of size 1 are generated. The supports of the candidate sets are then counted over the first interval of the database. Based on these supports, a new candidate pattern of size 2 is already generated if all of its subsets are already known to be frequent, and its support is counted over the database together with the patterns of size 1. In general, after every interval, candidate patterns are generated and counted.

2.1.4.2. Frequent-pattern tree based methods

Frequent-pattern tree based methods use the compact data structure of frequent-pattern trees (FP-tree), which was described previously. The FP-growth algorithm proposed by Han et al. in [HPYM04] is the most well known FP-tree based algorithm that faces the frequent itemset mining problem. The FP-growth algorithm stores the actual transactions from the database within an FP-tree, facilitating the finding procedure of all frequent items' support.

2.2. Fault-tolerant Frequent Itemsets

Real-world data tend to be dirty. As a result, the discovery of knowledge over large real-world data requires the development of fault-tolerant data mining methods. The goal of those methods is the extraction of approximate and more general fault-tolerant patterns from database, instead of finding exact patterns.

On the other hand, frequent pattern mining often generates a large number of frequent itemsets, which reduces not only the efficiency, but also the effectiveness of mining. This happens due to the fact that users have to sift through a large number of mined results to find the useful ones. Therefore, the effectiveness of frequent pattern mining is improved by fault-tolerant frequent pattern mining.

An itemset can be characterized as an *approximate frequent itemset* if a percentage of its items is frequent in the transaction database. As a result, all fault-tolerant frequent itemsets can be produced by using this slight relaxation of the frequent itemsets' notion.

Consider the transaction database shown in Fig 2.5, if the minimal support threshold is set to 3, there exists no pattern with more than two items, as there are many short patterns, with low support counts. However, longer approximate frequent patterns with support count equal to 3 or more can be extracted from such a database. For example, transactions 10, 30 and 50 contain four out of five items: a, b, c, e and f.

Transaction ID	Items
10	b, c, e, f
20	d, e, g
30	a, b, c, e
40	a, d, f
50	a, b, e, f

Fig. 2.5. Transaction Database TDB [Peth01]

2.2.1. Fault-tolerant Frequent Itemsets Definition

Given a fault tolerance $\delta(\delta > 0)$ and an itemset P such that $|P| > \delta$, a transaction $T =$ *(tid, X)* is said to *FT-contain* itemset P if and only if there exists $P' \subseteq P$ such that $P' \subseteq X$ and $|P'| \geq (|P| - \delta)$, which is equivalent to $|P \cap X| \geq (|P| - \delta)$. The number of transactions in a database FT-containing itemset P is called the *FT-support* of P, denoted as $\overline{\sup}(X)$.

The set of transactions FT-containing itemset X is called the *FT-body* and is denoted as $\overline{B}(X)$. Given a frequent-item support threshold min_sup^{item} and an FT-support threshold min_sup^{FT}, an itemset X is called a *fault-tolerant frequent pattern*, or an *FT-pattern*, if and only if:

1. $\overline{\sup}(X) \geq \min\_sup^{FT}$ and
2. for each item $x \in X$, $\sup_{\overline{B}(X)}(x) \geq \min\_sup^{item}$, where $\sup_{\overline{B}(X)}(x)$ is the number of transactions in $\overline{B}(X)$ containing item x.

The frequent-item support threshold is used to filter out infrequent items, whereas FT-support threshold is used to capture frequent patterns in the sense of allowing at most δ mismatches.

Apart from the two thresholds mentioned above, there also exists the *length threshold* denoted as *min_l (min_l* > δ*)*, which is applied for having as an output only FT-patterns consisting of at least *min_l* items.

An item x is called a *global frequent item* if and only if $\overline{\sup}(X) \geq \min\_sup^{item}$, which means that it appears in more than $\min\_sup^{item}$ transactions. It holds that FT-patterns contain only global frequent items and $\overline{\sup}(X) \geq \sup(X)$, for any itemset X.

Considering the definitions listed above, the *Fault-Tolerant Itemset Mining Problem* can be defined as follows:
"Given a transaction database, a fault tolerance, a frequent-item support threshold, an FT-support threshold and a length threshold, the problem of fault-tolerant frequent pattern mining is to find the complete set of FT-patterns passing the length threshold." [Peth01]

Returning to the transaction database TDB shown in Fig. 2.5 and setting the frequent-item support threshold $min\_sup^{item} = 2$, the FT-support threshold $min\_sup^{FT} = 3$ and the fault-tolerance $\delta = 1$, which means that only one mismatch is allowed, it holds that for itemset X = *abcef*, $\widetilde{B}(X)$ includes transactions 10, 30 and 50, each of them FT-contains X. Also, each item in X appears in at least two transactions in $\widetilde{B}(X)$. As a result, itemset *abcef* can be considered as an FT-frequent pattern.

A variant of the problem described above is the *top-K Fault-Tolerant Itemset Mining problem*, which requires to find only the top-K FT-frequent itemsets according to their fault-tolerant frequency.

2.2.2. Fault-tolerant Frequent Itemset Mining Methods

2.2.2.1. Apriori-based methods

Apriori-based fault-tolerant frequent itemset mining methods extend the Apriori heuristic in order to face the fault-tolerant frequent itemset mining problem and are based in the heuristic that follows up:

"if X ($|X|>\delta$) is not an FT-pattern, then none of its supersets is an FT-pattern, where δ is the fault tolerance"

Pei et al. based on this extended heuristic, implemented in [PeTH01] the FT-Apriori algorithm (Fault-Tolerant Apriori algorithm), which tackles efficiently the problem mentioned before.

2.2.2.2. Binary Vector-based methods

Those methods are based on design of binary vectors, called *Appearing Vectors* that are used for indicating the distribution of candidate fault-tolerant frequent itemsets in the transaction database.

Koh et al. in [KoYo05] proposed a vector-based algorithm, called VB-FT-Mine (Vector-Based Fault–Tolerant frequent pattern Mining), used for speeding up the process of mining fault-tolerant frequent patterns.

Yang et al., proposed in [YaFB01] the GGA algorithm (Greedy Growing Algorithm), which exploits the sparseness of the underlying data to find large itemsets that are correlated over database records. They took advantage of the transaction coverage notion, which allowed them to extend the algorithm and view it as a fast clustering algorithm for discovering segments of similar transactions in binary sparse data.

2.3. Dense Frequent Itemsets

If an itemset is found to be frequent, all of its items must co-occur sufficiently often, which is rare in real-world data. A generalization of frequent itemsets is given by replacing the requirement of perfect co-occurrence by partial co-occurrence, requiring that an itemset has at least a proportion $1-\varepsilon$ of items present in at least a proportion f of database rows, where f is the *ε-approximate frequency* and ε represents the *percentage of fault tolerance.*

This generalization, which was described in Section 2.2 leads to two problems. The first one has to do with the generation of many approximately frequent itemsets without meaningful information, whereas the second one is associated with the fact that the usual kind of itemset mining algorithms, like Apriori, are not easily generalized to the new task [SeMa04].

Those problems can be illustrated taking into consideration the two example databases (a) and (b) of Fig. 2.6.

A	B	C	D	E	F	G	H
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0

(a)

A	B	C	D
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1

(b)

Fig. 2.6. Two Example Databases [SeMa04]

Fig. 2.6 (a) can be used for the description of the first problem. It is obvious that itemset *ABCDE* is frequent. However, a multitude of approximately frequent sets exist with $\varepsilon = 0.5$, such as *ABCFGH*, *ABCDFGH*, *ABCDEFGH* etc. and beyond the fact that *ABCDE* is frequent, those sets give us no new information.

Fig. 2.6 (b) is used for the illustration of the second problem. Itemset *ABCD* has 0.5-approximate frequency 100%, but the approximate frequencies of its subsets are lower. For example, the approximate frequency for *A* is 50%, for *AB* is 83% and for *ABC* is 67%.

Thus a set can be approximately frequent having none of its nontrivial subsets frequent. This precludes pruning the candidate itemsets in the way that Apriori and other algorithms do.

The definition of dense itemsets aids to the avoidance of both problems.

2.3.1. Dense Itemsets Definition

An itemset X is *(σ, δ)-dense*, given two parameters σ and δ, if for any subset $Y \subseteq X$, there is a set r_Y of σ database rows such that in the subdatabase defined by Y and r_Y at least a fraction δ of items are present.

A *binary database* $DB = \langle R, r \rangle$ consists of a finite set R of *attributes*, also known as *items*, and a finite multiset $r = \{t_1, t_2, \ldots, t_n\}$ of *transactions*, which are subsets of R. The *frequency* of an itemset $X \subseteq R$ in a database $DB = \langle R, r \rangle$ is the number of transactions that include all the attributes of X, which can be typically defined as $freq(X) = |\{ t \in r \mid t \supseteq X \}|$.

The *weak density* of an itemset $X \subseteq R$, which can noted as *wdens(X, r)*, equals to:

$$wdens(X, r) = \frac{\sum_{t \in r} |X \cap t|}{|X| \cdot |r|}$$

and represents the average fraction of items that are present in a set of transactions. Given a number σ between zero and the size of the relation, the *weak density at support* σ of X can be defined as well and is equal to:

$$wdens(\sigma, X, r) = \max_{r'} wdens(X, r'),$$

where the maximum is taken over all σ-element submultisets r' of r.

Taking into account the definitions given above, an itemset can be characterized as *weakly* (σ, δ)*-dense*, if its weak density at support σ exceeds δ, where σ and δ are predefined parameters.

The *density* $dens(\sigma,X)$ of an itemset X at support level σ is the minimum of the weak densities of all non-empty subsets of X, which can be formally described as follows:

$$dens(\sigma, X) = \min_{\emptyset \neq Y \subset X} wdens(\sigma, Y)$$

Taking advantage of the definition stated above, an itemset X can be characterized as *(strongly)* (σ, δ)*-dense*, if it holds that $dens(\sigma,X) \geq \delta$.

Consider the example database of Fig. 2.7. The supports σ at which the listed sets can be characterized as *weakly* $(\sigma, 0.5)$*-dense* are illustrated in Fig. 2.8.

A	*B*	*C*	*D*	*E*	*F*
1	1	0	0	0	0
1	0	1	0	0	0
1	0	0	1	0	0
1	0	0	0	1	0
1	0	0	0	0	1
0	1	1	1	0	0
0	0	1	1	1	0
0	0	0	1	1	1
0	1	0	0	1	1
0	1	1	0	0	1

Fig. 2.7. Example Database [SeMa04]

set	support
A	10
B	8
AB	9
BC	8
ABC	8
BCD	8
$ABCD$	7
$ABCDE$	4
$BCDEF$	6
$ABCDEF$	5

Fig. 2.8. Supports σ (weakly (σ, 0.5)-dense listed sets) [SeMa04]

2.3.2. Dense Itemsets Mining Methods

Existing algorithms find all dense itemsets from large collections of binary data and are based on the familiar A-priori idea:

*"for each $h \geq 1$, given dense sets of size **h**, form candidate sets of size **h+1**, and then do a database pass to verify which candidates indeed satisfy the density condition".*

Seppänen et al. in [SeMa04] proposed the Dense-Sets algorithm, which performs a levelwise search to find all dense itemsets and can be extended into the variant problem of finding the *k densest sets*, with a given support, or the *k best supported sets* with a given density.

2.4. Association Rules

Mining of *association rules* is an important data mining problem. Mining association rules from a transaction database involves the finding of rules such as: *"A customer who buys item X and item Y is also likely to buy item Z in the same transaction"*, where *X*, *Y* and *Z* are initially unknown.

2.4.1. Association Rule Mining Problem

The association rule mining problem can be decomposed into two subproblems:

1. *Find out all frequent itemsets, which are the sets of items that are contained in a sufficiently number of transactions, with respect to a minimum support threshold*
2. *From the set of frequent itemsets found, find out all the association rules that have a confidence value exceeding a minimum confidence threshold*

From the two problems mentioned above, the second one is straightforward, whereas the first one has been a subject of many major research efforts [ChLK97].

Let $I = \{i_1, i_2, ..., i_m\}$ be the set of items and D the transaction database. For each transaction T of the transaction database it holds that $T \subseteq I$.

An *association rule* can be characterized as an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ *and* $X \cap Y = \emptyset$.

An association rule $X \Rightarrow Y$ holds in the database D with *confidence c%*, if no less than *c%* of the transactions in D that contain X, also contain Y. An association rule $X \Rightarrow Y$ has *support s%* in D, if $\sigma_{X \cup Y} = |D| \times s\%$, where $\sigma_{X \cup Y}$ is the support count of the itemset $X \cup Y$.

If *s%* is the given support threshold, the *association rule mining problem* is reduced to the problem of finding the set $L = \{X \mid X \subseteq I \wedge \sigma_X \geq |D| \times s\%\}$ or the set L_k, where symbol L_k denotes the set of all frequent *k-itemsets* in L, where each k-itemset contains exactly k items.

The corresponding set of association rules that are extracted from the example transaction database $\mathcal{D}$ in Section 2.1.1 (Fig. 2.1), according to solution of the problem described above, is shown in Fig. 2.9.

Rule	Support	Frequency	Confidence
{beer} ⇒ {chips}	2	50%	100%
{beer} ⇒ {wine}	1	25%	50%
{chips} ⇒ {beer}	2	50%	66%
{pizza} ⇒ {chips}	1	25%	50%
{pizza} ⇒ {wine}	1	25%	50%
{wine} ⇒ {beer}	1	25%	50%
{wine} ⇒ {chips}	1	25%	50%
{wine} ⇒ {pizza}	1	25%	50%
{beer, chips} ⇒ {wine}	1	25%	50%
{beer, wine} ⇒ {chips}	1	25%	100%
{chips, wine} ⇒ {beer}	1	25%	100%
{beer} ⇒ {chips, wine}	1	25%	50%
{wine} ⇒ {beer, chips}	1	25%	50%

Fig. 2.9. Association Rules and their Support and Confidence in *D* [Goet03]

2.5. Maintenance of Association Rules

Transaction databases are not static databases, because several updates are constantly being applied to them. More specifically, new records (transactions) are added to record purchase activities. Older records in the database are deleted from the database and existing records may be edited or changed, due to corrections of manual operational errors or other reasons.

Consequently, new association rules may appear in the database and at the same time, some existing association rules would become invalid. The problem that arises involves the *maintenance of discovered association rules*, according to the *insertions*, *deletions* or *modifications* of the transactions in the transaction database.

2.5.1. Update Problem of Association Rules

The *update activities* take place in a transaction database *D* include *insertions* and *deletions*. Also, *modification activities* can be treated as deletions followed by insertions. Δ^- denotes the *set of deleted transactions*, while Δ^+ denotes the set of *newly added transactions*. The *updated database*, which is denoted as *D'* equals to

$D' = (D - \Delta^-) \cup \Delta^+$. D^- denotes the set of *unchanged transactions* and it is equal to $D^- = D - \Delta^- = D' - \Delta^+$.

The definitions of all the symbols described above are include in Table 2.5.1 that follows up.

Table 2.1. Definitions of Several Symbols

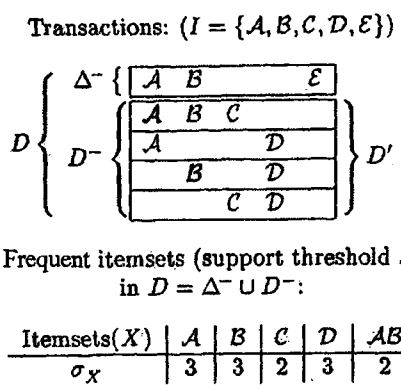
database	Support count of itemset v	Frequent k-itemsets
Δ^+	δ_X^+	-
D^-	-	-
Δ^-	δ_X^-	-
$D = \Delta^- \cup D^-$	σ_X	L_k
$D' = D^- \cup \Delta^+$	σ'_X	L'k

The new support count of an itemset X in the updated database D' is defined as σ'_X. The set of frequent itemsets in D' is denoted as L', whereas L'_k denotes the set of frequent k-itemsets in L'. The support count of an itemset X in the database Δ^+ is denoted as δ_X^+ and δ_X^- is the corresponding support count in Δ^-.

As a result of the previous mining procedure on the old database D, L and $\sigma_X \forall X \in L$ are known. Consequently, the *update problem* can be defined as follows:

"Find L' and $\sigma'_X \forall X \in L'$ efficiently, given the knowledge of D, D', Δ^-, D^-, Δ^+, L and $\sigma_X \forall X \in L$".

Fig. 2.10 illustrates the deletion of a transaction belonging to the depicted transaction database.



Frequent itemsets (support threshold $s = 25\%$) in $D = \Delta^- \cup D^-$:

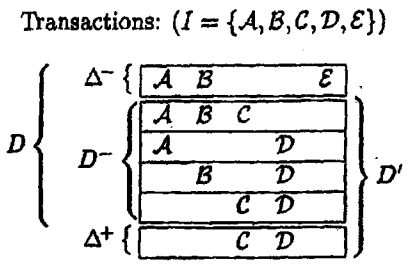
Itemsets(X)	A	B	C	D	AB
σ_X	3	3	2	3	2

Fig. 2.10. $\Delta+ = \varnothing$ [ChLK97]

By observing this transaction database, it is found out that the original database D, before the deletion of any transaction, contains 5 transactions. In this state of the transaction database, if the support threshold is set to 25% the frequent itemsets that are extracted to L are those with support count no less than $5 \times 25\% = 1.25$.

The next state of transaction database D results from the deletion of transaction (1, {A, B, E}), which belongs to Δ^-. As a result from this deletion, database D consists of 4 transactions and the frequent itemsets belonging in L' are those in D' with support count no less than $4 \times 25\% = 1$.

Fig. 2.11 illustrates the insertion of a transaction into the transaction database.



Frequent itemsets (support threshold $s = 25\%$) in $D' = D^- \cup \Delta^+$:

Itemsets(X)	A	B	C	D	CD
σ'_X	2	2	3	4	2

Fig. 2.11. $|\Delta+| > 0$ [ChLK97]

The difference from the previous example is that apart from the deletion of transaction (1, {A, B, E}) from database *D*, an insertion of transaction (6, {C, D}) takes also place into database *D*.

The original transaction database consists of 5 transactions. After inserting transaction (6, {C, D}), which belongs to database Δ^+ and deleting transaction (1, {A, B, E}), which belongs to Δ^-, the resulting updated transaction database consists of 5 transactions. Consequently, frequent itemsets in *L'* are those itemsets in *D'* with support count no less than $5 \times 25\% = 1.25$.

2.5.2. Methods for Maintaining Discovered Association Rules

2.5.2.1. Apriori-based methods

Apriori-based methods use the anti-monotone Apriori principle in order to generate smaller number of candidates, meanwhile taking advantage of knowledge acquired from a previous mining procedure. Cheung et al. in [CHNW96] proposed the FUP algorithm, which handles only the case of transaction insertions in the database. Cheung et al. in [ChLK97] proposed the FUP_2, which handles insertions, as well as deletions and modifications.

2.5.2.2. Frequent-pattern tree based methods

Frequent-pattern tree based methods are based on the structure of frequent-pattern trees. Koh et al. in [KoSh04] proposed the AFPIM algorithm, which handles insertions, deletions and modifications and adjusts or reconstructs the structure of the FP-tree according to the changes that take place in the transaction database.

2.6. Field Matching Techniques

Database entries are usually mismatched due to typographical variations or errors within the string data. Multiple techniques have been developed in order to extract the similarity of strings, taking into consideration potential typographical variations.

Some of the field matching techniques, which are used for data de-duplication purposes, will be presented in the following paragraphs.

2.6.1. Character-based similarity metrics

Character-based similarity metrics are designed to deal with typographical errors. The main character-based similarity metrics are: (i) *Edit distance*, (ii) *Affine gap distance*, (iii) *Smith-Waterman distance*, (iv) *Jaro distance metric* and (v) *Q-gram distance.*

The *edit distance* between two strings σ_1 and σ_2 is the minimum number of edit operations of single characters needed to transform the string σ_1 into σ_2. There are three types of edit operations: (i) *insert* a character into the string, (ii) *delete* a character from the string, and (iii) *replace* one character with a different character. In its simplest form, the cost for each edit transformation is equal to 1. This distance is also referred to as *Levenshtein distance.* Needleman and Wunsch [NeWu70] modified the original edit distance model, and allowed for variable costs for different edit distance operations.

A main string variation includes the truncation or the shortening of a specific string. For example, the entity *"John A. Smith"* could be written as *"Jonathan Abraham Smith"*. The *affine gap distance* can handle with this problem introducing two extra operations: (i) *open gap* and (ii) *extend gap.*

Smith-Waterman distance is an extension of the edit and affine gap distances. This metric considers that mismatches at the beginning and the end of strings have lower

costs than mismatches in the middle, allowing better substring matching. Therefore, the strings *"Prof. John A. Smith, University of Illinois"* and *"John A. Smith, Prof."* can match within short distance using the Smith-Waterman distance, since the prefixes and suffixes are ignored.

Jaro distance metric was mainly used for comparison of last and first names. The Jaro metric for strings σ_1 and σ_2 is computed following the next steps:

1) Compute the sting lengths $|\sigma_1|$ *and* $|\sigma_2|$
2) Find the *"common characters"* c in the two strings; common are all the characters $\sigma_1[j]$ and $\sigma_2[j]$ for which $\sigma_1[j] = \sigma_2[j]$ and for which $\sigma_1[j] = \sigma_2[j]$ and $|i-j| \le \frac{1}{2}\min\{|\sigma_1|,|\sigma_2|\}$
3) Find the number of transpositions t; the number of transpositions is computed as follows: compare the i_{th} common character in σ_1 with the i_{th} common character in σ_2. Each non-matching character is a transposition.

The Jaro comparison value is:

$$Jaro(\sigma_1,\sigma_2) = \frac{1}{3}\left(\frac{c}{|\sigma_1|} + \frac{c}{|\sigma_2|} + \frac{c-t/2}{c}\right).$$

The *q-gram distance* is computed using the *q-grams*. A q-gram is a short character substring of length q of the database strings. The intuition behind the use of q-grams as a foundation for approximate string matching is that two strings σ_1 and σ_2 are similar if they share a large number of q-grams in common. Given a string σ, its q-grams are obtained by *"sliding"* a window of length q over the characters of σ.

Letter q-grams, including trigrams, bigrams, and/or unigrams, have been used in a variety of ways in text recognition and spelling correction. One natural extension of q-grams are the positional q-grams, which also record the position of the q-gram in the string. Gravano et al. [Grav+01] showed how to use positional q-grams to locate efficiently similar strings within a relational database.

2.6.2. Token-based similarity metrics

Character-based similarity metrics work well for typographical errors. However, it is often the case that typographical conventions lead to rearrangement of words (e.g., *"John Smith"* vs. *"Smith, John"*). In such cases, character-level metrics fail to capture the similarity of the entities. Token-based metrics try to compensate for this problem.

Monge and Elkan [MoEl96] proposed a basic algorithm for matching text fields based on atomic strings. An atomic string is a sequence of alphanumeric characters delimited by punctuation characters. Two atomic strings match if they are equal, or if one is the prefix of the other. Based on this algorithm, the similarity of two fields is the number of their matching atomic strings divided by their average number of atomic strings.

Cohen [Cohe98] described a system named *WHIRL* that adopts from the information retrieval the cosine similarity combined with the *tf.idf* weighting scheme to compute the similarity of two fields. Cohen separates each string σ into words and each word w is assigned a weight

$$u_{\sigma}(w) = \log(tf_w + 1) \cdot \log(idf_w),$$

where tf_w is the number of times that w appears in the field and idf_w is $\frac{|D|}{n_w}$, where n_w is the number of records in the database D that contain w. The *tf.idf* weight for a word w in a field is high if w appears a large number of times in the field (large tf_w) and w is a sufficiently *"rare"* term in the database (large idf_w). For example, given a set of company names, infrequent terms such as *"IBM"* or *"Sun"* will have higher idf_w values than frequent terms such as *"Corp"*. The *cosine similarity* of σ_1 and σ_2 is defined as

$$sim(\sigma_1, \sigma_2) = \frac{\sum_{j=1}^{|D|} u_{\sigma_1}(j)\, u_{\sigma 2}(j)}{\|u_{\sigma_1}\|_2 \cdot \|u_{\sigma_2}\|_2}$$

The cosine similarity metric works well for a large variety of entries, and is insensitive to the location of words, allowing natural word moves and swaps. For example, the cosine similarity metric regards *"John, Smith"* as equivalent to *"Smith, John"*. Also, introduction of frequent words affects only minimally the similarity of the two strings due to the low *idf* weight of the frequent words. For example, *"John Smith"* and *"Mr. John Smith"* would have similarity close to one.

Unfortunately, this similarity metric does not capture word spelling errors, especially if they are pervasive and affect many of the words in the strings. For example, the strings *"Compter Science Department"* and *"Deprtment of Computer Scence"* will have zero similarity under this metric. Bilenko et al. [Bile03] suggest the *SoftTF-IDF* metric to solve this problem. In the *SoftTF-IDF* metric, pairs of tokens that are *"similar"* and not necessarily identical are also considered in the computation of the cosine similarity. However, the product of the weights for non-identical token pairs is multiplied by the similarity of the token pair, which is less than one.

Gravano et al. [GIKS03] extended the *WHIRL* system to handle spelling errors by using q-grams, instead of words, as tokens. In this setting, a spelling error minimally affects the set of common q-grams of two strings, so the two strings *"Gteway Communications"* and *"Comunications Gateway"* have high similarity under this metric, despite the block move and the spelling errors in both words. This metric handles the insertion and deletion of words nicely. The string *"Gateway Communications"* matches with high similarity the string *"Communications Gateway International"* since the q-grams of the word *"International"* appear often in the relation and have low weight.

2.6.3. Phonetic similarity metrics

Character-level and token-based similarity metrics focus on the string-based representation of the database records. However, strings may be phonetically similar even if they are not similar in a character or token level. For example, the word *"Kageonne"* is phonetically similar to *"Cajun"* despite the fact that the string representations are very different. The phonetic similarity metrics are trying to address such issues and match such strings.

Russell invented *Soundex*, which is the most common phonetic coding scheme. Soundex is based on the assignment of identical code digits to phonetically similar groups of consonants and is used mainly to match surnames.

The *New York State Identification and Intelligence System (NYSIIS)* was proposed by Taft [Taft70]. The NYSIIS system differs from Soundex in that it retains information about the position of vowels in the encoded word by converting most vowels to the letter A. Furthermore, NYSIIS does not use numbers to replace letters; instead it replaces consonants with other, phonetically similar letters, thus returning a purely alpha code.

Philips suggested the *Metaphone* algorithm as a better alternative to Soundex. Philips suggested using 16 consonant sounds that can describe a large number of sounds used in many English and non-English words. Double Metaphone is a better version of Metaphone, improving some encoding choices made in the initial Metaphone and allowing multiple encodings for names that have various possible pronunciations. For such cases, all possible encodings are tested when trying to retrieve similar names. The introduction of multiple phonetic encodings greatly enhances the matching performance, with rather small overhead.

2.6.4. Numeric similarity metrics

While multiple methods exist for detecting similarities of string-based data, the methods for capturing similarities in numeric data are rather primitive. Typically, the numbers are treated as strings (and compared using the metrics described above) or simple range queries, which locate numbers with similar values. Koudas et al. [KoMS04] suggest, as direction for future research, consideration of the distribution and type of the numeric data, or extending the notion of cosine similarity for numeric data to work well for duplicate detection purposes.

2.7. Duplicate Record Detection

One of the most important tasks in data cleaning is the de-duplication of records, i.e., the detection of multiple representation of a single entity. This procedure implies matching between records, a procedure which is not straightforward in real world problems. For example, duplicate records may be erroneous due to a combination of factors such as transcription errors or incomplete information.

Elmagarmid et al. in [ElIV06] describe the methods that deal with the problem of data deduplication. The presented methods can be broadly divided into two categories:

- Approaches that rely on *training data to "learn"* how to match the records. This category includes (some) probabilistic approaches and supervised machine learning techniques.
- Approaches that rely on *domain knowledge* or on *generic distance metrics* to match records. This category includes approaches that use declarative languages for matching, and approaches that devise distance metrics appropriate for the duplicate detection task.

Elmagarmid et al. in [ElIV06] classified the data de-duplication methods in five main categories, which are described in the following paragraphs.

2.7.1. Notation

The tables that need to be matches are denoted as A and B and it is assumed, without loss of generality, that A and B have n comparable fields. In the duplicate detection problem, each tuple pair $<a, b>$, $(a \in A, b \in B)$ is assigned to one of the two classes M and U.

The class M contains the record pairs that represent the same entity *("match")* and the class U contains the record pairs that represent two different entities *("non-match")*.

Each tuple pair $<a, b>$ is represented as a random vector $\underline{x} = [x_1,\ldots,x_n]^T$ with n components that correspond to the n comparable fields of A and B. Each x_i shows the level of agreement. of the *ith* field for the records a and b. Many approaches use binary values for the x_i's and set $x_i = 1$ if field i agrees and let $x_i = 0$ if field i disagrees.

2.7.2. Probabilistic Matching Models

Newcombe et al. [NKAJ59] were the first to recognize duplicate detection as a Bayesian inference problem. Then, Fellegi and Sunter [FeSu69] formalized the intuition of Newcombe et al. introducing the notation described above.

The comparison vector $\underline{x}$ is the input to a decision rule that assigns $\underline{x}$ to U or to M. The main assumption is that $\underline{x}$ is a random vector whose density function is different for each of the two classes. Then, if the density function for each class is known, the duplicate detection problem becomes a *Bayesian inference problem.* Various techniques have been developed for addressing this *"general"* decision problem. Some of those techniques are: (i) *Bayes Decision Rule for Minimum Error,* (ii) *Bayes Decision Rule for Minimum Cost,* and (iii) *Decision with a Reject Region.* Each method mentioned above takes advantage of a decision rule based on probabilities. This decision rule is used in order to decide whether $\underline{x}$ belongs to U or M.

2.7.3. Supervised and Semi-Supervised Learning

The development of new classification techniques in the machine learning and statistics communities prompted the development of new de-duplication techniques. The supervised learning systems rely on the existence of training data in the form of record pairs, pre-labeled as matching or not.

One set of supervised learning techniques treat each record pair *<a, b>* independently, similarly to the probabilistic techniques mentioned in the previous paragraph. Cochinwala et al. [CKLS01] used the well-known *CART* algorithm, which generates classification and regression trees, a linear discriminant algorithm, which generates linear combination of the parameters for separating the data according to their classes, and a *"vector quantization"* approach, which is a generalization of nearest neighbor algorithms. The experiments which were conducted indicate that CART has the smallest error percentage.

Bilenko et al. [Bil+03] use *SVMlight* to learn how to merge the matching results for the individual fields of the records. Bilenko et al. showed that the SVM approach usually outperforms simpler approaches, such as treating the whole record as one large field. A typical post-processing step for these techniques is to construct a graph for all the records in the database, linking together the matching records. Then, using the transitivity assumption, all the records that belong to the same connected component are considered identical. However, the transitivity assumption can sometimes result in inconsistent decisions.

The supervised clustering techniques described above have records as nodes for the graph. Singla and Domingos [SiDo04] observed that by using attribute values as nodes, it is possible to propagate information across nodes and improve duplicate record detection. For example, if the records *<Microsoft, CA>* and *<MicrosoftCorp., California>* are deemed equal, then *CA* and *California* are also equal, an information that can be useful for other record comparisons.

Pasula et al. [Pas+02] proposed a semisupervised probabilistic relational model that can handle a generic set of transformations. While the model can handle a large number of duplicate detection problems, the use of exact inference results in a computationally intractable model. They proposed the use of a Markov Chain Monte Carlo (MCMC) sampling algorithm to avoid the intractability issue. However, it is unclear whether techniques that rely on graph-based probabilistic inference can scale well for data sets with hundreds of thousands of records.

2.7.4. Active-Learning-Based Techniques

One of the problems with the supervised learning techniques is the requirement for a large number of training examples. While it is easy to create a large number of training pairs that are either clearly non-duplicates or clearly duplicates, it is very difficult to generate ambiguous cases that would help create a highly accurate classifier. Based on this observation, some duplicate detection systems used active learning techniques to automatically locate such ambiguous pairs. Unlike an *"ordinary"* learner that is trained using a static training set, an *"active"* learner actively picks subsets of instances from unlabeled data, which, when labeled, will provide the highest information gain to the learner.

Sarawagi and Bhamidipaty [SaBh02] designed *ALIAS*, a learning based duplicate detection system, that significantly reduces the size of the training set. The main idea behind *ALIAS* is that most duplicate and non-duplicate pairs are clearly distinct. For such pairs, the system can automatically categorize them in U and M without the need of manual labeling. ALIAS requires humans to label pairs only for cases where the uncertainty is high.

ALIAS starts with small subsets of pairs of records designed for training, which have been characterized as either matched or unique. This initial set of labeled data forms the training data for a preliminary classifier. In the sequel, the initial classifier is used for predicting the status of unlabeled pairs of records. The initial classifier will make clear determinations on some unlabeled instances but lack determination on most.

The goal is to seek out from the unlabeled data pool those instances which, when labeled, will improve the accuracy of the classifier at the fastest possible rate. Pairs whose status is difficult to determine serve to strengthen the integrity of the learner. Conversely, instances in which the learner can easily predict the status of the pairs do not have much effect on the learner. Using this technique, ALIAS can quickly learn the peculiarities of a data set and rapidly detect duplicates using only a small number of training data.

Tejada et al. [TeKM01], [TeKM02] used a similar strategy and employed decision trees to teach rules for matching records with multiple fields. Their method suggested that by creating multiple classifiers, trained using slightly different data or parameters, it is possible to detect ambiguous cases and then ask the user for feedback.

2.7.5. Distance-Based Techniques

Active learning techniques require some training data or some human effort to create the matching models. In the absence of such training data or ability to get human input, supervised and active learning techniques are not appropriate.

One way of avoiding the need for training data is to define a distance metric for records. Using the distance metric and an appropriate matching threshold, it is possible to match similar records, without the need for training data.

One approach is to treat a record as a *long field*, and use one of the *distance metrics* to determine which records are similar. Monge and Elkan [MoEl96], [MoEl97] proposed a string matching algorithm for detecting highly similar database records. The basic idea was to apply a general purpose field matching algorithm, especially one that is able to account for gaps in the strings, to play the role of the duplicate detection algorithm.

Cohen [Cohe00] suggested to use the *tf.idf* weighting scheme, together with the *cosine similarity* metric to measure the similarity of records. Koudas et al. [KoMS04] presented some practical solutions to problems encountered during the deployment of such a string-based duplicate detection system at AT&T.

Distance-based approaches that conflate each record in one big field may ignore important information that can be used for duplicate detection. A simple approach is to measure the distance between individual fields, using the appropriate distance metric for each field, and then compute the weighted distance between the records. In this case, the problem is the computation of the weights, which is very similar to the probabilistic setting described in previous paragraph.

An alternative approach, proposed by Guha et al. [GKMS04] is to create a distance metric that is based on ranked list merging. The basic idea is that if only one field is compared from the record, the matching algorithm can easily find the best matches and rank them according to their similarity, putting the best matches first. By applying the same principle for all the fields, each record is associated with n ranked lists of records, one for each field. Then, the goal is to create a rank of records that has the minimum aggregate rank distance when compared to all the n lists.

Guha et al. map the problem into the *minimum cost perfect matching problem*, and developed efficient solutions for identifying the *top-k matching records*. The first solution was based on the *Hungarian Algorithm*, a graph-theoretic algorithm that solves the minimum cost perfect matching problem. Guha et al. also present the *Successive Shortest Paths algorithm* that works well for smaller values of k and is based on the idea that it is not required to examine all potential matches to identify the top-k matches.

The distance-based techniques described so far, treat each record as a flat entity, ignoring the fact that data is often stored in relational databases, in multiple tables. Ananthakrishna et al. [AnCG02] describe a similarity metric that uses not only the textual similarity, but the "co-occurrence" similarity of two entries in a database. For example, the entries in the state column *"CA"* and *"California"* have small textual

similarity; however, the city entries *"San Francisco"*, *"Los Angeles"*, *"San Diego"* and so on, often have foreign keys that point both to *"CA"* and *"California"*. Therefore, it is possible to infer that *"CA"* and *California"* are equivalent.

Ananthakrishna et al. showed that using *"foreign key co-occurrence"* information substantially improves the quality of duplicate detection in databases that use multiple tables to store the entries of a record.

One of the most important problems of the distance-based techniques is the definition of an appropriate value for the matching threshold. An appropriate threshold value could be computed by supervised techniques. However, the main advantage of distance-based techniques lies in their ability to operate without training data.

2.7.6. Rule-Based Approaches

A special case of distance-based approaches is the use of *rules* to define whether two records are the same or not. Rule-based approaches can be considered as distance-based techniques, where the distance of two records is either 0 or 1.

Wang and Madnick [WaMa89] proposed a rule-based approach for the duplicate detection problem. For cases in which there is no global key, Wang and Madnick suggest the use of rules developed by experts to derive a set of attributes that collectively serve as a *"key"* for each record. For example, an expert could define the following rule:

IF *age* < 22	THEN	*status* = UNDERGRADUATE
	ELSE	*status* = GRADUATE

By using such rules, Wang and Madnick attempted to generate unique keys that can cluster multiple records representing the same real-world entity. Lim et al. [LSPR93] also used a rule-based approach, but with the extra restriction that the result of the

rules must always be correct. Therefore, the rules should not be heuristically-defined but should reflect absolute truths and serve as functional dependencies.

Hernandez and Stolfo [HeSt98] further developed this idea and derived an equational theory that dictates the logic of domain equivalence. This equational theory specifies an inference about the similarity of the records. For example, if two persons have similar name spellings, and these persons have the same address, we may infer that they are the same person. Specifying such an inference in the equational theory requires declarative rule language. For example, for an employee database, the following rule could be developed:

```
FORALL (r1, r2) in EMPLOYEE
  IF r1.name is similar to r2.name AND
     r1.address = r2.address
  THEN r1 matches r2
```

In such a rule similarity is measured by using a string comparison technique and matching implies that both records are meant to be duplicates.

AJAX [Galh01] is a prototype system that provides a declarative language for specifying data cleaning programs, consisting of SQL statements enhanced with a set of primitive operations to express various cleaning transformations. AJAX provides a framework wherein the logic of a data cleaning program is modeled as a directed graph of data transformations starting from some input source data.

Four types of data transformations are provided to the user of the system. The *mapping* transformation standardizes data, the *matching* transformation finds pairs of records that probably refer to the same real object, the *clustering* transformation groups together matching pairs with a high similarity value, and finally, the *merging* transformation collapses each individual cluster into a tuple of the resulting data source.

Typically, rule-based systems operate with high accuracy. However, those systems require huge manual efforts from human experts in order to exploit the critical generated rules.

2.8. Experimental Methodology of Existing Methods

This paragraph describes the datasets and the experimental parameters that are used in common methods that cope with problems such as the duplicate elimination and off-line cleaning problem. More specifically, the following paragraphs include a description of datasets' nature for each method and the experimental parameters used for the evaluation of the method's performance.

2.8.1. Duplicates

Chaudhuri et al. in [CGGM03] used a clean *Customer[name, city, state, zip code]* relation consisting of about 1.7 million tuples from an internal operational data warehouse. They created input datasets by introducing errors in randomly selected subsets of Customer tuples. All characteristics of real data such as variations in token lengths and frequencies of tokens are preserved in the erroneous input tuples.

They considered two types of error injection methods. The type I method introduces errors in tokens with equal probability, i.e., all tokens in a column are equally likely to become erroneous, whereas Type II method introduces errors in tokens with a probability that is directly proportional to their frequency, i.e., tokens with higher frequency are more likely to become erroneous.

According to set of signatures they evaluated *Normalized Elapsed Time* and *Accuracy*, whose description is stated below.

Normalized Elapsed Time refers to the elapsed time for processing the set of input tuples using the fuzzy match algorithm divided by the elapsed time to process one

input tuple using the naïve algorithm which compares an input tuple with each reference tuple. If the normalized time for a fuzzy match algorithm is less than the number of input tuples, then it outperforms the naive algorithm.
Accuracy describes the percentage of input tuples for which a fuzzy match algorithm identifies the seed tuple, from which the erroneous input tuple was generated, as the closest reference tuple is its accuracy.

Chaudhuri et al. in [ChGK06] performed all of their experiments by using a *customer relation* from an operational data warehouse. Using variant edit similarity threshold values they estimate *the time needed for implementing similarity joins* on a relation *R* of 25.000 customer addresses with itself.

Yuan et al. in [SuLS02] evaluate the performance of their method by using synthetic databases containing records of 13 fields. The errors they introduced in duplicate records range from small typographical changes to large changes of some fields.

In order to test the performance of their method, they use variant sliding window sizes for finding similarities between records belonging in the same window and variant database duplicate ratios and database sizes as well. More specifically, according to those variant parameters they evaluated the *time required to run each method*, *the number of duplicate pairs found* and the *number of comparisons.*

2.8.2. Off-line cleaning

Bhattacharya et al. in [BhGe04] used as datasets, cliques of entities containing information about authors. Those data were transformed by adding noise using a probabilistic model.

They evaluate their algorithm by measuring the quality of the clusters generated using different group and clique size . In order to estimate the cluster quality they use *entity dispersion* and *cluster diversity* as measures of cluster quality. *Entity dispersion* reflects the number of different clusters that references corresponding to the same

entity are spread over, which means that a perfect de-duplication has dispersion 1, whereas *cluster diversity* quantifies the number of distinct entities that have been put in the same cluster.

Leung in [Leun04] used a transaction database of 100k records with an average transaction length of 10 items and a domain of 1000 items. He estimated the run-time of algorithms according to variant percentages of frequent itemsets satisfying succint constrains processed before tightening or before relaxing the constraint.

Zhu et al. in [ZhWC03] used synthetic data from IBM (IBM Synthetic Data) and real-world datasets from UCI repository (Blake and Merz, 1998). They introduced class noise in some data instances using a probabilistic noise. In order to estimate their method's performance they evaluate the classification accuracy of instances according to different levels of noise.

Low et al. in [LoLL01] used two real world datasets, including a company dataset, which requires complex matching logic and data manipulation and a patient dataset, which is a much larger dataset containing many errors. According to different numbers of duplicate identification rules, they evaluated the run-time needed, the error percentage and the measure of recall.

CHAPTER 3. PROBLEM DESCRIPTION AND PROPOSED METHOD

3.1. Problem Description

In this chapter, we will mainly deal with the approximate match of tuples based on string-valued attributes. More specifically, the procedure of approximate matching involves the retrieval of clean tuples, whose similarity with the incoming tuple is above a threshold value. As shown in Fig. 3.1, if the similarity between an input customer tuple and its closest reference tuple is higher than some threshold, then the correct reference tuple is loaded. Otherwise, the input is routed for further cleaning before considering it as referring to a new customer. A fuzzy match operation that is resilient to input errors can effectively prevent the proliferation of fuzzy duplicates in a relation, which represent multiple tuples describing the same real world entity.

The critical ingredient of a fuzzy match operation is the similarity function used for comparing tuples. In typical application domains, the similarity function must definitely handle string-valued attributes and possibly even numeric attributes.

Given the similarity function and an input tuple, the goal of the fuzzy match operation is to return the reference tuple being closest to the input tuple. An extension is to return the closest K reference tuples enabling users, if necessary, to choose one among them as the target, rather than the closest. A further extension is to only output K or fewer tuples whose similarity to the input tuple exceeds a user-specified minimum similarity threshold.

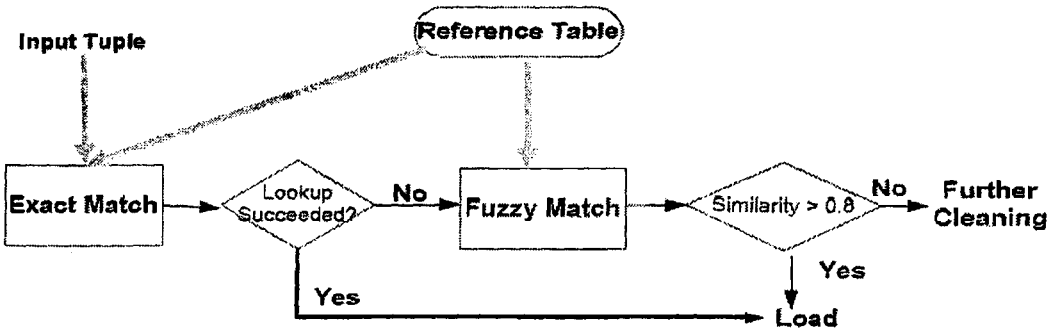


Fig. 3.1 Template for Using Fuzzy Match [CGGM03]

In our problem, the result of a fuzzy match operation applied on an input tuple could be one of the following classifications:

- *exactly matched*, i.e., a reference tuple exactly matched with the input tuple
- *approximately matched*, i.e., a reference tuple approximately matched with the input tuple
- *not resolved*, i.e., a set of K reference tuples enabling users, if necessary, to choose one among them
- *new record*, i.e., no reference is matched with the input tuple

Every input tuple is classified according to its maximum similarity with reference tuples. If this similarity is above a maximum threshold value, then the input is classified either as exactly matched, either as approximately matched. If the maximum similarity is below a minimum threshold, the input tuple is classified as a new record. Otherwise, the input tuple is classified as not resolved. This procedure is shown in Fig. 3.2.

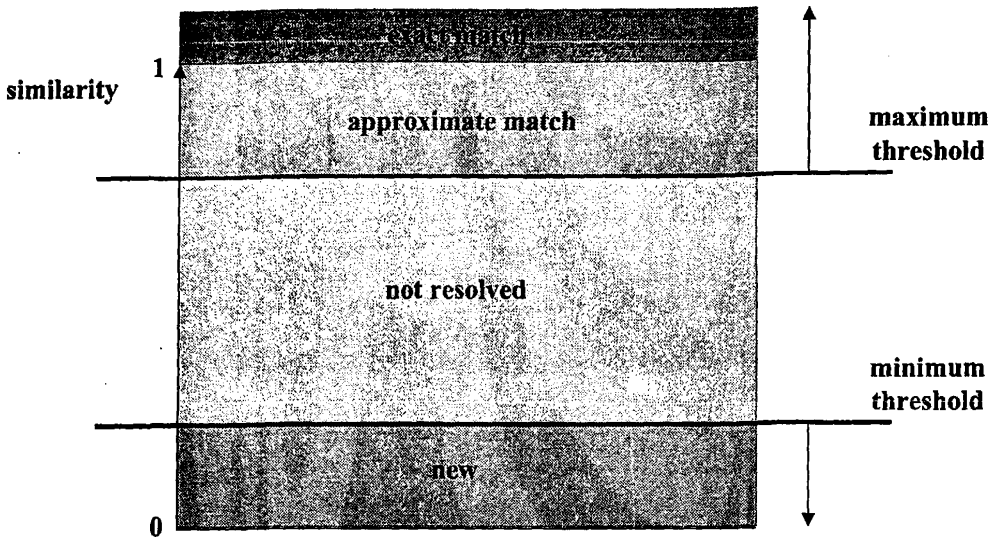


Fig. 3.2. Classification of Input Tuple According to Maximum Similarity

3.2. Baseline Method (Fuzzy Match Data Cleaning)

Chaudhuri et al. in [CGGM03] adopt a probabilistic approach where the goal is to return the closest K reference tuples with high probability. The author's method preprocesses the reference relation to build an index relation, called the error tolerant index (ETI) relation, for retrieving at run time a small set of candidate reference tuples, which are compared with the input tuple. Their retrieval algorithm retrieves with high probability a superset of the K reference tuples closest to the input tuple. The index relation ETI is implemented and maintained as a standard relation.

The authors propose a fuzzy match similarity function that explicitly considers IDF token weights and input errors while comparing tuples, an error tolerant index and a probabilistic algorithm for retrieving the K reference tuples closest to the input tuple, according to the fuzzy match similarity function.

3.2.1. Fuzzy Similarity Function (fms)

Informally, the similarity between an input tuple and a reference tuple is the cost of transforming the former into the latter. Low transformation costs of input tuples denote high similarity.

A transformation operation is applied upon a set of tokens derived from the attributes of a tuple. Supposing v is a reference tuple, the set of tokens included in attribute i is denoted by *tok[v(i)]*. For example, if *v(i) = "Boeing Company"*, then the resulting set of tokens coming from the tokenization function on *v(i)* equals to *tok[v(i)] = {Boeing, Company}*.

Each of the following transformation operations: (a) *token replacement*, (b) *token insertion* and (c) *token deletion* is associated with a cost that depends on the weight of the token being transformed. The weight function for each token can be defined as:

$$w(t,i) = IDF(t,i) = \log\frac{|R|}{freq(t,i)},$$

where *freq(t,i)* denotes the *frequency* of a token t in column i and equals to the number of tuples v in R such that *tok(v[i])* contains t.

More specifically, considering the case where u is an input tuple and v is a reference tuple, the cost of operations taking place in order to transform u into v is defined as follows:

- *Token replacement*: The cost of replacing a token *t1* in *tok(u[i])* by token *t2* from *tok(v[i])* is $ed(t_1, t_2) \cdot w(t_1, i)$, where *tok(u[i])* is the set of tokens held on attribute *u[i]* and $ed(t_1, t_2)$ is the edit distance between t_1 and t_2. If t_1 and t_2 are from different columns, the cost is defined to be infinite.
- *Token insertion*: The cost of inserting a token t into *u[i]* is $c_{ins} \cdot w(t, i)$, where the *token insertion factor* c_{ins} is a constant between 0 and 1.
- *Token deletion*: The cost of deleting a token t from *u[i]* is $w(t, i)$.

Transforming u into v requires each column *u[i]* to be transformed into *v[i]* through a sequence of transformation operations, whose cost is defined to be the sum of costs of all operations in the sequence. The *transformation cost tc(u[i], v[i])* is the cost of the minimum cost transformation sequence for transforming *u[i]* into *v[i]*. The cost *tc(u, v)* of transforming *u* into *v* is the sum over all columns *i* of the costs *tc(u[i], v[i])* of transforming *u[i]* into *v[i]* and equals to:

$$tc(u,v) = \sum_i tc(u[i], v[i])$$

The fuzzy match similarity function *fms(u, v)* between an input tuple *u* and a reference tuple *v* in terms of the transformation cost *tc(u, v)* can be defined as:

$$fms(u,v) = 1 - \min\left(\frac{tc(u,v)}{w(u)}, 1.0\right),$$

where *w(u)* is the sum of *weights* of all tokens in the token set *tok(u)*. Token set *tok(u)* denotes the multiset union of sets $tok(a_1),\ldots,tok(a_n)$ of tokens from the tuple $u[a_1,\ldots,a_n]$.

Suppose an example reference relation *R* with attributes (r_1, r_2) and $|R| = 10$ and tuples *u = ("John", "Ford")* and *v = ("Jahn", "Ford")*. If two reference tuples contain attribute values *"John"* and *"Ford"* in attributes r_1 and r_2 respectively, the transformation cost and the *fms* function are computed as follows:

$$\begin{aligned} tc(u,v) &= tc(\text{"John"}, \text{"Jahn"}) + tc(\text{"Ford"}, \text{"Ford"}) \\ &= ed(\text{"John"}, \text{"Jahn"}) \cdot w(\text{"John"}, r_1) + ed(\text{"Ford"}, \text{"Ford"}) \cdot w(\text{"Ford"}, r_2) \\ &= 1 \cdot \log\frac{|R|}{freq(\text{"John "}, r_1)} + 0 \cdot \log\frac{|R|}{freq(\text{"Ford "}, r_2)} = \log\frac{10}{2} = 0.699 \end{aligned}$$

$$tok(u) = tok(u_1) \cup tok(u_2) = \{\text{"John"}, \text{"Ford"}\}$$

$$w(u) = w(u_1) + w(u_2) = w(\text{"John"}, r_1) + w(\text{"Ford"}, r_2) = \log\frac{10}{2} + \log\frac{10}{2} = 1.398$$

$$fms(u,v) = 1 - min\left(\frac{tc(u,v)}{w(u)}, 1.0\right) = 1 - min\left(\frac{0.699}{1.398}, 1.0\right) = 1 - 0.5 = 0.5$$

3.2.2. Fuzzy Match

Given an input tuple u, the goal of the fuzzy match algorithm is to identify the approximate matches, i.e., the K reference tuples closest to **u**. Particularly, the *K-Fuzzy Match Problem* can be defined as follows:

Given a reference relation R, a minimum similarity threshold c $(0 < c < 1)$, the similarity function *fms*, and an input tuple u, find the set $FM(u)$ of *fuzzy matches* of at most K tuples from R such that:

1. $fms(u, v) \geq c$, for all v in $FM(u)$
2. $fms(u, v) \geq fms(u, v')$ for any v in $FM(u)$ and v' in $R-FM(u)$

A naïve algorithm scans the reference relation R comparing each tuple with u. A more efficient approach is to build an "index" on the reference relation for quickly retrieving a superset of the target fuzzy matches. Standard index structures like B+-tree indexes cannot be employed in this context because they can only be used for exact or prefix matches on attribute values. Therefore, during a pre-processing phase, additional indexing information can be gathered for efficiently implementing the fuzzy match operation. The additional information can be stored as a standard database relation and be indexed using standard B+-trees to perform fast exact lookups. Chaudhuri et al. in [CGGM03] refer to this indexed relation as the *error tolerant index (ETI)*. The authors' challenge was to identify and effectively use the information in the indexed relation. The authors' solution was based on the insight of deriving from *fms* an easily indexable similarity function fms^{apx} with the following characteristics:

1. fms^{apx} upper bounds *fms* with high probability,
2. The *error tolerant index (ETI) relation* can be built for retrieving efficiently a small candidate set of reference tuples whose similarity with the input tuple u, as per fms^{apx}, is greater than the *minimum similarity threshold* c.

Therefore, with a high probability the similarity as per *fms* between any tuple in the candidate set and u is greater than c. From this candidate set, the K reference tuples closest to u can be returned as the fuzzy matches.

The authors used fms^{apx} as an approximation of *fms* for which they could build an indexed relation. fms^{apx} is obtained (a) by ignoring differences in ordering among tokens in the input and reference tuples, and (b) by allowing each input token to match with the "closest" token from the reference tuple. Since disregarding these two distinguishing characteristics while comparing tuples can only increase similarity between tuples, fms^{apx} is an upper bound of *fms*.

For example, the tuples *[boeing company, seattle, wa, 98004]* and *[company boeing, seattle, wa, 98004]* which differ only in the ordering among tokens in the first field are considered identical by fms^{apx}.

In fms^{apx}, the authors measured the closeness between two tokens through the similarity between sets of substrings of tokens, called *qgram sets*, instead of edit distance between tokens used in *fms*. Further, this *qgram set similarity* is estimated well by the commonality between small probabilistically chosen subsets of the two qgram sets. This property can be exploited in order to build an indexed relation for fms^{apx}, because for each input tuple only the reference tuples whose tokens share a number of chosen qgrams with the input tuple must be identified.

Given a string s and a positive integer q, the *qgram set* denoted by $QG_q(s)$ is the set of all size q substrings of s. For example, the *3-gram set* QG_3*("boeing")* is {*boe, oei, ein, ing*}.

In order to estimate fms^{apx}, it is necessary to compute the *token min-hash signature* and the *min-hash similarity* between two tokens. Let U denote the universe of strings over an alphabet Σ, and $h_i: U \rightarrow N$, $i = 1,...,H$ be H hash functions mapping elements of U uniformly and randomly to the set of natural numbers N. Let S be a set of strings.

The *min-hash signature mh(S)* of S is the vector $[mh_1(S), ..., mh_H(S)]$ where the i^{th} coordinate $mh_i(S)$ is defined as $mh_i(S) = \arg\min_{a \in S} h_i(a)$.

The intuition behind the hash functions h_i is to isolate qgrams in specific coordinates. It is obvious that variable H indicates the number of qgrams being isolated. The selection is random, but each hash function returns a qgram standing in a specific coordinate of the token.

Token similarity can be defined in terms of the *min-hash similarity* between their qgram sets. Let q and H be positive integers. Let $I[X]$ denote an *indicator variable* over boolean X, i.e., $I[X] = 1$ if X is true, and 0 otherwise. The *min-hash similarity* $sim_{mh}(t_1,t_2)$ between tokens $t1$ and $t2$ is:

$$sim_{mh}(t_1,t_2) = \frac{1}{H}\sum_{i=1}^{H} I[mh_i(QG(t_1)) = mh_i(QG(t_2))]$$

From the above definition, it is obvious that the min-hash similarity computes the average number of common qgrams returned from the same hash function.

Suppose tokens t_1 = *"William"* and t_2 = *"Williams"* and $H = 3$. A possible min-hash signature might be $mh(QG(t_1)) = [Wil, lli, iam]$ and $mh(QG(t_2)) = [Wil, lli, ams]$ for t_1 and t_2 respectively. Two out of three hash functions returned the same qgram for both tokens. That means that tokens t_1 and t_2 have 2 qgrams in common and the min-hash similarity $sim_{mh}(t_1, t_2)$ is equal to $\frac{1}{3} \cdot 2$.

Taking into consideration the previous definition of *min-hash similarity*, the *fms* approximation fms^{apx} can be defined as follows:

Let u, v be two tuples $d_q = (1-1/q)$ be an adjustment term.

$$fms^{apx}(u,v) = \frac{1}{w(u)}\sum_{i}\sum_{t \in tok(u[i])} w(t) \cdot \underset{r \in tok(v[i])}{Max}\left(\frac{2}{q} sim_{mh}(QG(t), QG(r)) + d_q\right)$$

Suppose u is the input tuple and v is a reference tuple. From the above definition, it is obvious that fms^{apx} searches for every attribute token of input tuple u the most similar corresponding attribute token of tuple v, in terms of min-hash similarity. Maximum similarities are computed and multiplied with the weight of the matched input token. This means that infrequent tokens play an important role in fms^{apx}, since their weight is greater than a frequent token. fms^{apx} is computed by dividing this weighted sum with the overall weight of the input tuple u.

3.2.3. The Error Tolerant Index (ETI)

The primary purpose of ETI is to enable, for each input tuple u, the efficient retrieval of a candidate set S of reference tuples whose similarity with u is greater than the minimum similarity threshold c.

From the definition of fms^{apx}, $fms^{apx}(u,v)$ is measured by comparing min-hash signatures of tokens in *tok(u)* and *tok(v)*. Therefore, for determining the candidate set, it is essential to efficiently identify for each token t in *tok(u)*, a set of *tids* corresponding to reference tuples sharing min-hash qgrams with that of t. In order to identify such sets of *tids*, ETI holds each qgram s along with the list of all *tids* of reference tuples with tokens whose min-hash signatures contain s.

Suppose R is the reference relation and H the size of the min-hash signature. *ETI* as shown in Fig. 3.3 is a relation with the following schema: *[QGram, Coordinate, Column, Frequency, Tid-list]*. For each tuple e in *ETI* it holds that *e[Tid-list]* contains the list of tids of all reference tuples containing at least one token t in the field *e[Column]* whose *e[Coordinate]-th* min-hash coordinate is *e[QGram]*. The number of tids included in *e[Tid-list]* is stored in *e[Frequency]* attribute.

Q-gram	Coordinate	Column	Frequency	Tid-list
oei	1	1	1	{R1}
ing	2	1	1	{R1}
com	1	1	2	{R1,R3}
pan	2	1	2	{R1,R3}
bon	1	1	1	{R2}
orp	1	1	1	{R2}
ati	2	1	1	{R2}
sea	1	2	3	{R1,R2,R3}
ttl	2	2	3	{R1,R2,R3}
wa	1	3	3	{R1,R2,R3}
980	1	4	3	{R1,R2,R3}
004	2	4	1	{R1}
014	2	4	1	{R2}
024	2	4	1	{R3}

Fig. 3.3. ETI Relation Example [CGGM03]

3.2.4. Query Processing Algorithm

Chaudhuri et al. in [CGGM03] proposed an algorithm for processing fuzzy match queries. Such queries ask for K fuzzy matches of an input tuple u whose similarities as per fms with u are above a minimum similarity threshold c.

The authors' goal was to reduce the number of lookups against the reference relation by effectively using the ETI. Their proposed algorithm fetches tid-lists by looking up ETI of all qgrams in min-hash signatures of all tokens in u. For efficient lookups, the authors assume that the reference relation R is indexed on the *Tid* attribute, and the ETI relation is indexed on the *[QGram, Coordinate, Column]* attribute combination.

More specifically, the algorithm for processing the fuzzy match query given an input tuple u is as follows. For each token t in $tok(u)$, its IDF weight $w(t)$ is computed, requiring the frequency of t. Those frequencies can be stored in the ETI and be fetched by issuing a SQL query per token. Then the minhash signature $mh(t)$ of each token t is determined and each qgram in $mh(t)$ is assigned a weight equal to $w(t)/|mh(t)|$. Using the ETI, it is feasible to generate the candidate set S of reference tuple tids whose similarity, as per fms^{apx} and hence fms, with the input tuple u is greater than c. All tuples in S are fetched from the reference relation in order to verify

whether or not their similarities with u as per *fms* are truly above c. Among those tuples which passed the verification test, the K tuples with the K highest similarity scores are returned.

The candidate set S is computed as the union of sets S_k, one for each qgram q_k in the min-hash signatures of tokens in *tok(u)*. For a qgram q_k, which is the i^{th} coordinate in the min-hash signature *mh(t)* of a token t in the j^{th} column, S_k is the tid-list from the tuple $[q_k, i, j, freq(q_k, i, j), S_k]$ in ETI. The lookup for $[q_k, i, j, freq(q_k, i, j), S_k]$ is efficient because of the index on the required attribute combination of ETI.

Each tid in S_k is assigned a score that is proportional to the weight *w(t)* of the parent token t. If a tuple with tid r is very close to the input tuple u, then r is a member of several sets S_k and hence gets a high overall score. Otherwise, r has a low overall score. The candidate set is constituted by tids that have an overall score greater than $w(u) \cdot c$ minus an adjustment term, which represents a correction to approximate the edit distance between tokens with the similarity between their qgram sets.

During the process of looking up tid-lists corresponding to qgrams, the scores of tids belonging in these tid-lists are maintained in a hash table. The score of a tid equals the sum of weights of all qgrams whose tid-lists it belongs to. The weight $w(q_k)$ assigned to a qgram q_k in the min-hash signature $mh(t_i)$ of a token t_i is $w(t_i)/|mh(t_i)|$. If a tid in S_k is already present in the hash table, then its score is incremented by $w(q_k)$. Otherwise, the tid is added to the hash table with an initial score of $w(q_k)$. After all qgrams in the signatures of input tokens are processed, a tid r is selected and added to the candidate set S only if its score is above $w(u) \cdot c$ minus the adjustment term.

The query processing algorithm proposed by Chaudhuri et al. in [CGGM03] is summarized in Fig. 3.4.

```
FuzzyMatch(Input tuple u, H, ETI, R, c)
1. Initialize hash table TidScores; AdjustmentTerm = 0
2. Tokenize u and compute min-hash signatures Q of all tokens
3. Assign token weights; RemWt = sum of all token weights
4. threshold = c·RemWt
5. For each q-gram s in Q s.t. s = mh_i(t) of token t in column col
6.    if (mh_i(t) is the first q-gram of mh(t) to be looked up)
7.       AdjustmentTerm += w(t)·(1-1/q)
8.    Fetch tid-list(s) by looking up (s, i, col) against ETI
9.    Update TidScores
         a. Increment scores of existing tids by w(t)/|mh(t)|
         b. If RemWt ≥ threshold, insert new tids with score w(t)/|mh(t)|.
10.   RemWt -= w(s)
11. Fetch tuples from R for TIDs with score ≥ c–AdjustmentTerm
12. Compare, using f, each of these tuples with u
13. Return K (or less) most similar tuples with similarity above w(u)·c
```

Fig. 3.4. Query Processing Algorithm [CGGM03]

3.3. Improvements: Online Data Cleaning using Qgram tries

As already mentioned, our goal is to deal with the problem of approximate matching between reference and input tuples. More specifically, we face the problem of classifying each input tuple as an existing or a new tuple, before its input to the reference table. This means that an input tuple might be an erroneous representation of a reference tuple. This may occur due to typing errors or others types of noise.

Our goal is to successfully classify input tuples within a short period of time. In cases of erroneous input tuples, it is a challenge to determine with a high probability whether the tuple can be matched with an existing reference tuple or characterize the tuple as a new record. It is also critical to avoid mismatches of input tuples that are already stored in the reference table. Due to the fact that the whole procedure must not exceed a time threshold, our main target is to choose the appropriate data structures that will effectively clean a stream of input tuples.

Specifically, we provide the following extensions to the method proposed by Chaudhuri et al. in [CGGM03]:

- replacement of ETI index with a similar index that holds more information about the reference table
- use of a data structure held in main memory for the retrieval of frequently accessed candidate tuples
- proposal of an algorithm combining the above structures to effectively classify input tuples

3.3.1. Word Index

The proposed method uses a structure called *word index* that is quite similar to ETI. The *word index* holds information about the attribute values stored in the reference table. *Word index* structure consists of five fields: *(a) qgram, (b) coordinate, (c) column, (d) tid-list* and *(e) frequency* being described as follows:

- *qgram* field corresponds to a sequence of Q characters.
- *coordinate* field represents the occurrence position of the corresponding *qgram* within a string value. For example, if this string value s begins with a qgram q, then the coordinate value of q for s equals to 1.
- *column* field indicates the string-valued attribute that holds the specific value.
- *tid-list* field contains a list created from tuple ids that include *qgram* Q in the position which is denoted by the *coordinate* field.
- *frequency* field represents the number of the tuple ids belonging to the tid-list.

The word index structure is used for the retrieval of tuple ids that probably match the input tuple. Processing the qgrams within an input attribute value, the tids with attributes that share common qgrams in a specific position within the word are retrieved.

Before deciding whether the input tuple is clean or not, a candidate set of tuple ids is retrieved from the word index. The candidate set includes the tuples that possibly

match the input tuple. This candidate set is generated by returning the tid-lists of tids with attribute values that share common qgrams in same positions. The attribute values correlated with the returned tids are cached in the qgram trie.

Supposing that the word index is indexed on *{qgram, coordinate, column}* set of fields, the retrieval of the candidate tuple ids can take place efficiently.

3.3.2. Qgram Trie

As mentioned above, the purpose of a *qgram trie* is the caching of clean attribute values that probably match the input string value. The proposed *qgram trie* is defined as follows:

1. The trie consists of one root labeled as "null", a set of word-prefix subtrees as the children of the root, and a header table.
2. Each node in the word-prefix subtree consists of four fields:
   a) *qgram*, which records the item that this node represents
   b) *count* that records the number of the tuple ids represented by the portion of the path reaching this node
   c) *node-link* that links to the next node in the trie carrying the same qgram, or null if there is none
   d) *tid-list*, which records the set of tids with attribute values that share this node in the trie representation.
3. Each entry in the header table consists of two fields: (a) the *qgram* and (b) the *head of node-link*, which is a pointer pointing to the first node in the trie carrying the qgram. On the top of the header table are hold the last inserted header items.

According to the *qgram trie* definition, words with same prefixes share a number of nodes within a path of the trie. For example, if words *"Ric"*, *"Rica"* and *"Ricus"*,

with ids 1, 2 and 3 respectively are retrieved, the resulting *qgram trie* being built in memory is shown in Fig. 3.5.

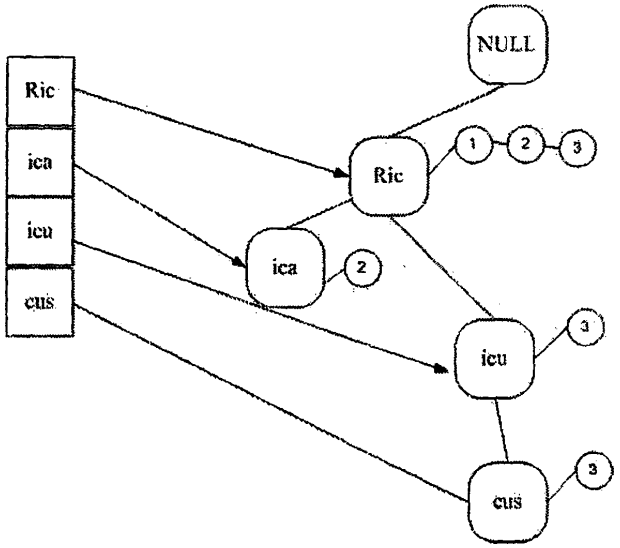


Fig. 3.5. Qgram Trie Example

Having stored all the candidate words that share common qgrams in same positions with the input value, the qgram trie described above is searched according to the *qgram sequence* of the input value. Then the set tuple ids with attribute values whose similarity with the input word is above a similarity threshold can be returned. The matching procedure, which is discussed in detail in section 3.3.5., is implemented by searching paths of the trie that hold qgrams of the input attribute value.

3.3.3. Qgram Trie Searching Algorithm

The proposed matching algorithm, which is described in section 3.3.5, takes advantage of the qgram trie described in the previous section. Taking into consideration that the qgram trie contains the *qgram sequences* of candidate clean attribute values that probably match the input attribute value, it is efficient to extract from it the most similar clean attribute in terms of *qgram set similarity*. This means that attribute values containing the most common qgrams with the input value are very similar to it and can be returned as fuzzy matches of the possibly dirty input

value. Given the qgram trie and a suitable searching algorithm, the qgram set similarity can be determined efficiently.

Suppose that the input attribute value is *"Ricuss"* with *qgram sequence {Ric, icu, cus, uss}*. The matching procedure for the specific word searches the paths of trie that hold the specific *qgram sequence.* Starting from nodes with qgrams *"Ric", "icu", "cus"* and *"uss"*, the matching procedure searches the occurrence of the qgram sequence *{Ric, icu, cus, uss}* in paths of the trie.

During the qgram trie searching procedure, a *score table* maintains the matching scores between the input value and the clean words. After every successful matching between nodes of trie and qgrams of subsequences, the *score table* is updated by incrementing the scores of the tuple ids that belong to the *tid-list* of the matched node. Ending this searching procedure, it is possible to retrieve a set of tuple ids with attribute values very similar to the input value.

Using the qgram trie shown in Fig. 3.5, if the input attribute value is *"Ricuss"* the scores of words *"Ric", "Rica"* and *"Ricus"* are 1, 1 and 3 respectively, denoting that *"Ricus"* is the closest clean word to the input value. The searching procedure is summarized in Fig. 3.6:

Input: *input tuple u*
Output: *K closest tuples to u*
1. For each attribute value *a* of *u*
    a. Generate qgram sequence *s* of input value *a*
        - Find first qgram *q* of *s* in header table
            i. Access all nodes holding *q*
            ii. Search all possible paths of trie with nodes holding the qgram subsequence *s* beginning with *q*
            iii. Update score table in case of successful match
2. Sort score table
3. Return *K* tuple ids with most similar attribute values according to their score

Fig. 3.6. Searching Procedure

3.3.4. Main Memory Maintenance of Qgram Trie

As already mentioned, a qgram trie is used in order to cache attribute values of frequently accessed reference tuples. This means that it must be updated after every processing procedure of incoming tuples.

Due to the fact that the qgram trie must not exceed a maximum size, the two update operations are (i) the *insertion* and the (ii) *pruning* operation. The insertion operation adds a branch in the qgram trie and puts the correlated qgrams in the top of the header table of the qgram trie. For example, if reference tuple *("John", "Ford")* with tuple id equal to 1 is cached in the qgram trie, then the qgram trie will be constructed as it is shown in Fig. 3.7.a. If the reference tuple *("John", "Palm")* with tuple id equal to 2 is about to be inserted to the qgram trie, after the insertion the qgram trie will be updated as shown in Fig.3.7.b.

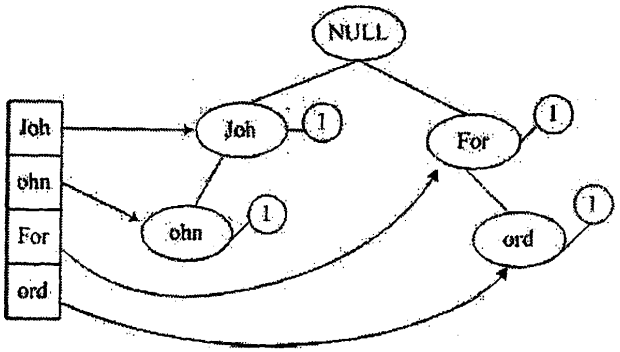


Fig. 3.7.a Qgram Trie Before Insertion

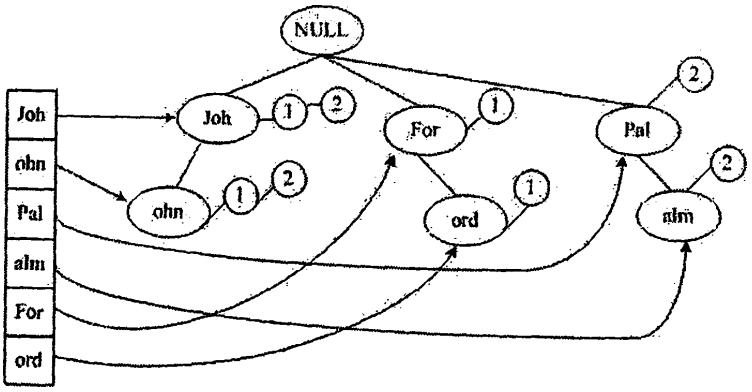


Fig. 3.7.b Qgram Trie After Insertion

As it is shown in Fig.3.7.b, after the insertion operation, the qgrams included in the attributes of the second reference tuple are put on the top of the header table. In other words, the frequent qgrams are moved to the top of the header table, indicating their recent access. This procedure takes place in order to facilitate the pruning procedure.

The pruning operation takes place when the qgram trie size reaches the maximum size. As mentioned above, the elements of the header table are sorted according to the last time they were accessed. The pruning of the qgram trie is implemented by using the least recently used (LRU) algorithm. More specifically, the less frequent qgrams, that lie in the bottom of the header table, are the first to be extracted from the qgram trie. Following the path beginning from the bottom header table items, the pruning procedure crops a qgram trie branch with a leaf holding the specific qgram. If a header table doesn't point any trie node, then it is removed from the header table. The whole procedure is completed when the qgram trie takes up a specific size of main memory.

Suppose the pruning procedure must be applied on the qgram trie shown in Fig. 3.7.b. If two trie nodes have to be cropped, the cropping algorithm makes the following steps:

1. Access the most infrequent header table item (in our example *"ord"*)
2. Follow the path of nodes beginning from the node indicated by this header table
3. For every leaf trie node *n* in the path
    - delete from the branch the tids held by node *n*
    - delete leaf trie node
    - delete all nodes holding no tid lists in the branch
4. If the trie didn't reach a specific size, execute iteratively step 3 for the next less frequent header table items (in our example *"For"*) and remove the header table items that don't link to any trie nodes.

The steps described above are shown in Fig. 3.8.a and Fig. 3.8.b.

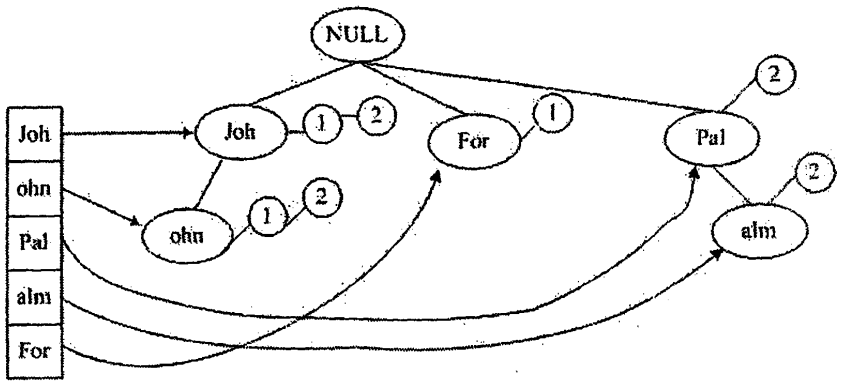


Fig. 3.8.a Pruning Procedure Results – Steps 1-3

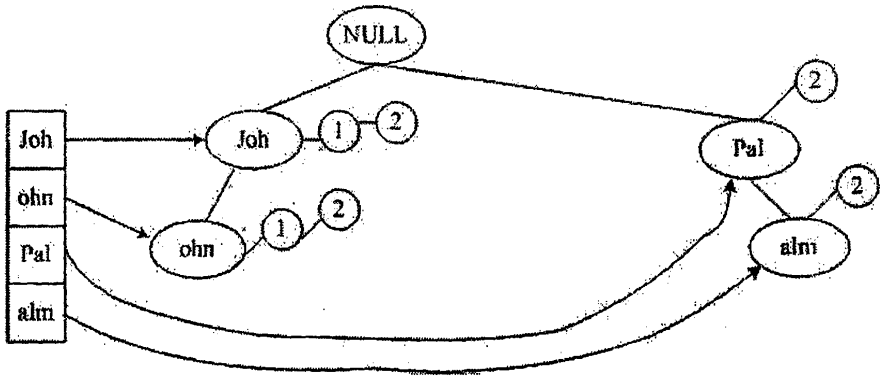


Fig. 3.8.b Pruning Procedure Results – Step 4

3.3.5. Matching Procedure

Chaudhuri et al. in [CGGM03] proposed a query processing algorithm in order to handle fuzzy match queries. The proposed method instead of the error tolerant index uses the word index described in section 3.3.1 and extends their algorithm using the qgram trie structure in order to accelerate the fuzzy matching procedure.

The word index is proposed for more accurate matching, since it holds more information than the error tolerant index (ETI). ETI holds for each attribute value only a subset of qgrams, which is chosen randomly and indicates approximate positions of qgrams within the attribute value, such as the prefix or the suffix of the attribute. This means that ETI can be constructed faster and requires less hard disk space than the

word index. However, this random selection may lead to inaccurate results, since input tuples with dirty attribute values have few qgrams in common with valid tuples.

The intuition behind the use of a qgram trie is its suitability for caching purposes, since the trie is maintained in main memory. More specifically, the trie will hold at any time the most frequent attribute values, avoiding redundant I/O activities that might occur when an input value is being repeated and needs to be processed. This way the whole procedure can be accelerated in the presence of same input attribute values.

More specifically our method is separated in two main parts. In the first part, using the qgram trie and the searching procedure described in the section 3.3.3., we examine whether the input tuple matches a tuple included in the qgram trie. In this way, if the input tuple matches a reference tuple stored in the qgram, we avoid any I/O activities and classify it as an existing tuple.

If the procedure fails to match the input tuple in the first step, then it uses the word index in order to retrieve candidate tuples. Specifically, for each qgram of the input attribute values, we retrieve from the word index all tuple ids sharing the specific qgram in the same position. For each tuple id we maintain a score in a hash table indicating the common qgrams of reference tuple with the input tuple. After this procedure is completed, the set of candidate reference tuples is that with a score above a minimum threshold.

For each candidate tuple, we check if there is an exact match with the input tuple. If a candidate tuple is found to be exactly similar, then the input tuple is classified as existing. Otherwise, we compute the similarity of the two tuples using the fms similarity function. If no candidate reference tuples match exactly the input tuple, then the reference tuple with the highest similarity value that exceeds a minimum similarity threshold, corresponds to the approximate match of the input value. If the highest similarity value is below a maximum similarity threshold, the input tuple is classified as a new record. Otherwise, the tuple is classified as not resolved.

If the input tuple is matched exactly or approximately with a reference tuple, the attributes of the reference tuple are stored in the qgram trie, in order to be retrieved in future matches.

The proposed algorithm is summarized in Fig. 3.9.

Input: Stream tuple *t*
Output: Classification of *t* as new or existing tuple

1. ∀ attribute value *v* of *t*
   - 1.1. Check if *v* exists in qgram trie
     - 1.1.1. if *v* exists
       - *v* is clean
       - retrieve tids contain *v*
     - 1.1.2. else not determined
2. If all attributes values are clean
   - 2.1. find the retrieved tids containing all input attribute values
3. Else
   - 3.1. initialize the hash table with score equal to 0 for all tids
   - 3.2. retrieve from word eti all tids sharing common qgrams with *t* in same positions
   - 3.3. ∀ qgram *q* of *v* increment the corresponding score of tids containing *q* in same position with *t*
   - 3.4. retrieve tids with *count(qgrams) > qgram_threshold*
   - 3.5. check if there is exact match with the retrieved tuples
     - 3.5.1. if there is exact match
       - classify *t* as existing tuple
     - 3.5.2. else find the retrieved tuple *r* with the highest *fms* value
       - if *fms(t, r) > approx_match_threshold*
         - classify *t* as approximately existing tuple
       - else if *fms(t, r) < new_threshold*
         - classify *t* as new tuple
       - else
         - classify *t* as not resolved
   - 3.6. if *t* is classified as existing tuple
     - update qgram trie with the clean attribute values of the existing tuple

Fig. 3.9. Matching Procedure

We illustrate the above procedure using the following example:

Suppose the reference table is the table shown in Fig. 3.10 and the current qgram trie is that of Fig.3.6.b holding the attribute values of tuple with ids 1 and 2.

tid	name	surname
1	John	Ford
2	John	Palm
3	Jack	Smith

Fig. 3.10. Example Reference Table

Suppose that the first input tuple is the tuple *("John", "Palm")*. At first, the algorithm will search the qgram trie to check if the input tuple matches exactly a reference tuple cached in the trie. The searching procedure in the trie will return the tid list *<1, 2>* for the input attribute *"John"* and the tid list <2> respectively for the input attribute *"Palm"*. Both tid lists have in common the tid *1*. That means that the input tuple is a valid tuple and the procedure classifies it as an existing tuple.

Suppose the second input tuple is *("John", "Lord")*. The searching procedure in the trie will return only the tid list *<1, 2>* for the attribute *"John"*. That means that the input tuple is not cached in the qgram trie and the algorithm will continue the process of the tuple using the word index. If the *qgram_threshold* is equal to 2, the procedure will generate a candidate set including tuple ids *1* and *2*, since the number of common qgrams are equal to 3 and 2 respectively for reference tuples with ids *1* and *2*. Reference tuple with tid *1* will be the most similar retrieved candidate tuple having the maximum *fms* value. If the *fms* value is above the *approx_match_threshold* value, then input tuple will be classified as an approximately existing tuple. If the *fms* value is below the *new_threshold* value the input tuple will be classified as a new record. If the *fms* value is between the two thresholds, the input value will be classified as not resolved.

CHAPTER 4. EXPERIMENTAL METHODOLOGY

4.1. Data generation

We have taken the data from U.S. Census Bureau [USCB07], which embarked on a names list project involving a tabulation of names from the 1990 Census. Data are divided in 3 files containing only the frequency of a given name, without any specific individual information. Specifically, each file contains last names, male first names and female first names.

We have generated a *reference* relation *R(full_name)* of 100K and created different data sets of streaming data with sizes obtained as a *s*% of the original reference relation. The stream of data signifies transactions that people whose names are possibly stored in the reference relation *R* have done (e.g., we have a *Customer* reference relation and a *Sales* stream of possibly erroneous data). The values that we have used for *s* are: 0.1, 1 and 10.

For the streaming data, we have intentionally created problems to the data. Given a certain percentage *p*% of noise level (i.e., errors in the names), we have created the following anomalies in equal probabilities:

- character addition

- character deletion
- character update
- character transposition

The values we have used for the noise level p are 0.1, 0.5, 10 and 20. Moreover, the streaming data also contain a percentage $r\%$ of repeating tuples. We have generated streams with repetition percentage $r\%$ equal to 0%, 10% and 20%.

4.2. Alternative methods for cleaning using qgram tries

The state-of-the-art in the area, and thus our adversary in this work is the [CGGM03] paper. We have implemented the ETI method at Berkeley DB v.4.6.

Our method operates on top of the ETI index described in [CGGM03], by using the word index described in Chapter 3, in the following way:

- A trie of qgrams is built *at-runtime*. In other words, to avoid the huge size of the a-priori trie, as soon as we (i) load the reference relation R with data, we (ii) populate the word index. Then, as tuples come, we incrementally add the clean part of trie. The intuition is that the most popular names will eventually be cached in main memory, without having to store all the trie.

The employed algorithm is simple: each incoming tuple is checked against the trie, ETI index, reference relation. We will refer to this triad as the reference database. Each tuple is classified as one of the following:

- *Clean* (originally clean or cleansed in an unambiguous way). A clean tuple in the stream can be a detected existing tuple (i.e., a tuple exists in the reference relation) or *New* (a respective tuple did not previously exist in the database)
- *Not-resolved* (because there are many candidates and manual attention is needed).

To determine whether a full name (i) exists in the reference database exactly, (ii) approximately matches an existing tuple, (iii) does not exist, or (iv) possibly exists but

there are many candidates for its value, we need a distance metric. The distance metric of choice in our examples has been the fms similarity function already explained in section 3.2.1.

A second alternative of the problem is to give a *restricted memory budget M* to our algorithm keeping only the *interesting* parts of the trie.

4.3. Experimental parameters and measures

The measured measures (y-axis) are:

- time to complete (from which a throughput can be determined)
- precision of classification
- memory used
- cache hits

The varied parameters are:

- the stream size
- the noise level
- available memory
- repetition of input tuples
- reference table size

Table 4.1. Varied parameters

Parameter	Description	Possible values
Stream Size	(*s*% of |*R*|)	0.1, 1, 10
Noise Level	(*p*% of |*R*|)	1, 5, 10, 20
Available memory	(*q*% of |*R*|)	10, 15, 20
Repetition of input tuples	(*r*% of |*s*|)	0, 10, 20
Reference Table Size	|*R*|	10K, 50K, 100K tuples

4.4. Experimental results

We executed a number of experiments in order to evaluate the measures described in the previous paragraph. More specifically, we evaluated the execution time, the precision of classification and the memory used according to variant parameters such as the stream size, the noise level, the size of qgram trie and size of the reference table.

In the following paragraphs we will comment the effect of the variant parameters on execution time, precision and memory consumed.

4.5. Execution time

In this paragraph we represent the execution time of both methods. The following graphical representations show the effect of the variant parameters described previously on execution time. More specifically, we measured the execution time for reference tables with size 10000, 50000 and 100000 tuples. In our setting, there are three basic scenarios. According to the first scenario, the input stream contains no repeating tuples. In the second and third scenario the input stream contains 10% and 20% duplicate input tuples, respectively. For each reference table size, we examined the effect of available memory on execution time. We measured the execution time for available memory equal to 10%, 15% and 20% of the reference table size.

4.5.1. The effect of noise on execution time

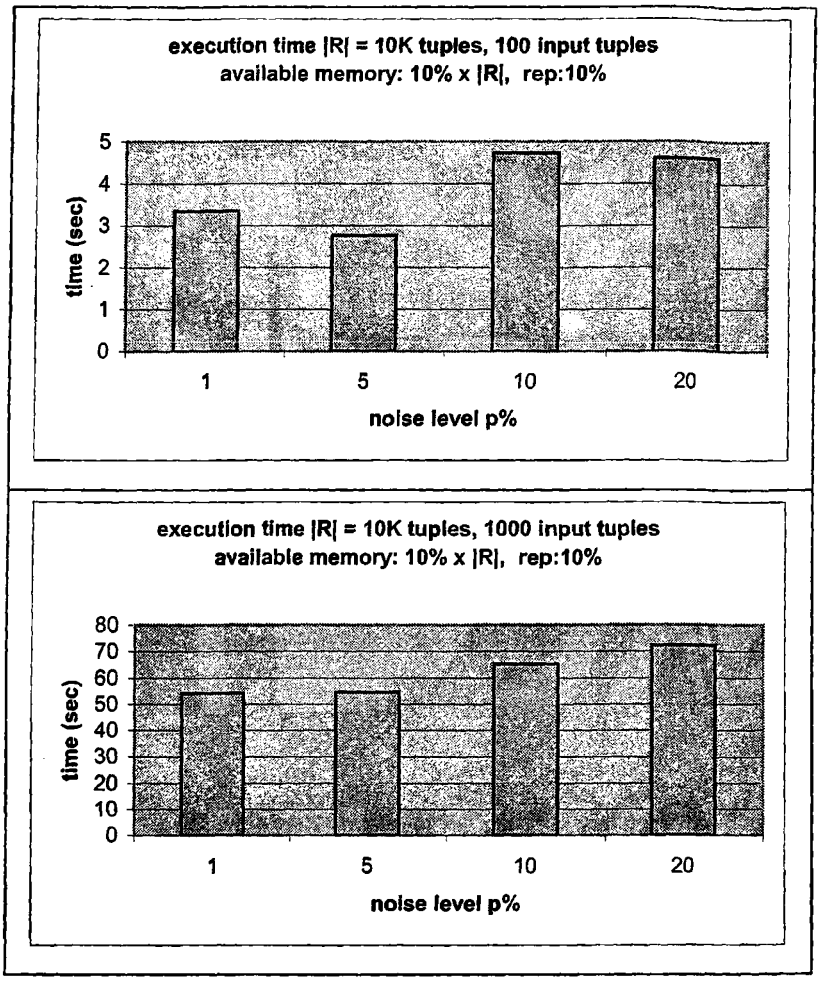


Fig. 4.1. Effect of Noise on Execution time

From the graphs represented in Fig. 4.1, it is obvious that the proposed method is sensitive to noise. Specifically, as the noise level increases, the execution time also increases but with a slowly increasing tendency.

4.5.2. The effect of repetition on execution time

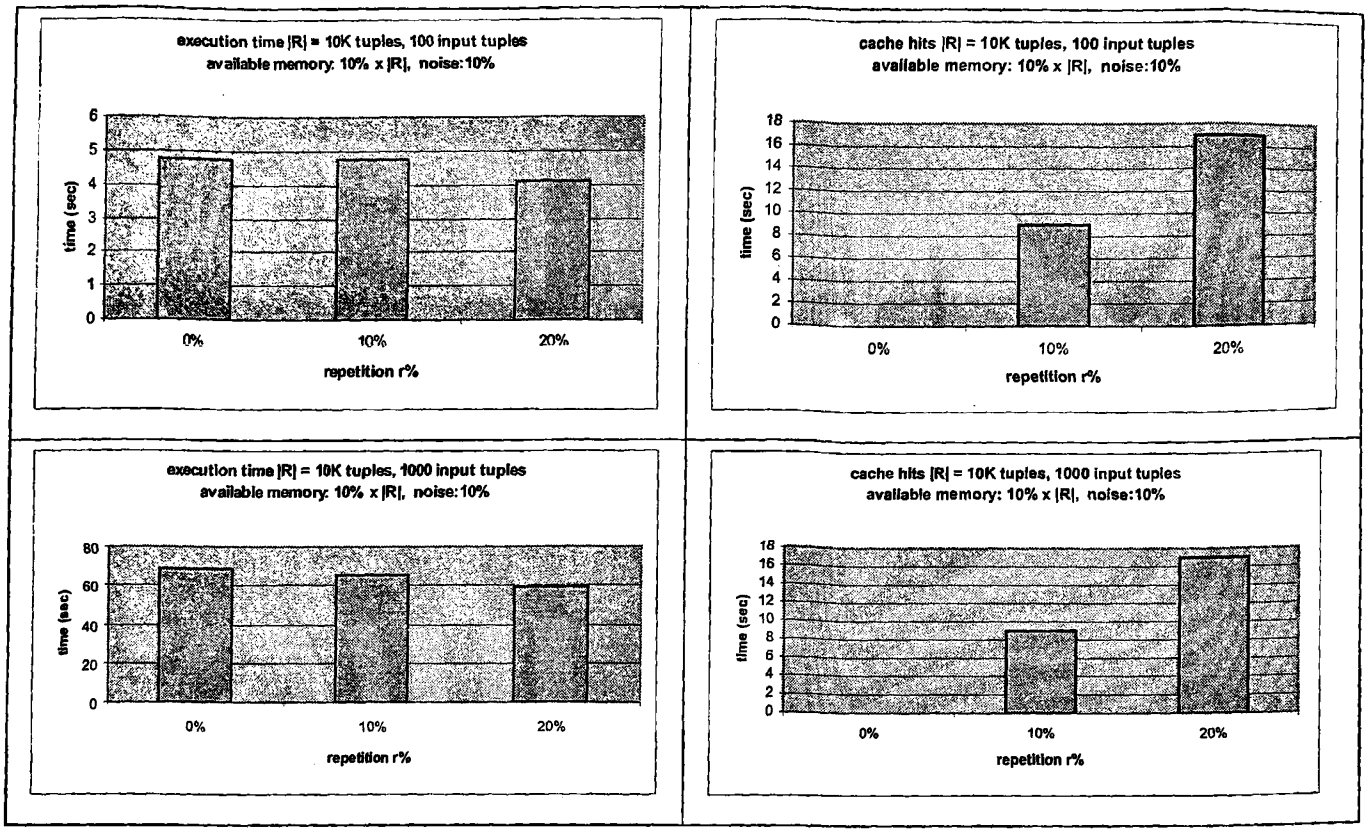


Fig. 4.2. Effect of Repetition on Execution time

The repetition of incoming tuples decreases the execution time. As shown in Fig. 4.2, repetition leads to more successful cache hits, thus, avoiding I/O activities.

4.5.3. The effect of available memory on execution time

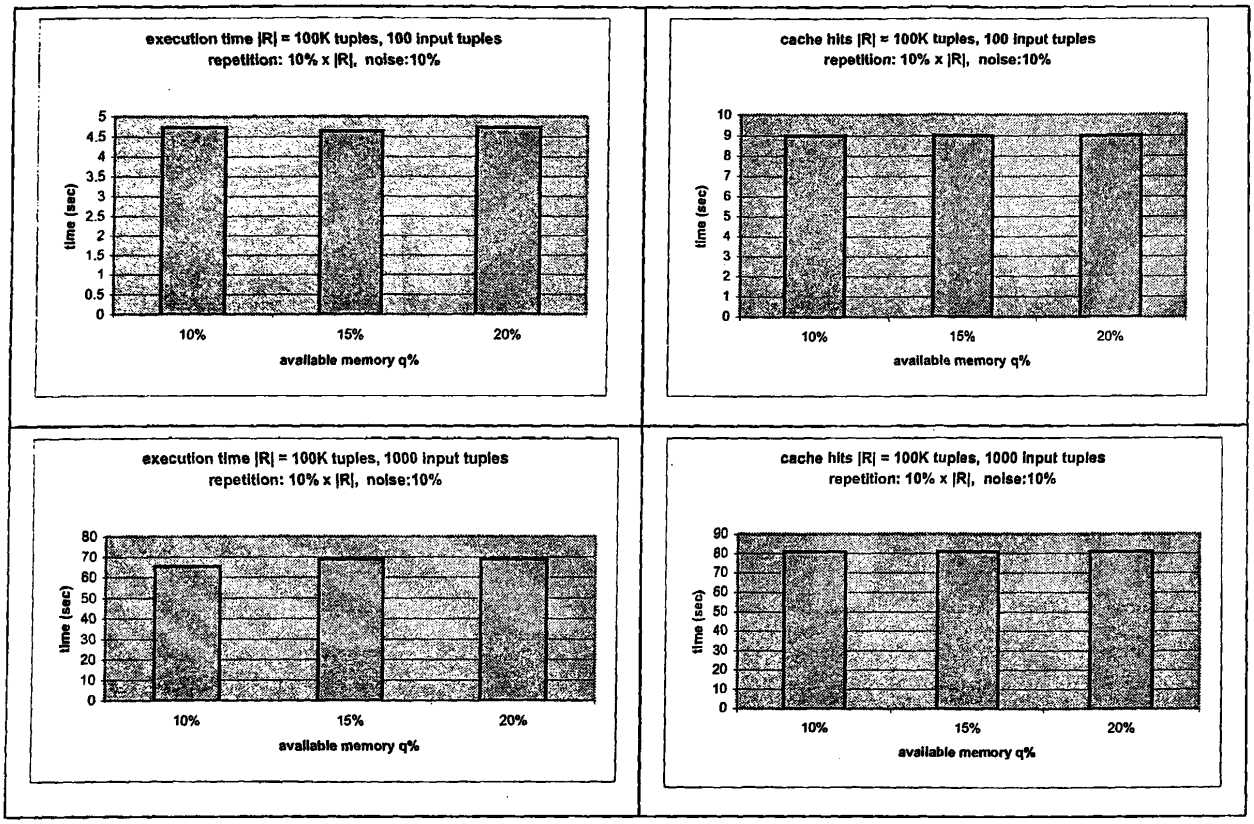


Fig. 4.3. Effect of Available Memory on Execution Time

We have experimented with three different budgets of memory. From Fig. 4.3, we conclude that execution time remains stable for available memory 10%, 15% and 20% of reference table size.

4.5.4. The effect of reference table size on execution time

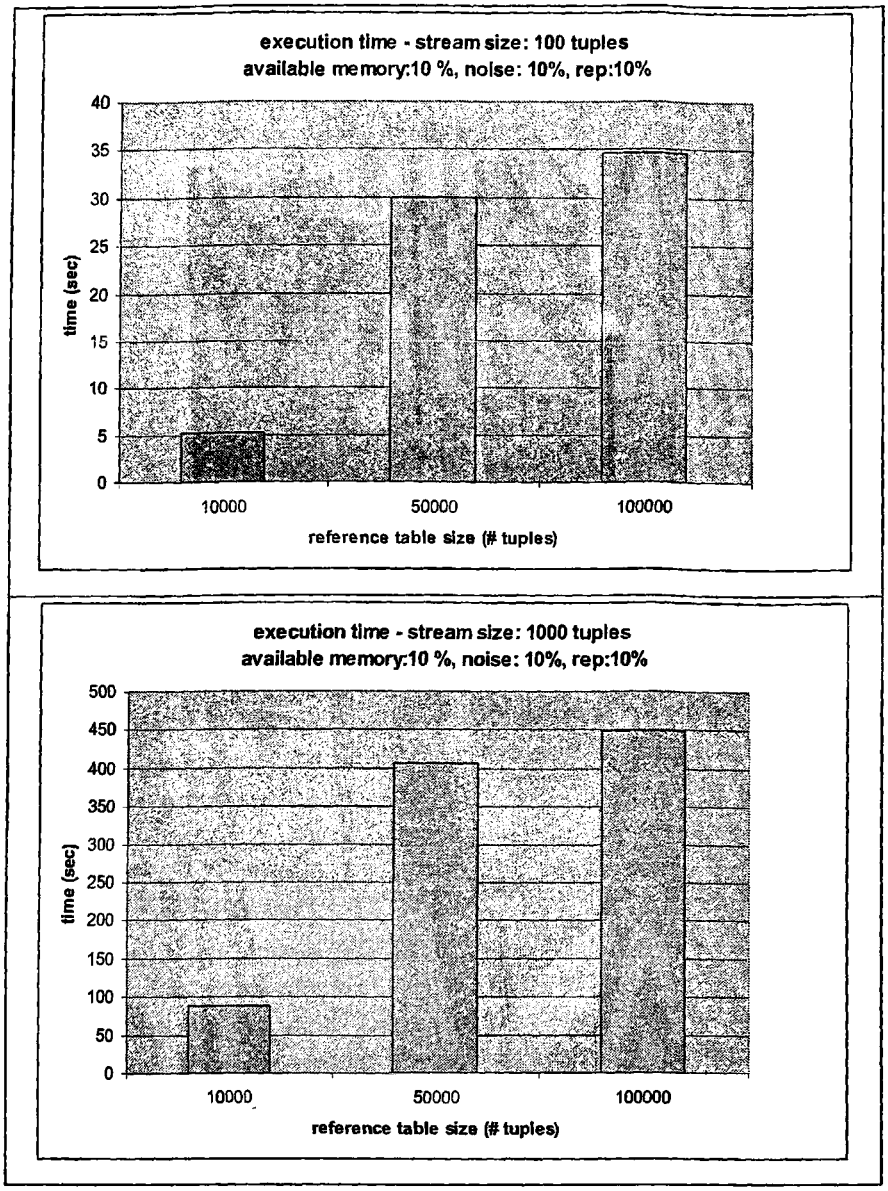


Fig. 4.4. Effect of Reference Table Size on Execution Time

As shown in Fig. 4.4, execution time is clearly affected by the reference table size. More specifically, as the reference table size increases, the execution time increases too. We note that for 50K and 100K reference tuples the execution time is slightly different, whereas for 10K reference tuples, execution time is significantly less.

In all cases though, the increase in execution time is sublinear and this is probably due to the effect of the trie.

4.5.5. Comparison with the state-of-the art method

For reference table size 10000 tuples, variant values for repetition of input tuples, available memory, noise level and stream size, the execution time is shown in Fig. 4.5 and Fig. 4.6.

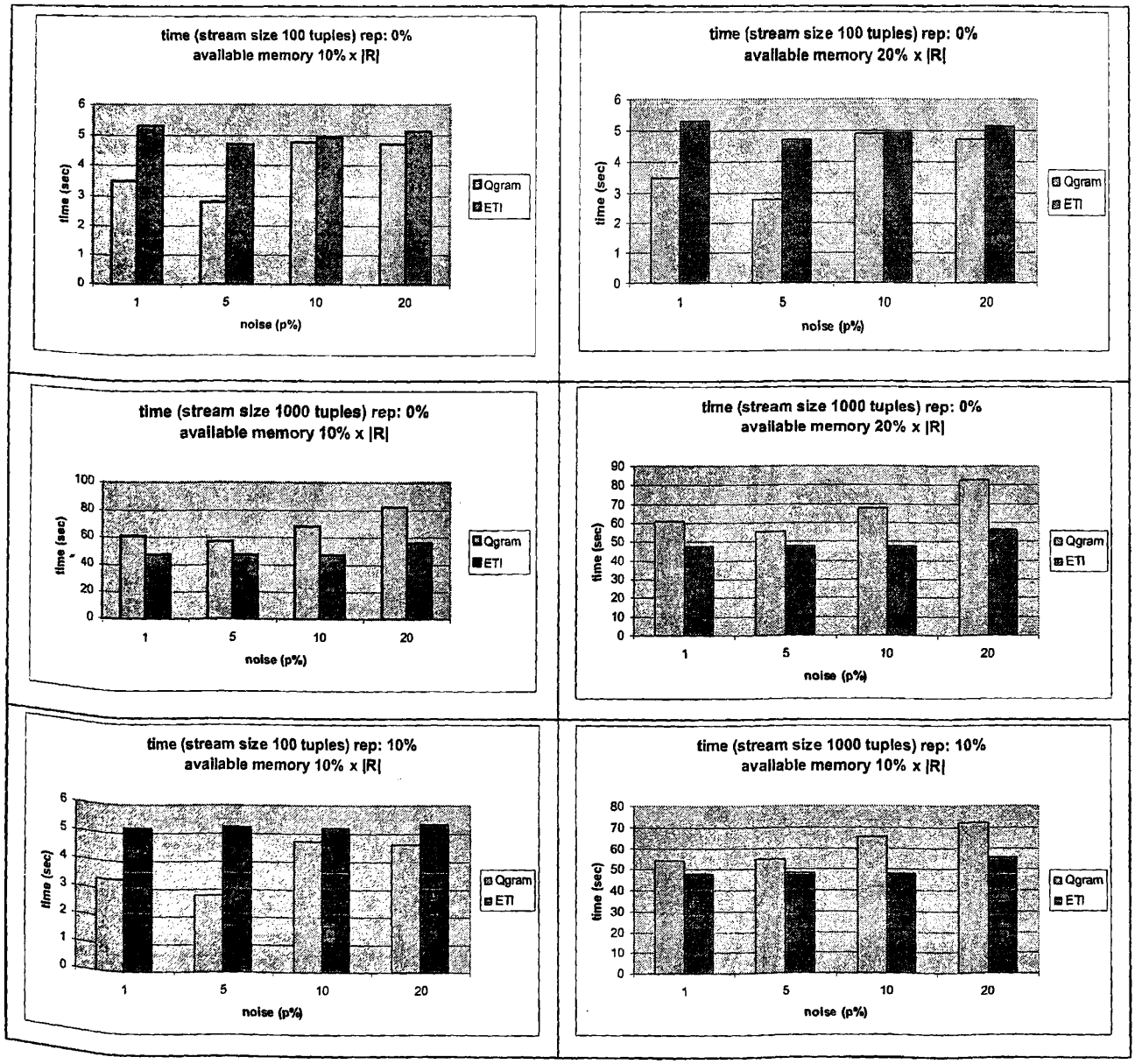


Fig. 4.5. Execution Time (|R|=10K tuples, variant repetition - available memory)

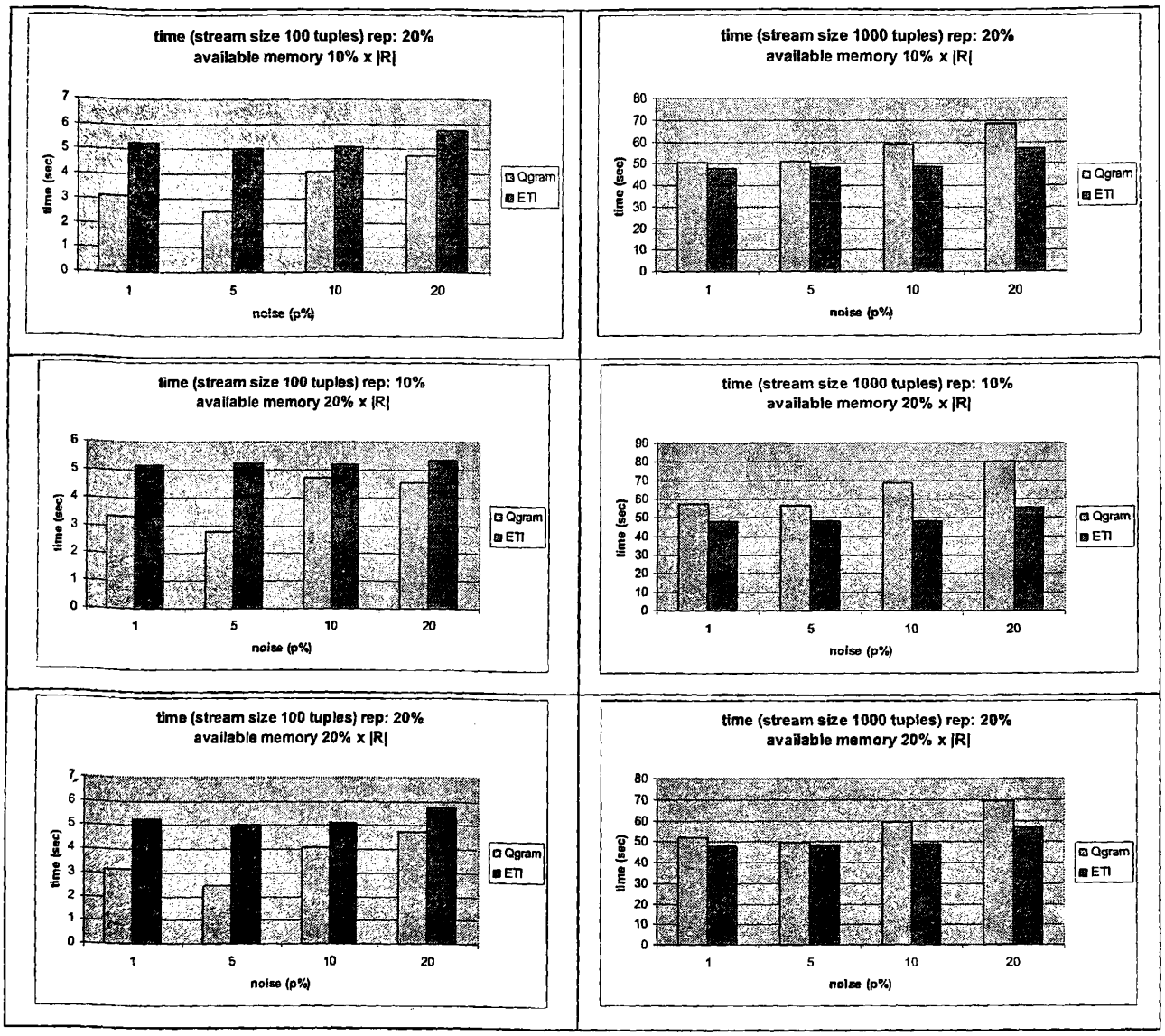


Fig. 4.6. Execution Time (|R|=10K tuples, variant repetition - available memory)

For reference table size 100000 tuples, variant values for repetition of input tuples, available memory, noise level and stream size, the execution time is shown in Fig. 4.7.

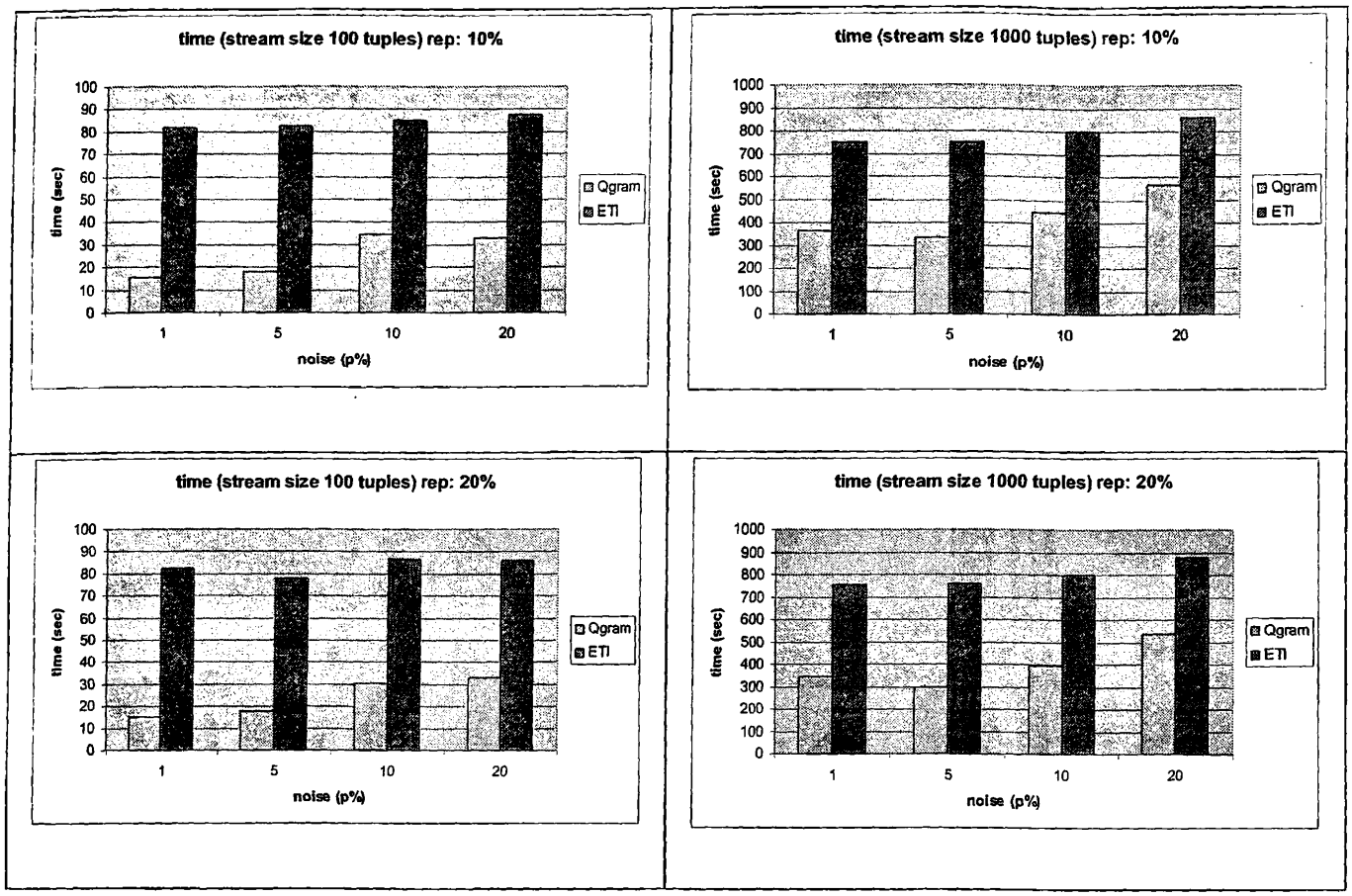


Fig. 4.7. Execution Time (|R|=100K tuples, , variant repetition - available memory)

We observe that our method is sensitive to the input stream size (Fig. 4.5, Fig. 4.6 and Fig. 4.7). More specifically, our method outperforms the state-of-the-art methods when the input stream does not exceed a specific size. This occurs due to the time needed for maintaining the qgram trie in main memory. For large streams and great percentage of repetition, our method works as efficiently as the state-of-the-art method. That means that many input tuples are already cached in main memory and the whole procedure is accelerated avoiding redundant I/O activities.

4.6. Precision of classification

In this paragraph we present the precision of classification of both methods, according to the three scenarios described in the previous paragraph. The following graphical representations show the effect of the variant parameters on the precision of

classification. More specifically, for each scenario we measure the number of correct matches (category *existing*), the number of correct classifications for new tuples (category *new*) and the number of not resolved tuples.

4.6.1. Effect of noise on precision of classification

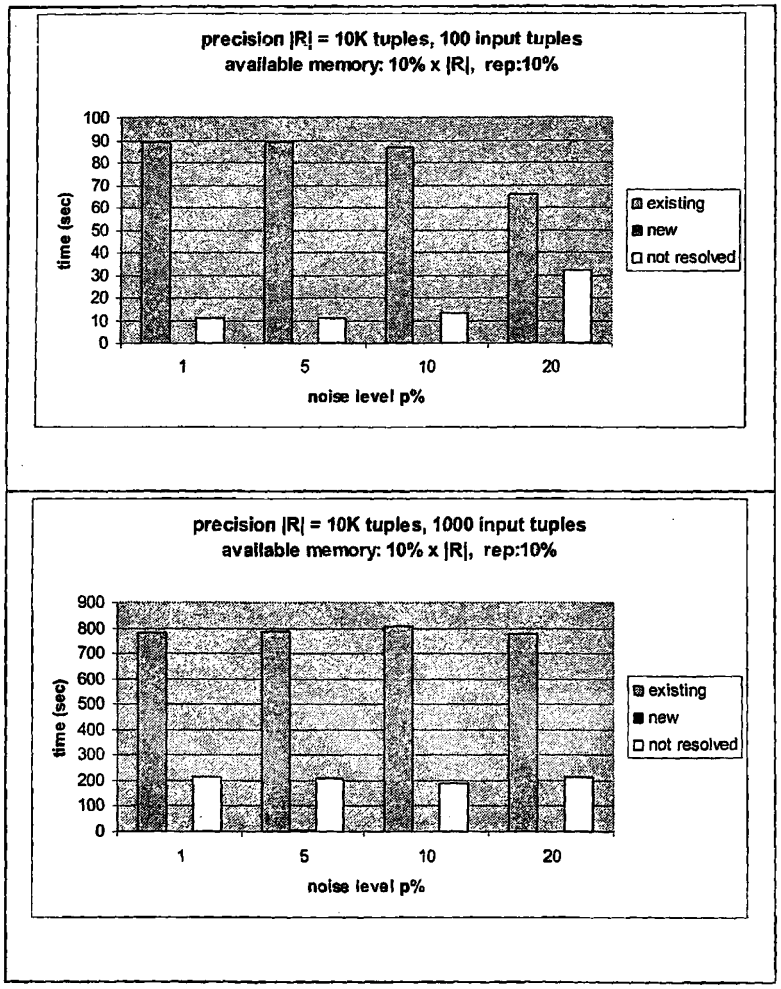


Fig. 4.8. Precision (|R|=10K tuples, repetition 10%, available memory 10%)

Fig. 4.8 depicts the sensitivity of our method to noise. For 20% noise level, the number of not resolved input tuples is increased, whereas the number of matches for existing tuples is decreasing. For smallest amounts of noise, the precision of classification remains stable. For appropriate values of similarity thresholds, there are no misclassifications of input tuples. Specifically, a dirty input tuple is not identified

as a new record and a new tuple is not approximately matched with an existing reference tuple. The effect of similarity thresholds on classification is described in detail in section 4.6.3.

4.6.2. Effect of repetition on precision of classification

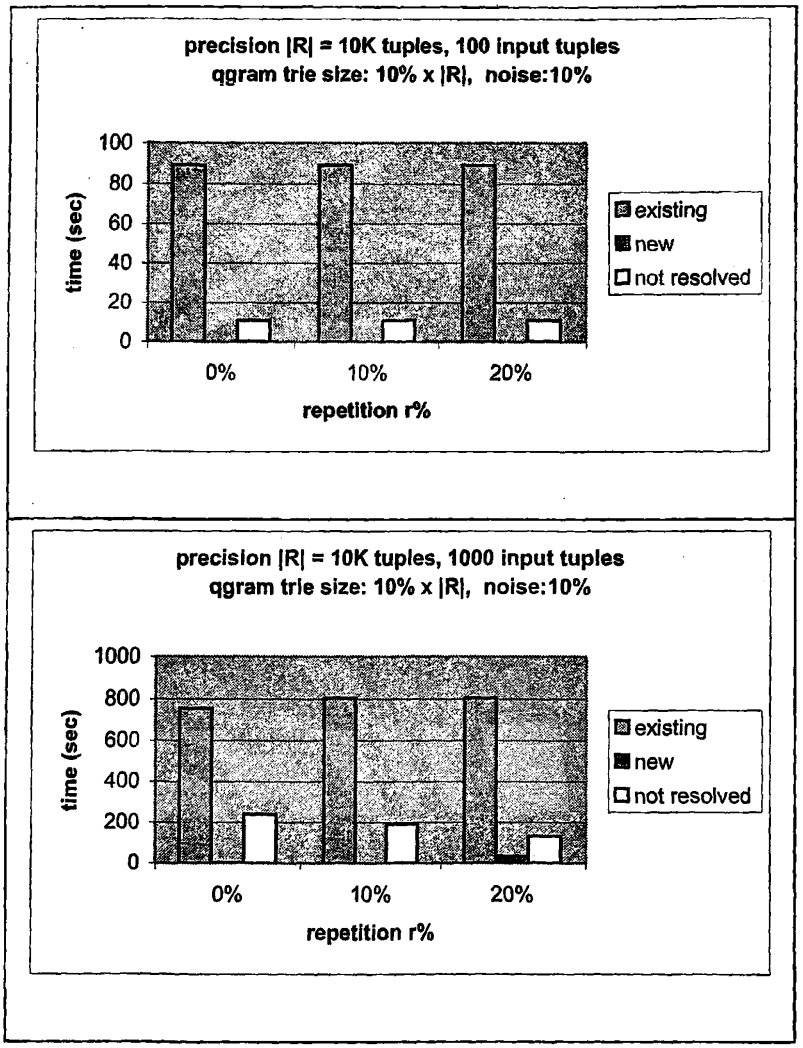


Fig. 4.9. Precision (|R|=10K tuples, noise 10%, available memory 10%)

We observe that repetition of incoming tuples does not affect the precision of classification (Fig. 4.9).

4.6.3. Effect of similarity thresholds on precision of classification

The precision of classification is greatly affected by the selection of threshold values. Changing the threshold value *thres_new* for determining an input tuple as a new tuple, we realize that the classification results change. In Fig. 4.10, we represent the classification results for *thres_new* equal to 0.2 and 0.5. We also represent the true classification for the specific input stream.

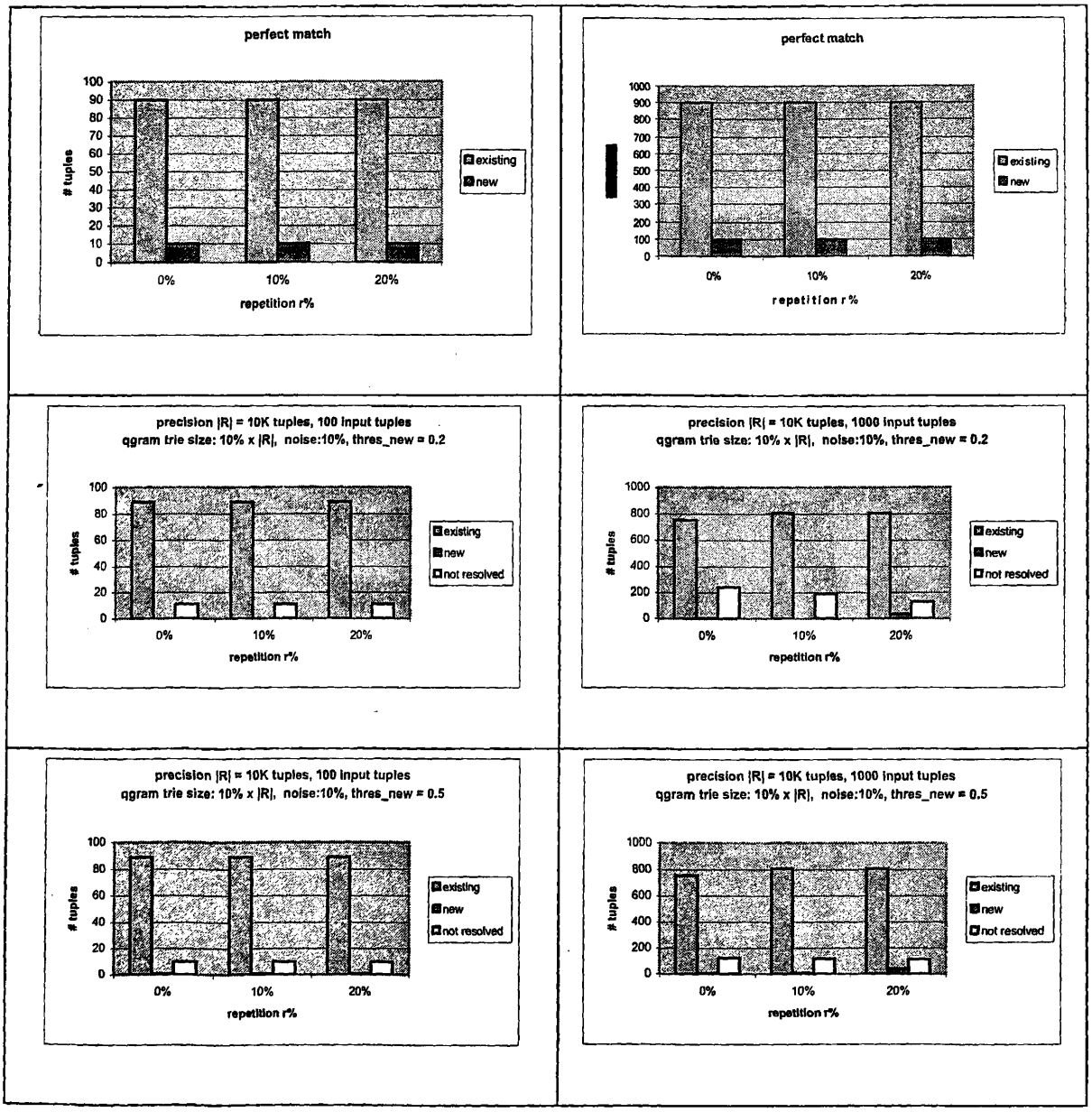


Fig. 4.10. Effect of Threshold Values on Precision

We realize that setting threshold *thres_new* to 0.5, affects the proper classification of input tuples. More specifically, dirty input tuples having maximum similarity value under this threshold are misclassified, because they are identified as new records. To visualize this erroneous classification, we depict the number of misclassified input tuples in the Fig. 4.11.

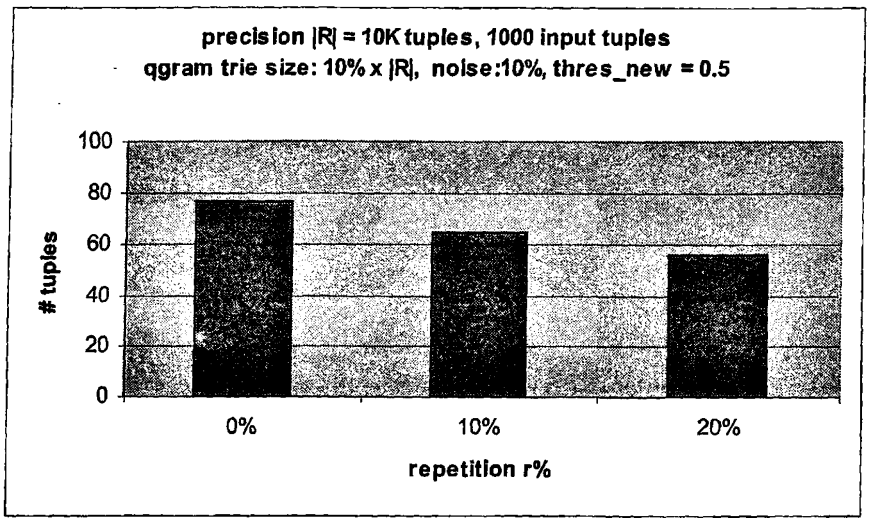


Fig. 4.11. Misclassifications of Dirty Input Tuples as New Records

4.6.4. Comparison with the state-of-the-art method

For reference table size 10000 tuples, variant values for repetition of input tuples, available memory, noise level and stream size, the precision of classification is shown in Fig. 4.12 and Fig 4.13.

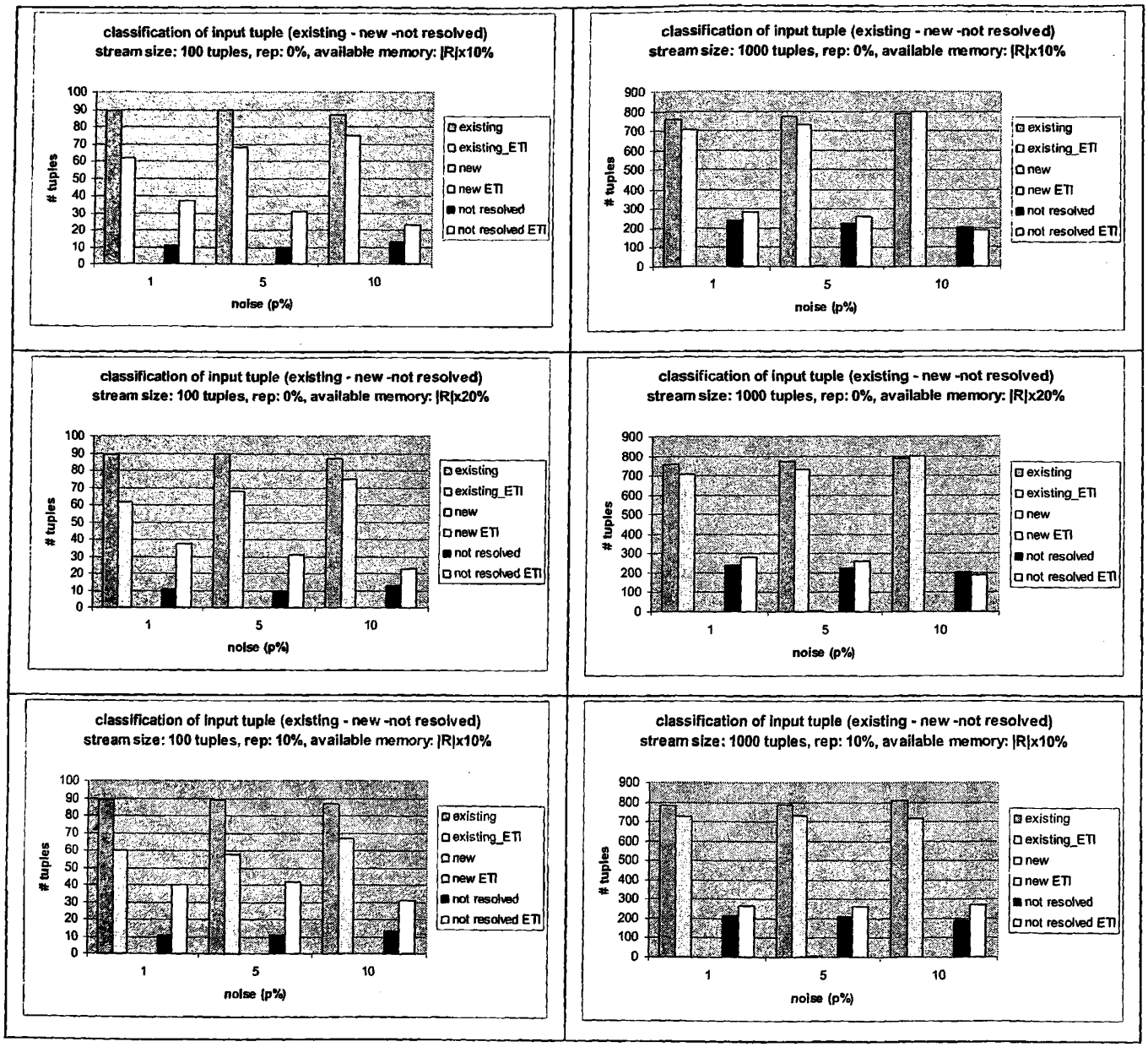


Fig. 4.12. Precision (|R|=10K tuples, , variant repetition - available memory)

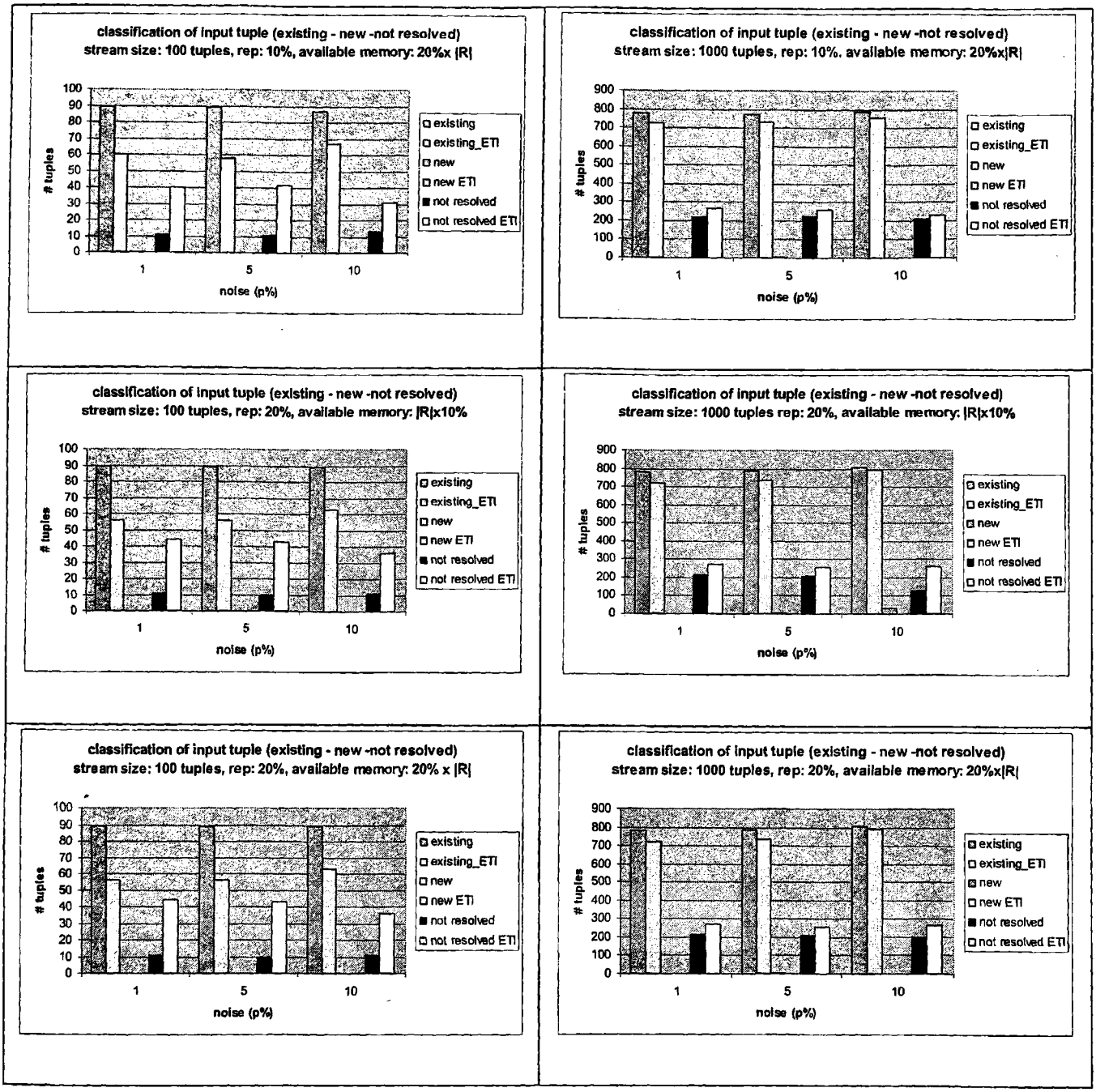


Fig. 4.13. Precision (|R|=10K tuples, , variant repetition - available memory)

For reference table size 100000 tuples, variant values for repetition of input tuples, available memory, noise level and stream size, the precision of classification is shown in Fig. 4.14.

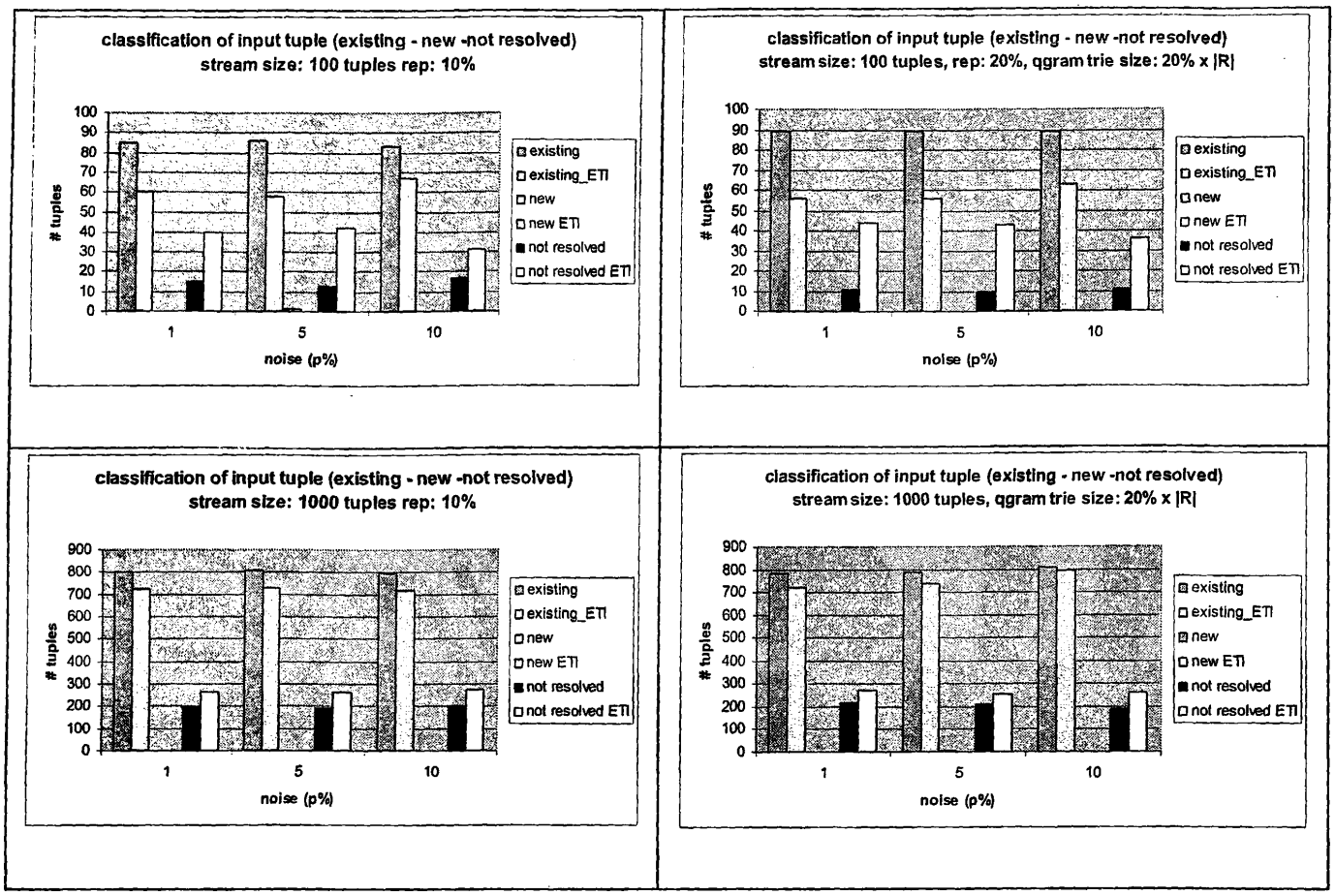


Fig. 4.14. Precision (|R|=100K tuples, , variant repetition - available memory)

We observe that our method outperforms the state-of-the-art method in the precision of classification for any reference table and input stream size (Fig. 4.12, Fig. 4.13 and Fig. 4.14). This occurs due to the fact that Word ETI holds more information about the reference tuples leading to more precise classification.

4.7. Memory Consumption

In this paragraph we present the memory consumption of our method. More specifically, we depict the main memory needed for the execution of specific experiments.

For reference table 100000 tuples and available memory 10% the maximum memory according to different noise levels is shown in Fig. 4.15.

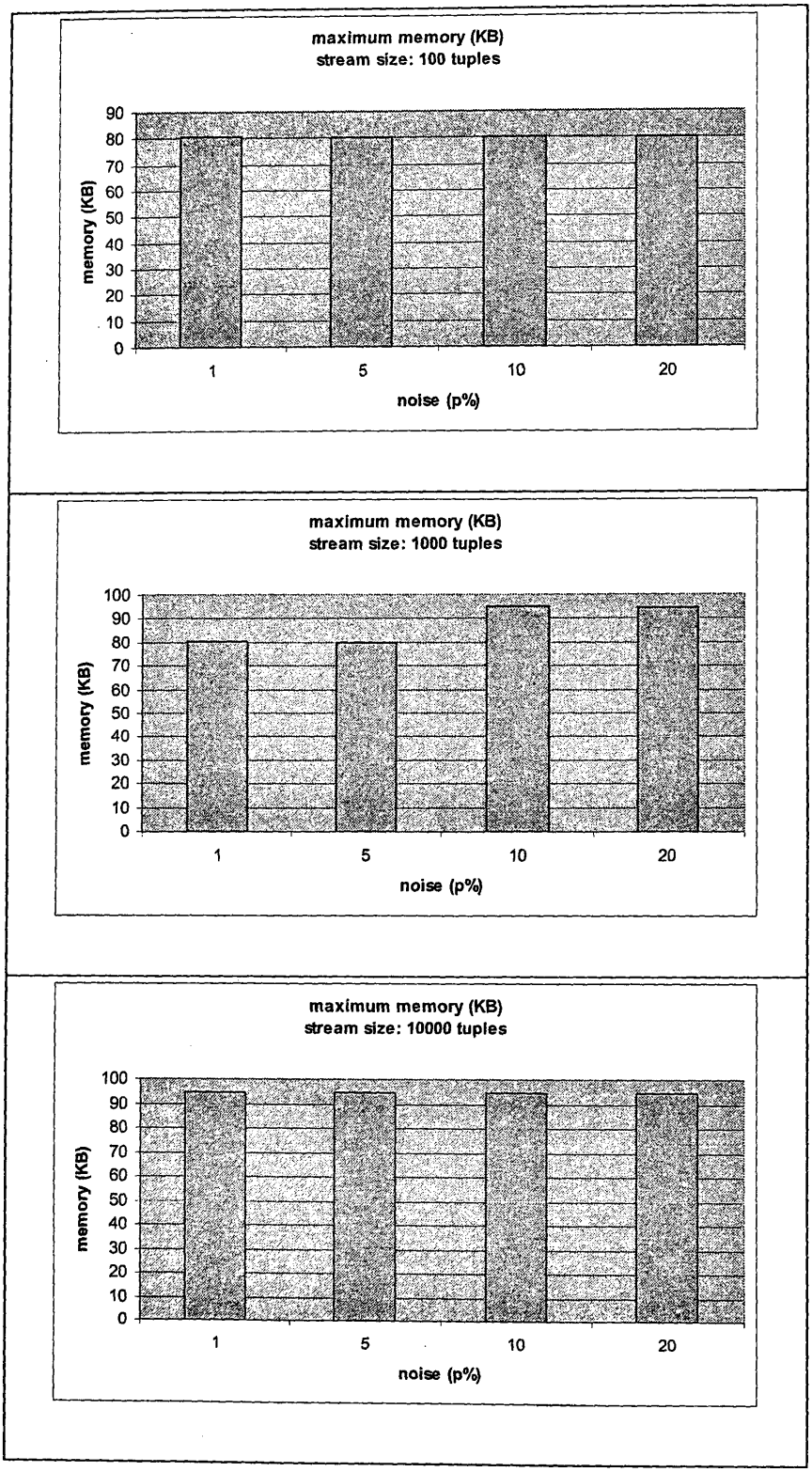


Fig. 4.15. Maximum Memory (|R|=100K tuples, repetition 10%, available memory 10%)

From the graphical representations it is obvious that the maximum memory consumption remains stable and is independent to noise. This occurs due to the pruning operation on qgram trie, which is applied for keeping its size fixed. For a better visualization of the memory consumption, we depict the memory amounts needed at runtime in Fig. 4.16.

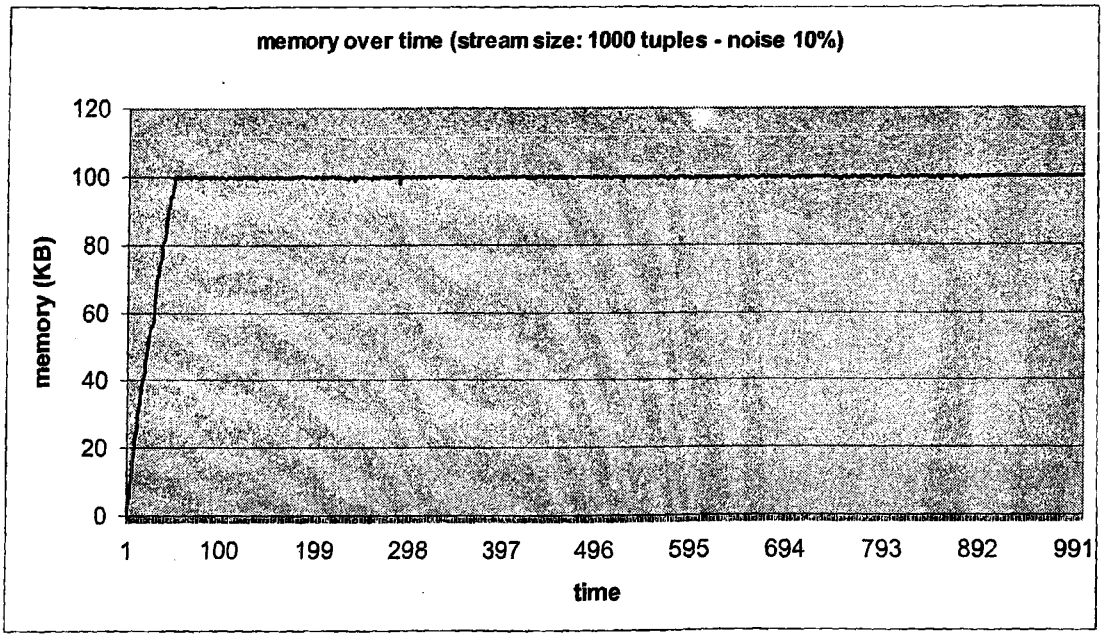


Fig. 4.16. Memory at Runtime (|R|=100K tuples, noise 10%, repetition 10%, available memory 10%)

CHAPTER 5. CONCLUSIONS

5.1. Conclusions – Summary

The problem we have dealt with was the approximate matching of reference data. Our approach is associated with the implementation of an effective method for on-line detecting similarity between input and reference records. More specifically, we have proposed on a cleaning procedure that classifies a stream of incoming tuples, before their insertion to a reference table, as existing or not existing reference tuples.

Our approach is based on a structure called *Word Index*, which is a table holding information about the attribute values stored in the reference table. This structure is used for the retrieval of reference tuples that probably match input tuples according to qgram similarity.

Moreover, we have proposed a trie structure called *Qgram Trie* that is maintained in main memory and is used for the caching of the frequently retrieved attribute values. This way, we avoid redundant I/O activities and accelerated the whole procedure. Additionally, we have applied the LRU algorithm as a replacement policy in case the size of trie exceeds a specific percentage of main memory. Using this replacement policy we assured that the size of trie was kept fixed and contained all the recently accessed attribute values.

Our experiments have indicated that:

- Our method outperforms the state-of-the-art method in precision for any noise level
- The precision of classification can be significantly improved using the Word ETI
- The execution time is improved when the streaming data contain frequent input tuples
- Our method slows down in the case of large streams, for main memory maintenance tasks
- The selection of appropriate similarity thresholds is crucial for the precision of classification in terms of misclassified tuples.

5.2. Future Work

As already mentioned in previous chapters, the main target of our method was to effectively classify a stream of input tuples before their insertion to a table holding valid tuples. We have implemented the qgram trie structure in order to cache frequent input tuples and avoid redundant I/O activities. An interesting topic for future work is the qgram trie space optimization. Specifically, it would be interesting to develop a new procedure for building the trie in main memory, in order to avoid the repetition of information and cache more valid tuples. The specific trie described in Section 3.3.2 encapsulates a set of frequent tid lists. However, since a trie node contains its own tid list, it is obvious that nodes belonging to the same branch hold the same information about reference tids. Therefore, a compression procedure can be implemented using appropriate algorithms for optimization of the overall size of the trie.

REFERENCES

[Baya98] Roberto J. Bayardo Jr. Efficiently Mining Long Patterns from Databases. In Proc. SIGMOD Conference 1998, pp. 85-93.

[BhGe04] Indrajit Bhattacharya, Lise Getoor. Iterative Record Linkage for Cleaning and Integration. In Proc. 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD 2004), pp. 11-18, Paris, France, June 13, 2004.

[CGGM03] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, Rajeev Motwani. Robust and Efficient Fuzzy Match for Online Data Cleaning. In Proc. 2003 ACM SIGMOD International Conference on Management of Data, pp. 313-324, San Diego, California, USA, June 9-12, 2003.

[Chau+05] Surajit Chaudhuri, Kris Ganjam, Venky Ganti, Rahul Kapoor, Vivek Narasayya, Theo Vassilakis. Data Cleaning in Microsoft SQL Server 2005. In Proc. ACM SIGMOD International Conference on Management of Data, pp. 918-920, Baltimore, Maryland, USA, June 14-16, 2005.

[ChCC02] Moses Charikar, Kevin Chen, Martin Farach-Colton. Finding Frequent Items in Data Streams. In Proc. Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, pp. 693-703, Malaga, Spain, July 8-13, 2002.

[ChGK06] Surajit Chaudhuri, Venkatesh Ganti, Raghav Kaushik: A Primitive Operator for Similarity Joins in Data Cleaning. ICDE 2006: 5

[Dai+06] Bing Tian Dai, Nick Koudas, Beng Chin Ooi, Divesh Srivastava, Suresh Venkatasubramanian. Column Heterogeneity as a Measure of Data Quality. CleanDB, 2006.

[DaMa05] Rajanish Dass, Ambuj Mahanti. Fast Frequent Pattern Mining in Real-Time. In Proc. 11th International Conference on Management of Data Advances in Data Management, pp. 156-167, Goa, India, January 6-8, 2005.

[FaPS96] Usama Fayyad, Gregory Piatetsky - Shapiro, Padhraic Smyth. The KDD Process for Extracting Useful Knowledge from Volumes of Data. Communications of the ACM, 39(11), pp. 27-34, Nov. 1996.

[Goet03] B. Goethals, "Survey on Frequent Pattern Mining", Manuscript, 2003. http://www.cs.helsinki.fi/ u/goethals/publications/survey.ps

[JiGr06] Nan Jiang, Le Gruenwald. Research issues in data stream association rule mining. SIGMOD Record, 35(1), pp. 14-19, Mar. 2006.

[KoKu06] Jia-Ling Koh, Yu-Ting Kung. An Efficient Approach for Mining Top-K Fault-Tolerant Repeating Patterns. In Proc. 11th International Conference on Database Systems for Advanced Applications (DASFAA), pp. 95-110, Singapore, April 12-15, 2006.

[LaLN03] Laks V. S. Lakshmanan, Carson Kai-Sang Leung, Raymond T. Ng. Efficient dynamic mining of constrained frequent sets. ACM Transactions on Database Systems (TODS), 28(4), pp. 337-389, Dec. 2003.

[LeLN02] Carson Kai-Sang Leung, Laks V. S. Lakshmanan, Raymond T. Ng: Exploiting succinct constraints using FP-trees. SIGKDD Explorations, 4(1), pp. 40-49, Jun. 2002.

[Leun04] Carson Kai-Sang Leung. Dynamic FP-Tree Based Mining of Frequent Patterns Satisfying Succinct Constraints. In Proc. 1st International Symposium on Applications of Constraint Databases (CDB'04), pp.117-132, Paris, June 12-13, 2004.

[LoLL01] Wai Lup Low, Mong-Li Lee, Tok Wang Ling. A knowledge-based approach for duplicate elimination in data cleaning. Information Systems, 26(8), pp. 585-606, Dec. 2001.

[PeHa02] Jian Pei, Jiawei Han. Constrained frequent pattern mining: a pattern-growth view. SIGKDD Explorations, 4(1), pp. 31-39, Jun. 2002.

[SeMa04] Jouni K. Seppänen, Heikki Mannila. Dense itemsets. In Proc. 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 683-688, Seattle, Washington, USA, August 22-25, 2004.

[SuLS02] Sam Yuan Sung, Zhao Li, Sun Peng. A fast filtering scheme for large database cleansing. In Proc. 2002 ACM CIKM International Conference on Information and Knowledge Management, pp. 76-83, McLean, VA, USA, November 4-9, 2002.

[TsCh04] Pauray S. M. Tsai, Chien-Ming Chen. Mining interesting association rules from customer databases and transaction databases. Information Systems, 29(8), pp. 685-696, Dec. 2004.

[USCB07] Frequently Occurring First Names and Surnames From the 1990 Census. U.S. Census Bureau. Available at http://www.census.gov/ genealogy/names/

[XHYC05] Dong Xin, Jiawei Han, Xifeng Yan, Hong Cheng. Mining Compressed Frequent-Pattern Sets. In Proc. 31st International Conference on Very Large Data Bases, pp. 709-720, Trondheim, Norway, August 30 - September 2, 2005.

[YaFB01] Cheng Yang, Usama M. Fayyad, Paul S. Bradley. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In Proc. 7th ACM SIGKDD International Conference on Knowledge discovery and Data Mining, pp. 194-203, San Francisco, CA, USA August 26-29, 2001.

[YWYH02] Jiong Yang, Wei Wang, Philip S. Yu, Jiawei Han. Mining Long Sequential Patterns in a Noisy Environment. In Proc. 2002 ACM SIGMOD International Conference on Management of Data, pp. 406-417, Madison, Wisconsin, June 3-6, 2002.

[ZhWC03] Xingquan Zhu, Xindong Wu, Qijun Chen, Bridging Local and Global Data Cleansing: Identifying Class Noise in Large, Distributed Data Datasets. In Proc. 20th International Conference Machine Learning, pp. 920–927, Washington D.C., USA, 2003.

SHORT CV

Ioannis Krommydas was born in Ioannina in 1982 and finished high school in 2000. He obtained his B.Sc. in Computer Science in 2004 from the Computer Science Department of the University of Ioannina. Ioannis Krommydas has enrolled in the Graduate Program of the Computer Science Department of the University of Ioannina as an M.Sc. candidate in the academic year 2004 - 2005. His research interests are Databases, Data Cleaning and Data Mining.