Efficient Maintenance of Common Keys in Archives of Continuous Query Results from Deep Websites

Fajar Ardian Sourav S Bhowmick

School of Computer Engineering, Nanyang Technological University, Singapore
assourav@ntu.edu.sg

***Abstract*—In many real-world applications, it is important to create a local archive containing versions of structured results of *continuous queries* (queries that are evaluated periodically) submitted to autonomous database-driven Web sites (e.g., deep Web). Such history of digital information is a potential gold mine for all kinds of scientific, media and business analysts. An important task in this context is to *maintain* the set of *common keys* of the underlying archived results as they play pivotal role in data modeling and analysis, query processing, and entity tracking. A set of attributes in a structured data is a *common key* iff it is a key for all versions of the data in the archive. Due to the data-driven nature of key discovery from the archive, unlike traditional keys, the common keys are *not temporally invariant*. That is, keys identified in one version may be different from those in another version. Hence, in this paper, we propose a novel technique to maintain common keys in an archive containing a sequence of versions of evolutionary continuous query results. Given the current common key set of existing versions and a new snapshot, we propose an algorithm called COKE (COmmon KEy maintenance) which *incrementally* maintains the common key set without undertaking expensive minimal keys computation from the new snapshot. Furthermore, it exploits *certain* interesting evolutionary features of real-world data to further reduce the computation cost. Our exhaustive empirical study demonstrates that COKE has excellent performance and is orders of magnitude faster than a baseline approach for maintenance of common keys.**

I. Introduction

Due to the proliferation of database-driven Web sites, there is an enormous volume of structured data on the Web. For instance, a recent study [4] showed that the deep Web contains more than $450,000$ Web databases and these are mostly *structured* data sources ("relational" records with attribute-value pairs) – with a dominating ratio of $3.4 : 1$ versus unstructured sources. Web users typically retrieve relevant data from a deep Web source by submitting an HTML form with valid input values. Search engines, on the other hand, may employ a technique called *surfacing* which automatically submits a large number of queries through the form with valid input values to crawl the content of a deep Web site [16]. In the paper, we address an important problem targeted to deep Web users community instead of search engines.

An important characteristic of deep Web data sources is that they are evolutionary in nature. Consequently, the data content published by a site in response to a query may evolve with time. Hence, Web users may pose *continuous* queries [5], [14] (queries that are evaluated periodically on a source) to retrieve relevant data over time. In many applications, it is important to create archives containing previous versions of such evolutionary query results as such history of digital information is a potential gold mine for all kinds of scientific, media and business analysts [17]. For example, one may be interested in finding how the average salaries of people at different cities have changed during 2005 to 2008 to study the effect of subprime mortgage crisis.

EID	Name	Birthdate	City	Children	Income
e_1	Alice	1 Jan 80	Los Angeles	1	10000
e_2	Bob	2 Feb 79	Los Angeles	2	20000
e_3	Carol	3 Mar 78	Chicago	3	20000
e_4	Alice	4 Apr 77	Los Angeles	4	20000
e_5	Dave	5 May 76	Houston	2	30000

(a) R_1

EID	Name	Birthdate	City	Children	Income
e_4	Alice	4 Apr 77	Los Angeles	5	40000
e_5	Dave	5 May 76	Houston	2	30000
e_6	Eve	5 May 76	Houston	2	30000
e_7	Bob	6 Jun 75	Los Angeles	5	20000
e_8	Isaac	7 Jul 74	Phoenix	6	50000

(b) R_2

Fig. 1. Continuous query results at times (a) τ_1 and (b) τ_2.

EID	Title	Bid	Price	Date	Location
e_1	Apple..Digital	21	350.00	Feb 18 07:54	Florida
e_2	Apple..Grade A	29	202.50	Feb 18 08:23	Iowa
e_3	Apple..New	21	405.00	Feb 18 08:58	N.Y.
e_4	Slightly..16GB	24	227.50	Feb 18 09:21	Illinois
e_5	Apple..Grade A	32	138.00	Feb 18 09:36	Iowa

(a) R_1 (Week *n-1*)

EID	Title	Bid	Price	Date	Location
e_1	Slightly..16GB	24	227.50	Feb 18 09:21	Illinois
e_2	Apple..Grade A	34	152.50	Feb 18 09:36	Iowa
e_3	Apple..Used	21	202.50	Feb 25 10:34	N.Y.
e_4	Apple...World	23	202.50	Feb 25 11:10	N.Y.
e_5	Apple..MA627LL	27	192.50	Feb 20 09:41	Georgia

(b) R_2 (Week *n*)

Fig. 2. Continuous query results from an auction site.

The most common strategy to store continuous query results is to first extract structured records from the HTML pages using existing data extraction techniques and then store them in relational table(s) [1], [3], [6]. Consequently, data extracted from a remote Web source Q during a time period τ_1 to τ_n can be represented in the local archive as a sequence of relations $S = \langle R_1, R_2, \ldots, R_n \rangle$. For example, consider two sets of persons' records shown in Figure 1 that are extracted from the results of a continuous query at times τ_1 and τ_2, and stored in relational tables R_1 and R_2, respectively. Each tuple or record in the table represents a person entity in the archive. Note that the EID attribute is not part of the records but is used only to facilitate discussions. Notice that *children* and *income* attributes of entity e_4 are updated in R_2; entity e_8 has been inserted in R_2; and entity e_3 in R_1 has been deleted. Figure 2 depicts another example of relational representations of continuous query results from an auction website at two different weeks.

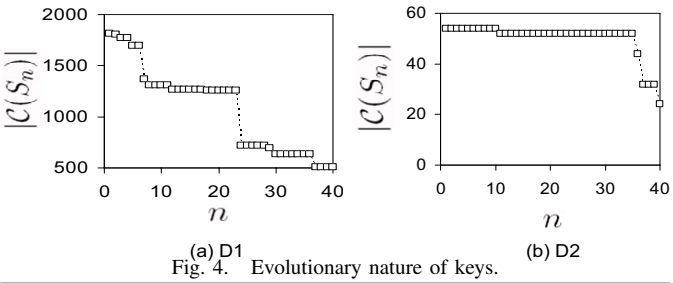
A. Motivation

Given a sequence of versions of relation S in the archive, the identification of *common keys* is important for accurate tracking of entities, version management, and query processing over archived relations (we shall elaborate on some of these issues in Section II). A key[1] c of a relation R_i in S is

[1] A set of attributes is a *key* of a relation iff there is no two tuples in this relation with the same value for all attributes in this set.

ID	Web site	Query	No. of Attr.	Avg. no of tuples
D1	www.careerbuilder.com	Information Technology (Category), California (State), Database Administrator (Keywords), 50 (Miles)	12	274
D2	stores.tomshardware.com	MP3 Player (Find Products)	7	554

Fig. 3. Real-world data



(a) D1 (b) D2

Fig. 4. Evolutionary nature of keys.

a *common key* iff it is a key for all the versions of the relation. That is, c is a key of $R_i \ \forall \ 0 < i \leq |S|$. For example, some of the common keys of R_1 and R_2 in Figure 1 are $\{birthdate,\ name\}$, $\{name,\ children\}$, and $\{name,\ income\}$. In Figure 2, {*date*, *title*, *location*} and {*title*, *bid*, *location*} are two examples of common keys. c is a *minimal common key* iff none of its proper subset is also a common key. For instance, {*date*} and {*title*, *bid*} are the minimal common keys in Figure 2. In this paper, we present an efficient *data-driven* algorithm to identify a set of common keys in S.

Automated identification of *all* common keys in an archive containing versions of continuous query results is a non-trivial problem for several reasons. Firstly, due to privacy reasons, the remote Web site Q may not publish *explicit identifier(s)* (e.g., social security number) of entities. Hence, one may not be able to manually identify such identifiers as keys. Secondly, even if such identifiers do exist (e.g., Vehicle Identification Number (*VIN*)), an explicit list of all keys is often difficult to find manually. Thirdly, identification of common keys should be completely transparent from the Web users and their interaction behaviors with Q should not be affected. Lastly and more importantly, due to lack of availability of the remote database schema of Q, it is not possible for the local archive to take the traditional approach of inferring keys from the schema of Q. Note that this issue is more challenging for typical Web users compared to search engines. The latter may submit *many* queries to retrieve a large number of records with *different* attribute values at different time points in order to crawl Q [16]. In this case, it is easier to determine the common keys quickly. In contrast, the former is only interested in a subset of data in Q as the Web users' goal are *not to crawl* Q. Thus, they may submit relatively *fewer* continuous queries to retrieving fewer attributes values, making it harder to discover common keys quickly.

Evolutionary Nature of Keys. Discovering keys, especially *composite* keys (keys consisting of two or more attributes), is known to be a computationally difficult problem as the number of possible keys increases exponentially with the number of database attributes [23]. To address this problem, recently Sismansis et al. [23] proposed a practical algorithm, called GORDIAN, for discovering composite keys in a collection of real-world entities. Consequently at first glance, it may seem that the common key discovery problem can be solved by first discovering the set of keys $\mathcal{K}(R_i)$ of *any* relation R_i in S where $0 < i \leq |S|$ using an existing data-driven key discovery technique (e.g., GORDIAN [23]). Next, assign $\mathcal{K}(R_i)$ to be the common keys of all the versions in S. Since traditionally keys are temporally invariant, $\mathcal{K}(R_i)$ should remain the same across all versions and therefore looking at any one version is sufficient to choose common keys of S.

Interestingly, the above approach does not work due to the evolutionary nature of keys in the archive. We observed that the keys discovered from R_i by any existing data-driven key discovery technique may *not be temporally invariant*. That is, a key in relation R_i archived at time τ_i may not be a key in R_{i+1} downloaded at time τ_{i+1}. For instance, consider R_2 in Figure 1. The keys $\{birthdate\}$, $\{children,\ city\}$, and $\{children,\ income\}$ of R_1 are no more keys in R_2. On the other hand, $\{name\}$ is now a key of R_2. Similarly, in Figure 2 $\{price\}$ is a key in R_1 but not in R_2. Hence, the set of keys discovered from R_i cannot be automatically assigned as the common key set of the entire sequence S.

To get a better understanding of this problem, we experimented with some versions of real-world continuous query results from a set of deep Web sites. Figure 3 shows two representative sites and continuous queries that we analyzed over a time period. A query is shown in the "*Query*" column and the string inside the bracket of each query corresponds to the "*field*" in the search form. Each data set contains a relational archive of a sequence of query results for a continuous query submitted during 20 November, 2007 and 28 February, 2008. We periodically issued the query every two days and collected 40 snapshots for each of the data sets. The "*No. of Attr.*" column specifies the number of attributes in each record in the results and the "*Avg. no. of tuples*" column specifies the *average* number of records returned by the query at every time point.

Clearly, if an existing common key in the sequence S is not a key in a new relation R_n, then the existing common key set, denoted by $\mathcal{C}(S)$, needs to be modified. In other words, "invalidity" of a common key results in evolution of $\mathcal{C}(S)$. Observe that whenever a common key is invalid it results in a decrease in the cardinality of $\mathcal{C}(S)$. Hence, we empirically observe the evolutionary nature of keys by analyzing the change in cardinality of the common key set over time. Figure 4 graphically elaborates on the evolution of the cardinality of common key sets against the number of relations over time for the two representative data sets. Observe that the number of common keys evolves with time for both the data sets and the frequency of evolution varies widely. For instance, in Figure 4(a) the common keys changed frequently (13 times) whereas in Figure 4(b), it only changed five times. Also, the cardinalities of common key sets for both these sources kept changing throughout the time period. Consequently, the *final common keys may not converge quickly and as a result we may not be able to determined them by observing only first few versions of the query results*.

B. Overview

Given the current common key set $\mathcal{C}(S)$ of a sequence of relations S and a new relation R_n, at first glance, it may seem that the common key maintenance problem can be solved in two steps. First, find the set of minimal keys of R_n by applying an existing data-driven key discovery technique [15], [23]. The set of all keys can be derived from the set of minimal keys. Second, we can maintain the common keys by computing the intersection of $\mathcal{C}(S)$ and keys of R_n. While this approach clearly works, it is computationally expensive as the minimal key discovery step is of high computational cost [23] and we have to execute it for *every* new relation.

Instead, in Section IV we propose an algorithm called COKE (**CO**mmon **K**ey Maintenanc**E**) which exploits *certain* evolutionary features of real-world data and *incrementally* maintains $\mathcal{C}(S)$ after the arrival of R_n *without* computing the set of minimal keys of R_n. Using an efficient *partitioning plan*, COKE first splits R_n based on each $C \in \mathcal{C}(S)$ into a set of *non-trivial sub-relations* where each sub-relation has at least two tuples and any two tuples in the same sub-relation have the same value for all attributes in C. Further, any two tuples in two different sub-relations have different value for at least one attribute in C. Observe that if C is no longer a key in R_n, then its non-trivial sub-relation contains at least two tuples. Next for each C and for each non-trivial sub-relation T, COKE creates the *projected sub-relation* from T by removing all the attributes in C from T. Then, it computes the minimal keys of the projected sub-relation using an existing data-driven key discovery algorithm. Next, the algorithm efficiently computes the set of minimal *proxy keys* of the projected sub-relation over C. A *proxy key* P is a key of the projected sub-relation and $P \supseteq C$. Lastly, the set of minimal common keys of the new sequence is efficiently computed from the proxy keys. Note that COKE can efficiently determine when the common keys remain unchanged in the new version as the non-trivial sub-relation set is empty in this case. Consequently, it discards subsequent steps for computing projected sub-relations and proxy keys.

The above strategy of common keys maintenance is significantly faster than computing the entire set of keys from R_n directly for the following reasons. Firstly, we can ignore the computation of all sub-relations that are *not* non-trivial (i.e., contains only one tuple). Interestingly, our analysis of a variety of real-world continuous query results in Section V revealed that there are relatively very few non-trivial sub-relations. Consequently, COKE only needs to consider a small number of such sub-relations. Secondly, the number of tuples in each non-trivial sub-relation is often very small. This further improves the performance of COKE by reducing the execution cost of a data-driven key discovery algorithm. We conduct an extensive set of experimental evaluations in Section VI by comparing the performance of the COKE algorithm with respect to a baseline method on both synthetic and real-world data sets. The experimental results show that COKE is more scalable and orders of magnitude faster than the baseline approach.

II. Usefulness of Common Keys

In this section, we highlight the usefulness of common keys by discussing how evolutionary nature of keys may adversely affect the accuracy of tracking entities and in query processing. In this context, we outline how the knowledge of common keys can help us to alleviate these problems.

Accuracy of tracking entities in a deep Web archive. Monitoring the evolution of deep Web query results is beneficial for several applications such as trend analysis, event tracking and notification, etc. For example, a user may wish to be notified whenever the *price* of any *ipod* drops by 50%. A monitoring algorithm should output high *Quality of Data (QoD)* so that the quality of overall monitoring process can be improved. QoD can be measured in different ways, one of which is *accuracy* of tracking. That is, a tracking algorithm should be able track an entity *correctly* in the query results sequence. For example, consider the results in Figure 2. Suppose that two entities $e_1 \in R_i$ and $e_2 \in R_j$ are regarded as the same entity if they have same values for the attribute set $\{title, date, location\}$. We refer to such attribute set as *identifier*. If the tracking algorithm is "identifier-aware", then it can track the evolution of the results with high accuracy. For instance, consider the second entity in Figure 2(b). Although it shares the same *title* with the second and fifth entities in Figure 2(a), it is only identical to the fifth entity as the *title*, *date*, and *location* values match for these two entities. Observe that the values of identifier of an entity e must be identical in two different results' snapshots of a continuous query. That is, if e occurs in R_i and R_j then there must exists a one-to-one mapping between the occurrences of e.

The knowledge of common keys can be used to detect potential identifier(s) when such information is not explicitly available from the source. An identifier must satisfy following two criteria. First, *an identifier must be a common key in* S. For instance, one of the common keys in Figure 2, namely $\{date, title, location\}$ and $\{title, bid, location\}$, *may be* used as an identifier. Second, *the value of a common key of an entity* e *must be conserved in the historical results sequence of a query*. In other words, if e exists in both R_1 and R_2 then the common key values of e in these two relations must be identical and must not evolve with time. For example, the common key $\{date, title, location\}$ is an identifier for each *auction* entity as the values are conserved in R_1 and R_2 for every entity. Once we are aware of this conserved feature, we can determine that the fourth and fifth entities in R_1 are identical to the first and second entities in R_2, respectively, as they have same common key values. On the other hand, the common key $\{title, bid, location\}$ cannot be an identifier as the value of *bid* of an entity may evolve with time.

Note that traditional techniques for duplicate record detection or reference reconciliation [7], [8] cannot be used to identify identifiers. They assume that the attributes of the records of the same real-world entity may be represented in different ways due to typing error, etc., and *all* attributes of the records can be used to decide whether or not the records

Symbol	Description
S	Sequence of historical relations
R_n	New relation
$\mathcal{K}(R)$	Set of keys of relation R
$\mathcal{MK}(R)$	Set of minimal keys of R
$\mathcal{C}(S)$	Set of common keys of S
$\mathcal{MC}(S)$	Set of minimal common keys of S
$\mathcal{P}(S, C)$	Set of proxy keys of S over C
$\mathcal{MP}(S, C)$	Set of minimal proxy keys of S over C
$\mathcal{T}(R, C)$	Set of non-trivial sub-relations of R over C
T^p	The projected sub-relation of $T \in \mathcal{T}(R, C)$

TABLE I
SYMBOLS.

refer to the same entity. However, in our problem this is not possible as *some of the attributes of the entities are evolving*.

Query optimization. Consider the archive in Figure 2. Suppose a user wishes to find all *distinct titles* with number of *bids* less than 25 in the last two weeks (R_1 and R_2). To execute this query, the query processor needs to sort the records in R_1 and R_2 based on the attribute $title$ in order to eliminate duplicates. Since $title$ is a key of R_2, the query optimizer may rewrite the SQL query by removing the *distinct* clause. As a result, duplicate elimination can be avoided. However, observe that it is not a key in R_1. Consequently, the above rewriting strategy will generate incorrect results for R_1 (if we assume keys are temporally invariant). In contrast, the knowledge of common keys of R_1 and R_2 can accurately guide the query processor to determine when to avoid duplicate elimination. In the aforementioned example, it cannot be avoided as *title* is not a common key. However, if one wishes to find *distinct* (*title*, *bid*) pairs then duplicate elimination can be avoided as the pair is a common key. Common keys are also useful in *order optimization*. The readers may refer to [2] for details.

III. PRELIMINARIES

We begin by discussing our strategy for modeling a sequence of versions of structured continuous query results. Then, we formally introduce the notion of common keys. Finally, we define the problem that we address in this paper. The set of symbols used in this paper is summarized in Table I.

A. Model of Structured Web Data Sequence

We represent structured continuous query results from an autonomous deep Web source Q as a pair (Q, Γ), where $\Gamma = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ is the timestamp sequences recording the times when data was retrieved from Q periodically. We take an *entity view* of the data of Q. We consider query results from Q at time τ_i as primarily a set of entities: $G_{Q_{\tau_i}} = \{G_1, G_2, G_3, \ldots, G_r\}$. Each entity $G_j \in G_{Q_{\tau_i}}$ consists of a set of elements E_j where each element $e \in E_j$ is a pair (a, v) where a is the *attribute* and v is the *literal value* (possibly empty) of e. We say that two entities $G_\ell \in G_{Q_{\tau_i}}$ and $G_m \in G_{Q_{\tau_j}}$ $\forall 1 \leq i < j \leq n$ are *identical*, denoted as $G_\ell = G_m$, if they represent the same real-world entity.

Let A be the set of all attributes for $G_{Q_{\tau_i}}$. Then, $G_{Q_{\tau_i}}$ can be stored in a relational table $R_i(a_1, a_2, \ldots, a_{|A|})$ where $a_k \in A$ $\forall 1 \leq k \leq |A|$ and each record u_j represents an entity $G_j \in G_{Q_{\tau_i}}$. As Q disseminates data periodically over a time period, the collection of historical data of Q can be represented as a sequence $\langle G_{Q_{\tau_1}}, G_{Q_{\tau_2}}, \ldots, G_{Q_{\tau_n}} \rangle$ where $\tau_1 < \tau_2 < \ldots < \tau_n$. Also, we assume that $A_i = A_j$ $\forall 1 \leq i < j \leq n$ where A_i and A_j are the sets of attributes for $G_{Q_{\tau_i}}$ and $G_{Q_{\tau_j}}$, respectively. Consequently, data from Q can be represented as a sequence of relational tables $S = \langle R_1(a_1, a_2, \ldots, a_{|A|}), R_2(a_1, a_2, \ldots, a_{|A|}), \ldots, R_n(a_1, a_2, \ldots, a_{|A|}) \rangle$. In the sequel, the set of attributes (schema) of R_i $(a_1, a_2, \ldots, a_{|A|})$ is omitted if it is understood in the context.

B. Common Key

Informally, a key in relation R_i is a *common key* iff it is a key for all the versions of the relation in the sequence S.

Definition 1: (**Common Key**). *Let $S = \langle R_1, R_2, \ldots, R_m \rangle$ be a sequence of relations from a Web source Q with a set of attributes A. Let $C \subseteq A$ and $C \neq \emptyset$. Then, C is a **common key** of S iff C is a key of R_i $\forall 1 \leq i \leq m$. C is a **minimal common key** of S if and only if C is a common key of S and none of its proper subset is also a common key of S.*

We denote the sets of common keys and minimal common keys of S as $\mathcal{C}(S)$ and $\mathcal{MC}(S)$, respectively. For example, consider the relations R_1 and R_2 shown in Figure 1. Let $S = \langle R_1, R_2 \rangle$. The set of attributes $\{name, children\}$ is a common key of S since it is a key of both R_1 and R_2. This set of attributes is also a minimal common key of S since none of its proper subsets (i.e., $\{name\}$ and $\{children\}$) is also a common key of S. Also, $\mathcal{MC}(S) = \{\{name, birthdate\}, \{name, children\}, \{name, income\}\}$.

C. Common Key Maintenance Problem

Let $S = \langle R_1, R_2, \ldots, R_n \rangle$ be a sequence of relations from a deep Web source Q with a set of attributes A. Note that the keys of R_1 can be computed using an existing data-driven key discovery algorithm. Then for each relation R_i where $i > 1$, we can compute the common keys of the new sequence of relations that incorporates the new relation R_i by examining the common keys of the "old" sequence, and selecting those that are the keys of R_i.

Definition 2: **(Common Key Maintenance Problem)** *Let $S_o = \langle R_1, R_2, \ldots, R_m \rangle$ be the sequence of relations from source Q at times $\langle \tau_1, \tau_2, \ldots, \tau_m \rangle$. Let R_n be the new relation from Q at time $\tau_n > \tau_m$. Given $\mathcal{MC}(S_o)$ and R_n, the problem of* **common key maintenance** *is to find $\mathcal{MC}(S_n)$ of the updated sequence $S_n = \langle R_1, R_2, \ldots, R_m, R_n \rangle$.*

IV. ALGORITHM COKE

In this section, we propose an efficient algorithm, called COKE (**CO**mmon **K**ey Maintenanc**E**), for computing $\mathcal{MC}(S_n)$. We restrict this algorithm to in-memory processing where $\mathcal{MC}(S_o)$ and R_n reside in memory. Due to space constraints, the proofs of theorems and lemmas presented in the sequel are given in [2]. We begin by defining *proxy key*.

Definition 3: (**Proxy Key**) *Given a R with attributes A, let $P \subseteq A$, $Y \subseteq A$, $P \neq \emptyset$ and $Y \neq \emptyset$. Then, P is a **proxy key** of R over Y if and only if P is a key of R and $P \supseteq Y$. P is a **minimal proxy key** of R over Y if and only if P is a proxy key of R over Y and none of its proper subset is also a proxy key of R over Y.*

Algorithm 1: Algorithm COKE

Input: $\mathcal{MC}(S_o)$, R_n
Output: $\mathcal{MC}(S_n)$
1 $\mathcal{T}(R_n, \cdot) \leftarrow$ SubRelationConst($\mathcal{MC}(S_o), R_n$);
2 $\mathcal{MC}(S_n) \leftarrow$ CommonKeyGen($\mathcal{MC}(S_o), \mathcal{T}(R_n, \cdot)$);
3 **return** $\mathcal{MC}(S_n)$

We denote the sets of proxy keys and minimal proxy keys of R over Y as $\mathcal{P}(R, Y)$ and $\mathcal{MP}(R, Y)$, respectively. For example, consider the relation R_1 shown in Figure 1. The set of attributes $\{name, children\}$ is a proxy key of R_1 over $\{name\}$ since it is a key of R_1 and it is a superset of $\{name\}$. This set of attributes is also a minimal proxy key of R_1 over $\{name\}$ since none of its proper subsets (i.e., $\{name\}$ and $\{children\}$) is also a proxy key of R_1 over $\{name\}$. Also, $\mathcal{MP}(R_1,\{name\}) = \{name, children\}$, $\{name, income\}$, $\{name, birthdate\}\}$.

A. Overview

The number of possible common keys is exponential to the number of attributes. To tackle this problem, we break the search space into several sub-search spaces, where each sub-search space corresponds to each minimal common key $C \in \mathcal{MC}(S_o)$, and consists of all the supersets of C. For each sub-search space, we compute its $\mathcal{MP}(R_n, C)$ by first splitting R_n based on C into a set of *non-trivial sub-relations* (denoted by $\mathcal{T}(R_n, C)$). Then, we compute $\mathcal{MP}(R_n, C)$ from $\mathcal{T}(R_n, C)$. After that, from these sets of minimal proxy keys of the sub-search spaces, we compute $\mathcal{MC}(S_n)$ efficiently. We now formally define *non-trivial sub-relations*.

Definition 4: **(Sub-relation)** *Give a relation R with attributes A, let $C \subseteq A$ and $C \neq \emptyset$. Let $G = \{T_1, T_2, \ldots, T_n\}$ be a collection of sets of records of R. Then, G is a* ***set of sub-relations*** *of R over C if and only if G satisfies the following four conditions: (a) $R = T_1 \cup T_2 \cup \ldots \cup T_n$, (b) $T_i \cap T_j = \emptyset$ $\forall 1 \leq i < j \leq n$, (c) $u[C] = v[C]$ $\forall u, v \in T_i$ $\forall 1 \leq i \leq n$, and (d) $u[C] \neq v[C]$ $\forall u \in T_i, v \in T_j$ $\forall 1 \leq i < j \leq n$. We call $T_i \in G$ as a* ***trivial sub-relation*** *if $|T_i| = 1$, and as a* ***non-trivial sub-relation*** *if $|T_i| > 1$.*

For example, consider R_1 shown in Figure 1. The collection of sets of records $\{\{e_1, e_4\}, \{e_2\}, \{e_3\}, \{e_5\}\}$ is a set of sub-relations of R_1 over $\{name\}$ since it satisfies all the four conditions above. The sub-relation $\{e_2\}$ is a trivial sub-relation since it contains only one record, while the sub-relation $\{e_1, e_4\}$ is a non-trivial sub-relation since it contains more than one record.

The above idea of computing $\mathcal{MC}(S_n)$ is realized in COKE by the following two phases as shown in Algorithm 1. The *non-trivial sub-relations computation phase* computes sets of non-trivial sub-relations $\mathcal{T}(R_n, \cdot)$ from $\mathcal{MC}(S_o)$ and R_n (Line 1). The *common keys computation phase* then computes $\mathcal{MC}(S_n)$ from $\mathcal{MC}(S_o)$ and $\mathcal{T}(R_n, \cdot)$ (Line 2). For ease of detailed exposition of these phases, we use the following running example.

Example 1: Consider the relations in Figure 1. Let S_o = $\langle R_1 \rangle$ and R_n = R_2. Also, $\mathcal{MC}(S_o)$ = $\{\{birthdate\}$, $\{name, children\}$, $\{name, income\}$, $\{children, income\}$, $\{children, city\}\}$. ∎

Algorithm 2: SubRelationConst

Input: $\mathcal{MC}(S_o)$, R_n
Output: $\mathcal{T}(R_n, \cdot)$
1 $root \leftarrow$ call PartitioningPlan($\mathcal{MC}(S_o)$);
2 **foreach** $w \in root.children$ **do**
3 SubRelationStub($R_n, w, \{w.label\}$);
4 **return** $\mathcal{T}(R_n, \cdot)$
5 **procedure**: SubRelationStub(T, w, C);
6 $G \leftarrow$ partition T based on $w.label$;
7 **if** w *is a leaf* **then**
8 **foreach** *non-trivial sub-relation* $T' \in G$ **do**
9 $\mathcal{T}(R_n, C) \leftarrow \mathcal{T}(R_n, C) \cup \{T'\}$;
10 **else**
11 **foreach** *non-trivial sub-relation* $T' \in G$ **do**
12 **foreach** $w' \in w.children$ **do**
13 SubRelationStub($T', w', C \cup \{w'.label\}$);

B. Non-trivial Sub-relations Computation (Phase 1)

Sub-relation Computation Using Partitioning Plan. One method for computing $\mathcal{T}(R_n, \cdot)$ from $\mathcal{MC}(S_o)$ and R_n is that, for each $C \in \mathcal{MC}(S_o)$ we can partition the records in R_n recursively based on each attribute in C. However, the computation cost can be further reduced if some of the intermediate partitioning steps can be shared among the minimal common keys in $\mathcal{MC}(S_o)$. For example, the minimal common keys $\{name, children\}$ and $\{children, income\}$ share the attribute *children*, and thus they can share some of the intermediate partitioning steps. We explain our main idea below. First, we compute $\mathcal{T}(R_n, \{children\})$ by partitioning the records in R_n based on the attribute $children$. Second, we compute $\mathcal{T}(R_n, \{name, children\})$ by partitioning the records in each sub-relation in $\mathcal{T}(R_n, \{children\})$ based on the $name$. Third, we compute $\mathcal{T}(R_n, \{children, income\})$ by partitioning the records in each sub-relation in $\mathcal{T}(R_n, \{children\})$ based on $income$.

We formalize the above steps of computing $\mathcal{T}(R_n, \cdot)$ by using a *partitioning plan*. The partitioning plan is a tree in which each of its nodes, except the root node, is labeled with an attribute, and each of its paths from the root to the leaf node represents each minimal common key $C \in \mathcal{MC}(S_o)$, and vice versa. For example, consider the $\mathcal{MC}(S_o)$ and R_n in Example 1. Then, one possible partitioning plan is shown in Figure 5. Notice that the minimal common key $\{name, children\}$ is represented as the path of "$children \rightarrow name$", and the path of "$children \rightarrow income$" represents the minimal common key $\{children, income\}$ (the root node is not included in the path since it is unlabeled).

Algorithm 2 summarizes the steps of computing $\mathcal{T}(R_n, \cdot)$ using the partitioning plan approach. First, we create a partitioning plan from $\mathcal{MC}(S_o)$ (Line 1). We will discuss the method of creating the partitioning plan in the next subsection.

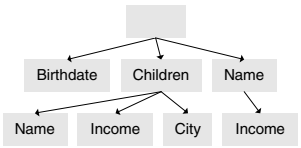


Fig. 5. Partitioning plan computed from $\mathcal{MC}(S_o)$.

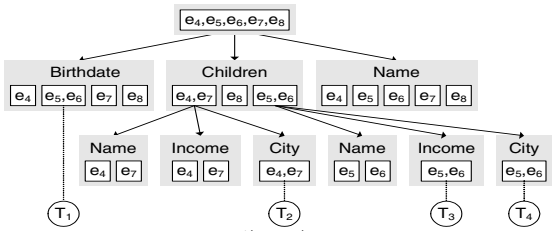


Fig. 6. Computation of $\mathcal{T}(R_n, \cdot)$ using the partitioning plan.

The algorithm starts from the root setting R_n as the *current relation*, and recursively visits each of the child nodes (Lines 2-3). At each node, except the root, it partitions the records in the current relation into multiple sub-relations based on the node's label (Line 6). It then checks whether the current node is a leaf node (Line 7). If it is, then it adds all non-trivial sub-relations into $\mathcal{T}(R_n, C)$ where C is the minimal common key represented by the path from the root to the current node (Lines 8-9). If it is not, for each non-trivial sub-relation the algorithm sets it as the current relation, and again recursively visit each of the child nodes (Lines 11-13). Note that the algorithm does not continue the traversal of the partitioning plan for the trivial sub-relations because at the leaf node such sub-relations will not be added into $\mathcal{T}(R_n, C)$. We shall justify the reason for not adding the trivial sub-relations into $\mathcal{T}(R_n, C)$ in Section IV-C.

LEMMA *1:* $C \in \mathcal{K}(R_n)$ *if and only if* $\mathcal{T}(R_n, C) = \emptyset$.

Example 2: The partitioning plan computed from $\mathcal{MC}(S_o)$ is shown in Figure 5. The process of computing $\mathcal{T}(R_n, \cdot)$ using the partitioning plan is shown in Figure 6. Each box in each node represents a sub-relation in that node, and each number in each box represents a record (represented by its *EID*) in that sub-relation.

We illustrate the computation using $\mathcal{T}(R_n, \{children, income\})$ as an example. We start from the root node, set R_n as the current relation, and visit the node $children$. We partition the current relation, i.e., R_n, based on the attribute $children$, and obtain three sub-relations, i.e., $\{e_4, e_7\}$, $\{e_5, e_6\}$, and $\{e_8\}$. Consider the sub-relation $\{e_4, e_7\}$. Since it is non-trivial, we set it as the current relation, and visit the node $income$. We partition the current relation, i.e., $\{e_4, e_7\}$, based on $income$ and obtain two sub-relations, i.e., $\{e_4\}$ and $\{e_7\}$. Since we are at the leaf node and the sub-relations $\{e_4\}$ and $\{e_7\}$ are trivial, we do not add them into $\mathcal{T}(R_n, \{children, income\})$. Now consider $\{e_5, e_6\}$. Since it is non-trivial, we set it as the current relation, and visit the node $income$. We partition the current relation based on $income$ and obtain one sub-relation, i.e., $\{e_5, e_6\}$. Since we are at the leaf node and the sub-relation $\{e_5, e_6\}$ is non-trivial, we add it into $\mathcal{T}(R_n, \{children, income\})$ (denoted as T_3 in Figure 6). Lastly, we consider $\{e_8\}$. Since this is a trivial sub-relation, we prune it. Thus, $\mathcal{T}(R_n, \{children, income\}) = \{T_3\}$. Notice that the node $income$, the child of the node $name$, in the partitioning plan is not traversed since every sub-relation in the node $name$ is trivial. Also, the set of sub-relations $\mathcal{T}(R_n, \{children, name\})$ is an empty set and the corresponding minimal common key $\{children, name\}$ is in $\mathcal{K}(R_n)$. This agrees with Lemma 1. The remaining sets of sub-relations are $\mathcal{T}(R_n, \{birthdate\}) = \{T_1\}$ and $\mathcal{T}(R_n, \{children, city\}) = \{T_2, T_4\}$. ∎

Algorithm 3: PartitioningPlan

Input: $\mathcal{MC}(S_o)$
Output: $root$ (root of the partitioning plan)

```
1  root ← new node;
2  foreach C ∈ MC(S_o) do
3      foreach a ∈ C do
4          a.frequency ← a.frequency + 1;
5  foreach C ∈ MC(S_o) do
6      sort the attributes in C according to decreasing frequency;
7  foreach C ∈ MC(S_o) do
8      w ← root;
9      foreach a ∈ C in order of decreasing frequency do
10         w' ← find the node with label a in w.children;
11         if w' does not exist then
12             w' ← new node;
13             w'.label ← a;
14             w.children ← w.children ∪ {w'};
15         w ← w';
16 return root
```

Partitioning Plan Construction. Our goal is to construct the partitioning plan such that the computation cost required to construct it and to compute $\mathcal{T}(R_n, \cdot)$ is minimum. This problem is challenging as in one hand, we need to take into account the pruning of the sub-relations and on the other hand, we cannot preprocess R_n too much since the construction of the plan may be too costly. We propose to create a plan that contains the minimum number of nodes and ignore the pruning of the sub-relations. We choose this strategy because it can be computed only from the set of minimal common keys of S_o. Further, a plan that contains lesser number of nodes is likely to reduce the total computation cost.

Theorem 1: The problem of constructing partitioning plan containing minimum number of nodes from $\mathcal{MC}(S_o)$ is NP-Hard.

We develop efficient heuristics to construct the partition plan. The basic idea is inspired by the FP-Tree construction algorithm [10]. The objective is to assign the attributes that exist in large number of minimal common keys in $\mathcal{MC}(S_o)$ as close as possible to the root node in order to maximize the number of shared nodes. Consequently, the number of nodes in the plan is minimized. The algorithm is shown in Algorithm 3 and consists of the following three steps. Firstly, it computes the *frequency* of each attribute. The *frequency* of an attribute is defined as the number of minimal common keys in $\mathcal{MC}(S_o)$ that contains the attribute (Lines 2-4). Secondly, it sorts the attributes in each $C \in \mathcal{MC}(S_o)$ based on the descending order of frequency (Lines 5-6). Attributes with same frequency

EID	Name	City	Children	Income
e_5	Dave	Houston	2	30000
e_6	Eve	Houston	2	30000

(a) T_1^p

EID	Name	Birthdate	Income
e_4	Alice	4 Apr 77	40000
e_7	Bob	6 Jun 75	20000

(b) T_2^p

EID	Name	Birthdate	City
e_5	Dave	5 May 76	Houston
e_6	Eve	5 May 76	Houston

(c) T_3^p

EID	Name	Birthdate	Income
e_5	Dave	5 May 76	30000
e_6	Eve	5 May 76	30000

(d) T_4^p

Fig. 7. Projected sub-relations.

Algorithm 4: CommonKeyGen

Input: $\mathcal{MC}(S_o)$, $\mathcal{T}(R_n, \cdot)$
Output: $\mathcal{MC}(S_n)$

1 **foreach** $C \in \mathcal{MC}(S_o)$ **do**
2 **foreach** $T \in \mathcal{T}(R_n, C)$ **do**
3 $T^p \leftarrow$ create a projection of T;
4 $\mathcal{MK}(T^p) \leftarrow$ call key discovery algorithm on T^p;
5 $\mathcal{MP}(T, C) \leftarrow \bigcup_{K \in \mathcal{MK}(T^p)} \{K \cup C\}$;
6 **foreach** $C \in \mathcal{MC}(S_o)$ **do**
7 **if** $\mathcal{T}(R_n, C) \neq \emptyset$ **then**
8 $\mathcal{MP}(R_n, C) \leftarrow MIN\left(\bigotimes_{T \in \mathcal{T}(R_n, C)} \mathcal{MP}(T, C)\right)$;
9 **else**
10 $\mathcal{MP}(R_n, C) \leftarrow \{C\}$;
11 $\mathcal{MC}(S_n) \leftarrow MIN\left(\bigcup_{C \in \mathcal{MC}(S_o)} \mathcal{MP}(R_n, C)\right)$;
12 **return** $\mathcal{MC}(S_n)$

are ordered arbitrarily. Thirdly, for each $C \in \mathcal{MC}(S_o)$, the algorithm creates a path from the root to a leaf node, and share its prefix nodes maximally with other paths having the same prefix (Lines 7-15).

Example 3: Consider Example 2. The frequency of $name$ is two since there are two minimal common keys (i.e., $\{name, children\}$ and $\{name, income\}$) that contain this attribute. The frequencies of $children$, $income$, $birthdate$, and $city$ are three, two, one, and one, respectively. Hence, $children > name > income > birthdate > city$. After sorting the attributes in each $C \in \mathcal{MC}(S_o)$, we obtain $\mathcal{MC}(S_o)$ = $\{\{birthdate\}, \{children, name\}, \{name, income\}, \{children, income\}, \{children, city\}\}$. Note that the partitioning plan depends on the ordering of attributes. The partitioning plan constructed from this $\mathcal{MC}(S_o)$ is shown in Figure 5. Notice that the minimal common key $\{name, income\}$ is represented as the path $name \rightarrow income$. Furthermore, the paths $children \rightarrow name$, $children \rightarrow income$, and $children \rightarrow city$ share the prefix node $children$. ∎

Observe that in the aforementioned approach we continue partitioning the new sub-relation until we reach the leaf node of the plan. This is because we want to compute the sets of sub-relations over the minimal common keys where each minimal common key is represented as a path from the root to the leaf node. Hence, if we stop the partitioning before reaching the leaf node, the resulting sets of sub-relations will not be over the minimal common keys.

C. Common Keys Computation (Phase 2)

We compute the set of minimal common keys of the updated sequence S_n (denoted as $\mathcal{MC}(S_n)$) from $\mathcal{T}(R_n, \cdot)$, which is generated in the previous phase. Note that if the common keys remain unchanged in the new sequence then non-trivial sub-relation set is empty (Lemma 1). Hence, in this case Phase 2 does not need to be executed. We begin by defining two operators, namely *minimization* and *pairwise union*, that we shall be using later.

Let X be a collection of sets of elements. The *minimization* of X, denoted by $MIN(X)$, is defined as the collection of every set in X whose none of its proper subset is also in X, i.e., $\{x | x \in X \wedge \nexists y \in X \text{s.t } y \subset x\}$. For example, let X = $\{\{name\}, \{birthdate\}, \{name, city\}\}$. Then, $MIN(X)$ is $\{\{name\}, \{birthdate\}\}$. Notice that $\{name, city\}$ is not in $MIN(X)$ since one of its proper subset ($\{name\}$) is in X.

Let X and Y be collections of sets of elements. The *pairwise union* of X and Y, denoted by $X \otimes Y$, is defined as the collection of the union of every set in X with every set in Y, i.e., $\{x \cup y | x \in X \text{ and } y \in Y\}$. For example, let X = $\{\{name\}, \{birthdate\}\}$ and Y = $\{\{name\}, \{city\}\}$. Then, $X \otimes Y$ is $\{\{name\}, \{name, birthdate\}, \{name, city\}, \{birthdate, city\}\}$. It follows from its definition that pairwise union operator is commutative (i.e., $X \otimes Y = Y \otimes X$) and associative (i.e., $X \otimes (Y \otimes Z) = (X \otimes Y) \otimes Z$). For simplicity, we use the notation $\bigotimes_{i=1}^{n} X_i$ as a short form of $X_1 \otimes X_2 \otimes \cdots \otimes X_n$.

Algorithm of Phase 2. The steps for Phase 2 is presented in Algorithm 4. It consists of three key steps as follows.

Step 1: Computation of Proxy Keys of Sub-Relations. Given $C \in \mathcal{MC}(S_o)$ it computes $\mathcal{MP}(T, C)$ from $T \in \mathcal{T}(R_n, C)$. First, it creates the *projected sub-relation* T^p from T by removing all the attributes in C from the attributes of T (Line 3). Second, it computes the minimal keys of T^p ($\mathcal{MK}(T^p)$) using an existing key discovery algorithm (Line 4). Third, $\mathcal{MP}(T, C)$ is computed by taking the union of attributes in C and $K \in \mathcal{MK}(T^p)$ (Line 5).

Example 4: We illustrate the computation of $\mathcal{MP}(T_1, \{birthdate\})$. We create the projected sub-relation T_1^p from T_1 by removing the attribute $birthdate$ from T_1 as shown in Figure 7(a). By using an existing key discovery algorithm, we obtain $\mathcal{MK}(T_1^p)$ = $\{\{name\}\}$. Then, we can obtain $\mathcal{MP}(T_1, \{birthdate\})$ = $\{\{name\} \cup \{birthdate\}\}$ = $\{\{name, birthdate\}$.

Similarly, we can compute the projected sub-relations of T_2, T_3, and T_4. These are depicted in Figures 7(b)-(d) (denoted as T_2^p, T_3^p, and T_4^p, respectively). The sets of minimal proxy keys of the sub-relations are $\mathcal{MP}(T_2, \{children, city\})$ = $\{\{name, children, city\}, \{income, chidren, city\}, \{birthdate, children, city\}\}$, $\mathcal{MP}(T_3, \{children, income\})$ = $\{\{name, children, income\}\}$, and $\mathcal{MP}(T_4, \{children, city\})$ = $\{\{name, children, city\}\}$. ∎

Step 2: Computation of Proxy Keys of R_n**.** Given $C \in \mathcal{MC}(S_o)$, in this step our goal is to compute the set of minimal proxy keys of R_n over C (denoted as $\mathcal{MP}(R_n, C)$) from

$\mathcal{MP}(\cdot, C)$. Note that $\mathcal{MP}(\cdot, C)$ has already been computed in the preceding step. We consider the following two cases: (a) If $\mathcal{T}(R_n, C) = \emptyset$. Based on Lemma 1, it follows that $C \in \mathcal{K}(R_n)$. Consequently, $\mathcal{MP}(R_n, C) = \{C\}$ (Line 10). (b) If $\mathcal{T}(R_n, C) \neq \emptyset$, then we compute $\mathcal{MP}(R_n, C)$ by using the following theorem (Line 8). Intuitively, for each C we first apply the pairwise union on the proxy key sets of all the non-trivial sub-relations ($\mathcal{MP}(T, C)$). Then, we extract the minimal proxy keys by removing the supersets (using minimization).

Theorem 2: Let $C \in \mathcal{MC}(S_o)$. *Given* $\mathcal{MP}(\cdot, C)$, $\mathcal{MP}(R_n, C) = MIN\left(\bigotimes_{T \in \mathcal{T}(R_n, C)} \mathcal{MP}(T, C)\right)$

Note that the above theorem holds regardless of whether or not we add trivial sub-relations into $\mathcal{T}(R_n, C)$. This is the reason why we do not add trivial sub-relations in $\mathcal{T}(R_n, C)$ as discussed in Algorithm 2 (Line 8). Also, as the number of elements in $\bigotimes_{T \in \mathcal{T}(R_n, C)} \mathcal{MP}(T, C)$ is exponential to the number of sub-relations in $\mathcal{T}(R_n, C)$, we propose a more efficient recursive method based on the following lemma. It reduces the number of elements in the intermediate computation by applying minimization operator each time we apply pairwise union operator to remove unnecessary elements from its results.

LEMMA *2: Let* $\{X_1, X_2, \ldots, X_n\}$ *be a collection of sets of elements. Then,* $MIN\left(\bigotimes_{i=1}^{n} X_i\right) = MIN\left(X_1 \otimes MIN\left(\bigotimes_{i=2}^{n} X_i\right)\right)$

Example 5: We first illustrate the computation of $\mathcal{MP}(R_n, \{name, children\})$. Since $\mathcal{T}(R_n, \{name, children\}) = \emptyset$, we set $\mathcal{MP}(R_n, \{name, children\}) = \{\{name, children\}\}$. Let us now illustrate $\mathcal{MP}(R_n, \{children, city\})$. Since $\mathcal{T}(R_n, \{children, city\}) = \{T_2, T_4\}$, we compute $\mathcal{MP}(R_n, \{children, city\})$ from $\mathcal{MP}(T_2, \{children, city\})$ and $\mathcal{MP}(T_4, \{children, city\})$. Using Theorem 2, we obtain $\mathcal{MP}(R_n, \{children, city\}) = MIN(\{\{name, children, city\}, \{income, children, city\}, \{birthdate, children, city\}\} \otimes \{\{name, children, city\}\}) = \{\{name, children, city\}\}$. Similarly, we can compute remaining minimal proxy key sets: $\mathcal{MP}(R_n, \{name, income\}) = \{\{name, income\}\}$, $\mathcal{MP}(R_n, \{birthdate\}) = \{\{name, birthdate\}\}$ and $\mathcal{MP}(R_n, \{children, income\}) = \{\{name, children, income\}\}$. ∎

Step 3: Computation of Common Keys of S_n**.** Finally, we compute the set of minimal common keys in S_n ($\mathcal{MC}(S_n)$) from $\mathcal{MP}(R_n, \cdot)$ generated from the preceding step by applying the minimization operator on the sets of proxy keys (Line 11 in Algorithm 4). Note that the minimization operation ensures removal of super keys from this set.

Example 6: We obtain $\mathcal{MC}(S_n) = MIN(\{\{name, children\}\} \cup \{\{name, income\}\} \cup \{\{name, birthdate\}\} \cup \{\{name, children, income\}\} \cup \{\{name, children, city\}\}) = \{\{name, children\}, \{name, income\}, \{name, birthdate\}\}$. ∎

D. Complexity Analysis

We now summarize the time and space complexities of COKE with respect to the parameters shown in Table II. Let

Parameter	Description	Default value
N_s	# attributes in R_n	20
N_t	# records in R_n	50,000
N_k	# minimal common keys in $\mathcal{MC}(S_o)$	30
N_a	# attributes in each minimal common key in $\mathcal{MC}(S_o)$	5
N_u	# minimal common keys in $\mathcal{MC}(S_o)$ whose set of non-trivial sub-relations is not empty	10
N_p	# non-trivial sub-relations in R_n associated with each minimal common key in $\mathcal{MC}(S_o)$ whose set of non-trivial sub-relations is not empty	20
N_q	# records in each non-trivial sub-relation in R_n associated with each minimal common key in $\mathcal{MC}(S_o)$ whose set of non-trivial sub-relations is not empty	10

TABLE II
PARAMETERS FOR ANALYSIS.

$O(\mathbb{F}(R))$ and $O(\mathbb{G}(R))$ be the time and space complexities, respectively, of computing $\mathcal{MK}(R)$ using key discovery algorithm. Then,

Theorem 3: The time and space complexities of the COKE *algorithm are* $O(N_k N_a N_t \log N_t + N_u N_p \mathbb{F}(T^p))$ *and* $O(N_s N_t + \mathbb{G}(T^p))$*, respectively.*

Remark. Recall that we traverse the partitioning plan in a depth first fashion. The space that we require in each node in the partitioning plan is at most N_t. We only store the identifier of the records. Since the depth of the partitioning plan is at most N_a, then the space that we require is at most $N_t N_a$. In contrast, the space that we require to store the new relation is $N_t N_s$. Since $N_s > N_a$, then the space that we require to generate the sets of sub-relations is less than the space required for storing the new relation. Furthermore, the space required to *store* the sets of sub-relations is $N_u N_p N_q$. Note that one record can be in multiple sub-relations in multiple minimal common keys. Since N_u, N_p, and N_q are small, this space is less than the space required to store the new relation.

In COKE, a data-driven key discovery algorithm is executed against projected sub-relation T^p. Since T^p has smaller number of attributes and significantly smaller number of records than R_n, the cost of each execution of key discovery algorithm in COKE is much cheaper. However, in COKE, key discovery algorithm is executed $N_u N_p$ times, which is the total number of non-trivial sub-relations in R_n. Thus, if $N_u N_p$ is sufficiently large, it is possible that the total cost of execution of key discovery algorithm in COKE can be expensive. However, as we shall show in the next section, such scenario is improbable in real-world data sets.

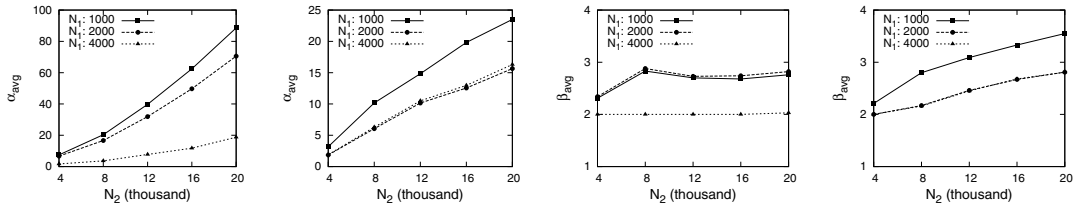
V. ANOREXIC NATURE OF SUB-RELATIONS

Recall that we made the assumption that the number of non-trivial sub-relations in R_n and the number of their records are relatively small. This "anorexic" nature of the sub-relations reduces the total execution cost of a key discovery algorithm. Hence, a critical question that needs to be answered is *whether this assumption is practical for evolutionary real-world continuous query results?* In this section, we show that it is indeed the case.

Factors affecting anorexic behavior. In general, there are two factors that affect the extent to which anorexic characteristics are present in relation R_n: (a) the domain size of the minimal common keys in $\mathcal{MC}(S_o)$, and (b) the number of

	AMZN1	AMZN2	CNET1	CNET2	DIGG1	DIGG2	EBAY1	EBAY2
Web site	amazon.com		cnet.com		digg.com		ebay.com	
Query	mp3	digital camera	camera	laptop	tech	world	ipod touch	laptop computer
# records in R_1 (10th Nov, 2008)	3051	3708	10100	10100	10200	20400	10489	21950
# records in R_2 (17th Nov, 2008)	3996	3998	10200	10100	10200	20400	10635	21951
# attributes in R_1 and R_2	13	14	24	25	18	18	30	29

TABLE III
REAL-WORLD DATA SETS.



(a) DIGG2 (b) EBAY2 (c) DIGG2 (d) EBAY2

Fig. 8. Anorexic nature of real-world data sets.

records in R_n. Given two relations R_1 and R_2, we can show the effect of these two factors using the following method. We sample N_1 records from R_1, compute the set of minimal keys of these N_1 records, and use it as $\mathcal{MC}(S_o)$. We can vary the first factor by varying N_1. Specifically, we can increase the domain size of the minimal common keys by increasing N_1. This is because in a relation with large number of records, it is more likely to get duplicate records for sets of attributes with smaller domain size than with larger domain size, and thus, the minimal keys of this large relation are more likely to have larger domain size. Next, we sample N_2 records from R_2, and use it as R_n. We can vary the second factor by varying N_2.

We study the effect of the above factors by analyzing two parameters: α_{avg} and β_{avg}. Let $\widehat{\mathcal{MC}}(S_o)$ be the set of minimal common keys of S_o whose set of non-trivial sub-relations is not empty, i.e., $\widehat{\mathcal{MC}}(S_o) = \{C | C \in \mathcal{MC}(S_o)$ and $\mathcal{T}(R_n, C) \neq \emptyset\}$. Then, α_{avg} is defined as the average number of non-trivial sub-relations associated with each minimal common key whose set of non-trivial sub-relations is not empty, i.e.,

$$\alpha_{avg} = \frac{\sum_{C \in \widehat{\mathcal{MC}}(S_o)} |\mathcal{T}(R_n, C)|}{|\widehat{\mathcal{MC}}(S_o)|}$$

β_{avg} is defined as the average number of records in each non-trivial sub-relation, i.e.,

$$\beta_{avg} = \frac{\sum_{C \in \widehat{\mathcal{MC}}(S_o)} \sum_{T \in \mathcal{T}(R_n, C)} |T|}{\sum_{C \in \widehat{\mathcal{MC}}(S_o)} |\mathcal{T}(R_n, C)|}$$

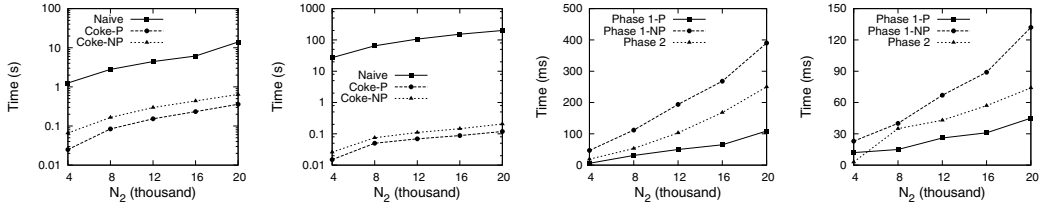
Analysis of α_{avg} and β_{avg}. We applied the above method on real-world data sets DIGG2 and EBAY2 (Table III). Due to space constraints, the results of the remaining data sets are given in [2]. Each data set is obtained by submitting the query shown in row "*Query*" to the deep Web site shown in row "*Web site*" using the API provided by the site. For each data set, we submitted the same query to the same site on $10th$ and $17th$ November, 2008. Thus, for each data set we have two relations representing R_1 and R_2, respectively. The number of attributes and records in these relations are shown in rows "*# attributes*" and "*# records*", respectively. Figure 8 shows the variability of α_{avg} and β_{avg} for these data sets. We can see that α_{avg} and β_{avg} tend to decrease and increase with the increase in N_1 and N_2, respectively. This is due to the fact that when the domain size of the minimal common keys is large, it is less likely that there are records in R_n with the same value for all attributes in these minimal common keys. Also, when the number of records in R_n is large, it is more likely that there are records with the same value for all attributes in these minimal common keys.

It is interesting to observe that the size of each non-trivial sub-relation not only grows very slowly with respect to the number of records in R_n, but also its value is very small. This is beneficial for COKE as the computation cost of the key discovery algorithm on these sub-relations becomes inexpensive. The number of non-trivial sub-relations however increases roughly linear to the number of records in R_n. This can be expensive in COKE since the number of executions of the key discovery algorithm is linear to the number of non-trivial sub-relations. However, when the domain of the minimal common keys is large, the number of non-trivial sub-relations is relatively small. In many real-world data sets, this is true since the minimal common keys are computed from a large relation.

VI. PERFORMANCE EVALUATION

We have implemented two variants of COKE algorithm, namely COKE-P and COKE-NP. COKE-P uses partitioning plan in computing the sets of non-trivial sub-relations, while COKE-NP *does not*. The purpose of implementing COKE-NP is to study the effectiveness of using partitioning plan in computing the sets of non-trivial sub-relations. Recall that the execution time of Phase 1 of COKE is affected by the usage of partitioning plan. Hence, we distinguish between the Phase 1 of COKE-P and COKE-NP. We denote the Phase 1 of COKE-P and COKE-NP as Phase 1-P and Phase 1-NP, respectively. The Phase 2 of these two variants, which is not affected by the partitioning plan, is denoted as Phase 2.

We also implemented a baseline algorithm to compute common keys, called NAÏVE. In this approach, we first compute $\mathcal{MK}(R_n)$ from R_n by using a data-driven key discovery algorithm. Next, we compute $\mathcal{MC}(S_n)$ by applying minimization operation on the results of pairwise union of $\mathcal{MC}(S_o)$



(a) DIGG2 (b) EBAY2 (c) DIGG2 (d) EBAY2

Fig. 9. Execution times of NAÏVE, COKE-P, and COKE-NP on real-world data sets.

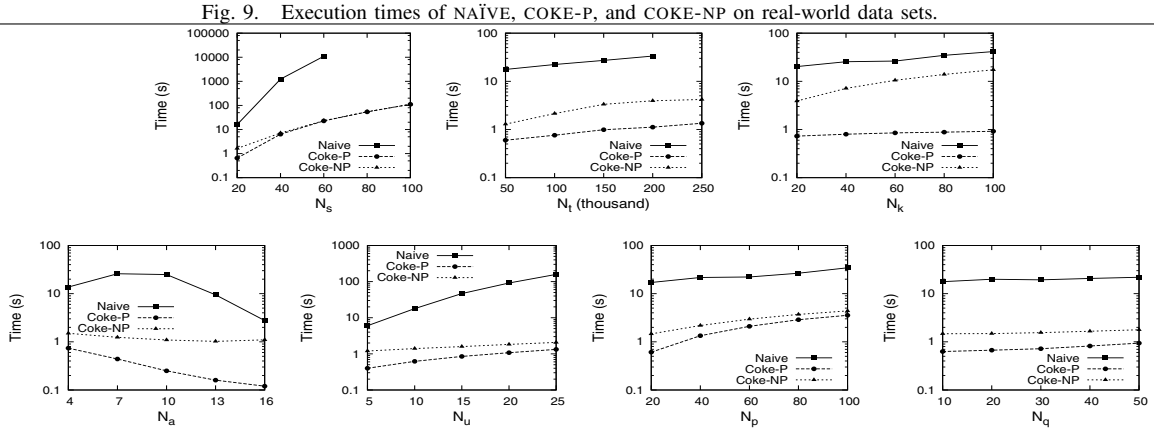


Fig. 10. Execution times of NAÏVE, COKE-P, and COKE-NP.

and $\mathcal{MK}(R_n)$. The formal description of the algorithm is given in [2]. Note that NAÏVE reflects the scenario where in the absence of COKE, we can only adopt a data-driven key discovery algorithm *without any non-trivial enhancement* to maintain common keys.

We use GORDIAN [23] as the key discovery algorithm for NAÏVE as it is efficient for relation with large number of records. We use AGREE SET [15] as the key discovery algorithm for the two variants of COKE as it is efficient for relation with small number of records (non-trivial sub-relations contain small number of records). Note that, although GORDIAN uses pruning techniques to eliminate large number of combinations of attributes, it may still suffer from the exponential size of the combinations of attributes. In contrast, although AGREE SET does not suffer from exponential size of the combinations of attributes, it suffers from quadratic size of the pairs of records. Thus, when the relation contains a small number of records and a large number of attributes, AGREE SET is potentially better than GORDIAN.

We implemented all algorithms in Java. All experiments were performed on a Windows XP machine with Pentium DC 3.40 GHz processor and 2.99 GB RAM.

A. Experiments on Real-World Data Sets

Eight data sets from various deep Web sites as shown in Table III are used for our study. Details related to retrieval of these data sets are discussed in Section V. Due to space constraints, we only show the results for data sets DIGG2 and EBAY2. Results of other data sets are qualitatively similar and can be found in [2].

For each data set, we first generate $\mathcal{MC}(S_o)$ by randomly sampling N_1 records from R_1, compute the set of minimal keys of these N_1 records, and use it as $\mathcal{MC}(S_o)$. For each data set, we generate R_n by randomly sampling N_2 records from R_2, and use it as R_n. We set N_1 to 1000 for both DIGG2 and EBAY2 data sets. The reason for choosing a relatively small value is to ensure that each minimal common key in $\mathcal{MC}(S_o)$ is associated with many non-trivial sub-relations, each with many records. Notice that this setting is unfavorable to COKE.

We measured the performance of the algorithms w.r.t the number of records in R_n by varying N_2. We ensure that R_n with smaller number of records is a subset of R_n with larger number of records. This will increase the numbers of non-trivial sub-relations and their records as we increase N_2.

Comparison of NAÏVE, COKE-P, and COKE-NP. Figures 9(a) and 9(b) show the performances of COKE-P, COKE-NP, and NAÏVE. We can make the following observations. Firstly, COKE-P is orders of magnitude faster than NAÏVE for both data sets. Secondly, it is evident that partitioning plan provides slight benefits to the algorithm. In the next section, using synthetic data, we shall discuss scenarios when it provides *significantly large* benefits.

Comparison of the phases of COKE-P and COKE-NP. Figures 9(c) and 9(d) report the performances of the two phases. Execution times of both phases increase roughly linearly with the number of records in R_n. Phase 1 behaves linearly as increase in number of records increases the number of records that need to be partitioned. Recall that the performance of Phase 2 is influenced by the number of non-trivial sub-relations and their sizes. Hence, as we increase the number of records in R_n, the number of non-trivial sub-relations increases roughly linear fashion, and their sizes increase very slowly. Furthermore, the linear behavior of Phase 2 justifies the usage of AGREE SET for key discovery. Although it is quadratic to the number of records in the non-trivial sub-relations, the linearity of Phase 2 is due to the fact that there

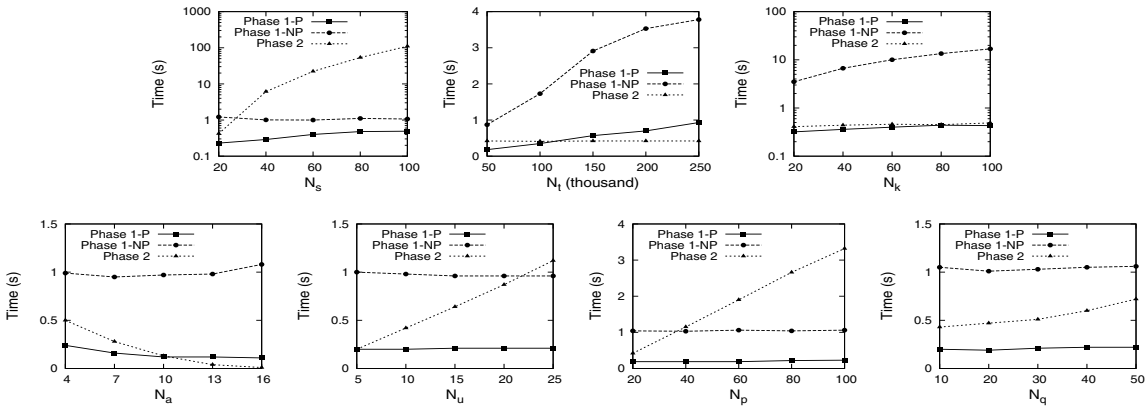


Fig. 11. Execution times of the phases of COKE-P and COKE-NP.

is not much variation in the number of records in the non-trivial sub-relations with increasing number of records in R_n. Additionally, the impact of a quadratic algorithm is small since each non-trivial sub-relation contains small number of records.

B. Experiments on Synthetic Data Sets

We report the execution times w.r.t each of the seven parameters in Table II by varying one parameter and setting remaining ones to their default values.

Data sets generation. We synthetically generate $\mathcal{MC}(S_o)$ by using the following procedure. We create N_k minimal common keys. Each of this is created by randomly taking N_a attributes from the attributes of R_n. We generate R_n by using the following procedure. We create N_t records each with N_s attributes. We set the values of the attributes of these records such that each attribute of R_n is a key of R_n. We randomly take N_u minimal common keys from $\mathcal{MC}(S_o)$. For each of these N_u minimal common keys, we associate it with a set of non-trivial sub-relations, created by randomly taking N_pN_q records from R_n, and dividing them into N_p non-trivial sub-relations each with N_q records. We set the values of the attributes of these N_q records as the following. For each attribute in the minimal common key, we set the values of this attribute to same value. We then randomly take two attributes that are not in the minimal common key. For each of these attributes, we randomly take three records from N_q records and set the values of this attribute of these records to same value. The reason for taking these two attributes is to enforce that there are at least two super sets of each minimal common key that are not a key of R_n. The objective of this is to vary the minimal common keys in $\mathcal{MC}(S_n)$.

Comparison of efficiency and scalability of NAÏVE and COKE-P. Figure 10 reports the performances of the algorithms with respect to various parameters. Observe that COKE-P is more efficient and scalable than NAÏVE. The superiority of COKE-P can be mostly seen for the parameters N_s and N_u. Specifically, COKE-P executes the key discovery algorithm on non-trivial sub-relations. These sub-relations have smaller number of records as well as smaller number of attributes. Although COKE-P has the overhead of executing the key discovery algorithm multiple times, the total execution time is still orders of magnitude faster than the NAÏVE. Observe that COKE-P is able to handle a data set with 100 attributes in only 110 seconds. Such size of data set is intractable for NAÏVE. Also, the parameter N_u affects the performance of NAÏVE adversely as it needs to find the minimal common keys when it traverses the possible combinations of attributes. However, in COKE-P, the non-trivial sub-relations are anorexic, and thus it is cheaper for the key discovery algorithm to find the minimal common keys.

Interestingly, the execution time of the NAÏVE algorithm increases up to a certain value of N_a and then decrease due to the pruning techniques used by GORDIAN. It is fast when the minimal keys contain small number of attributes or large number of attributes. In contrast, the execution time of COKE-P decreases with N_a due to decrease in execution time of Phase 2. This is because the number of attributes in the projected non-trivial sub-relation is equal to $N_s - N_a$. Thus, when N_a increases, the projected non-trivial sub-relation contains smaller number of attributes. Hence it is faster for the key discovery algorithm to compute its minimal key set.

Comparison of COKE-P and COKE-NP. Figure 10 reports the advantage of using partitioning plan. Specifically, it can be mostly seen for parameters N_t and N_k. Note that N_t determines the number of records that the algorithms need to partition. Although the execution times of both COKE-P and COKE-NP increase with N_t, COKE-P has smaller rate of increase. This is expected since by using partitioning plan, some of the partitioning can be shared. Similar to N_t, although the execution times of both COKE-P and COKE-NP increase with N_k, COKE-P has a much smaller rate of increase. In practice N_k can be large, since the number of possible combinations of attributes is exponential. In this case it is beneficial to use partitioning plan. These results also show that our heuristic algorithm for plan construction is effective in reducing the effect of N_t and more importantly N_k.

Comparison of the phases of COKE-P and COKE-NP. Figure 11 reports the performance comparison of different phases in COKE-P and COKE-NP. The parameters that significantly affect the execution times of Phases 1 and 2 are N_t and N_s, respectively. Although the number of possible combinations of attributes is exponential to N_s, the execution time of Phase

2 does not increase exponentially with N_s. This is because COKE-P uses AGREE SET. The other parameters that also affect the execution time of Phase 2 are N_u, N_p, and N_q. The execution time of Phase 2 is linear to N_u and N_p. This is as expected since the number of execution of key discovery algorithm is linearly affected by N_u and N_p. In contrast, the execution time of Phase 2 is super linear to N_q.

VII. Related Work

Quality of monitoring the dynamic Web. Monitoring the dynamic Web in response to continuous queries have recently triggered a lot of interest. In this context, optimizing the QoS (Quality of Service) and QoD have been the focus of several research efforts. For example, multi-query optimization has been exploited in [5] to improve the system throughput, optimization of freshness and currency of query results were investigated in [20], [22], and Kukulenz and Ntoulas [13] have studied quality/freshness tradeoff for *bounded* continuous search queries. However, we are not aware of any prior work that studied the discovery of common keys in order to improve accuracy of tracking entities. Note that existing monitoring systems [5], [14], [19] deploy variants of *HTMLDiff* and XML *diff* algorithms to detect and track changes to the underlying data. However, these algorithms are "identifier-oblivious" and as a result they may be confused by similar values for different attributes associated with each entity. Consequently, these algorithms may adversely affect the QoD by producing erroneous mapping between entities over time. Hence, our work is complimentary to these efforts.

Also, none of the existing efforts [1], [3], [6] in extracting entity-relation tuples from text into relational databases focus on maintenance of common keys.

Discovery of keys and functional dependencies. More germane to our work are efforts related to key discovery problem in structured databases and mining of functional dependencies (FD). GORDIAN [23] and AGREE SET [15] compute the set of minimal keys indirectly by computing the set of maximal non keys and then converting it into the set of minimal keys. There has also been a great deal of work related to mining strict and approximate FDs from the data. TANE [11], FUN [18], and *FD_Mine* [25] take a candidate *generate-and-test* approach where levelwise search strategy is used to explore the search space. They reduce the search space by eliminating candidates using pruning rules. Similar to our approach, TANE uses a partitioning of the tuples with respect to their attribute values to check validity of functional dependencies. It implicitly identifies keys to prune the search space. FastFDs [24] and *Dep-Miner* [15] employ first-depth search and levelwise search strategies, respectively, to discover FDs by considering pairs of tuples. Several association rules and sampling-based techniques have also been proposed to make FDs less restrictive by allowing some exceptions to the FD rules [12], [21]. Our work differs from these approaches as follows. First, these techniques focus on discovering keys or FDs from *static* data whereas we focus on incrementally maintaining common keys. These techniques do not assume evolving nature of keys or FDs and hence they are not designed to efficiently maintain the common keys. Second, COKE exploits anorexic characteristics of real-world data to devise efficient solution to this problem. Such characteristics are not exploited in traditional key or FD discovery techniques.

VIII. Conclusions and Future Work

In this paper, we have described a novel technique for efficiently maintaining common keys in a sequence of versions of archived continuous query results from deep Web sites. This is crucial for developing robust techniques for modeling evolutionary Web data, query processing, and tracking entities over time. We have proposed an algorithm called COKE to discover common keys from the archived versions of structured query results represented as relations. It generates minimal common keys without computing the minimal key set of the new relation. Importantly, it exploits certain anorexic properties of real-world data to provide efficient solution to this problem. Our exhaustive empirical study has demonstrated that COKE has excellent real-world performance. We are currently exploring techniques to detect *conserved* common keys to support *identifiers* discovery in order to track entities accurately in the historical query results sequence.

References

[1] A. Arasu, H. Garcia-Molina. Extracting Structured Data from Web Pages. *In SIGMOD*, 2003.
[2] F. Ardian, S. S. Bhowmick. coke: Efficient Maintenance of Common Keys in Evolutionary Web Archives. *Technical Report*. Available at *http://www.cais.ntu.edu.sg/∼assourav/TechReports/coke-TR.pdf*, 2010.
[3] M. J. Catarella et al. Structured Querying of Web Text. *In CIDR*, 2007.
[4] K. C.-C. Chang, B. He, C. Li et al. Structured Databases on the Web: Observations and Implications. *In ACM SIGMOD Record*, 33(3), 2004.
[5] J. Chen, D. DeWitt, F. Tian et al. NiagaraCQ: A Scalable Continuous Query System for the Internet Databases. *In SIGMOD*, 2000.
[6] E. Chu, A. Baid et al. A Relational Approach to Incrementally Extracting and Querying Structure in Unstructured Data. *In VLDB*, 2007.
[7] X. Dong, A. Y. Halevy, J. Madhavan. Reference Reconciliation in Complex Information Spaces. *In SIGMOD*, 2005.
[8] A. K. Elmagarmid, P. G. Ipeirotis, V. S. Verykios. Duplicate Record Detection: A Survey. *In IEEE TKDE*, 2007.
[9] D. Gunopulos, R. Khardon, H. Mannila, et al. Discovering all most Specific Sentences. *In ACM Trans. Database Systems*, 28(2):140–174, 2003.
[10] J. Han, et al. Mining Frequent Patterns without Candidate Generation. *In SIGMOD*, 2000.
[11] Y. Huhtala et al. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *In Computing J.*, 42(2):100–111.
[12] J. Kivinen, H. Mannila. Approximate Dependency Inference from Relations. *Theoret. Comp. Sci.*, 149:129–149, 1995.
[13] D. Kukulenz, A. Ntoulas. Answering Bounded Continuous Search Queries in the World Wide Web. *In WWW*, 2007.
[14] L. Liu, C. Pu, W. Tang. WebCQ: Detecting and Delivering Information Changes on the Web. *In CIKM*, 2000.
[15] S. Lopes, J. M. Petit, et al. Efficient Discovery of Functional Dependencies and Armstrong Relations. *In EDBT*, 2000.
[16] J. Madhavan, D. Ko, et al. Google's Deep Web Crawl. *In VLDB*, 2008.
[17] J. Masanès. Web Archiving. Springer, New York, Inc., Secaucus, N.J., 2006.
[18] N. Novelli, R. Cicchetti. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. *In ICDT*, 2001.
[19] B. Nguyen, et al. Monitoring XML Data on the Web. *In SIGMOD*, 2001.
[20] S. Pandey, K. Ramamritham, S. Chakrabarti. Monitoring the Dynamic Web to Respond to Continuous Queries. *In WWW*, 2003.
[21] D. Sánchez, J. M. Serrano et al. Using Association Rules to Mine For Strong Approximate Dependencies. *In DMKD*, 16:3143–348, 2008.
[22] M. A. Sharaf, A. Labrinidis, et al. Freshness-Aware Scheduling of Continuous Queries in the Dynamic Web. *In WebDB*, 2005.
[23] Y. Sismanis, P. Brown, et al. GORDIAN: Efficient and Scalable Disocovery of Composite Keys. *In VLDB*, 2006.
[24] C. Wyss, et al. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances. *In DAWAK*, 2001.
[25] H. Yao, H. J. Hamilton. Mining Functional Dependencies from Data. *In DMKD*, 16:197–219, 2008.