Liu XL, Wang HZ, Li JZ *et al.* EntityManager: Managing dirty data based on entity resolution. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(3): 644–662 May 2017. DOI 10.1007/s11390-017-1731-1

# EntityManager: Managing Dirty Data Based on Entity Resolution

Xue-Li Liu, Hong-Zhi Wang*, *Member, CCF*, Jian-Zhong Li, *Fellow, CCF*, and Hong Gao, *Senior Member, CCF*

*Massive Data Computing Laboratory, Harbin Institute of Technology, Harbin 150001, China*

E-mail: xueli.hit@gmail.com, {wangzh, lijzh, honggao}@hit.edu.cn

Received February 29, 2016; revised January 10, 2017.

**Abstract** Data quality is important in many data-driven applications, such as decision making, data analysis, and data mining. Recent studies focus on data cleaning techniques by deleting or repairing the dirty data, which may cause information loss and bring new inconsistencies. To avoid these problems, we propose EntityManager, a general system to manage dirty data without data cleaning. This system takes real-world entity as the basic storage unit and retrieves query results according to the quality requirement of users. The system is able to handle all kinds of inconsistencies recognized by entity resolution. We elaborate the EntityManager system, covering its architecture, data model, and query processing techniques. To process queries efficiently, our system adopts novel indices, similarity operator and query optimization techniques. Finally, we verify the efficiency and effectiveness of this system and present future research challenges.

**Keywords** dirty data, entity resolution, uncertain attribute, query processing, query optimization

## 1 Introduction

Dirty data exists in many systems. It can be caused by information integration, error input, incompleteness, and imprecision. Due to the frequent appearance of dirty data, many research efforts have been devoted to analyzing and processing the dirty data[1-3].

In dirty data management, a fundamental problem is to handle data with conflicts[4-6]. Here a conflict datum refers to different representations of the same real-world object, which is viewed as the intrinsic part of dirty data. For example, different names in different data resources may refer to the same person. Managing and querying conflict data is important in dirty data analysis, and it is also a cornerstone of many real-world applications, such as entity resolution[7], truth discovery[8], and data recovery[9].

Due to its importance, many efforts have been made on conflict data management[4-6]. The majority of these studies are based on data cleaning, which is designed for the conflict data to smooth the inconsistency. After data cleaning, the inconsistencies are eliminated and a single and unified representation of each real-world object is created. Then existing algorithms can be directly applied to the cleaned data to generate results for any user-specified queries. However, these data cleaning methods only produce a representative value of the conflict data, but not preserve the heterogeneous information within the conflict data, which may be more useful in some applications. For example, in trade analysis[10], outdated information is valuable in predicting future data. Thus, it is more desirable to design a system which cannot only losslessly maintain the information within the conflict data but also support general queries efficiently.

In this paper, we design a new system called EntityManager. We regard the entity as the building block in our system, while the tuple is regarded as the basic data unit in previous studies. Here each entity in the data corresponds to a unique object in the application in the real world. In our system, we organize all the entity tuples as different entities, each of which contains

Regular Paper

This work was partially supported by the National Key Technology Research and Development Program of the Ministry of Science and Technology of China under Grant No. 2015BAH10F01, the National Natural Science Foundation of China under Grant Nos. U1509216, 61472099, and 61133002, the Scientific Research Foundation for the Returned Overseas Chinese Scholars of Heilongjiang Province of China under Grant No. LC2016026, and the Ministry of Education (MOE)-Microsoft Key Laboratory of Natural Language Processing and Speech, Harbin Institute of Technology.

*Corresponding Author

a number of conflict representations. Then, the queries are performed on an entity-wise manner, that is, by comparing the constraint of the queries on each entity. We can benefit a lot from utilizing the entity as the basic data unit in organizing data. On one hand, conflicts among data are preserved and resolved within each entity. On the other hand, queries performed on entities can tolerate much more errors on the constraint of the queries, i.e., the query with inaccurate constraints is more likely to obtain accurate results within a whole entity, but not within each individual tuple.

The conflicts in the same attribute of a tuple could be regarded as uncertain, but not mutually exclusive, i.e., each descriptive form of a conflict uncertain datum is meaningful in the real world. Each of the conflict values can represent a possible entity attribute value. Thus, we describe multiple values as uncertain values. Even though many studies using probabilistic model have been proposed to represent uncertain data, such models cannot solve the uncertainty problem caused by conflicts. Indeed, the conflict uncertain data cannot be represented by the possible world model, which is not designed to manage the uncertainty led by conflicts from entity resolution. The possible world model is based on the concept of "possible world"[1], which is not suitable to define data with conflicts.

Instead of using "possible world", we develop the EntityManager system by using the entity as the basic unit and using conflict attribute values as an uncertain attribute value. In our system, the entity resolution technique is performed on the datasets and the tuples corresponding to the same real-world entity are merged as an uncertain entity tuple. For example, Table 1 contains the information of three individuals. By using the entity resolution technique, the relation is represented as Table 2, which means the three individuals are recognized as the same person. Since the attributes NAME, CITY, and PNO have conflicts in their values, such attribute contains an uncertain value with two possible values and their probabilities.

Due to the uncertainty in the attributes and possible errors in query constraint, the queries in our system have different semantics from the queries in traditional databases. EntityMangager modifies common SQL statements as its query statements, and it returns results with probability value representing the extent to what degree this entity matches the query. With such possibilities, users could judge whether the results should be accepted.

Due to the uncertainty in the attributes, the query processing techniques and query optimization techniques in EntityManager are different from those in traditional methods. On one hand, by incorporating the uncertainty of each attribute, the indices in our system can group similar values together. On the other hand, the similarity query processing techniques incorporate the quality of the attributes values and take each entity as a processing unit. Consequently, the query results are organized according to referred entities. Also, our system adopts query optimization techniques based on the novel estimation methods, which estimate the selectivity on multi-value attributes with uncertainty attached to each value.

The contributions of this paper include three parts. Firstly, we propose a novel data model: the entity model to organize tuples. This model introduces uncertainty as the representation of different descriptions of one entity attribute. Secondly, we provide an overview of query processing techniques and query optimization techniques especially for the proposed entity model. Finally, we test the efficiency of query processing and optimization techniques in our system. We also test the

**Table 1**. Original Relation

| ID | NAME | CITY | ZIPCODE | PNO (Phone Number) | SALARY |
|---|---|---|---|---|---|
| 1 | John Smith | Beijing | 100010 | 010-80325789 | 4k |
| 2 | John Smith | BJ | 100010 | 010-80103389 | 5k |
| 3 | John Smith | Beijing | 100010 | 010-80325789 | 6k |

**Table 2**. Entity Relation: PERSON

| EID | NAME | CITY | ZIPCODE | PNO | SALARY |
|---|---|---|---|---|---|
| 1 | (John Smith, 0.67) | (Beijing, 0.67) | (100010, 1.0) | (010-80325789, 0.67) | (4k, 0.33) |
| | (J. Smith, 0.33) | (BJ, 0.33) | | (010-80103389, 0.33) | (5k, 0.33) |
| | | | | | (6k, 0.33) |

Note: EID: Entity ID.

effectiveness of our system using case study and usability study.

The remaining parts of this paper are organized as follows. Section 2 introduces related work. Section 3 describes the structure of the EntityManager system. Section 4 provides the data model in our system. Section 5 discusses the query processing techniques. In Section 6, the system performance is verified by experiments. Further challenges and work are discussed in Section 7. Finally, Section 8 concludes this paper.

## 2 Related Work

We divide the related work into three categories: dirty data management, similarity-based query operators, and query estimation technique.

### 2.1 Dirty Data Management

Existing studies on processing dirty data fall into three categories. The first category is data cleaning[11], which is to detect and remove errors to improve data quality. However, data cleaning cannot clean the dirty data thoroughly and excessive data cleaning may lead to the loss of information. Besides, existing data cleaning techniques are generally time-consuming. Especially when massive data updates frequently, the frequent data cleaning operation will greatly reduce the efficiency. To avoid data loss, the algorithms in the second category perform queries directly on dirty data and obtain query results with a clean degree[1-3]. While these methods are suitable for special cases, they cannot handle general cases. Third category includes several general models of dirty data management without data cleaning[4-6]. Most of these models only consider the uncertainty in the attribute value, while the quality degree of each value of uncertain attributes and the relationship of the attributes are ignored.

In this paper, we focus on entity-based relational database model in which one entity tuple refers to an object in the real world. This model reflects real-world objects and preserves the relationship of real attribute values.

### 2.2 Similarity-Based Query Operators

*Similarity Search.* A natural way for string similarity search is to build an appropriate index structure for strings. Currently, the most popular index structure for similarity string matching is the inverted table, which splits strings into grams and measures strings by edit distance metric[12-18]. Even though gram-based method can process similarity string matching efficiently in many cases, it has some weaknesses. Firstly, it cannot deal with data update effectively. Secondly, the inverted tables may introduce many collection operations, which increase the complexity of the query.

There are still a lot of non-inverted list index structures supporting the similarity string search. For example, Zhang *et al.*[19] proposed an edit distance tree structure which hashes each string into a number and inserts it into a B+ tree structure. This method can support data update well, but it imposes too much on the ordering of the string and cannot pay full attention to the similarity of strings. Therefore, a large number of similar strings cannot be allocated in the same leaf node, which results in defective filtering.

In our system, entity similarity search is closely related to the similarity search. Similarity search is to find a string set in which each string is similar to the query string, and entity similarity search is to return the entities in which their attribute is similar to the query string. Although entity similarity search and similarity search have the same query form, their query semantics have some differences. The entity similarity search needs to consider the quality of each attribute value. It means that even if there is one string in the attribute similar to the query string, we cannot ensure that the entity is in the result set since we must consider the influence of the quality of each attribute value. Existing similarity technology can be applied to our system, but the efficiency would be poor. Therefore, our system adopts a new technique for entity similarity search.

*Similarity Join.* Similarity join is an essential operation in many applications such as data integration, data cleaning and fraud detection. There has been a lot of work on similarity join in the academic and the industrial community. In the academic community, existing studies usually employ a filter-and-refine framework such as Part-Enum[12], All-Pairs-ED[20], Ed-join[17], PPjoin[18] and so on. In the filter process, they generate a signature set for each string, build an inverted index for each signature, and then compare the signatures to filter the unsimilar string pairs. The most common signature is $q$-gram. In the refining step, they compute the similarity value of string pairs in the candidate set produced in the filter process and then return the final results. However, they are inefficient when the string length is not larger than 30[21]. Also, the index and candidates to be verified may be oversize. To address this problem, Wang *et al.*[21] proposed a trie-

based method. This method builds a prefix index for string sets based on the idea that strings often share the common prefix. This method uses string pairwise join, while our EntityManager uses fuzzy set similarity join. Therefore their method is not suitable for our system.

Current studies of the similarity join based on set usually use Jaccard distance as similarity measure function[12,22-23] and adopt the filter-and-verify framework. The framework takes items in the set as signatures, and then uses filtering measures to generate candidate sets. It is not suitable for our system because it assumes the exact match between the two sets while our system uses the fuzzy match.

To improve the efficiency of similarity join operator, the method in [24] changes the fixed size gram to changed length gram, and the method in [25] optimizes the fixed prefix length and chooses the best prefix length for each string.

Different from the existing similarity join, the entity similarity join is the fuzzy match between two weighted string sets. It is more complicated than the similarity join for a string pair and a string set pair.

### 2.3 Query Estimation Technique

Estimation of the size of the results of a query operator is crucial in the query optimization process. There has been a lot of work on query estimation for traditional relational database management systems. Most approaches are based on histogram[26], which records the distribution of data. Here, a histogram on an attribute $A$ can be constructed by partitioning the data distribution of $A$ into buckets and approximating the frequencies of values in each bucket in some common fashion.

Another method is based on sampling. The sampling methods avoid scanning all of the data. These methods sample partial tuples from the database, and then estimate query size based on the result obtained from samples. The classic sampling methods include simple random sampling[27-29], systematic sampling[30], and LSH (locality-sensitive hashing) sampling[31]. In the simple random sampling, each tuple is selected with equal probability. Systematic sampling divides the dataset into $K$ intervals, and then takes a fixed number of samples from each interval. It requires that the dataset has been ordered already. Lee *et al.*[31] proposed LSH sampling, which is the first sampling method to solve the estimation size of set similarity join. This method first aggregates the similarity sets into some clusters, and then samples from clusters to estimate the result size. However, it cannot be applied directly in the EntityManager system because the element in the entity set has a quality degree.

## 3 System Overview

The goal of our system is to manage dirty data without cleaning it and retrieve query results according to the quality requirements. To achieve this goal, we study various techniques of dirty data management, such as data model, indexing, query optimization and query processing.

In this section, we provide an overview of the EntityManager system. More details on both its data model and query processing implementation techniques will be discussed in Section 4 and Section 5, respectively.

### 3.1 Data Model and Query Model

To avoid cleaning dirty data, data storage and query processing in EntityManager are based on entity data model. The entity data model takes entity as the basic storage unit. It roots in entity resolution technique and further gathers various descriptions referring to the same real-world entity in the results of entity resolution into one entity tuple. When a query comes, the system returns the entities matching the query as the results. The query is different from traditional queries, though the form is similar to SQL. Considering that users need to get query answers satisfying a quality requirement, we introduce a similarity query model. This similarity query model also has other advantages. For example, it supports fuzzy query when the user cannot describe the constraint exactly. Also, when the dirty data has errors such as spelling mistakes, it can meet user requirements when the query constraint is relaxed.

### 3.2 Query Processing

We developed the following three new techniques for processing query efficiently on our entity database.

• *Similarity-Based Operators.* With the selection and the join operators different from the traditional relational database, an entity similarity search algorithm[32] and an entity similarity join algorithm[33] for the EntityManager system were developed.

• *Indices.* To process entity similarity search and join efficiently, Fgram-Tree[32] and bi-layer prefix[33] indices were designed for entity similarity search and entity similarity join, respectively. The Fgram-Tree index

handles similarity weight string search, and the bi-layer prefix index handles both the string similarity and the string set similarity.

• *Query Optimization.* For new operators in our system, Zhang *et al.* designed novel result estimation algorithms for the entity similarity search and join operators in [34]. Additionally, since the selectivity of join operation on multiple relations is difficult to estimate, we proposed a greedy algorithm for selectivity estimation and a join order selection algorithm based on the selectivity[35].

### 3.3 Architecture

The framework of query processing in the EntityManager system is similar to that of traditional relational databases[36]. As depicted in Fig.1, the framework includes a data indexing module, syntax parsing module, query executor module and query optimization module. However, the EntityManager system has a different module: entity resolution module. The input of this module is dirty data, and through the entity resolution algorithm, this module organizes tuples into entities. The entities are the units of query processing in the EntityManager system.

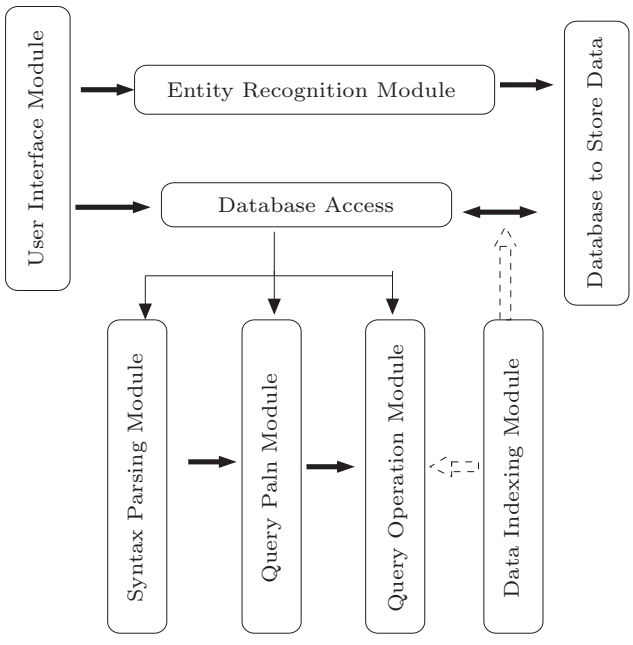



Fig.1. EntityManager architecture.

## 4 Data Model

In this section, we introduce the data model in our system and the semantics of the queries. In the data model of our system, the definitions of database and relation are similar to those in traditional database[37], while the definitions of attributes and tuples are different.

Since the data model is different from the traditional relational model, the query in our system has different semantics compared with the traditional relational database. On one hand, the comparison between attribute values should consider the uncertainty of each value. On the other hand, the comparison between attributes and query constraints is an approximate comparison rather than the accurate comparison in the traditional relational database. We use the following example to show these differences.

*Example* 1. To select the telephone number of person "John Smith", the selection constraint is NAME $\approx$ "John Smith". From the entities in Table 2, the attribute NAME of $e1$ has a probability of $\frac{2}{3}$ to match the attribute perfectly. Suppose that the threshold is set to 0.3, we get the results set {(010-80325789, 0.67), (010-80103389, 0.33)}.

### 4.1 Entity-Based Data Model

Based on above description, firstly we define the uncertain attribute value in Definition 1. An uncertain attribute value contains not only the possible values, but also its corresponding possibility. Then we give the definition of entity in Definition 2. The entity is the basic unit in the entity-based relational database system, containing a set of uncertain attributes.

**Definition 1** (Uncertain Attribute). *An uncertain attribute is a pair* $(attr, A)$*, where attr is the name of the attribute and* $A = \{(v, p)|v$ *is a possible value of the attribute and* $p$ *is the possibility of the value* $v\}$ *is the set of possible values of attribute attr. A is called an attribute value set.*

Clearly, the sum of possibilities of all possible values for an uncertain attribute value is 1.

**Definition 2** (Entity). *An entity is a pair* $E = (K, A)$*, where A is a set of uncertain attributes and K is a set of keys that is to identify the entity uniformly (e.g., entity-ID).*

Since we introduce the possibility of the values of uncertain attributes, we need to define a new concept to judge whether a tuple satisfies a query, and we call this concept "similarity". We define the similarity for numeric type attribute and string type attribute.

**Definition 3** (Similarity for Numeric Type Attribute). *For an uncertain attribute* $(a, V)$ *and an atom*

*constraint $C$ in form of $a@v$ where @ is a predicate symbol (e.g., $>, <, \cdots$) and $v$ is a constraint, the similarity between them is defined as follows*:

$$sim(V@C) = \sum_{(v_i,p_i)\in V} sim(v_i@v) \times p_i,$$

*where*

$$sim(v_i@v) = \begin{cases} 1, \ if \ v_i@v, \\ 0, \ otherwise. \end{cases}$$

For a selection query with a constraint $a < v$, the similarity between one tuple and a query can be calculated according to Definition 3. We use an example to illustrate it.

*Example* 2. Let $t$ be a tuple with value ((20, 0.1), (25, 0.4), (27, 0.4), (36, 0.1)) in attribute $a$. For two given queries, $Q1$: $a < 22$ and $Q2$: $a < 28$, the similarity between this tuple and query $Q1$ is $1\times0.1+0\times0.4+0\times0.4+0\times0.1 = 0.1$, and the similarity between this tuple and query $Q2$ is $1\times0.1+1\times0.4+1\times0.4+0\times0.1 = 0.9$.

**Definition 4** (Similarity for String Type Attribute). *For uncertain attributes $(a, V_1)$ and $(a, V_2)$, and the edit distance threshold $\tau$, the similarity between $(a, V_1)$ and $(a, V_2)$ is defined as follows:*

$$sim(V@C) = \sum_{(v_i\in V_1, v_j\in V_2)\&d(v_i,v_j)\leqslant\tau} sim(v_i@v_j) \times p_i,$$

*where*

$$sim(v_i, v_j) = 1 - \frac{d(v_i, v_j)}{\max\{|v_i|, |v_j|\}}.$$

*Here $d(v_i, v_j)$ is the edit distance of string $v_i$ and $v_j$.*

With these definitions, we define some query operators.

### 4.2 Queries

As discussed in Section 3, our query model is based on similarity; hence the query language in our system has different semantics with traditional query languages. To facilitate users to use this system, we modify traditional SQL query language to get our query language: EQL (entity query language). In this subsection, we use examples to illustrate key features of the EQL language. The example queries are to be performed on the tuples in Table 2.

#### *4.2.1 Basic Queries*

$Q_1$. Simple EQL query: find the phone numbers of person named "John Smith" and the quality degree is 0.9.

```
SELECT    PNO
FROM      PERSON
WHERE     PAERSON.NAME ≈ "John Smith"
SATISFY   THRESHOLD = 0.9
```

Note that EQL uses SATISFY to denote the possibility requirement of the results defined by users. Here "≈" denotes the entity similarity search operator for string type attributes, which will be defined in Subsection 4.3. It expresses that the search is a similarity search.

$Q_2$. Similarity grouping and function in EQL: unlike group operator in SQL, the group in EQL is similarity group. It means that the similar value will be taken as one group to process. For example, if a user wants to know which city has a higher average salary, we use the SGROUP BY expression.

```
SELECT    PAERSON.CITY, SALARY
FROM      PAERSON
SGROUP    BY PERSON.CITY
```

In our example, we note that the grouping adopts similarity measures. It means that "Beijing" and "BJ" will be included in one group because they indicate the same city.

With uncertain entity attributes in our system, we compute new functions, namely SSUM, SVAG, SMIN, SMAX to replace corresponding functions SUM, VAG, MIN, and MAX in SQL. The differences between the new functions and functions in SQL lie in that the new functions return the result with its possibility. The COUNT function in EQL is the same with the COUNT function in SQL.

$Q_3$. Similarity join in EQL: suppose we have another table CAROWNER whose schema is (NAME, CAR) with NAME as the person name and CAR as the car that the person owns. If a user wants to know the number of cars registered in the same city, the query involves aggregation and similarity join. Assume that the quality degree is defined to be 0.9.

```
SELECT    CAROWNER.CAR, COUNT(*)
FROM      PERSON, CAROWNER
WHERE     PAERSON.NAME ≈ CAROWNER.NAME
SATISFY   THRESHOLD = 0.9
SGROUP    BY PAERSON.CITY
```

If the join is not the similarity equi-join and join attribute is numeral form, we take the expected value as the comparing item.

#### 4.2.2 Update Queries

Updating is a basic operation in the database system. The updating in EQL has some differences from the update, insert, delete operations in SQL. For the update operator in EQL, the condition is "similarity" rather than "equality". The "similarity" is measured by a similar function with a similarity threshold. In our system, the similar function can be defined based on edit distance, Jaccard distance or Hamming distance for the string type. For the numeric type, we still use exact match, but we also take into account the quality of the numeric attribute value. Here the similarity threshold is assigned by the system. For example, suppose there is an update query of "update the salary of John Smith to 8k" for Table 2. In the executing process, the EntityManger system first finds out the tuples meeting NAME condition to obtain $e_1$ tuple, and then add 8k to SALARY attribute values in $e_1$ (note that we should recompute the possibility for each value). For the insert operator, EntityManager supports inserting new entity directly, which means in the insert clause, the user must give all the uncertain values and their corresponding quality degrees. For the delete clause, we first find entities matching the delete condition with the similarity search operation, and then delete the entity tuple directly.

### 4.3 Operators of the Entity Data Model

In this subsection, we describe basic query operators in the EntityManager system. The semantics of group, project and update operator adopts similarity matching. The definition of "similarity" is the same with that of the similarity in the select operator. Therefore this subsection focuses on the search and join operators which are redefined with new semantics in terms of the similarity between the query constraint and multi-value attributes.

#### 4.3.1 Selection Operator

As shown in example 1, the query results are organized in an entity table and each entity in the results satisfies the query with a similarity value. Since a result with a low similarity is generally less interesting to the user than value answers with higher similarities, we consider those results with a similarity value less than a threshold $\tau$ (this parameter can be a default value in the system or provided by the user) as unsatisfied for a query. Therefore, the results of queries should be those answers that have a similarity exceeding the threshold $\tau$. Therefore a query for numeric type attribute with constraint $a <_\tau x$ can be processed by the operator defined as follows:

$sim(a < x) > \tau \Longleftrightarrow \sum_{(v_i,p_i)\in V} sim(v_i < x) \times p_i > \tau$.

In example 2, if the similarity threshold is fixed to 0.5, the tuple does not satisfy query $Q1$ but satisfies query $Q2$.

For a query for string types, the attribute with constraint $a \approx_\tau x$ can be processed by the operator defined as follows:

$sim(a \approx x) > \tau \Longleftrightarrow \sum_{(v_i,p_i)\in V} sim(v_i, x) \times p_i > \tau$.

#### 4.3.2 Top-k Entity Similarity Join Operator

Similarity join between two sets of tuples returns pairs of tuples satisfying that the similarity between two tuples in each pair is above a given threshold[17]. Currently, many methods to measure the similarity are proposed[18], such as edit distance, Hamming distance, Jacquard similarity, cosine similarity for strings, and the exact match for the numeric type. All of them could be applied in our system. We use function *sim* to compute the similarity value. Considering that query results have a corresponding quality degree and users want to get a high quality answer, we define the similarity join operator from two aspects: one is top-$k$ entity similarity join, and the other is threshold-based entity similarity join.

Top-$k$ search has wide applications in many systems such as the web, the multimedia search, and the traditional relation database[38]. Since all possible values have a possibility in the entity database, the results of similarity join will also have a probability representing the quality of the result. To find the join results with a high probability in the proper size, the top-$k$ operator is in demand in the EntityManager system, where the answers ranked by the quality and users specify parameter $k$. We define the top-$k$ entity similarity join as follows.

**Definition 5** (Top-$k$ Entity Similarity Join). *Given two uncertain attribute value sets $R$, $S$, the numeric value $k$ and the edit similarity threshold $\tau$, for each $(r,s) \in R \times S$, the rank value $rv(r,s) = \sum sim(v_{r_i}, v_{s_j}) \times p_{r_i} \times p_{s_j}$ where $p_{r_i}$ is the possibility of the $i$-th possible value in attribute value $r$, and the top-$k$ entity similarity join returns all attribute value pairs $D$ satisfying*: 1) $|D| \leqslant k$, 2) *for each* $(r,s) \in D$, $(r', s') \notin D$, $rv(r,s) \geqslant rv(r', s')$.

There are some practical applications that users are only interested in results with a relatively high proba-

bility and the results with lower possibilities will be ignored. Hence we consider another similarity join whose result reaches a probability threshold, which is defined as follows.

**Definition 6** (Threshold Entity Similarity Join). *Given entity tables $R$, $T$, entity join attribute $S$, possibility threshold $\theta$, and similarity threshold $\tau$, the entity similarity join operator on $S$ of $R$ and $T$ returns the pairs satisfying*: $\{(r,s)|\sum sim(v_{r_i},v_{s_j})\times p_{r_i}\times p_{s_j} \geqslant \theta\}$, *where $p_{r_i}$ is the possibility of the $i$-th possible value in attribute value $r$. For convenience, $sim(v_{r_i},v_{s_j})\times p_{r_i}\times p_{s_j}$ is called as quality affection of string pairs $v_{r_i},v_{s_j}$ to entity pair $(r,s)$.*

## 5 Query Processing Techniques of EntityManager

In this section, we introduce the query processing techniques in EntityManager. We first introduce the query execution techniques in Subsection 5.1, and then introduce the query optimization techniques in Subsection 5.2.

### 5.1 Query Execution in EntityManager

The major part of the query execution in EntityManager is the implementation algorithms of the query operators, which include the entity similarity search, the entity similarity join, the update, and the group. Section 4 has introduced the executing process for the update operator. Since the group operator adopts the similarity match for group condition, we adopt the self threshold entity similarity join method to aggregate similar uncertain attribute values into one group. There are also some types of query especially for numeric type attributes, such as the range query and the non-equal entity similarity join. For these queries, the system first computes the expectation value for each attribute value, and then compares the expectation value with the query constraint to obtain the results. Therefore, we focus on the entity similarity search and the entity similarity join in this subsection.

#### *5.1.1 Similarity Search*

Both numeric type and string type are important in the real applications. For the numeric type attribute, given a numeric threshold $\tau$, firstly we need to compute the similarity between the uncertain attribute value and the search value. If the similarity is no less than $\tau$, we add the numeric value into the result set. Here, the threshold $\tau$ (ranging from 0 to 1) is specified by the system or an expert. To improve query efficiency, EntityManager builds B-tree index for frequently-used attributes in queries.

For string type, given a query string $t$, and an entity set $S$, string similarity search returns all entities in $S$ which are similar to $t$. The results of the entity similarity search can be got through the results of similarity search by verifying the probabilities of similar strings. Therefore we focus on improving the efficiency of the current similarity search techniques. Nowadays, indexing is an efficient technique to implement similarity string search. Existing index techniques often construct a prefix tree to index each string or split a string into a feature gram set, and then build an inverted table for each $q$-gram appearing in the string sets. However, both of them cannot effectively hash similar strings to the same index items and the dissimilar items to different index items. This weakness makes current index inefficient for similarity search.

Thus, Tong and Wang proposed a new string index, Fgram-Tree[32], which filters strings based on $q$-gram feature. Fgram-Tree groups similar strings into the same index node and locates dissimilar strings into different nodes. Hence it supports data updating and greatly improves the efficiency of the entity similarity query. Fgram-Tree can also be applied to different types of search such as top-$k$ and threshold entity similarity search.

We start with the definition of the structure of Fgram-Tree.

**Definition 7** (Fgram-Tree)[32]. *An Fgram-Tree is a tree structure, where each leaf node is represented as a triple $(bs,cbs,ids)$, and each intermediate node is represented as a tuple $(bs,cbs)$. Here $bs$ is a tuple set, and each tuple consists of two elements: a $q$-gram in strings of ids and its occurrence frequency; $cbs$ is the center of $bs$, storing the $q$-gram sets whose frequency is no less than the threshold $\tau$; $ids$ is the set of the strings attached to the node.*

Example 3 explains the specific structure of Fgram-Tree.

*Example* 3. Assume that we have strings {(0, Joey), (1, Joel), (2, Janson), (3, Janet), (4, Joe)} and the threshold $\tau = 2$. The general index structure is shown in Fig.2. Similar strings $s_0$, $s_1$, $s_4$ are stored in LNode1, and strings $s_2$ and $s_3$ are stored in LNode2.

Based on the Fgram-Tree index, two filter conditions used in the pruning process are proposed and are necessary conditions for similar strings. The first filter-

ing condition concerns on the common gram ($q$-gram is the gram with $q$ characters) set of the query string $t$ and the $bs$ of node $c$. If the cardinality of their common gram set is less than the threshold $\tau$, any string belonging to node $c$ must not be similar to $t$. Here the threshold can be obtained by existing count filtering method[17]. The second filtering condition is about the common grams of query string $t$ and the $cbs$ of node $c$. If $t$ belongs to the cluster node, it must have common gram with $cbs$.

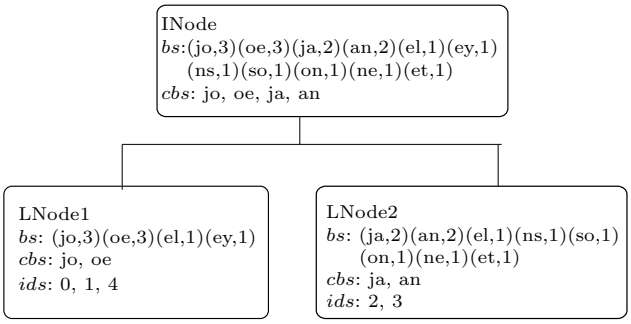



Fig.2. Fgram-Tree structure.

For threshold-based search based on Fgram-Tree, we use these two conditions to prune leaf nodes. Then, each string $s$ in the obtained leaf nodes needs to be verified whether $s$ satisfies the constraint.

For top-$k$ search method, first we find the leaf nodes satisfying constraint $t$ by setting the threshold to 0, and the $k$ most similar strings attached in one leaf node are chosen as the initial results. Then we visit other leaf nodes and update the results as follows: assume that string $s$ in the initial results has the smallest similarity with the query string, if the similarity value of the query string with a string $s$ in the leaf nodes is larger than the similarity value of the query string with $s$, we remove $s$ and add $t$ to the results. Thus we only need to access the index once.

### 5.1.2 Entity Similarity Join

In this subsection, we discuss the threshold entity similarity join and the top-$k$ entity similarity join.

*Threshold Entity Similarity Join.* Given two entity tables $S$ and $R$, and the join attribute $A$ on $S$ and $R$, the threshold entity similarity join returns all entity pairs in $R \times S$ such that the similarity of each entity pair is no less than a predefined similarity threshold. For numerical attributes, we need to compute the expected value for each uncertain attribute, and then compute the absolute difference of the expected values of the entity pair. If the absolute difference is no more than the similarity threshold, we add the entity pair into the final results. EntityManager uses a hash-based method to process the numerical entity similarity join. Firstly, we compute the expected value for each join attribute value, and then we hash them into hash buckets. Finally we check the buckets to get the final results. Here the hash function is constructed by a division-based method. In hash step, all the similarity entity pairs fall into the same buckets. Therefore in the final checking step, we only need to check the buckets whose hash values are in the threshold range.

For the string type attribute, existing similarity join algorithms are inefficient to solve the threshold entity similarity join due to redundant calculations. We proposed the algorithm ES-JOIN[33], which adopts the traditional filter-and-verify framework but filters dissimilar entity pairs with a new filtering method through novel index: bilayer-prefix index.

Unlike existing string-based prefix indices taking $q$-grams as index entries, the bilayer-prefix index has two layers. The first layer takes $q$-gram as index entries. The second layer takes entity ID $eid$ as index entries and stores a list with entries of the form $(cid, Pro)$, where $cid$ is the string ID in the attribute value of the entity denoted by $eid$ and $Pro$ is the probability of the string $cid$. We use example 4 to illustrate the bilayer-prefix index.

*Example* 4. Assume that we have two entities {1, {(abc, 0.9), (abcd, 0.1)}} and {2, {(abce, 0.5), (bcd, 0.5)}}. The bilayer-prefix index structure is shown in Fig.3.

To get similar entity pairs, firstly we merge the bilayer-prefix indices of the two join tables to get the candidate results by bilayer-prefix filtering. Then two novel filtering methods are introduced to prune the candidates. According to Definition 6, the similarity of two entity attribute values is the sum of quality affection of string pairs in entity pairs. The quality affection includes two parts: the similarity of each string pair and the product of $Pro$ of each string pair. Note that the similarity of a string pair is no larger than 1.

To save unnecessary computing, the $(cid, Pro)$ list of an entity is ordered by descending $Pro$. Let $k$ be the longest length of the $(cid, Pro)$ list of an entity, and the key idea of our first filtering method is that the computation of the similarity of string pairs in an entity pair can be skipped if the product of $Pro$ in the first string pair is less than $\tau/k$. In fact, according to the pigeon-

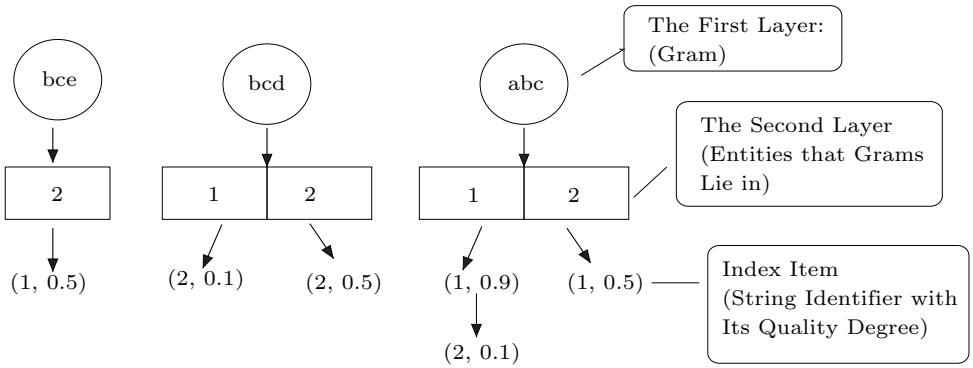



Fig.3. Bilayer-prefix index structure.

hole principle, only if at least one of the similarity of string pairs is no less than $\tau/k$, the sum of the similarity of all the $k$ pairs will have a chance to be no less than the threshold $\tau$. It can be easily verified whether the product of $Pro$ of the first string pair is larger than $\tau$.

For candidate entity pairs which meet the first filtering condition, we compute the sum of the products of $Pro$ in the string pairs of $(cid, Pro)$ list until the sum value is larger than $\tau$. This is the second filtering condition.

The filtering methods reduce the number of candidate pairs and save lots of computing cost. But even if an entity pair meets both the above filtering conditions, it may not be the final results. We should verify the candidate pairs.

*Top-k Entity Similarity Join.* We modify the threshold-based entity similarity join method to process the top-$k$ entity similarity join. Firstly, we set the threshold to a small value to get the initial results. Then we check the number of results: if the number of the results is less than $k$, we loose the threshold to get more results. After repeating this process until the result number is no less than $k$ or the threshold value is 0, we rank the quality value of the results and return the top-$k$ values as the final results. In experiments, we find that if an adequate threshold value is set, there is no need to loose the threshold value in most cases, which avoids recomputing the initial results. The algorithm especially for the top-$k$ entity similarity join will be developed in the future work.

## 5.2 Query Optimization in Entity-Based Databases

The query optimization in EntityManager includes two steps. The first step is estimating the query result size. The second step is generating the query plan based on the estimation results. For the entity similarity search, the system uses an index to obtain the results[32]. For the group operator, the result size estimation technique over the threshold entity similarity join can be used for estimating the self-join size[39]. Apparently, the update operator does not require size estimation. Therefore we focus on the estimation of the result size over the range query and the entity similarity join query. In this subsection, we discuss techniques for the result size estimation of the range query (Subsection 5.2.1), the result size estimation of the threshold entity similarity join (Subsection 5.2.2) and the order selection of the entity similarity multi-join (Subsection 5.2.3). We adopt the strategy in traditional relation databases[37] to generate the query plan.

### 5.2.1 Range Query Estimation

Existing query estimation methods are not suitable for the range query in EntitManager, and they lead to an overestimate since the uncertainty of the attribute is not considered. To solve this problem, we propose a histogram-based estimation technique. We start it with an unbounded range query $Q$ by $a <_\tau x$, where $a$ is an uncertain attribute value and $\tau$ is the similarity threshold. This query returns all entity tuples satisfying $sim\,(a < x) > \tau$, which means that $a$ satisfies the following relationship: $\sum_{(v_i,p_i)\in a} sim\,(v_i < x) \times p_i > \tau$.

If all possible values of an uncertain attribute are stored in EntityManager, the calculation of the similarity in Definition 3 is equivalent to calculating the cumulative distribution function $F_a(x)$, where $F_a\,(x) = \sum_{v_i<x} p_i$. The query results are the values which satisfiy $F_a\,(x) > \tau$.

Histogram is often used to estimate the cumulative distribution functions, and we use it to estimate the

range query. Fig.4 shows an example of the cumulative distribution functions (CDF) over several tuples on attribute $A$. The tuples are shown in Table 3, where $A$ is the attribute name. In Fig.4, each stacked line represents one tuple. The $x$-axis and the $y$-axis represent the attribute value and the similarity value, respectively. Each stacked line can be regarded as the cumulative distribution function of each uncertain attribute value. Therefore, with such a figure containing all tuples, given a query $Q$ ($a <_\tau x_0$), the total number of tuples which satisfy query $Q$ can be estimated by the number of stacked lines crossing the line segment $l$ constrained by $x = x_0$, $\tau < y \leqslant 1$.

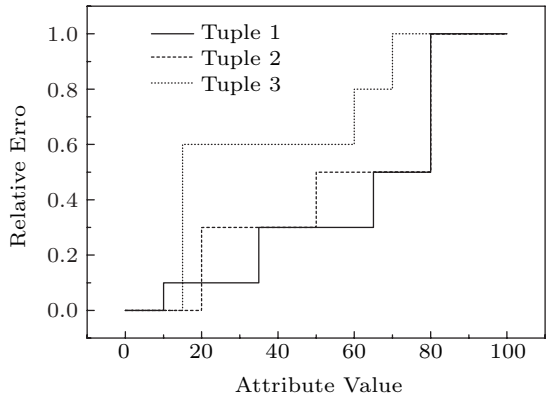



Fig.4. Example for the histogram structure.

**Table 3.** Data Fragment

| ID | $A$ |
|---|---|
| 1 | ((10, 0.1), (35, 0.3), (65, 0.5), (80, 0.1)) |
| 2 | ((20, 0.3), (50, 0.5), (80, 0.2)) |
| 3 | ((15, 0.6), (60, 0.2), (70, 0.2)) |

We define a basic two-dimensional histogram. The range input values are partitioned into $n \times m$ buckets, where $n$ and $m$ are the lengths of the two dimensions respectively. A histogram bucket $H(i, j)$ covers the area given by $((i \times \delta_x, j \times \delta_s), ((i + 1) \times \delta_x, (j + 1) \times \delta_s))$, where $\delta_x$ and $\delta_s$ are the horizontal width and the vertical width of each histogram bucket. Each histogram bucket records the number of tuples whose stacked lines intersect with this bucket.

We find that the estimation errors using this histogram will not exceed the number of inflection points of the stacked line in the buckets. In order to make the estimation more accurate, we need to ensure that the number of the inflection points in each bucket $H(i, j)$ is small enough. In our method, the histogram is firstly partitioned into $p$ equal-width buckets, and the number of the inflection points in each bucket should not exceed $\varepsilon$ ($\varepsilon = M/p$, where $M$ is the total number of inflection points, which equals the number of all possible attribute values). When a bucket contains more than $\varepsilon$ inflection points, this bucket is partitioned into $q$ equal-width buckets (generally, $q \ll p$, and $q$ can be considered as a constant). Then we set each new bucket containing $\varepsilon/q$ inflection points. The bucket not meeting the requisition is partitioned until the number of its inflection points is less than $\varepsilon$.

*Query Estimation Method.* With the above histogram structure, we can easily estimate the result size of the range query. Given a range query $a <_\tau x_0$, the result size is estimated as the number of the buckets meeting the similarity threshold in $x_0$.

*General Range Query Estimation.* For general range queries, we add another dimension to the above histogram. The meanings of the original dimensions do not change (the $x$-axis and the $y$-axis represent the end point of the query and the similarity, respectively), and the new additional dimension ($z$-axis) represents the left constraint of the query. Given a general range query $Q(x_1 < a < x_2)$, we can estimate the result size by counting the number of stacked lines crossing the line segment $l$ given by $x = x_2, \tau < y \leqslant 1$ and $z = x_1$. That is equivalent to executing a query $Q'$ ($a <_\tau x_2$) on the plane, where $z = x_1$.

We call such new histogram as the improved histogram. In this histogram, every plane on the $z$-axis is a basic histogram, corresponding to the constraint $z \leqslant x < v_{\max}$. The width of a bucket on the $z$-axis is controlled by an input parameter $\delta_z$ (in general, $\delta_z$ can be equal to $(v_{\max} - v_{\min})/p$).

### 5.2.2 Threshold Similarity Join Size Estimation

This subsection focuses on the result size estimation of the threshold entity similarity join. The existing methods for join result size estimation are mainly based on sampling[30]. However, these methods can only be applied to our system when the threshold value is relatively small and the join result size is relatively large. When it comes to a larger threshold and a relatively small join result size, the estimation accuracy cannot be guaranteed. This is because the samples satisfying join condition take small proportion in sample space $R \times S$. To solve this problem, we use locality sensitive hashing (LSH)[31] to cluster similar objects of $R$ and $S$. Assume that the clusters on $R$ and $S$ are $(C_{R_1}, \cdots, C_{R_m})$ and $(C_{S_1}, \cdots, C_{S_n})$, respectively. The result size of the threshold entity similarity join can be obtained by multiplying the estimation of the result size of the thresh-

old entity similarity join of $C_{R_i}$ $(1 \leqslant i \leqslant m)$ and $C_{S_j}$ $(1 \leqslant j \leqslant n)$. Note that if we take more samples from the sample space $C_{R_i} \times C_{S_j}$, the sample results would reflect the actual result size more precisely.

*Framework of Estimation Algorithm.* The framework of the threshold similarity join size estimation algorithm is divided into two parts. The system first partitions the dataset into clusters based on the similarity of attribute values, and then samples from the cluster set to estimate the join result size.

*Hash Cluster Based on Quality Degrees.* The partition process has two steps. We first use LSH to hash entity uncertain attribute values to get the similar relationship of the uncertain attribute values, and then apply a clustering algorithm based on the similar relationship.

Our system uses the edit distance as the similarity measure function and takes $q$-gram as the signature of strings in the threshold entity similarity join process. The $q$-gram set of an uncertain attribute value is the union of the $q$-gram set of all strings in the uncertain attribute value. For example, the gram set of the uncertain attribute value {(Robert, 0.6), (Bob, 0.4)} is { Ro, ob, be, er, rt, Bo, ob}. Note that the gram set allows repeated elements. Based on count filtering[17], if the edit distance of two strings $s_1, s_2$ is less than $k$, the common $q$-gram number is at least $\max(|s_1|, |s_2|)+1-(k+1)\times q$ (hereafter, it is denoted as $L(s_1, s_2)$). From Definition 6, the threshold entity similarity join has the following features: the similarity value between string pair $(r_i, s_j)$ increases as the similarity value between entity pair $(r, s)$ increases.

Besides, by counting the $q$-gram set of each uncertain attribute value, the attribute value could transfer to a high-dimension vector. For each vector, each dimension corresponds to a unique $q$-gram, and the number of the dimensions is the $q$-gram number in the uncertain attribute value set. This process in our system is called as the vectorization of uncertain attribute values. We have the following properties.

**Property 1**. *Suppose that $(r, s)$ is an entity uncertain attribute value pair, the more similar $r$ and $s$ are, the more common $q$-grams the corresponding $q$-gram set has, and the more similar the corresponding high-dimension vector is, where the vector is the representation of the $q$-gram set of $(r, s)$.*

**Property 2**. *Suppose that the quality of entity uncertain attribute value pair $(r, s)$ is $p$, where $p = \sum_{ed(v_{r_i}, v_{s_j})\leqslant k} p_{r_i} \times p_{r_j}$. Then the larger $p$ is, the larger $p_{r_i}$ and $p_{s_j}$ are. That is, the higher quality the entity tuple $(r, s)$ has, the higher quality the corresponding common $q$-gram value has.*

Based on Property 2, the process of the vectorization of an uncertain attribute value can be improved by adding the influence of quality degree. It makes the result vector more accurate to represent the corresponding attribute value. In the vectorization process, we do not count the frequency of each $q$-gram, but count the quality value of each $q$-gram. For example, without considering the impact of different quality degrees in the vectorization process, the vectorization result of {(Robert, 0.9), (Bob, 0.1)} is (1.9, 0, 1, 1.8, 0); otherwise, the vectorization result is (1.1, 0, 1, 0.2, 0).

In the vectorization process, each vector is mapped into a hash value through LSH hash function family[31]. Then we can obtain all the similar relationship with the same hash value. Taking each uncertain attribute value as a vertex and each similar pair as an edge between two vertices, the uncertain attribute value set can be represented as a graph. The graph reflects the similar relationship of all the uncertain attribute value sets. Therefore, the problem of clustering similar attribute values could be converted to the problem of graph clustering or community detection[40]. Recently many studies have been done on graph clustering and community detection. In EntityManager, we adopt the CNM[41] algorithm to get the cluster.

*Sample Method to Estimate Result Size.* The sample method is based on random sampling. Our estimation method randomly takes samples from each cluster to get sample sets, and then performs the threshold entity similarity join on this sample sets to estimate the result size. Suppose that $R$ and $S$ are two join entity tables, $C_{R_i}$ and $C_{S_j}$ are the $i$-th and the $j$-th cluster set of $R$ and $S$, respectively. $S_{C_{R_i}}$ and $S_{C_{S_j}}$ are the sample set from $C_{R_i}$ and $C_{S_j}$ respectively. If the threshold entity similarity join size of $S_{C_{R_i}}$ and $S_{C_{S_j}}$ is $n$, the estimation join size of $C_{R_i}$ and $C_{S_j}$ is $|C_{R_i}| \times |C_{S_j}| \times n/(|S_{C_{R_i}}| \times |S_{C_{S_j}}|)$.

### 5.2.3 Entity Similarity Multi-Join Order Selection in EntityManager

Entity similarity multi-join is the entity similarity join related to multiple entity tables. Its cost associates with the size of the intermediate results generated in the join process and the indices on the join tables. Based on these two factors, we establish the corresponding cost model. By the cost model, the entity similarity multi-join order selection method adopts traditional multi-join optimization algorithm[36] to decide the order of

the join between entity tables.

The most important parameter in the cost model is the size of the intermediate results during the join process. If the join tables with a small result size take precedence on execution, the time and memory performance will be improved. However, it is insufficient to consider the intermediate size only. If the join tables have indices in the join attribute, we can avoid scanning disk in the actual join execution process, which will greatly reduce I/O cost.

Based on the above facts, we use intermediate result size and index as the major cost to optimize multi-join order. The cost model can be expressed as a function defined as follows:

$$\begin{aligned} cost(R, S : a) = \ & \alpha(Index(R : a) + Index(S : a) + \\ & (1 - \alpha)(cost(R) + cost(S) + \\ & mresult(R) + mresult(S)), \end{aligned}$$

where $R$ and $S$ are the tables involving the join process, $cost(R, S : a)$ is the join cost of $R$ and $S$ on attribute $a$, $cost(R)$ is the cost of generating $R$, $mresult(R)$ is the size of generated temporary table in the join process, and $Index(R)$ represents I/O cost, which works only when $R$ is an original table. Evidently, if $R$ is the original table, $cost(R) = 0$ and $mresult(R) = 0$. Besides, $\alpha$ is the weight of index structure in the total cost. We suppose that the temporary table is stored in memory, and hence it does not need the I/O operation. The $Index()$ function is defined as follows.

$$Index(R : a) = \begin{cases} |R|, & \text{if } bool(R : a) = 0, \\ 0, & \text{if } bool(R : a) = 1, \end{cases}$$

where $bool\,(R : a)$ denotes whether $R$ has index on attribute $a$. If there exists an index over $R$, $bool(R : a) = 1$; otherwise, $bool(R : a) = 0$.

Based on the cost model, we use an existing multi-join sequence selection algorithm[36] to optimize the order of multi-join. When the number of entity join tables is small (no more than 5), the dynamic programming method[36] can be used to return the optimal order of join. However, when the number of entity join tables is large, dynamic programming will result in the calculation cost increasing exponentially with the number of entity tables. However, we can use a heuristic search strategy, such as the greedy algorithm[36] to get a better join order.

## 6 Experiments

We conducted experimental evaluations on EntityManager using both real and synthetic datasets. The system was evaluated from three aspects, namely the efficiency of the system under different query settings for different data types, the effectiveness of the system compared with the traditional relation database, and the usability of the system through question testing.

### 6.1 Experimental Setting

All the experiments were conducted on a PC with 2.93 GHz Inter® Core™2 Duo CPU with 4 GB main memory.

We used two real-life datasets. The first one is publication dataset. We extracted 1) 100M publications in DBLP①, which includes 1.5M authors, 2.5M papers, and 8k venues (conferences/journals), 2) 30M publications in ACM② including 0.7M papers and 0.4M authors from IEEE Xplore③, and 3) 65M publications including 1.5M papers and 1M authors from IEEE Xplore③. We adopted entity resolution method in [42] to obtain two entity tables: PAPER and AUTHOR. The schema of the table PAPER is (TITLE, VENUE, YEAR), and the schema of table AUTHOR is (NAME, TITLE), where the TITLE is the paper title. The entity table PAPER has 3 567 342 tuples and the entity table AUTHOR has 987 453 tuples. We found that 56% of the table PAPER and 63% of the table AUTHOR have multiple values in their attributes, and 90% of multi-value occurs in the attributes of TITLE and AUTHOR. We then extracted 1M tuples from entity tables AUTHOR and PAPER. Note that we only extracted tuples which have multi-value attribute tuples. For these extracted data, we retrieved the relevant data in the raw tables (before entity recognition), aiming to compare the effectiveness of our system with traditional systems. The second one is electronic commerce data. We crawled book information in computer science category from eBay④ and Amazon⑤. Then we adopted existing entity resolution method[42] to get 1.6M entity tuples. Each tuple

①http://dblp.uni-trier.de/, Mar. 2017.

②www.acm.org/dl, Mar. 2017.

③http://ieeexplore.ieee.org, Mar. 2017.

④http://www.ebay.com, Mar. 2017.

⑤http://www.amazon.com, Mar. 2017.

includes four entity attributes: BOOKTITLE, AUTHOR, PUBLICATION, and PRICE. The average number of the attribute value is 3 in AUTHOR, 2 in BOOKTITLE, 1.5 in PUBLICATION and 3 in PRICE. We divided the entity tables into two tables, and then did the entity similarity join process at attribute BOOKTITLE to get the relative similarity information between their PRICE and AUTHOR attributes.

We also developed a data generator to produce entity data including numeric type attributes and string type attributes. It is controlled by the number of entity tuples $m$ and the number of attributes $n$. Specifically, for each entity, the number of each attribute value is uniformly distributed between 1 and 5. The quality degree of each possible attribute value is randomly generated from 0.01 to 1, and these possibilities sum up to 1 for each attribute. For the string type attributes, we controlled the string length in 35 characters. For the numeric type attributes, we produced the numeric value with different distributions.

*Experimental Map.* To verify the efficiency of query processing in EntityManager, we made the following two evaluations over the syntactic dataset and the real-life dataset: 1) the scalability of the range query processing and 2) the scalability of the entity similarity search and join, compared with Bed-trees[19] for the string similarity search and Ed-Join algorithm[17] for the string similarity join. We revised these two algorithms by adding a checking stage to get the entity results to implement the entity similarity search and join operators.

To check the performance of the query optimization techniques in EntityManager, we made the following two evaluations: 1) the result size estimation algorithms for the range query and the entity similarity join query respectively, and 2) the indices for the entity similarity search and the entity similarity join queries.

To demonstrate the effectiveness of the system, we did a case study and a usability study over the publication dataset for the entity similarity search and the join queries. The experimental results are reported in Fig.5.

## 6.2 Efficiency of EntityManager

### 6.2.1 Efficiency of Query Processing

Firstly, we evaluated the efficiency with average time by randomly performing five times for 1) range queries for numeric type attributes, and 2) EQL queries including the entity similarity search and the join for string type attributes. The threshold is set to a default value of 0.5. Fig.5(a) reports their performance on the publication data and the electronic commerce data. We found that the queries with the entity similarity join operator take more time than the queries with the entity similarity search operator. It is evident that the join operator needs to handle more string pairs than the search operator. We also found that the range queries have good performance. Here the range queries were tested on the attributes PRICE and YEAR, and took B-tree as their index structures.

### 6.2.2 Scalability of Query Processing

In the next stage, we evaluated the scalability of the entity similarity search, the entity similarity join, and the range query. Here we generated synthetic data by setting the number of entity tuples in the table to $m = 10$ k, 50 k, 100 k, 200 k, 400 k, 800 k and 1 M. And we produced numeric data with uniform distribution. All the queries were performed five times and the average time was taken as the present results. In addition, we compared our algorithms with existing algorithms: Bed-tree[19] for the entity similarity search and Ed-Join algorithm[17] for the entity similarity join. These two algorithms have a good performance in the similarity search and the similarity join applications. Since similarity search and join algorithms in related work have been compared with Ed-Join in [19] and Bed-tree in [17], we did not compare our algorithms with them. Figs.5(b)∼5(d) report their performance. We have the following three observations: 1) ES-JOIN has better performance than Ed-Join, since the bilayer-prefix index can filter more entity candidates than Ed-Join; 2) the Fgram-Tree based algorithm performs better than the Bed-tree algorithm, since Fgram-Tree ensures that the similar entities are grouped into the same index nodes and lots of redundant computation is avoided; 3) the computation time increases as the number of entity tuples increases for both the string type and the numeric queries.

Here, we did not compare the range query algorithm with existing methods because we just took B-tree as the index, which is the common method used in the range search. We will put a novel algorithm especially for the range query in the further work.

### 6.2.3 Performance of Query Optimization

The query optimization techniques used in EntityManager are estimating the result size for the range query and the entity similarity join query. The system

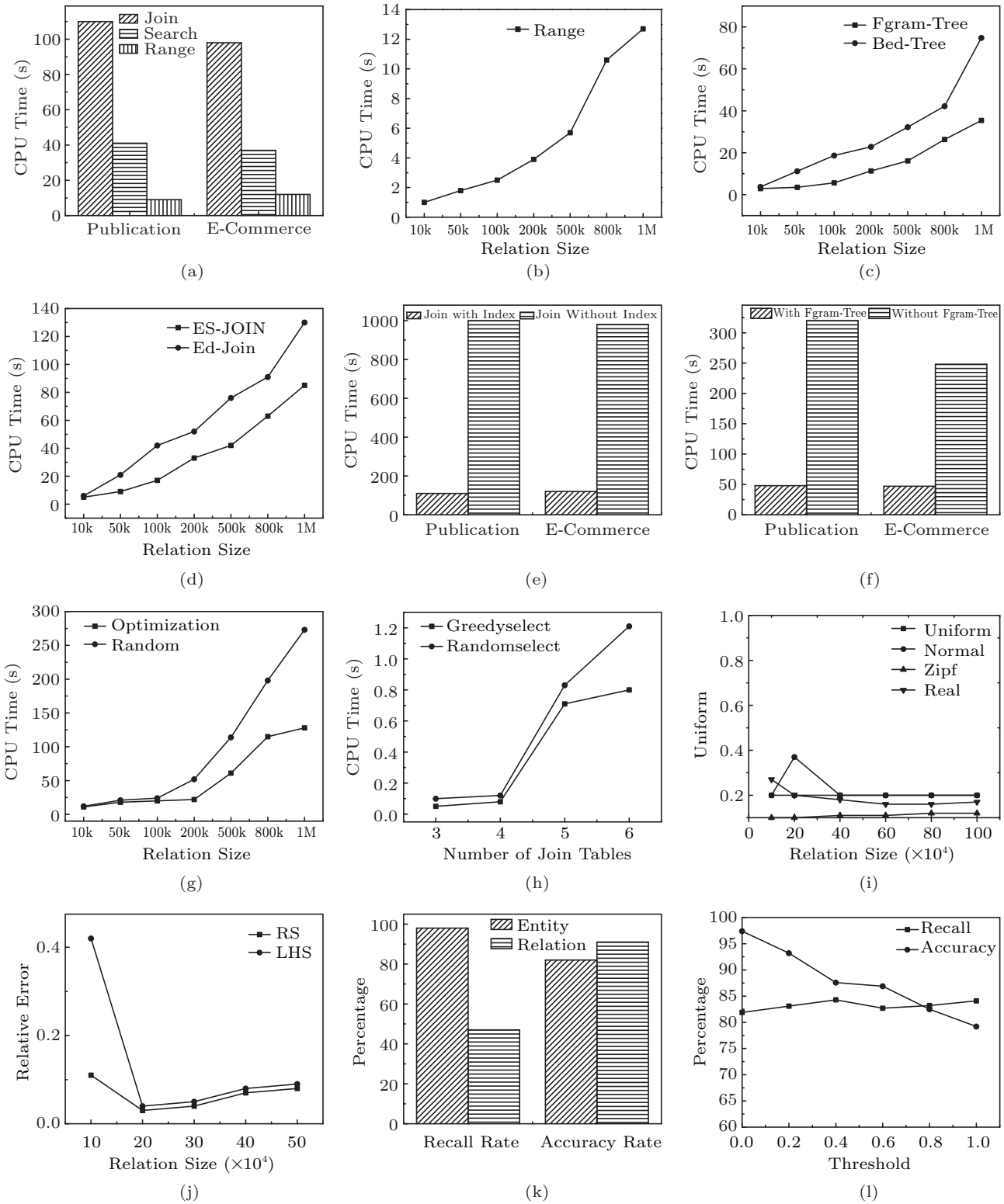



Fig.5. Performance evaluation of the EntityManager system. (a) Query processing. (b) $m$ (range query). (c) $m$ (search query). (d) $m$ (join query). (e) Bilayer-prefix index. (f) Fgram-Tree index. (g) $m$ (optimization). (h) Join-table number. (i) $m$ (range). (j) $m$ (join) (k) Case study. (l) Threshold.

uses the estimation results to generate query plans. We tested the performance of the estimation methods over the synthetic dataset. The entity tuple size varies from 10k to 1M. Firstly, we compared the efficiency of the query plan generated by estimation methods with the query plan generated by random selection. Here, the query combines the range query and the entity similarity join query involved four join tables. Fig.5(g) shows the results. We found that using our optimization technique, the query processing has better performance and the accelerated ratio is close to 2.

We then tested how the number of entity tables affects the performance of the entity similarity join algorithm. Here we increased the number of join tables involved the query from 3 to 6. The number of tuples of each table is set to 10k. Fig.5(h) shows the results. We found that the optimization query plan always performs better than the random selection query plan.

We also evaluated the time cost of the query plan generation. We tested the cost of the result size estimation for the range query and the entity similarity join query. Table 4 and Table 5 report the results. We found that the estimation stage costs very little time in the whole query processing time.

**Table 4.** Cost of Entity Similarity Join

| Dataset Size | Estimation Time (s) | Entity Join Time (s) | Ratio (%) |
|---|---|---|---|
| 10k | 0.100 | 5 | 2.0 |
| 50k | 0.172 | 11 | 1.5 |
| 100k | 0.428 | 23 | 1.8 |
| 200k | 0.719 | 36 | 1.9 |
| 500k | 1.234 | 46 | 2.6 |
| 800k | 2.356 | 67 | 3.5 |
| 1M | 3.345 | 89 | 3.7 |

**Table 5.** Numeric: Range Query

| Dataset Size | Estimation Time (s) | Query Time (s) | Ratio (%) |
|---|---|---|---|
| 10k | 0.010 | 1.0 | 1.0 |
| 50k | 0.032 | 1.5 | 2.1 |
| 100k | 0.112 | 3.0 | 3.7 |
| 200k | 0.151 | 4.0 | 3.7 |
| 500k | 0.255 | 6.0 | 4.2 |
| 800k | 0.436 | 10.8 | 4.0 |
| 1M | 0.523 | 12.5 | 4.1 |

Finally, we tested the relative error for the estimation methods. For the range query, we used the histogram to estimate its result size. Since the data distribution will affect the accuracy of the estimation results, we tested the relative error over data with uniform, normal, zipf and real data distributions. For the entity similarity join query, we compared our method with the simple random sampling method. Fig.5(i) and Fig.5(j) show the results. We found that both of the estimation methods have a low relative error.

### 6.3 Effectiveness of EntityManager

We used a case study and a usability study to evaluate the effectiveness of EntityManager.

#### *6.3.1 Case Study*

We first demonstrated the effectiveness of EntityManager through comparing it with the traditional relation system on the publication dataset. We used 100 queries including range queries over PRICE attribute and entity similarity search queries and entity similarity join queries over string attributes. Each query was tested five times to get the average results. We used recall and accuracy to show the comparison results. Here, the recall is the fraction of the returned answers that should be returned. The accuracy is the fraction of the corrected answers in the returned answers. Fig.5(k) shows the results. We found the following issues. 1) The accuracy of EntityManager is a bit lower than that of the traditional relation system, since EntityManager may return false positive results. 2) The recall of EntityManager is higher than that of the traditional relation system for both the numeric type and the string type queries. This is because for a specific entity, the query results produced by the traditional relational database may miss some tuples representing this entity. 3) Although the recall and the accuracy of EntityManager cannot achieve 100%, they are higher than 80%, which can be acceptable by most users and applications. Note that the accuracy of EntityManager depends on the accuracy of the entity resolution techniques; thus our system will perform better with the improvement of the precision of the entity resolution techniques.

Next, we tested how the threshold parameter impacts the recall and the accuracy of EntityManager. We set the threshold value to 0.1, 0.3, 0.6, 0.9 and 1.0 respectively. The results are shown in Fig.5(l). We observed that 1) the recall rate decreases as the threshold increases, because of the fact that EntityManager filters out the low-quality results but these results may satisfy the query constraint; 2) the accuracy rate keeps

the same in different threshold settings, since the accuracy of the results only depends on the accuracy of the entity resolution algorithm.

### 6.3.2 Usability Study

We used question testing to test the usability of our system. The used testing data is the publication dataset, and the testing question is shown in Table 6. When a user began to test, he/she was asked to run the same query task (randomly chosen from the query set) in both EntityManager and the traditional relation database: Mysql. We called the above systems as $A$ and $B$, respectively. After using both the systems, the user was asked to provide optional answers to the question of which system you think is more useful. Note that, in our evaluation, no hints and guides were provided. Such a method was also used in the usability test of [43].

**Table 6.** Results of Usability Study about the Question Which System You Think Is More Useful

| Feedback | Percentage of Participants |
|---|---|
| A is much more useful than B | 50 |
| A is slightly more useful than B | 40 |
| A and B are roughly the same | 5 |
| B is slightly more useful than A | 3 |
| A is much more useful than B | 2 |

We asked 30 undergraduate students to participate the testing anonymously and voluntarily. Eventually, we got all feedback. The feedback is shown in Table 6. As we can see in Table 6, overall 90% of participants think that the usability of EntityManager is better than that of the traditional relation database. Only 10% of participants have the opposite opinion.

## 7 Research Challenges

*Big Data and Parallel Implementations.* When datasets get very large, the traditional centralized processing may bring efficiency problems and be infeasible with limited system resources (e.g., CPU, I/O, and memory). To address these problems, we attempt to design novel parallel algorithms to do query processing, and the parallel algorithms bring research challenges including the storage strategy to adapt parallel implementation, and the extension from centralized algorithms to parallel algorithms.

*OLAP on the System.* Now EntityManager only supports aggregation with several standard aggregation functions. In existing e-bossiness applications, with information integrated from multiple data sources, the OLAP on dirty data is in requirement. With this motivation, we attempt to study the OLAP on dirty data with efficient operations including cube, slicing, roll-up and drill-down.

*Transaction Management.* Currently, the transaction management in our system is very simple. A practical database management system requires effective transaction management. The concurrency control of our system requires the lock on the entity level and new strategies should be adopted to ensure the correct results for concurrent operations. Thus, we attempt to study novel transaction management techniques for EntityManager.

## 8 Conclusions

In this paper, we described EntityManager, a system for managing dirty data with the entity as the basic unit and keeping conflicts in data as uncertain attributes. We introduced the reality demands and current challenges to manage dirty data, which are the motivation of our system. We gave an overview of query processing techniques and query optimization techniques especially designed for EntityManager. Finally, the effectiveness and the efficiency of our strategy were verified by experiments.

More efficient OLAP and transaction management techniques and a parallel system will be developed in our future work.

## References

[1] Andritsos P, Fuxman A, Miller R J. Clean answers over dirty databases: A probabilistic approach. In *Proc. the 22nd ICDE*, April 2006, Article No. 30.

[2] Fuxman A D, Miller R J. First-order query rewriting for inconsistent databases. In *Proc. the 10th ICDT*, January 2005, pp.337-351.

[3] Fuxman A, Fazli E, Miller R J. Conquer: Efficient management of inconsistent databases. In *Proc. SIGMOD*, June 2005, pp.155-166.

[4] Boulos J, Dalvi N, Mandhani B, Mathur S, Ré C, Suciu D. MYSTIQ: A system for finding more answers by using probabili31 ties. In *Proc. SIGMOD*, June 2005, pp.891-893.

[5] Hassanzadeh O, Miller R J. Creating probabilistic databases from duplicated data. *VLDB J.*, 2009, 18(5): 1141-1166.

[6] Widom J. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. CIDR*, Jan. 2005, pp.262-276.

[7] Getoor L, Machanavajjhala A. Entity resolution: Theory, practice & open challenges. *PVLDB*, 2012, 5(12): 2018-2019.

[8] Waguih D A, Berti-Equille L. Truth discovery algorithms: An experimental evaluation. arXiv: 1409.6428, May 2014. https://arxiv.org/abs/1409.6428, Mar. 2017.

[9] Lipner S B, Balenson D M, Ellison C M, Walker S T. System and method for data recovery, September 1996. US Patent 5,557,765. https://www.google.com/patents/us5557765, Apr. 2017.

[10] Miles M B, Huberman A M. Qualitative Data Analysis: An Expanded Sourcebook. Sage Publications, Inc., 1994.

[11] Rahm E, Do H H. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 2000, 23(4): 3-13.

[12] Arasu A, Ganti V, Kaushik R. Efficient exact set-similarity joins. In *Proc. the 32nd VLDB*, September 2006, pp.918-929.

[13] Behm A, Ji S, Li C, Lu J. Space-constrained gram-based indexing for efficient approximate string search. In *Proc. ICDE*, March 29-April 2, 2009, pp.604-615.

[14] Goemans M X, Williamson D P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 1995, 42(6): 1115-1145.

[15] Hadjieleftheriou M, Chandel A, Koudas N, Srivastava D. Fast indexes and algorithms for set similarity selection queries. In *Proc. the 24th ICDE*, April 2008, pp.267-276.

[16] Hadjieleftheriou M, Koudas N, Srivastava D. Incremental maintenance of length normalized indexes for approximate string matching. In *Proc. ACM SIGMOD*, June 29-July 2, 2009, pp.429-440.

[17] Xiao C, Wang W, Lin X. Ed-Join: An efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 2008, 1(1): 933-944.

[18] Xiao C, Wang W, Lin X, Yu J X, Wang G. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems*, 2011, 36(3): 15:1-15:15.

[19] Zhang Z, Hadjieleftheriou M, Ooi B C, Srivastava D. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In *Proc. SIGMOD*, June 2010, pp.915-926.

[20] Bayardo R J, Ma Y, Srikant R. Scaling up all pairs similarity search. In *Proc. the 16th WWW*, May 2007, pp.131-140.

[21] Wang J, Li G, Feng J. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 2010, 3(1): 1219-1230.

[22] Sarawagi S, Kirpal A. Efficient set joins on similarity predicates. In *Proc. ACM SIGMOD*, June 2004, pp.743-754.

[23] Vernica R, Carey M J, Li C. Efficient parallel set-similarity joins using mapreduce. In *Proc. ACM SIGMOD*, June 2010, pp.495-506.

[24] Li C, Wang B, Yang X. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *Proc. the 33rd VLDB*, September 2007, pp.303-314.

[25] Wang J, Li G, Feng J. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In *Proc. ACM SIGMOD*, May 2012, pp.85-96.

[26] Ioannidis Y E. The history of histograms (abridged). In *Proc. the 29th VLDB*, Sept. 2003, pp.19-30.

[27] Haas P J, Naughton J F, Seshadri S, Swami A N. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 1996, 52(3): 550-569.

[28] Hou W C, Ozsoyoglu G, Dogdu E. Error-constrained COUNT query evaluation in relational databases. *ACM SIGMOD Record*, 1991, 20(2): 278-287.

[29] Olken F. Random sampling from databases [Ph.D. Thesis]. University of California, 1993.

[30] Ngu A H, Harangsri B, Shepherd J. Query size estimation for joins using systematic sampling. *Distributed and Parallel Databases*, 2004, 15(3): 237-275.

[31] Lee H, Ng R T, Shim K. Similarity join size estimation using locality sensitive hashing. *PVLDB*, 2011, 4(6): 338-349.

[32] Tong X, Wang H. Fgram-Tree: An index structure based on feature grams for string approximate search. In *Proc. the 13th WAIM*, August 2012, pp.241-253.

[33] Liu X, Wang H, Li J, Gao H. Similarity join algorithm based on entity. *Journal of Software*, 2015, 26(6): 1421-1437. (in Chinese)

[34] Zhang Y, Yang L, Wang H. Range query estimation for dirty data management system. In *Proc. the 13th WAIM*, August 2012, pp.152-164.

[35] Liu X, Wang H, Li J, Gao H. Multi-similarity join order selection in entity database. *Journal of Frontiers of Computer Science and Technology*, 2012, 6(10): 865-876.

[36] Garcia-Molina H, Ullman J D, Widom J. Database System Implementation. Prentice-Hall, 2000.

[37] Abiteboul S, Hull R, Vianu V. Foundations of Databases. Addison-Wesley, 1995.

[38] Ilyas I F, Beskales G, Soliman M A. A survey of top-$k$ query processing techniques in relational database systems. *ACM Computing Surveys* (*CSUR*), 2008, 40(4): 11:1-11:58.

[39] Zhang Y, Yang L, Wang H. Similarity join size estimation with threshold for dirty data. *Journal of Computers*, 2012, 35(10): 2159-2168. (in Chinese)

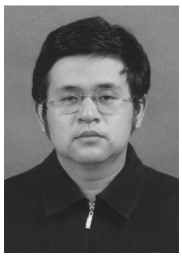
[40] Xu R, Wunsch D. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 2005, 16(3): 645-678.

[41] Clauset A, Newman M E, Moore C. Finding community structure in very large networks. *Physical Review E*, 2004, 70(6): 66-111.

[42] Li Y, Wang H, Gao H. Efficient entity resolution based on sequence rules. In *Proc. CSIE*, May 2011, pp.381-388.

[43] Kuang D, Li X, Ling C X. A new search engine integrating hierarchical browsing and keyword search. In *Proc. the 22nd IJCAI*, July 2011, pp.2464-2469.

**Xue-Li Liu** is a Ph.D. candidate in computer technology and science, Harbin Institute of Technology, Harbin. Her research interests include data quality and massive data management.



**Hong-Zhi Wang** is currently a professor at the School of Computer Science and Technology, Harbin Institute of Technology, Harbin. He obtained his B.S., M.E. and Ph.D. degrees in computer science and technology from Harbin Institute of Technology, Harbin, in 2001, 2003 and 2008 respectively. He has published more than 100 papers. He was awarded Microsoft Fellowship, Chinese Excellent Database Engineer and IBM Ph.D. Fellowship. His Ph.D. thesis was selected as a CCF outstanding Ph.D. thesis. His research interests include big data management, data quality, graph data management, and web data management.

**Jian-Zhong Li** is currently a professor at the School of Computer Science and Technology, Harbin Institute of Technology, Harbin. In the past, he worked as a visiting scholar with the University of California at Berkeley, Berkeley, as a staff scientist with the Information Research Group, Lawrence Berkeley National Laboratory, Berkeley, and as a visiting professor with the University of Minnesota, Minneapolis. His research interests include data management systems, sensor networks, and data-intensive computing. He has been involved in the program committees of major conferences and journals of computer science and technology, including TKDE, VLDB J., KDD, ICDE, CIKM, ICDM, EDBT, etc.

**Hong Gao** is a professor in the School of Computer Science and Technology at Harbin Institute of Technology, Harbin. Prof. Gao is the principal investigator for several National Natural Science Foundation Projects. She was also the winner of National Science and Technology Progress Award (Second Class) in 2005. Her research interests include wireless sensor network, cyber-physical systems, massive data management, and data mining.