We Challenge You to Certify Your Updates

Su Chen
National Univ of Singapore
chensu@comp.nus.edu.sg

Xin Luna Dong
AT&T Labs–Research
lunadong@research.att.com

Laks V.S. Lakshmanan
Univ of British Columbia
laks@cs.ubc.ca

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

Correctness of data residing in a database is vital. While integrity constraint enforcement can often ensure data consistency, it is inadequate to protect against updates that involve careless, unintentional errors, e.g., whether a specified update to an employee's record was for the intended employee. We propose a novel approach that is complementary to existing integrity enforcement techniques, to guard against such erroneous updates.

Our approach is based on (a) updaters providing an *update certificate* with each database update, and (b) the database system verifying the correctness of the update certificate provided before performing the update. We formalize a certificate as a (challenge, response) pair, and characterize good certificates as those that are easy for updaters to provide and, when correct, give the system enough confidence that the update was indeed intended. We present algorithms that efficiently enumerate good challenges, without exhaustively exploring the search space of all challenges. We experimentally demonstrate that (i) databases have many good challenges, (ii) these challenges can be efficiently identified, (iii) certificates can be quickly verified for correctness, (iv) under natural models of an updater's knowledge of the database, update certificates catch a high percentage of the erroneous updates without imposing undue burden on the updaters performing correct updates, and (v) our techniques are robust across a wide range of challenge parameter settings.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*

General Terms: Algorithms, Design, Experimentation

1. INTRODUCTION

Correctness of data residing in a database is of utmost importance for applications that rely on the data to make critical decisions. Much work has been done on analyzing and mining the data in a database in order to detect potential duplicates, likely errors and statistical outliers (see, e.g., [4, 3]). In this paper, we are interested in a complementary problem: starting with a correct database state, how do we minimize, if not prevent, the possibility that errors creep into the database as the data gets updated?

Ensuring correctness is not easy since databases are continuously being modified by human updaters (typically using an application) to reflect changes in reality. Consider, for example, databases supporting Project Management. Here, information about projects (such as the status of different parts of the projects, their current priorities, resource needs, and funding sources) and participants (such as roles of employees in different projects, individual project timelines, reporting of billable hours) need to be continuously updated over time; such updates are typically made by project administrators. Other such examples include Human Resources databases which contain dynamically changing information about employees (such as their compensation, disability and sick leave, and job functions) and organizations (such as management reporting structures), Inventory databases which contain continuously updated information about organizational assets, and so on.

Erroneous updates do happen in these databases, as many of us may have experienced, often with significant cost because of the difficulty of correcting the errors in the database and rolling back the decisions made due to these errors. For example, switching funding sources between projects in a Project Management database, or reporting a disability leave for the wrong employee in a Human Resources database, can create a lot of problems. In these scenarios, *it would be desirable to guard against such erroneous updates, even if it requires additional effort on the part of the database system, the application developers, and the updaters*.

A variety of classical techniques ranging from schema normalization to integrity constraint enforcement have been proposed to address this problem [14]. These existing techniques require the database designer to anticipate all possible errors and specify integrity constraints that must hold on all instances of the database schema, and which would detect (and reject) erroneous updates. However, a large variety of errors cannot be anticipated and arise due to carelessness on the part of updaters and by the inability of the database system to detect those updates as erroneous. The following example illustrates the idea.

EXAMPLE 1 (COMPLEX, BULK UPDATE). Consider the Corporate database shown in Fig. 1, which is a simplified Project Management and Human Resources database. The tables represent departments, employees, projects, and the association of employees with projects. An updater (for example, a project administrator) intends to make Dolores Quintana the Admin of all the projects whose responsible department is located in NJ, starting from 1985-02-01, but in the update, incorrectly specifies `location` as NY instead of NJ, `empId` as 000030 instead of 000130, and `stDate` as 1985-01-02 instead of 1985-02-01. For concreteness, we show below the SQL update issued by the application with which the updater interacted.

```
U1: update PROJ_EMP
    set empId = '000030', stDate = '1985-01-02'
    where job = 'Admin' and projId in (
       select PROJ.projId  from PROJ, DEPT
       where PROJ.deptId = DEPT.deptId and
          DEPT.location = 'NY' )
```


SIGMOD'11, June 12–16, 2011, Athens, Greece.

DEPT

deptId	deptName	mgrId	location
A00	Services	000010	NJ
B01	Planning	NULL	NY
C01	Information	000030	NJ

PROJ_EMP

empId	*projId*	job	stDate
000010	AD3100	Leader	1982-01-01
000010	MA2100	Leader	1982-02-01
000020	PL2100	Leader	1983-01-01
000030	IF1000	Leader	1983-06-01
000110	MA2100	Admin	1984-01-01
000130	IF1000	Admin	1984-07-01
000140	PL2100	Admin	1985-10-01
000140	IF2000	Admin	1985-03-01

EMP

empId	name	workDept	phone	sex	salary
000010	Christine Haas	A00	3978	F	52750
000020	Michael Thompson	B01	3476	M	41250
000030	Sally Kwan	C01	4738	F	38250
000110	Vincenzo Lucchessi	A00	3490	M	46500
000120	Sean OConnell	A00	2167	M	29250
000130	Dolores Quintana	C01	4578	F	23800
000140	Heather Nicholls	C01	4578	F	28420

PROJ

projId	projName	deptId	stDate	enDate
AD3100	Admin Services	A00	1981-01-01	1988-02-01
IF1000	Query Services	C01	1981-03-01	1987-02-01
IF2000	User Education	B01	1980-01-01	1986-02-01
MA2100	Line Automation	A00	1979-01-01	1986-03-01
PL2100	Line Planning	B01	1979-05-15	1987-09-15

Figure 1: Schema and Sample Data of a Corporate Database

Note that since valid `empId`, `stDate` and `location` values are specified in the update, integrity constraint checks may not be able to detect this erroneous update. Notice, all of the natural FDs applicable to the schema of Fig. 1 would be satisfied by this update. As a result, unintended records in the `PROJ_EMP` table are (incorrectly) updated, while the intended records are left untouched. □

The above example illustrates that catching erroneous updates to databases remains a challenging problem. In fact, almost all known classes of integrity constraints are unable to capture such updates as erroneous. One exception is the class of conditional functional dependencies (CFDs) proposed recently [6]. CFDs (with a slight extension) can assert value associations of the form "if job is Admin, then `empId` must start with 0001", thus capturing update U1 as erroneous. However, this requires the database designer to anticipate all possible correct future states of the database, and assert them as CFDs, which is not a particularly practical solution.

In this paper, we propose a novel approach to guard against erroneous updates that are "innocent mistakes" (as opposed to malicious ones). Our approach is based on (a) the updater providing, through an application, the database update and an "update certificate" that is relevant to the database update, and (b) the database system verifying the correctness of the update certificate provided before performing the update. An update certificate consists of a (challenge, response) pair, where the "challenge" intuitively asks the updater to provide additional information that is relevant to the specified database update, and the updater provides this information in the "response" to demonstrate that the updater indeed intended to make the specified update; in the absence of malice, this is evidence of correctness. The system verifies, as part of the update transaction, whether the response provided by the updater matches the challenge; if not, the specified update is rejected as being unintended. The following example illustrates the idea.

EXAMPLE 2 (CERTIFICATE CHALLENGES AND RESPONSES). Challenges for the update U1 above could include "Q1: What is the name of the employee with `empId` 000030?", "Q2: What is the day of the week of `stDate` 1985-01-02?" and "Q3: What is the name of an employee who works for a department located in NY?" If the updater responds "Dolores Quintana" to challenge Q1, or "Friday" to challenge Q2, or "Christine Haas" to challenge Q3, all of which match the *intended update* but not the *specified update* U1, the database system could reject U1 as unintended, thereby preventing any disastrous decision that might be made as a consequence of the erroneous update. □

1.1 Pragmatics of Certificates

For update certificates to be effective in guarding against erroneous updates, it requires additional effort on the part of the database system, updaters, and application developers. Our preceding discussion described the additional effort that would be incurred by the updaters (providing the certificate) and the database system (verifying the correctness of the update certificate).

A critical role also needs to be played by the applications that mediate between the updaters and the database system: that of taking the updaters' certificates and presenting them to the database system. A developer of such an application would need to pick one or more reasonable challenges for each type of update (such as U1) supported by the application, and bake them into the application. An updater using the application can choose any of the challenges relevant to the specified update and provide responses, as evidence that the updater indeed intended to make the specified update.

Note that updates need not be interactive for the certificate mechanism to be feasible. Since the challenges supported by the application are determined offline, a non-interactive updater can pass (i) the id of the chosen challenge, and (ii) the response to the challenge, as input to the application. This approach can also be used in conjunction with "batch" updates, where multiple update statements can be issued (along with their certificates) by the updater. The update transaction generated by the application would include the updates and their certificates for the system to verify.

It is important to emphasize that while the use of certificates adds some overhead to the update process, both for the updater and for the database system, and requires application developers to incorporate certificates into their applications, these overheads are significantly outweighed by the benefits of certificates in reducing errors in databases as the data gets updated. Such errors incur a significant cost for restoring the database to a correct state and rolling back the decisions made based on the incorrect state.

We next discuss various alternatives to update certificates and motivate our design choice of using certificates.

1.2 Alternatives to Certificates

Our main goal is to detect and prevent updates that might lead to innocent unintended errors. One of the simplest mechanisms for doing this, inspired by the approach taken by Unix to confirm that a user intended to delete a file, is asking the updater "Are you sure?" Another option, inspired by the practice used by most online secure systems for users to confirm a modified password, is to ask the updater to repeat their update specification. A third option is to show

the updater a sample of the records that are about to be updated and let the updater visually inspect the changes before deciding if this is the correct update.

All these approaches have the significant disadvantage that they are not suitable for updaters who use non-interactive applications to update databases. The first two approaches have the added disadvantage that updaters can easily answer "Yes" to the "Are you sure?" question, or use "cut and paste" as a quick way of repeating their update specification. Two recent studies, one on the (poor) security practices of users in dealing with online bank accounts [11], and the other on user practices in (not) reading and understanding end user license agreements [9], suggest that updaters are very likely to do so. Thus, such simplistic methods are unlikely to be effective to ensure correctness of updates.

What we need is a method that proves to be effective in catching erroneous updates while not imposing an undue burden on updaters. By the first objective, we mean that the percentage of erroneous updates let through should be very small. By the second objective, we mean that under natural models of an updater's knowledge of the data, the need for update certificates should not prevent most of their correct updates from going through easily.

We next spell out desiderata for update certificates, and argue that picking challenges in a principled manner is important to reducing errors in databases as the data gets updated.

1.3 Desiderata and Contributions

For update certificates to be useful in detecting erroneous updates, the most critical issue is the *design of certificate challenges* that are relevant to updates.

Database System Desiderata: Consider a challenge "Q4: What is the sex of the employee with `empId` 000030?" While Q4 is relevant to Update U1, the updater's (correct) response of "F" would not have helped the system detect the erroneous update, since both Sally Kwan and Dolores Quintana are female. Intuitively, from the database system's viewpoint, a challenge is not desirable if there are many unintended updates for which a correct response to the challenge is the same as that for the intended update.

Updater Desiderata: Suppose that the updater is asked "Q5: What is the phone number of an employee who works for a project whose responsible department is located in NY?" Such a challenge is too complex, so reduces understandability and increases the risk of a wrong response *even when the updater knows the correct response to the challenge*. Similarly, if the *only* challenges available were "Q6: What are the salary and the sex of the employee with `empId` 000030?" and "Q7: What are the salary and the phone number of the employee with `empId` 000030?", then an updater who does not know an employee's salary would be unable to choose either challenge and provide a correct response *even when she performs a correct update*. Thus, from the updater's viewpoint, it is desirable that challenges be easy to use, without undue expectation that she is familiar with every aspect of the data she updates.

We effectively address the desiderata important to updaters and to the database system and make the following contributions:

- First, we formalize an update certificate as a *(challenge, response)* pair. We identify properties that make for "good" challenges: high *discriminating power*, low *description complexity*, and high *diversity* (Sec. 2). Intuitively, the first property permits the database system to have high confidence that the specified update was intended, the second makes sure the challenges are easy to understand, while the third ensures that every updater can respond to some challenge.
- Second, we formulate two optimization problems for finding good challenges: (i) minimize description complexity of the challenges that meet specified discriminating power and diversity thresholds, and (ii) maximize diversity of the challenges that meet specified discriminating power and description complexity thresholds. We develop efficient algorithms to optimally solve these problems (Sec. 3).
- Finally, we experimentally evaluate our techniques on benchmark databases and demonstrate that (i) databases often have many good challenges, (ii) such challenges can be efficiently identified, (iii) responses provided by updaters can be quickly verified for correctness, (iv) under natural models of an updater's knowledge of the database, update certificates enjoy good precision and recall: they catch most erroneous updates without imposing undue burden on updaters performing correct updates, and (v) our techniques are robust across a wide range of challenge parameter settings (Sec. 4).

2. CERTIFICATES: PROPERTIES

Databases allow insertions, deletions and modifications of records, tables and table spaces. We focus on *record modifications*, which allow the values of specified attributes in a set of records of a specified table to be modified to specified values. For example, Update U1 is of this form. These updates present the most challenges, and their solutions can be adapted for other updates.

2.1 Certificates

An *update certificate* consists of a *challenge* presented to the updater and a *response* by the updater. We propose certificates for both the update condition (to ensure that the intended set of records are being updated) and for the updated values (to ensure that the modified values are the intended ones).

DEFINITION 2.1 (UPDATE CERTIFICATE). Consider update condition or updated value γ of the form `attr=val`.[1] An *update certificate* C_γ for γ is a (challenge, response) pair (Q_γ, R_γ), where (i) challenge Q_γ is a SQL query, whose `where` clause condition $Cond(Q_\gamma)$ implies γ,[2] and (ii) response R_γ is the answer provided by the updater to Q_γ. An *update certificate* $\mathbf{C}_U$ for a record modification update U is a set of certificates C_γ, one for each update condition and updated value γ in the update U. □

DEFINITION 2.2 (VALID CERTIFICATE). Given a database instance D, a certificate $C_\gamma = (Q_\gamma, R_\gamma)$ is said to be *valid* if R_γ is in the answer set of Q_γ on D. A certificate $\mathbf{C}_U$ is said to be *valid* if each $C_\gamma \in \mathbf{C}_U$ is valid. □

Ex. 2 presented challenges (Q1–Q3) associated with the certificates for update U1. Example certificates for U1 might be the pairs (Q1, Dolores Quintana), (Q2, Friday) and (Q3, Christine Haas), respectively, none of which is valid. However, certificates (Q1, Sally Kwan), (Q2, Tuesday), and (Q3, Michael Thompson) are valid.

We next present three properties that we consider important for guarding against erroneous updates: *discriminating power*, *description complexity* and *diversity*. These properties formalize the desiderata for certificates by the database system and by the updater.

2.2 Discriminating Power

Should all specified updates with valid certificates (e.g., (Q4, F)) be considered as intended by the updater, and accepted by the system? The notion of *discriminating power*, discussed below, quantifies the confidence provided by a valid certificate that the specified update was indeed intended.

[1] For the purpose of certificates, an updated value `attr=val` can be treated exactly like an update condition `attr=val`.

[2] This property ensures that the challenge is *relevant* to the update.

We first need the notions of a *challenge template* and a *CR-table*. Consider a challenge Q_γ. The *challenge template* Q_γ^T is the parameterized query obtained from Q_γ by replacing γ in its `where` clause by γ^T, obtained by replacing `val` in γ by a parameter `$val`. For example, the challenge template $Q1^T$ would be "`select name from EMP where empId = $val`".[3]

A challenge template Q_γ^T is associated with a *CR-table* $T_{Q_\gamma^T}(V, R)$, which contains answers R to challenges V that can be obtained from Q_γ^T. If materialized, the CR-table can be used to quickly validate an updater's response to any challenge. For example, the projection of table `EMP` on attributes `(empId, name)` would be the CR-table for challenge template $Q1^T$, and the projection of the join `DEPT` $\bowtie_{deptId=workDept}$ `EMP` on attributes `(location, name)` would be the CR-table for challenge template $Q3^T$.

We are finally ready to define the notion of discriminating power. Consider an update condition or updated value γ, and a valid certificate $C_\gamma = (Q_\gamma, R_\gamma)$ for γ. This certificate could be considered as a "lucky guess" if the updater had intended to specify a different update condition/value $\gamma_j \neq \gamma$, but response R_γ is also a valid response for Q_{γ_j}. Intuitively, the discriminating power of a valid certificate is the probability that the provided certificate is *not a lucky guess*. More formally, we have the following definition.

DEFINITION 2.3 (DISCRIMINATING POWER). Consider a database instance D, an update condition or updated value γ : `attr = val`, a valid certificate $C_\gamma = (Q_\gamma, R_\gamma)$ for γ, and the CR-table $T_{Q_\gamma^T}(V, R)$ for the challenge template Q_γ^T.

The *discriminating power* (DP) of C_γ is defined as:

$$DP(C_\gamma) = 1 - \frac{|\{T_{Q_\gamma^T}.V \mid T_{Q_\gamma^T}.V \neq \texttt{val} \,\&\, T_{Q_\gamma^T}.R = R_\gamma\}|}{|\{T_{Q_\gamma^T}.V \mid T_{Q_\gamma^T}.V \neq \texttt{val}\}|}$$

We define the minimum (resp. average) discriminating power of a challenge template Q_γ^T, denoted minDP(Q_γ^T) (resp., avgDP(Q_γ^T)) as the minimum (resp., average) DP over all possible *valid responses* in the CR-table $T_{Q_\gamma^T}$.[4] □

For example, the DP of certificate (Q4, F) is 3/6, and the minDP and avgDP of $Q4^T$ are 3/6 and 7/12, respectively.

Since valid certificates with low DP are not useful as evidence of the updater's intention, applications should use only challenges that yield certificates with a high discriminating power. Only then can the system have high confidence that the specified update will not make the database dirty.

2.3 Description Complexity

If providing certificates were to place an undue burden on updaters, then certificates may not be used despite their obvious benefits. Hence, it should be *easy* for updaters to provide certificates.

In practice, we expect that a challenge to be presented in natural language. To keep the natural language question simple, we require two properties of individual challenges. First, the challenge must be a chain join query, with the updated table at one end of the chain; such queries can be translated to natural language much more easily than challenges containing arbitrary joins. For example, this property holds for challenge Q3. Second, no challenge presented to an updater should be too verbose since that would reduce understandability. The two components that contribute to this verbosity

[3]Our techniques can be extended to handle non-equality update conditions. For example, if γ was "`stDate > 1984-01-01`", the challenge template could include "`stDate > $val`".

[4]Note that, given a CR-table $T_{Q_\gamma^T}$ for challenge template Q_γ^T, the discriminating power depends only on the response R_γ.

DM for {Q11, Q21, Q31}			
	D	D ⋈ E	D ⋈ P
Q11	1	0	0
Q21	0.05	1	0
Q31	0.05	0	1

DM for {Q11, Q21, Q22}		
	D	D ⋈ E
Q11	1	0
Q21	0.05	1
Q22	0.05	1

Figure 2: Diversity Matrices for Sets of Challenges

are the length of the chain join (i.e., the number of tables, which is #joins + 1) and the number of response values the updater is asked to provide. This intuition is captured by the following requirement.

DEFINITION 2.4 (DESCRIPTION COMPLEXITY). We define the *description complexity* of challenge Q_γ as follows.

$$DC(Q_\gamma) = w_1 * (\#\text{joins} + 1) + w_2 * \#\text{select-clause-attrs} \quad (1)$$

In the absence of other information, we set $w_1 = w_2 = 1$.

The *description complexity*, or size, of a challenge template Q_γ^T is defined to be the same as that of the challenge Q_γ. □

For example, $DC(\texttt{Q1}) = 2$ and $DC(\texttt{Q3}) = 3$. This captures the complexity of expressing these challenges in natural language.

2.4 Diversity

It is important to realize that an updater may not be familiar with every aspect of the data she updates. Hence, the application should present an updater with a set of sufficiently *diverse* challenges, with the hope that there will be at least one challenge whose response would be correctly known to the updater.

We propose an information theoretic notion of diversity based on the chain joins present in a set of challenges, that captures the following key intuitions: (i) the more the number of distinct chain joins, the higher the diversity; (ii) the smaller the similarities between the distinct chain joins, the higher the diversity; and (iii) the more uniform the distribution of distinct chain joins, the higher the diversity. For example, the set of challenge templates $Q6^T$ ("what are the salary and sex of an employee with `empId` $e?") and $Q7^T$ ("what are the salary and phone number of an employee with empId $e?") is not very diverse. This is because they both involve the same table (chain join prefix).

The above intuitions can be precisely captured using the information theoretic concept of entropy, as follows.

DEFINITION 2.5 (DIVERSITY). Consider a set of challenges $\mathcal{Q}_\gamma = \{Qi_\gamma \mid 1 \leq i \leq k\}$. Let $\{J_\ell \mid 1 \leq \ell \leq m\}$ denote the set of chain joins and their prefixes that are present in $\mathcal{Q}_\gamma$, and let α denote an arbitrary constant in $(0, \frac{1}{km})$.[5] The *diversity matrix* of $\mathcal{Q}_\gamma$ is a $k \times m$ matrix whose entries are defined as follows.

$$\begin{aligned} DM(Qi_\gamma, J_\ell) &= 1, \text{if } J_\ell \text{ is } Qi_\gamma\text{'s chain join,} \\ &= \alpha, \text{if } J_\ell \text{ is a proper prefix of } Qi_\gamma\text{'s chain join,} \\ &= 0, \text{otherwise} \end{aligned}$$

Given the diversity matrix DM, the *diversity* of the set of challenges $\mathcal{Q}_\gamma$, denoted $DV(\mathcal{Q}_\gamma)$, is defined as the entropy $H(X)$ $(= \Sigma_{\ell=1}^m p(X = \ell) * \log_2(1/p(X = \ell)))$ of the random variable X, where

$$p(X = \ell) = \frac{\Sigma_{i=1}^k DM(Qi_\gamma, J_\ell)}{\Sigma_{j=1}^m \Sigma_{i=1}^k DM(Qi_\gamma, J_j)}, 1 \leq \ell \leq m$$

The diversity of a set of challenge templates $\{Qi_\gamma^T, 1 \leq i \leq k\}$ is defined to be the same as that of $\mathcal{Q}_\gamma$.[6] □

[5]$\alpha < \frac{1}{km}$ is important for Lemma 3.1.

[6]Diversity can be naturally extended to incorporate response attributes.

EXAMPLE 3 (DIVERSITY). Consider condition `location = NY` of Update U1, and the following space of possible challenges, abbreviated using (chain-join, response attributes), where each $\bowtie$ denotes a (primary key, foreign key) left-outer-join:

- Q11: (`DEPT`, (`deptName`))
- Q21: (`DEPT` $\bowtie$ `EMP`, (`name`))
- Q22: (`DEPT` $\bowtie$ `EMP`, (`phone`))
- Q31: (`DEPT` $\bowtie$ `PROJ`, (`projName`))

The diversity matrices for two sets of challenges S_1 = {Q11, Q21, Q31} and S_2 ={Q11, Q21, Q22} are shown in Fig. 2, for $\alpha = 0.05$. Based on the diversity matrix, the diversity for S_1, $DV(S_1)$ is 1.58, while for S_2, $DV(S_2)$ is 0.94. Note, S_1 and S_2 have the same cardinality and description complexity, but S_2 has two challenges, Q21 and Q22, with the same chain join while challenges in S_1 have no chain joins in common. □

3. FINDING CHALLENGE TEMPLATES

The database system needs to identify a set of good challenge templates for every database column that may be updated or on which an update condition may be specified and publish these templates. The application developer can consult these published templates, pick one or more reasonable challenges for each type of update supported by the application, and bake them into the application that is used by the updater to perform updates. For identifying good challenge templates, we formulate two optimization problems in Sec. 3.1, and subsequently develop efficient algorithms to solve these problems in Sec. 3.2 and Sec. 3.3.

3.1 Optimization Problems

Sec. 2 identified the properties that make for a set of good challenges: high discriminating power, low description complexity, and high diversity, which naturally extend to challenge templates as well. While desirable, it is unfortunately impossible to simultaneously optimize for all these properties. For example, it may be possible to find a set of k challenge templates from the updated table and they have a low description complexity; however, their diversity will be 0. If, instead, we find a set of k challenges with different and long chain joins, they are likely to have high diversity but larger description complexity. We thus formulate two constrained optimization problems that trade off the two ease-of-use concerns, while requiring high enough discriminating power.

Problem MinimizeDC: Given an update column `attr`, threshold values $\tau_{minDP}, \tau_{avgDP}$ and τ_{DV}, find a set of k update-relevant challenge templates $\mathcal{Q} = \{Qi^T \mid 1 \le i \le k\}$ that minimizes $\max(\{DC(Qi^T) \mid 1 \le i \le k\})$, while satisfying

1. $minDP(Qi^T) \ge \tau_{minDP}, 1 \le i \le k$,
2. $avgDP(Qi^T) \ge \tau_{avgDP}, 1 \le i \le k$,
3. $DV(\mathcal{Q}) \ge \tau_{DV}$, and
4. there does not exist any other challenge template Q^T, such that $minDP(Q^T) \ge \tau_{minDP}, avgDP(Q^T) \ge \tau_{avgDP}$, Q^T's join path is a prefix of that of Q_i^T and its attributes form a subset of those of $Q_i^T, 1 \le i \le k$.

Problem MaximizeDV: Given an update column `attr`, threshold values $\tau_{minDP}, \tau_{avgDP}$ and τ_{DC}, find a set of k update-relevant challenge templates $\mathcal{Q} = \{Qi^T \mid 1 \le i \le k\}$ that maximizes $DV(\mathcal{Q})$, while satisfying

1. $minDP(Qi^T) \ge \tau_{minDP}, 1 \le i \le k$,
2. $avgDP(Qi^T) \ge \tau_{avgDP}, 1 \le i \le k$,
3. $DC(Qi^T) \le \tau_{DC}, 1 \le i \le k$, and
4. there does not exist any other challenge template Q^T, such that $minDP(Q^T) \ge \tau_{minDP}, avgDP(Q^T) \ge \tau_{avgDP}$, Q^T's join path is a prefix of that of Q_i^T and its attributes form a subset of those of $Q_i^T, 1 \le i \le k$.

We use both $minDP$ and $avgDP$ in our problem definitions since they provide complementary ways to characterize the discriminating power of challenge templates. For example, a medium value of τ_{minDP} and a high value of τ_{avgDP} ensure that challenge templates can be identified even if some instances of the challenge template don't have a high DP, while guaranteeing that none of the challenge template instances have a low DP. This is not possible using just $minDP$ or $avgDP$. The last condition in the definition of the two problems precludes a challenge template when a less complex one would work.

We experimentally show that our techniques are robust across a wide range of threshold values $\tau_{minDP}, \tau_{avgDP}, \tau_{DV}$ and τ_{DC}; this makes choosing of threshold values quite easy.

3.2 Minimizing Description Complexity

In this section, we present an algorithm called MINIMIZEDC for solving Problem MinimizeDC. We first give an overview (Sec. 3.2.1) and then explain it in greater detail (Sec. 3.2.2 and 3.2.3).

3.2.1 Overview of Algorithm MINIMIZEDC

Virtually, there is a chain-join tree where the root corresponds to the updated table (no join), each node at level-k corresponds to a join path with $k-1$ joins, and its parent corresponds to its immediate join path prefix (see Fig. 3). To minimize description complexity, we shall explore the tree *top-down*.

Algorithm MINIMIZEDC dynamically generates the join tree in a breadth-first fashion; for each node, it maintains a *set* of lists $\{L_n\}$, $n > 0$, where L_n is a list of challenge templates with description complexity n (size-n) and with the join represented by the node. For example, the node `PROJ_EMP0` represents all challenge templates sharing the join `EMP` $\bowtie_{empId=empId}$ `PROJ_EMP`. Algorithm MINIMIZEDC starts with the root of the join tree and the n-th round proceeds in three steps:

1. *Examine size-n candidate templates.* For each node N where $L_n \neq \emptyset$ and each template Q^T in $N.L_n$, (1) check if Q^T has enough discriminating power; if so, remove it from $N.L_n$, and (2) check if adding Q^T to the result set $\mathcal{Q}$ violates the diversity threshold; if not, add it to $\mathcal{Q}$. Return when $\mathcal{Q}$ contains k templates.
2. *Generate size-$(n+1)$ candidate templates by adding one more attribute, and store them in L_{n+1} for each node.* This step invokes APRIORIGEN, which we explain later.
3. *Extend all chain-join paths one step further and generate 1-attribute challenge templates for each new node.* Note that when we extend a path, we exclude the join condition that is the same as the last one in the chain join (so no loop join).

Optimizing diversity constraint checking: For each candidate template that has enough discriminating power, we need to check if adding it to $\mathcal{Q}$ does not violate the diversity constraint. However, repeated entropy computation is expensive; the following sufficient condition allows us to "translate" the global minimum threshold on diversity, τ_{DV}, to a "local threshold" on the maximum number of challenges that have the same join, which is easier to check.

LEMMA 3.1. *Let $\mathcal{Q}$ be a set of k challenge templates, and τ_{DV} be a threshold on diversity. If at most $k_l = \lceil k/2^{\tau_{DV}} \rceil$ templates in $\mathcal{Q}$ share the same chain join, then $DV(\mathcal{Q}) \ge \tau_{DV}$.*

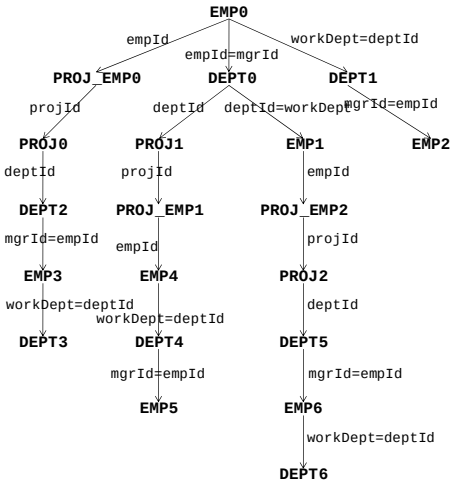


Figure 3: Chain-join tree in Ex. 4.

Proof Sketch: Recall the definition of diversity from Def 2.5. The nature of entries in the diversity matrix guarantees that with k challenge templates, the sum of all the entries in the diversity matrix will be in the closed-open interval $[k, k+1)$, since $\alpha < 1/km$. Since diversity of a set of challenge templates is computed using entropy, if $\ell \leq k$ chain joins are used by the k challenge templates in $\mathcal{Q}$, we know that $DV(\mathcal{Q}) < \log_2(\ell+1)$. Hence, the global minimum threshold on diversity, τ_{DV}, implies that at least $2^{\tau_{DV}}$ chain joins need to be used in $\mathcal{Q}$, with no more than $\lceil k/2^{\tau_{DV}} \rceil$ challenge templates that share the same chain join. Thus, ensuring this *local threshold* of $k_l = \lceil k/2^{\tau_{DV}} \rceil$ challenge templates would guarantee that $DV(\mathcal{Q}) \geq \tau_{DV}$. □

Algorithm APRIORIGEN**:** A key task of MINIMIZEDC is to generate candidate size-$(n+1)$ challenge templates for a node N_k. This task is non-trivial because we do not want to generate templates with a discriminating *sub-template* (i.e., with a subset of its attributes). However, checking if there exists such a sub-template can be very expensive. Inspired by the *A Priori* algorithm [1], our solution is based on two intuitions. First, we start from non-discriminating size-n challenge templates (in $N_k.L_n$) and check if any pair can be merged into a size-$(n+1)$ template. Second, for each generated template, rather than checking all of its sub-templates (exponential number), we check only those with one less attribute (linear number); such a template should be in some list $L_k, k < n$. For example, with 3-attribute templates $\{abc, abd, aef\}$ as input, the only pair that can be merged is abc and abd, resulting in a template with attributes $abcd$, which should be discarded as its size-3 sub-template acd is not included in the input set.

EXAMPLE 4. Consider an extended schema of that in Fig. 1:

```
DEPT(deptId, deptName, mgrId, location)
EMP(empId, name, workDept, phone, job, sex,
    birthdate, salary, bonus)
PROJ(projId, projName, deptId, stDate, enDate)
PROJ_EMP(empId, projId, job, stDate, enDate)
```

Fig. 3 shows the chain-join tree. The root is the table to be updated, i.e., `EMP`. Each node is named by the last table in the chain join plus a unique number to differentiate nodes with the same table. An edge represents a left outer join from a parent node to a child node on the attribute (or join predicate) labeled on the edge. For example, `DEPT0` identifies the result table of the chain join `EMP left-outer-join DEPT on empId=mgrId`.

The first column of Tbl. 1 lists the top-20 challenge templates for `EMP.empId` returned by MINIMIZEDC (in the order they are found) with parameters $k_l = 10$ and $\tau_{DC} = 10$. For example, entry (`EMP0.sex`, `DEPT0.deptId`) represents a challenge template with join `EMP left-outer-join DEPT`, asking for "the employee's sex and the Id of the department she manages".

MINIMIZEDC starts with the root node `EMP0`. The first round inserts eight candidate templates (each corresponding to an attribute other than `empId` in `EMP`) to `EMP0`.L_2, and inserts three nodes `PROJ_EMP0, DEPT0, DEPT1` to the join tree.

In the second round, the algorithm checks templates in `EMP0`.L_2 (Step 1). Assume it finds that 4 of them, each with one attribute `name, phone, birthdate,` or `salary`, have enough DP; accordingly, it removes them from `EMP0`.L_2 and adds them into $\mathcal{Q}$ as they also satisfy the diversity constraint. After this process, there are 4 attributes left in `EMP0`.L_2. The algorithm then invokes APRIORIGEN and generates 6 size-3 templates from them and puts the results into `EMP0`.L_3 (Step 2). Finally, it adds size-3 templates for nodes `PROJ_EMP0, DEPT0, DEPT1` and four more nodes `PROJ0, PROJ1, EMP1, EMP2` to the join tree (Step 3).

In the third round, the algorithm starts with node `EMP0`. It turns out that all of the 6 templates in `EMP0`.L_3 have enough DP and adding them into $\mathcal{Q}$ does not violate the diversity constraints. At this point, 10 templates are found. Then, the algorithm continues until 20 templates are found in the fourth round. □

We next explain Algorithms MINIMIZEDC and APRIORIGEN.

3.2.2 *Algorithm* MINIMIZEDC

Algorithm MINIMIZEDC is a breadth-first algorithm. In the n-th iteration, it examines candidate templates with description complexity n (size-n) (lines 6-13), then generates size-$(n+1)$ candidate templates by adding one more attribute (lines 14-17) or extending all chain joins one step further (lines 18-30).

MINIMIZEDC maintains the following data structures:

- $List$: a list of nodes, each corresponding to a node in the chain-join tree and of the form of $(parent, T, \langle attr1, attr2\rangle, level, \{L_k\})$, where $parent$ corresponds to the parent node in the join tree, T is the last table in the join path, $\langle attr1, attr2\rangle$ store the join attributes,[7] $level$ is the size of the join path, and L_k is a list of size-k challenge templates;
- $Queue$: a queue of nodes as in $List$, except that $\{L_k\} = \emptyset$. The nodes in $List$ and in $Queue$ never overlap;
- $\mathcal{Q}$: the set of already generated challenge templates;
- DM: the diversity matrix for templates in $\mathcal{Q}$.

The search starts with $n = 1$ and puts the root of the chain-join tree into $Queue$ (lines 1-4). At the beginning of the n-th round, $List$ contains all nodes that contain some size-n challenge templates [8]. The algorithm checks whether those challenge templates can be added to the result set $\mathcal{Q}$ or not. Specifically, for each such challenge template Qj^T (lines 6-7), we first check if Qj^T has enough discriminating power (line 8). If it does, we further check if adding Qj^T into the result set violates the diversity threshold (line 9). If not, Qj^T is added to the result set (line 10), and the algorithm returns once there are exactly k challenge templates in $\mathcal{Q}$ (lines 11-12). Once Qj^T has enough discriminating power, we remove it from the list L_n of the corresponding node N_i (line 13); as a result, after checking all size-n challenge templates, only the non-discriminating ones are left in the corresponding $N_i.L_n$; they will be used in generating size-$(n+1)$ challenges.

[7] Extension to the multi-attribute join attribute case is straightforward.

[8] In the first round, $List$ is empty because a template is at least size-2.

Algorithm 1: MINIMIZEDC($attr, T, k, \tau_{DP}, k_l$)

Input : update column $attr$ from table T,
k, # of challenge templates to find,
τ_{DP}, discriminating power threshold
k_l, local diversity threshold
Output : a set $\mathcal{Q}$ of k challenge templates

1 $\mathcal{Q} \leftarrow \emptyset$; $DM \leftarrow \emptyset$; $Queue \leftarrow \emptyset$; $List \leftarrow \emptyset$;
2 $n \leftarrow 1$;
3 $root \leftarrow$ Node(*null, T, null, 1,* $\emptyset$); // *root of the join tree*
4 $Queue$.enqueue(*root*);
5 **while** $Queue \neq \emptyset$ *or* $List \neq \emptyset$ **do**
// *P1. Check size-n challenge templates*
6 **foreach** *node* $N_i \in List$ **do**
7 **foreach** *challenge template* $Qj^T \in N_i.L_n$ **do**
8 **if** compute-dp(Qj^T) $\geq \tau_{DP}$ **then**
9 **if** check-diversity(Qj^T, DM, τ_{DP}) **then**
10 $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{Qj^T\}$;
11 **if** $|\mathcal{Q}| = k$ **then**
12 **return** $\mathcal{Q}$;
13 $N_i.L_n \leftarrow N_i.L_n - Qj^T$;
// *P2. Generate size-$(n+1)$ templates from size-n templates*
14 **foreach** *node* $N_i \in List$ **do**
15 AprioriGen(N_i, n); // *Generate size-$(n+1)$ templates*
16 **if** $N_i.L_{n+1} = \emptyset$ **then**
17 $List \leftarrow List - N_i$;
// *P3. Extend join paths, generate size-$(n+1)$ templates*
18 $Queue$.enqueue(*separator*);
19 **while** $true$ **do**
20 $N \leftarrow Queue$.dequeue();
21 **if** $N = separator$ **then**
22 **break**;
23 **foreach** *attribute* $A_j \in N.T$ **do**
24 $Q^T \leftarrow$ gen-challenge(A_j, N);
25 $N.L_{n+1} \leftarrow N.L_{n+1} + Q^T$;
26 **if** $N.L_{n+1} \neq \emptyset$ **then**
27 $List \leftarrow List + N$;
28 **foreach** *join from* $N.T.A_s$ *to* $T_i.A_t$ **do**
29 $N_{child} \leftarrow$ Node($N, T_i, \langle N.T.A_s, T_i.A_t \rangle, n, \emptyset$);
30 $Queue$.enqueue(N_{child});
31 $n \leftarrow n + 1$;

Algorithm 2: AprioriGen(N_k, n)

Input : N_k is a node in the join tree,
n is current size of challenge templates.
Output : N_k where $N_k.L_{n+1}$ is updated.

1 $N_k.L_{n+1} \leftarrow \emptyset$; // N_k *is the* k^{th} *node in* $List$
2 $\mathcal{Q}_{ex} \leftarrow \emptyset$; // *challenge templates generated, but not in* $N_k.L_{n+1}$
// *P1: generate size-$(n+1)$ challenge templates of* N_k
3 **for** $i \leftarrow 1$ *to* $|N_k.L_n| - 1$ **do**
4 **for** $j \leftarrow i$ *to* $|N_k.L_n|$ **do**
5 $Qi^T \leftarrow i$-th challenge template in $N_k.L_n$;
6 $Qj^T \leftarrow j$-th challenge template in $N_k.L_n$;
7 $m \leftarrow$ #attributes in Qi^T ;
8 **if** Qi^T *and* Qj^T *have* $(m-1)$ *attributes in common* **then**
9 $Q^T \leftarrow$ merge(Qi^T, Qj^T);
10 **if** $Q^T \in \mathcal{Q}_{ex}$ *or* $Q^T \in N_k.L_{n+1}$ **then**
11 **continue**;
12 **foreach** *challenge template* $Q^{T'}$ *with any* m *attributes of* Q^T, $Q^{T'} \neq Qi^T$ *and* $Q^{T'} \neq Qj^T$ **do**
13 $N \leftarrow$ the join path of $Q^{T'}$;
14 $p \leftarrow 1+$#joins in the join path of N;
15 **if** $Q^{T'} \notin N.L_{m+p}$ **then**
16 $\mathcal{Q}_{ex} \leftarrow \mathcal{Q}_{ex} \cup \{Q^T\}$;
17 **break**;
18 **if** $Q^T \notin \mathcal{Q}_{ex}$ **then**
19 $N_k.L_{n+1} \leftarrow N_k.L_{n+1} + Q^T$;
// *P2: add new basis (2-attribute challenge templates) on a cross node basis if necessary*
20 **if** $N_k.level = n - 1$ **then**
21 $N_p \leftarrow N_k.parent$;
22 **while** $N_p \neq null$ **do**
23 **foreach** $Qi^T \in N_k.L_n$ **do**
24 **foreach** $Qj^T \in N_p.L_{N_p.level+1}$ **do**
25 $Q^T \leftarrow$ merge(Qi^T, Qj^T);
26 $N_k.L_{n+1} \leftarrow N_k.L_{n+1} + Q^T$;
27 $N_p \leftarrow N_p.parent$;

After checking all size-n challenge templates, the second part of a round generates size-$(n+1)$ challenge templates by adding one more attribute using Algorithm APRIORIGEN for all nodes in $List$ (lines 14-17). If L_{n+1} for a node N_i is empty, meaning that there is no size-$(n+1)$ challenge template of this node, we remove N_i from the node list $List$ (lines 16-17).

Then, the third part of a round (lines 18-30) generates size-$(n+1)$ challenge templates by extending all chain join paths one step further. Specifically, $Queue$ contains all nodes with chain joins of size n at the beginning of the n-th round; 1-attribute challenge templates of such nodes are size-$(n+1)$ challenge templates. Therefore, the algorithm generates 1-attribute challenge templates of each node in $Queue$ and puts them into the list L_{n+1} of the node (lines 23-25). If there are size-$(n+1)$ challenge templates in L_{n+1}, the corresponding node is added to the node list $List$ (lines 26-27).

As the algorithm explores the chain-join tree in a breadth-first manner, while visiting a node at the n^{th} level, it puts all its children (with one more join) into $Queue$ (lines 28-30). The use of a $separator$ in $Queue$ helps to separate nodes on different levels, and knowing when the search is complete with all nodes on a given level. Finally, after checking all size-n challenge templates and generating all size-$(n+1)$ challenge templates, the round ends up with all nodes that have at least one size-$(n+1)$ challenge template in $List$ and all nodes on the $(n+1)^{th}$-level of the join tree in the queue $Queue$.

3.2.3 *Algorithm* APRIORIGEN

Algorithm APRIORIGEN maintains a list $\mathcal{Q}_{ex}$, storing challenge templates that have been generated but not inserted in $N_k.L_{n+1}$.

In the first part of the algorithm (line 3-19), we first check every pair of challenges in $N_k.L_n$, merge them for a new challenge template only if they share $m-1$ common attributes, where m is the number of attributes in a template in $N_k.L_n$ (lines 3-9). Then, it checks if the result Q^T should be added to $N_k.L_{n+1}$ (lines 10-18). First, if the template has been generated before (so either in $N_k.L_{n+1}$ or in $\mathcal{Q}_{ex}$), there is no need to consider it again (lines 10-11). Second, if a sub-template of Q^T has enough DP, we should discard it. We check every m-attribute sub-template $Q^{T'}$ of Q^T other than Qi^T or Qj^T (line 12). Note that it is possible that none of these m attributes are from table $N_k.T$, so the involved join path can be shorter; if we assume the join path has size p, the size of $Q^{T'}$ is $m + p(\leq n)$. Thus, $Q^{T'}$ should be in list L_{m+p} of the corresponding node if it does not have enough DP (lines 15-17).

If $N_k.L_n$ contains only 1-attribute challenge templates (line 20), we go to the second part of the algorithm (line 20-27) and generate across-table 2-attribute templates. We iteratively examine each ancestor node N_p of N_k. For each pair of a non-discriminating attribute in N_p and a non-discriminating attribute in N_k, we merge them and add the result template into $N_k.L_{n+1}$ (line 21-27). Note that since N_p has size $N_p.level$, its 1-attribute non-discriminating templates should be stored in $N_p.L_{N_p.level+1}$.

Based on Lemma 3.1, we can prove the following result.

THEOREM 3.2. *Algorithm* MINIMIZEDC *solves the Problem MinimizeDC.* □

3.2.4 Enhancing Diversity of MINIMIZEDC

MINIMIZEDC proceeds breadth-first, and if $\lceil k/2^{\tau_{DV}} \rceil$ challenge templates that satisfy the DP thresholds are available from a node in the chain join tree, it picks exactly that many *before moving to the next node*. The diversity of the challenge templates produced can be further boosted by greedily maximizing the diversity at each step, using a round-robin strategy for choosing discriminating challenge templates. The resulting algorithm, MINIMIZEDCHIGHDV, has the same optimality properties as MINIMIZEDC, but tries to produce more diverse challenge templates.

A second variant of MINIMIZEDC can obtain an even higher diversity in the set of returned challenge templates, at the cost of having slightly larger challenge templates. The key idea is that, instead of checking *all* size-n challenge templates in the join tree before moving to size-$(n+1)$ challenge templates, one can proceed in a more "local" fashion. Once a node N's size-n challenge templates have all been checked, one can check the size-$(n+1)$ challenge templates of N's children nodes in the join tree, together with the size-n challenges of other nodes (e.g., sibling nodes of N). The resulting algorithm, REDUCEDDCHIGHERDV, can return challenge templates with even higher diversity than MINIMIZEDCHIGHDV, but these may have a higher description complexity.

3.3 Maximizing Diversity

Problem MaximizeDV sets thresholds on discriminating power and description complexity, and seeks a set of k challenge templates that maximizes diversity. Recall from Def. 2.5 that challenges with longer chain joins yield a higher diversity; hence, we'll explore the join tree *bottom-up*. As a result, MAXIMIZEDV may return challenge templates with higher description complexity than those returned by previous algorithms. Algorithm MAXIMIZEDV (pseudo-code omitted for lack of space) proceeds in four steps:

1. Statically identify the chain-join tree that can yield challenge templates of size τ_{DC}.
2. Iteratively choose an unmarked node whose descendants are all marked and that maximizes the diversity, generate one template from this node, and mark the node as considered.
3. When all nodes are marked, unmark all nodes.
4. Repeat Steps 2-3 until we generate k challenge templates.

THEOREM 3.3. *Algorithm* MAXIMIZEDV *solves the Problem MaximizeDV.*

Proof Sketch: Let S_g and S_o be any sets of k challenge templates found by the greedy algorithm and present in any optimal solution respectively. Let Q be any template present in S_g but not in S_o. We will show $\exists P \in S_o - S_g$: replacing P by Q in S_o cannot decrease its diversity. By repeated application of this argument, it follows that S_g must be optimal.

Table 1: Top-20 challenge templates returned by MINIMIZEDC **and** MAXIMIZEDV **for column** `EMP.empId`**.**

MINIMIZEDC	MAXIMIZEDV
EMP0.name	EMP0.workDept, DEPT6.deptId
EMP0.phoneno	EMP0.workDept, EMP5.empId
EMP0.birthdate	DEPT2.location, DEPT3.deptId
EMP0.salary	EMP0.workDept, EMP6.empId
EMP0.job, EMP0.workDept	EMP0.workDept, DEPT4.deptId
EMP0.sex, EMP0.workDept	EMP0.job, EMP2.empId
EMP0.bonus, EMP0.workDept	DEPT2.location, EMP3.empId
EMP0.job, EMP0.sex	EMP0.workDept, DEPT3.deptId
EMP0.bonus, EMP0.job	EMP0.workDept, EMP4.empId
EMP0.bonus, EMP0.sex	DEPT2.deptId
PROJ_EMP0.projId	EMP0.workDept, PROJ2.projId
PROJ_EMP0.job	EMP0.job, DEPT1.deptId
PROJ_EMP0.stdate	EMP0.workDept, PROJ_EMP1.empId
PROJ_EMP0.endate	PROJ0.projId
EMP0.workDept, DEPT0.deptId	EMP0.workDept, PROJ_EMP2.projId
EMP0.job, DEPT0.deptId	EMP0.workDept, PROJ1.projId
EMP0.sex, DEPT0.deptId	PROJ_EMP0.projId
EMP0.bonus, DEPT0.deptId	EMP0.workDept, EMP1.empId
EMP0.workDept, DEPT0.deptname	EMP0.workDept, DEPT0.deptId
EMP0.job, DEPT0.deptname	EMP0.name

There are three cases to consider. Suppose $S_o - S_g$ contains a template P corresponding to an ancestor of Q. Then replacing P by Q will increase the diversity. Suppose $S_o - S_g$ contains a proper descendant of Q. By construction, Algorithm MAXIMIZEDV does not consider ancestors before it considers descendants, so this case cannot arise. So suppose no template in $S_o - S_g$ is an ancestor or descendant of Q. Consider replacing any template P in $S_o - S_g$ by Q. Since Q does not share a prefix with any template in $S_o - S_g$ and MAXIMIZEDV chooses Q to maximize the diversity of the solution it builds, it can be shown that replacing *any* template in $S_o - S_g$ by Q cannot decrease the diversity. □

EXAMPLE 5. Comparing with the challenges found by MINIMIZEDC in Ex. 4, Tbl. 1 also shows the top-20 challenge templates returned by MAXIMIZEDV. We observe obvious differences in the returned results. MINIMIZEDC first exhausts size-n templates from one node. MAXIMIZEDV, on the contrary, generates exactly the static tree as shown in Fig. 3, and proceeds from the bottom of the tree. It first picks leaf node `DEPT6` as it has the longest join path. Then, it picks `EMP5` and then `DEPT3`. At this time, the candidate nodes to choose from include `EMP3, DEPT4, EMP6, EMP2`, each from one branch. It picks `EMP6` next as choosing it maximizes the diversity. As there are 20 nodes in the tree, the 20-th returned template is from the root node. □

4. EXPERIMENTAL EVALUATION

We implemented the four algorithms in Sec. 3, namely, MINIMIZEDC, MINIMIZEDCHIGHDV, MAXIMIZEDV, REDUCEDDCHIGHERDV. Tbl. 2 shows the input parameters and their settings; unless otherwise specified, we used default values. We experimented on two sets of data: the Corporate database–a sample database provided by IBM DB2i [5] with size about 20KB (schemas shown in Ex. 4), and the TPC-H data [13]. All algorithms were implemented in Java. All experiments were conducted on a PC with a 2.33GHz Core2 Duo processor, 3.25GB RAM and 200G SATA Disk, running Windows XP Pro SP3. MySQL (v.5.0.67) was used as the back-end database.

In Sec. 4.1, we demonstrate the effectiveness and usability of the certificates mechanism via a simulation study; Sec. 4.2 presents efficiency results for verifying certificates at run-time; finally in Sec. 4.3, we report results that show the efficiency of challenge-identification algorithms.

Table 2: Input parameters, their value ranges, and default values (underlined).

Parameter	Value
k, desired number of challenge templates	10, <u>20</u>, 50, 100, 200
τ_{minDP}, threshold on minimum DP	0, 0.2, <u>0.4</u>, 0.6, 0.8
τ_{avgDP}, threshold on average DP	0, 0.2, 0.4, 0.6, 0.8 <u>0.90</u>, 0.92, 0.94, 0.96, 0.98
τ_{DC}, threshold on description complexity	2, 4, <u>5</u>, 6, 8, 10
k_l, maximum number of challenge templates with the same chain join	2, 4, 6, 8, <u>10</u>
k_a, maximum number of challenge templates with the same response attribute	2, 4, 6, 8, <u>10</u>
df, probability decreasing factor	0.1, 0.3, <u>0.5</u>, 0.7, 0.9
τ_p, probability threshold	0.1, 0.3, 0.5, 0.7, <u>0.8</u>, 0.9

4.1 Simulation Study

We perform a simulation study on the Corporate database to show that, under natural models of an updater's knowledge of a database, update certificates catch a high percentage of erroneous updates without imposing an undue burden on updaters; similar results are observed using TPC-H.

Data preparation: First, we describe how we generate the updates and updaters for the simulation study, and then present the two models that characterize different types of updaters.

Generating updates: We consider simple record modifications of the form "`UPDATE T SET attr=val' WHERE attr=val`". The updated table `T` and updated attribute `attr` are randomly selected from the database. Different values `val` and `val'` are randomly chosen from $\mathcal{V}_{\texttt{attr}}$, the set of all values of `T.attr` and other attributes linked to `T.attr` via foreign key relationships. Also, each update is associated with its intended value, $\texttt{val}_i \in \mathcal{V}_{\texttt{attr}}$. For a correct update, $\texttt{val=val}_i$; for an incorrect update, $\texttt{val}\neq\texttt{val}_i$.

Updater models: It is natural to consider updaters who only know information about the records they update, and joinable records from other tables. Among such updaters, we categorize them in our study into *local-schema-aware* and *global-schema-aware*, depending on whether the extent of an updater's knowledge about a joinable record depends on the length of its join path from the updated table or not. The *local-schema-aware* model characterizes updaters whose knowledge of the database is centered around the updated table: the longer the chain join needed to reach a joinable record, the less likely the updater is to know about it. In contrast, the *global-schema-aware* model characterizes updaters whose knowledge of joinable records is not correlated with their distance from the updated table. Further, different updaters may know about different subsets of attributes in the schema.

We generate updater knowledge in a probabilistic manner. In the two updater models, we utilize the same join tree as used by the challenge-identification algorithms (e.g., Fig.3), to assign probability values to nodes and attributes in the join tree, denoted $P(N_{ij})$ and $P(N_{ij}.attr_k)$, respectively, where N_{ij} is the j^{th} node on level i, and contains attribute $attr_k$. N_0 denotes the updated table (the root of the join tree) and $N_0.attr$ is the updated attribute. For a given parameter df (decreasing factor), the probabilities are assigned as follows.

1. $P(N_0) = P(N_0.attr) = 1.0$, in both updater models.
2. $\forall N_{ij}$ that is a child of $N (i > 0)$, $P(N_{ij})$ is uniformly drawn from the range $[df^i, P(N)]$, in the local-schema-aware model. $P(N_{ij}) = 1.0$ in the global-schema-aware model.
3. $\forall attr_k \in N_{ij}, P(N_{ij}.attr_k)$ is uniformly drawn from the range $[df^{i+1}, P(N_{ij})]$, in the local-schema-aware model. $P(N_{ij}.attr_k)$ is uniformly drawn from the range $[df, P(N_{ij})]$ in the global-schema-aware model.

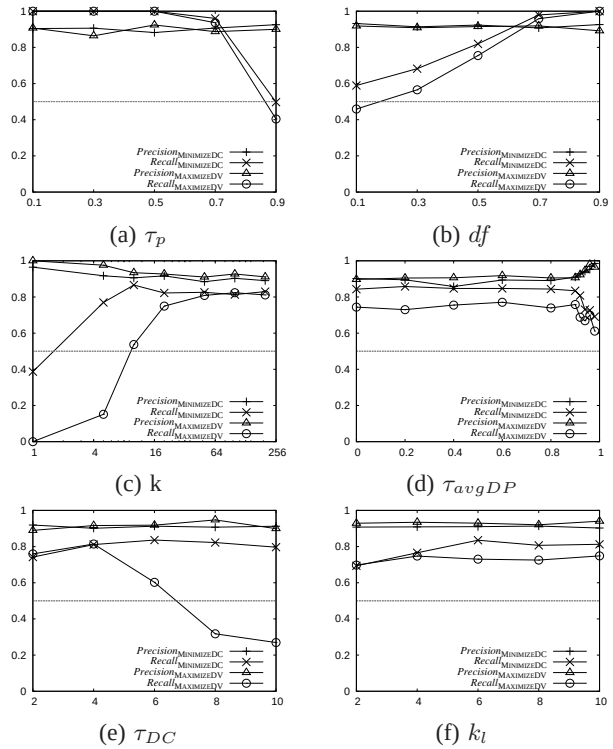


Figure 4: Local-schema aware: effect of various parameters

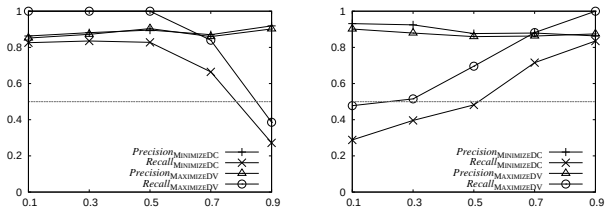
In both models, *an updater is able to answer a challenge only if she knows every attribute in the select clause of the challenge with a probability higher than a specific threshold parameter τ_p.*

Metrics: We use precision and recall of update verification as the performance metrics in the simulation study. *Precision* is defined as the fraction of the verified updates that are correct; this captures the effectiveness of update certificates in preventing unintended updates. *Recall* is defined as the fraction of correct updates that are verified; this captures the issue of ease of use of update certificates, as a correct update is not verified only if an updater's knowledge is insufficient to answer any presented challenge.

In our simulation study, we use average precision and recall for 1,000 updates consisting of half correct and half incorrect updates. Tbl. 2 lists the parameter values used in the simulation study. We show the results of MINIMIZEDC and MAXIMIZEDV – the other results are sandwiched between these two.

Results with Local-Schema-Aware Model: Fig. 4 shows the results of our simulation study on the Corporate data with the local-schema-aware model. The dotted lines show the precision (0.5) without using any update certificates. By applying the update certificates, the precision stays above 0.9 for both MINIMIZEDC and MAXIMIZEDV, across the entire range of parameters. This demonstrates that the certificate mechanism is very effective in reducing unintended updates. Hence, we focus the rest of the discussion on recall, which is a measure of ease of use of the certificates.

We observe in Fig. 4 that MAXIMIZEDV generally results in lower recall than MINIMIZEDC, especially when τ_{DC} is large. Recall from Sec. 3.3 that MAXIMIZEDV favors challenge templates with longer chain joins and results in challenge templates with higher description complexity. In the local-schema-aware model, updaters have lower ability to respond to challenges with long joins. However, with suitable parameter values, MAXIMIZEDV can get similar recall to that of MINIMIZEDC.



(a) probability threshold τ_p (b) decreasing factor df

Figure 5: Global-schema-aware: effect of various parameters

Probability threshold τ_p: Fig. 4(a) shows the results when we vary τ_p. Given a fixed probability of knowing an attribute, an updater's ability to answer challenges involving the attribute, and hence the recall, decreases with a higher τ_p. It is notable that even with a high threshold $\tau_p = 0.7$, about 95% of correct updates are verified.

Decreasing factor df: Fig. 4(b) shows the results as we vary df. With a smaller df, the probability of knowing a node in the join tree decreases faster with the depth of the node. Therefore, the recall increases significantly with larger df. When $df = 0.7$, the recalls are higher than 0.95. When $df = 0.5$ (the default value used), the recalls are about 0.8 and 0.7 for MINIMIZEDC and MAXIMIZEDV respectively; this models an updater with very limited knowledge, since the probability of knowing an attribute after one and two joins can be as low as 0.25 and 0.125, respectively; it is chosen to stress-test the ease of use issue.

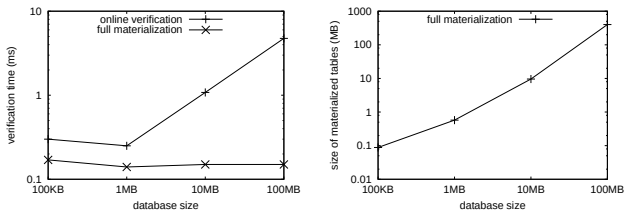
Number of challenges k: Fig. 4(c) shows the results as we vary k. When $k = 1$, the recall of MINIMIZEDC is about 0.4, which is the probability of knowing an attribute in the updated table (for $df = 0.5$ and $\tau_p = 0.8$). For MAXIMIZEDV, the recall is close to 0 since the only challenge involves a chain join which is too long (for $\tau_{DC} = 5$) to be known by the updater. With larger k, the recall increases quickly as many challenges are provided to the updater.

Average DP τ_{avgDP}: Fig. 4(d) shows the results as we vary τ_{avgDP}. When $\tau_{avgDP} < 0.9$, it does not affect the recall much; when $\tau_{avgDP} \geq 0.9$, the challenges become more complex, reducing the recall. Our simulation study also shows that τ_{minDP} has a similar effect as $\tau_{avgDP} \in [0.9, 1.0]$ (figure omitted).

Description complexity τ_{DC}: Fig. 4(e) shows the results when we vary τ_{DC}. When $\tau_{DC} = 2$, the choices of challenge templates are quite limited and almost the same for both algorithms. Therefore, both MINIMIZEDC and MAXIMIZEDV have low recall and high precision. When $\tau_{DC} \geq 6$, the recall of MAXIMIZEDV decreases significantly, since larger τ_{DC} allows complex challenges with longer joins in MAXIMIZEDV.

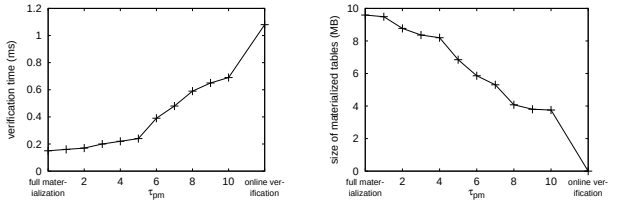
Join-level diversity k_l: Fig. 4(f) shows the results when we vary k_l. With the smallest k_l, there are at most 2 challenges available from the same node, so the average complexity of the challenges is high and the recall is low. With a higher k_l, more challenges are available from each node, so the diversity is lower, but we get a higher recall because the challenges contain fewer joins and the average complexity of the challenges is lower. The attribute-level diversity k_a has a similar effect as k_l; hence we omit the figure.

Results with Global-Schema-Aware Model: To highlight a key difference between the updater models, Fig. 5 shows results for join-challenges under the global-schema-aware model, when only a few challenges ($k = 5$) are provided. With the global-schema-aware model, the precision and recall show similar trends as a function of the various parameters, *except that* MAXIMIZEDV *consistently outperforms* MINIMIZEDC. For reasons of space, we only show results for τ_p and df.



(a) verification time (b) size of materialized tables

Figure 6: Verification overhead w. no and full materialization



(a) verification time (b) size of materialized tables

Figure 7: Verification overhead w. partial materialization.

With a more global knowledge of the schema, MAXIMIZEDV shows the added advantage of diversity, providing challenges involving different attributes and tables, giving more choice to the updaters. An updater who does not know a particular table will not be prevented from submitting her update with challenges from other tables. Note that the lower recalls in Fig. 5 compared to Fig. 4 are due to the smaller value of k used for the results of Fig. 5.

4.2 Certificate Verification

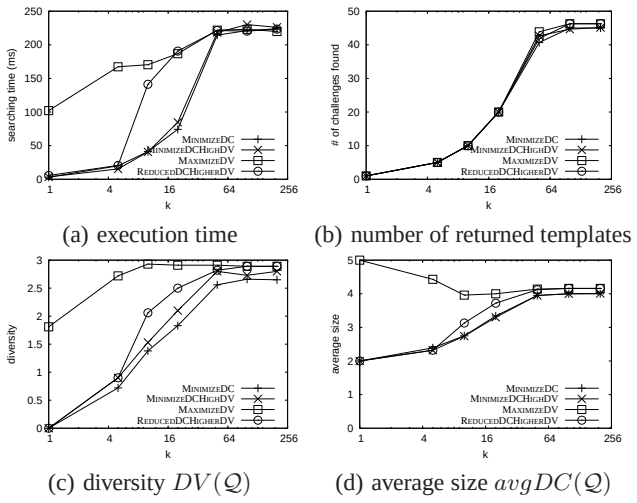
Certificate verification is an added computational cost incurred by the database system when supporting update certificates. For this reason, the efficiency and overhead of verifying certificates is an important consideration, which we investigate next.

We consider three methods: *online verification*, executing the challenge query on original tables; *full materialization*, materializing the CR-tables (see Sec. 2.2) and verifying using the materialized tables; and *partial materialization*, materializing only "critical" (which we define shortly) CR-tables, verifying on the materialized tables when such tables exist and on original tables otherwise. We take the union of the challenge templates returned by the four algorithms for each attribute in TPC-H and report (1) the average space for storing the materialized tables for all templates for an attribute, and (2) the average time for verifying on each possible update value using each template. In our implementation, we build indexes only for key and foreign-key columns; we store each materialized CR-table as a "covering index", which takes about the same space as the normal table but accelerates access to data.

Full vs no materialization: Fig. 6 shows the time and space for online verification and with full materialization on TPC-H data of different sizes. We see that without materialization, verification time increases linearly with the database size; on a 100MB database, it took 4.75ms on average. The maximum verification time (not shown) takes 2-8 times as much as the average time.

On the other hand, when we materialize all CR-tables, the average verification time is less than 0.2ms for all databases with various sizes, and the maximum verification time is less than 3 times the average, which is much faster than online verification. However, the space required for storing all materialized CR-tables for one update column grows with the size of the database, and can take as much space as the original database.

Partial materialization: As a trade-off between time and space,



(a) execution time (b) number of returned templates

(c) diversity $DV(\mathcal{Q})$ (d) average size $avgDC(\mathcal{Q})$

Figure 8: Results of various algorithms on Corporate.

we can materialize only some of the CR-tables. Selecting views for materialization to accelerate query answering has been well studied in the literature [10]. Here we explore a simple strategy, where we materialize a CR-table T if $\frac{\textit{verification time on original tables}}{\textit{verification time on materialized } T} \geq \tau_{pm}$, where τ_{pm} is a given threshold.

Fig. 7 shows the time and space cost on a 10MB TPC-H database as we varied τ_{pm} (we skip the results for 100MB TPC-H database as it shows similar trends). When $\tau_{pm} = 0$, we materialize all CR-tables; when $\tau_{pm} = 12$, we materialize no table as there does not exist any CR-table that can improve the efficiency by 12 times. As expected, the larger is τ_{pm}, the higher is the time cost for certificate verification and the lower is the space cost for CR-table materialization. As an example, when $\tau_{pm} = 5$, we can reduce the time by 78% compared with online verification, and reduce the space by 29% compared with full materialization.

4.3 Identifying Challenge Templates

In this section, we show that databases have many good challenges, and our algorithms can identify them efficiently.

Results on Corporate data: We first examine results of various algorithms on the Corporate database. Fig. 8 shows the execution time, the average number of returned challenge templates, their average size and diversity as we vary k. We have the following observations (they hold when we vary the other parameters as well).

- The algorithms run fast: even when we require top-200 templates, with the default setting, all algorithms finish in 0.25 seconds. MINIMIZEDC and MINIMIZEDCHIGHDV typically have the same running time, as they explore the same set of nodes in the join tree. When k is small, REDUCEDDCHIGHERDV is similar to MINIMIZEDC and MINIMIZEDCHIGHDV. When $k \geq 20$, REDUCEDDCHIGHERDV and MAXIMIZEDV often have similar running times: even though they explore the join tree in different directions, they end up checking most of the nodes in the join tree. The former two algorithms typically run faster than the latter two, as the latter two obtain higher diversity at the cost of exploring more complex challenge templates.
- For diversity of the results, in decreasing order the four algorithms are ranked as MAXIMIZEDV, REDUCEDDCHIGHERDV, MINIMIZEDCHIGHDV, MINIMIZEDC, consistent with our intuitions.
- For complexity of the results, in increasing order the algorithms are ranked as MINIMIZEDC/MINIMIZEDCHIGHDV (same size for these two algorithms), REDUCEDDCHIGHERDV, MAXIMIZEDV, also consistent with our intuitions.

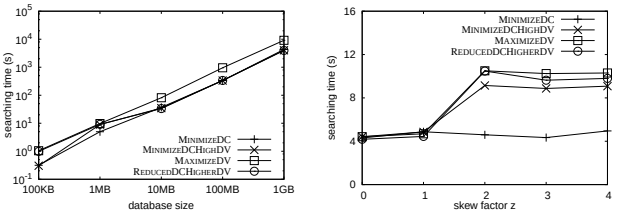


Figure 9: Effect of size **Figure 10: Effect of skew**

As k increases, typically the execution time, the number of returned templates, their diversity and average size all increase. This is because the higher the k, the more challenges returned (so often higher diversity), the larger search space to explore (so higher execution time), and the less flexibility of discarding a template to reduce complexity (so higher average size). The only exception is that when k is increased from 1 to 10, the average size of MAXIMIZEDV's results decreases. This is because to maximize diversity, MAXIMIZEDV starts with nodes with longer join paths; when k is increased from 1 to 10, MAXIMIZEDV is able to add templates with shorter joins (see Ex. 5), but when k is increased further, MAXIMIZEDV has to select templates from the same nodes to meet the requirement, so the average size goes up slightly again. Finally, note that measures for different algorithms converge when $k = 200$, as there are not enough results (on average 45.3 templates are returned) so all methods explore the full search space.

Results on TPC-H data: We next experiment on TPC-H and investigate the effect of data size and skew on our algorithms. We examine each update attribute in table `CUSTOMER` (which has many attributes and can form both (key, foreign key) join and (foreign key, key) join with other tables) and report average results.

We first generate uniform data and vary the size of the data from 100K to 1G. Fig. 9 reports the execution time. We have the following observations. (1) The efficiency of most of our algorithms is acceptable, since challenge identification is performed offline: on 1G data, MINIMIZEDC, MINIMIZEDCHIGHDV and REDUCEDDCHIGHERDV on average terminated in about 60 minutes, and MAXIMIZEDV terminated in about 150 minutes; in addition, the execution time increases linearly with the size of the data. MAXIMIZEDV takes longer time because it checks DP for certificate templates with more joins. (2) Unlike on Corporate data, the execution time of REDUCEDDCHIGHERDV is close to that of MINIMIZEDC and MINIMIZEDCHIGHDV in most cases. This is because each table in TPC-H has a large number of attributes and the data are uniform, leading to more 1-attribute template results and so REDUCEDDCHIGHERDV does not need to go deep in the join tree. (3) The Corporate database has a much lower execution cost than the smallest TPC-H data, showing that schema complexity affects the join-tree structure and the number of templates, and hence the execution time. We also measured the diversity and average size of the returned results (not shown), and observed that the results are quite stable, conforming to our intuition that, with uniformly distributed data, size should not affect result diversity and complexity.

We next generate 1M data with various skews using the tool in [13]. We vary the skew factor z (the Zipfan distribution) of our data from 0 (uniform) to 4 and report the results in Fig. 10. When z increases from 1 to 2, there is a jump in execution time for various methods. Intuitively, when the data is more skewed, the $minDP$ and $avgDP$ of a challenge template drops; when $z = 2$, they drop

to a level that cannot meet the DP thresholds, so we have to explore more complex challenges. Note that MINIMIZEDC is not affected much because it favors certificates with more attributes than those with longer chain joins, which can take shorter time for DP checking; however, if we increase k or τ_{DC}, the execution time of MINIMIZEDC would be closer to other methods. We also measured the diversity and average size of the returned results (not shown), and observed that the result complexity increases slightly when z increases from 1 to 2 (for the same reason as discussed above), but the result diversity remains stable.

4.4 Summary of Experimental Evaluation

We summarize our experimental results as follows.

- The use of update certificates successfully prevents most unintended updates, across a range of values for the DP parameters. Further, for moderate values of the complexity and diversity parameters, updaters can respond to at least one of the provided challenges, addressing the ease of use concern.
- Verifying certificates on the database is in general quite efficient, and we can further reduce the time cost by partially materializing the CR-tables.
- The challenge-identification algorithms have acceptable efficiency considering that the algorithms are executed offline.
- We have the following algorithm recommendations. When most updaters are local-schema-aware and complexity of the returned challenges is important, use MINIMIZEDCHIGHDV as it minimizes the average complexity, obtains higher diversity than MINIMIZEDC, and is almost as cheap as MINIMIZEDC. When most updaters are global-schema-aware and diversity of the returned challenges is important, use MAXIMIZEDV. To balance complexity and diversity, for a mixed workload of users use REDUCEDDCHIGHERDV.

5. RELATED WORK

Challenge-response techniques are widely used in practice for user identification; e.g., when a user needs to reset her password, she may be asked to provide her mother's maiden name. Such *user identification* is complementary to our problem of database update validation, and there has been no study of a formal notion of discriminating power of the challenges presented to the user.

Our DP measure to assess discriminating power is similar to frequency based measures (e.g., tf/idf) from Information Retrieval [12]. A key difference is that tf/idf is based on the occurrence frequency of a value, while DP is based on the frequency of a response value in the answer to all possible challenge queries. These two measures can be very different in practice.

Integrity constraints are quite powerful and useful to capture a class of erroneous updates [14]. For example, functional dependencies (FDs) and inclusion dependencies (IDs) can be used to ensure database consistency. Recent work has explored conditional variants of FDs and IDs to allow for more flexibility in having the constraint hold on a portion of the database, and also make assertions [6]. While these can achieve consistency under updates at a finer granularity, it is not practical to specify out of existence, every conceivable update error due to innocent mistakes, whether one uses CFDs or some other constraints, as illustrated by Ex. 1. In a similar spirit, the notion of matching dependencies (MDs) has been recently introduced [7, 2] as a means for capturing semantics of records in unreliable relations and in particular to help with record matching; the motivation for MDs is orthogonal to our work.

Finally, the notion of diversity has gained prominence in many areas such as recommender systems and query answering. [8] studies diversity from an axiomatic perspective and contains pointers to other works on diversity. Our notion of diversity is in the context of challenges, which are queries. To our knowledge, there has been no prior work on diversifying SQL query *expressions*.

6. CONCLUSIONS AND FUTURE WORK

The problem of ensuring that updates do not introduce errors into a database is an old and vexing problem, and existing techniques based on integrity constraints are inadequate to detect a large variety of errors that arise due to carelessness on the part of updaters. In this paper, we advocate the use of *update certificates*, a novel approach to detect erroneous updates that are unintended mistakes. We characterize good certificates as those with high *discriminating power*, low *description complexity*, and high *diversity*. We present algorithms to analyze databases and identify good challenges, and experimentally show that databases tend to have many good update certificates, these can be efficiently identified and verified, and are very effective in catching erroneous updates.

Our paper establishes the foundations of a novel approach to address the problem of erroneous updates in databases, but many interesting questions remain. A challenging problem is how to reduce the effort of updaters for multiple updates in an update transaction, without sacrificing the ability to detect erroneous updates. Another problem is to efficiently revise challenge templates as the database instance evolves in response to updates. Understanding how our techniques can work with conditional integrity constraints to comprehensively address the critical problem of erroneous updates is also an interesting direction of future work.

Acknowledgements

We would like to thank the anonymous reviewers for their many suggestions, which helped improve the quality of our paper.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
[2] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.
[3] L. Berti-Equille and T. Dasu. New directions in data quality mining. In *KDD*, 2009.
[4] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.
[5] Ibm db2 for i. http://www-03.ibm.com/systems/i/software/db2/sqldata.html.
[6] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.
[7] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2009.
[8] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, 2009.
[9] N. Good, J. Grossklags, D. K. Mulligan, and J. A. Konstan. Noticing notice: a large-scale experiment on the timing of software license agreements. In *CHI*, 2007.
[10] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *TKDE*, 17(1):24–43, 2005.
[11] M. Mannan and P. C. van Oorschot. Security and usability: The gap in real-world online banking. In *New Security Paradigms Workshop (NSPW)*, Sept.18-21 2007.
[12] G. Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, New York, 1983.
[13] Tools for controlling data skew of TPC-H data from Microsoft Research. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.
[14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, 1989.