Side-Effect Estimation: A Filtering Approach to the View Update Problem

Yun Peng, Byron Choi, Jianliang Xu, Haibo Hu, Sourav S Bhowmick

Abstract—Views and their updates have long been a fundamental technology required in a wide range of applications. However, it has been known that updates through views is a classical intractable problem. In this paper, we propose a novel, *data-oriented* approach to this problem that provides a *practical support* for view updates. In particular, we propose a summarization of the source database of views, which serves as an *update filter*. The update filter aims to efficiently reject untranslatable view updates by *estimating the side effects* of the updates, thereby avoiding costly translation analysis. For applications where estimation errors are not preferred, our update filter can be tuned to be exact. In this paper, we present our approach with SPJ views, an important class of view definitions. We first revise the notion of estimation errors to quantify the filter's qualities. We then propose a novel *join cardinality summary* (`JCard`) derived from cardinality equivalence. An estimation algorithm is proposed. Finally, we present optimizations enabling the construction of an accurate `JCard` through heuristics and sampling. Our extensive experiments show that update filters are efficient and can be easily tuned to produce accurate estimations on TPC-H and DBLP.

Index Terms—View update, side-effect estimation, relational database

1 Introduction

Views have been an important facility provided by DBMSs that allow users to access specific parts of a database. Over the years, views have remained useful in a wide range of emerging applications, such as data publishing or information dissemination [12], XML or RDF query rewriting [32], [37], query optimization [26] and tracing facility in P2P networks [35]. It is evident that these applications not only query, but also update views as if they were the actual database. Updates on views are translated to updates on the source databases of the views, and the views and its source databases must keep consistent after updates.

Example 1.1: To illustrate the view update problem, let us consider an example as shown in Fig. 1. Suppose we have a view V that joins `Product`, `Supplier`, `Order`, and `Agency`. (For illustration purposes, this example omits integrity constraints.) Suppose we insert (S_4, FL, P_6, C_3, A_5) into V as indicated by u_1. The only way is to insert (S_4, FL) into `Supplier` and (A_5, P_6) into `Agency`. However, since tuple p_2 is also joinable with tuple s_1, the insertions of (S_4, FL) and (A_5, P_6) cause an unspecified effect of inserting (S_4, LA, P_6, C_3, A_5) into V. In fact, we cannot translate u_1 without causing extraneous tuple(s) in V.

Updates through views have been one of the classical problems in databases. However, data are now ubiquitous and often provide insights to seemingly hard problems (*e.g.*, [8]). In this paper, we propose a new data-oriented approach

Product (P)

	supplier	*part*
p_1	S_5	P_4
p_2	S_4	P_6
p_3	S_1	P_1
p_4	S_2	P_2
p_5	S_5	P_2
p_6	S_6	P_3

Supplier (S)

	supplier	*location*
s_1	S_4	LA
s_2	S_1	LA
s_3	S_1	NY
s_4	S_2	NY
s_5	S_3	KS

Order (O)

	client	*part*
o_1	C_3	P_4
o_2	C_3	P_6
o_3	C_1	P_1
o_4	C_2	P_2

Agency (A)

	agent	*part*
a_1	A_4	P_1
a_2	A_3	P_2
a_3	A_1	P_3

View $V = \pi_{supplier,location,part,client,agent}(S \bowtie P \bowtie O \bowtie A)$

	supplier	*location*	*part*	*client*	*agent*
v_1	S_1	LA	P_1	C_1	A_4
v_2	S_1	NY	P_1	C_1	A_4
v_3	S_2	NY	P_2	C_2	A_3

view updates:
u_1: insert (S_4,FL,P_6,C_3,A_5)
u_2: insert (S_5,FL,P_6,C_3,A_4)

Fig. 1. An example of a view and source database tables

that extensively exploits data summaries in source databases (in addition to schema information and view definitions) to extend *practical support of view update(s)*. To our knowledge, excepting theoretical studies on complement views, prior work has not explicitly exploited summaries of source data.

View update analysis can often be computationally expensive. For instance, the view update problems under many settings are NP-hard [13], [14]. View update analysis includes two major steps: side-effect analysis and view update translation. Much extant work directly translates view updates, which can sometimes be inefficient. In contrast, we focus on side-effect analysis. In addition, we observe that in practice, view updates often cause side effects. We propose to reject (also referred to *filter*) such view updates early in our side-effect detection. Only side-effect free updates are passed to the necessarily heuristic update translation. More importantly, when *certain errors in the detection are allowed*, which have not been proposed before, the detection can be significantly faster. As a result, view updates that are (or estimated) untranslatable are rejected early, enabling less costly update translation.

More specifically, we propose a *Data-oriented View Updater*. The side-effect detector and update translator in our updater exploit data summaries of source databases, in addition

- *Yun Peng, Byron Choi, Jianliang Xu and Haibo Hu are with the Department of Computer Science, Hong Kong Baptist University, China.*
  E-mail: ypeng, bchoi, xujl, haibo@comp.hkbu.edu.hk
- *S.S. Bhowmick is with the School of Computer Engineering, Nanyang Technological University, Singapore.*
  E-mail: assourav@ntu.edu.sg

to database schemas and view definitions.

Overview of our updater. Fig. 2 depicts an overview of our proposed updater. (i) The side-effect detector efficiently estimates side effects caused by a view update. (ii) It rejects those with side effects. The detector can be tuned to be exact (*i.e., no estimation error*) such that no tuple in a source database is ever summarized. (iii) Updates of the view that are not filtered by the detector are processed by an update translator. (iv) Translatable view updates are applied to the database, and the view and the side-effect detector. Our updater is independent of translation algorithms. For illustration purposes, we assume the heuristics of SAT [12], amongst the ample work on translation algorithms. In this paper, we focus on the details of a side-effect detector.

Example 1.2: The performance improvement from a side-effect detector can be illustrated via a simple experiment with TPC-H benchmark dataset (1G bytes) and a view definition derived from a simplified Q7 of the benchmark. (The details of Q7 are presented in Sec. 10.) Note that the side-effect detector is tunable to compromise between estimation time and error. In the experiment, the detection time of Q7 is approximately 3s whereas the estimation time is smaller than one second. We can easily tune the detector to be error free. Let us denote the side-effect estimation time as t. Assume variable T captures the time for update translation which is at least 100s. That is, $t \ll T$. Further assume that 20% of view updates are side-effect free view updates, whereas 80% of the updates have side effects. Without the side-effect detector, the view update time is simply T (*i.e.*, at least 100s), whereas the view update time with a side-effect detector with no observed error is t+0.2T. Therefore, the view update analysis time is reduced by almost 80%, from T to t+0.2T.

Detecting side effects can be significantly more efficient than translating updates (recall Example 1.2), as a detector only requires to signify the *absence* and *presence* of side effects. To support this, we propose a *Join Cardinality Summary* (`JCard`). The novelty of `JCard` relies on the structures of *join cardinality equivalence classes* and *candidate view tuples*. First, (i) we summarize the tuples of a database by equivalence classes. `JCard` is often very small and side-effect estimation on `JCard` is efficient. Although the estimated count of side effects using the equivalence classes may be far from the exact one, it is sufficient to detect the presence of side effects. Second, (ii) to support accurate estimations, we propose to refine the classes by selecting certain candidate view tuples based on how likely they are to appear in views. Our experiments show that using a small number of candidate tuples leads to highly accurate estimations.

We remark that due to the nature of the view update problem, any practical algorithm that runs in PTIME necessarily rejects some translatable updates. Thus, side-effect estimation allows to introduce a small number of additional errors but further optimize the detection time. In Example 1.2, the detection time was around 3s and the estimation time was smaller than one second, whereas our summarization did not introduce errors.

Finally, the overall benefit of the side-effect detector also depends on the percentage of view updates with side effects. For instance, following up on Example 1.2, if view updates having side effects account for x% of all updates, our updater reduces the time of view update analysis by up to x%. We observe from popular real-world and synthetic benchmark datasets and random updates that view updates with side effects can be clearly more than those without. Thus, in practice, our side-effect detector has a high potential of avoiding potentially costly update translations.

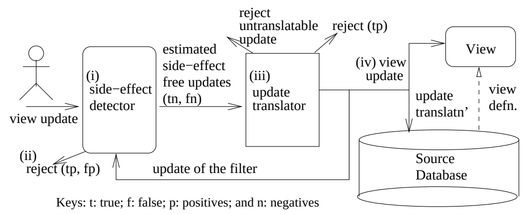


Fig. 2. The overview of our data-oriented view updater

Contributions. The main contributions are as follows.

1. We show that KL divergence can serve as an upper bound of the estimation error of our side-effect detector. Furthermore, we revise the notion of errors of estimation.

2. We propose a novel summary structure for our side-effect detector, called *Join Cardinality summary* `JCard`. At the core of `JCard` is the notion of cardinality equivalence of tuples. `JCard` comprises two structures: (i) database summaries and (ii) a set of candidate tuples. We present the construction of `JCard` and side-effect estimation algorithms.

3. We propose techniques to support side-effect estimations of *insertions, deletions, and replacements* on the SJ view. To support projections, we propose an extension of `JCard` based on value-cardinality equivalence.

4. We formally define two optimization problems in `JCard` construction, and propose a heuristic and a sampling solution to these problems, respectively. In particular, (i) we establish that our candidate tuple selection problem is equivalent to the Minimum Set Cover (MSC) problem. We therefore adopt heuristics for MSC to address our problem. (ii) To determine the optimal representation of a view definition, we employ a simple sampling technique (*e.g*, estimation of proportion) to quantify the estimation error of view definition representations.

5. We conduct a set of extensive experiments with synthetic and real datasets that verifies the effectiveness and efficiency of our proposed techniques and compare with one of the latest related work. Our experiments show that our side-effect detector can be easily tuned to be accurate.

Organization. The rest of this paper is organized as follows: Sec. 2 presents the related work. Sec. 3 provides preliminaries and the problem statement. We define the notion of errors for side-effect estimation in Sec. 4. Our join cardinality summary `JCard` is presented in Sec. 5, and Sec. 6 presents estimation algorithms on `JCard`. We study how `JCard` supports deletions and replacements in Sec. 7, and projections are supported in Sec. 8. Next, Sec. 9 addresses two optimization problems in the construction of `JCard`. An experimental evaluation is presented in Sec. 10. Finally, Sec. 11 concludes this work. All proofs are presented in Appendix A.

2 Related Work

The view update problem is one of the classical problems in databases [15], [18], [19]. In addition, view updates are involved in recent research and applications, such as [5], [11], [20], [36]. Due to space constraints, this section only presents certain non-exhaustive representative works relevant to our approach. One may consult the literature for more complete reviews of the problem (*e.g.*, [11], [30]).

Previous researchers [15], [19], [29] studied the view update problem under various view definition syntaxes (*e.g.*, select, project, acyclic join) and constraints (special forms of primary and foreign key constraints and functional dependencies) of source databases. Prior results suggest that limited support of view updates can be achieved in DBMSs. In particular, a seminal paper by Dayal and Bernstein [19] showed that the view update problem was undecidable under various settings. Different from other existing works, Keller [29] defined five criteria of correctness of view update translations and proposed algorithms to enforce correct update translations. In contrast, this paper focuses on capitalizing on a view's source data for practical support of view updates.

Bancilhon and Spyratos proposed the seminal work on view complements [15]. View updates could be translated without side effects if the updates resulted in unchanged view complements (*a.k.a* translation under a constant complement). The results of view complements were followed-up in the proposal of "consistent views" [24]. View complements are source data that are analyzed with view translation. Finding the minimum view complement is, in general, intractable [33], [34] and related research covers only the theoretical aspects of view complements. To our knowledge no research on the practical applications of view complements exists.

Other related research includes updates through XML views [4], [7], [12], [45]. One stream of research (*e.g.* [7]) cast an XML view into relational views and exploited relational techniques to solve the XML view update problem. Another stream of work (*e.g.*, [4], [12]) addressed updates through recursive XML views. Previous work [12] used source data to encode an XML view update as a SAT instance and used a SAT solver to determine update translation. Yet, how source data can be exploited to optimize update translation remains unexplored. Recently, a framework with polymorphic type inferences and lineage tracing approaches has been proposed to support updates through restricted XML views [20]. However, its advantages may not be obvious for relational views.

Recent progress on view updates includes the follows. Bi-directional transformation of *trees* [21] permits operations for universal data (concrete tree and abstract view) synchronization and relational view updates [3]. Kotidis et al. [31] introduced and exploited physical IDs for view updates, which requires an intrusion on the physical layer of DBMSs. Boneva et al. [5] proposed tree automata techniques for determining update programs for a fragment of XML view updates, with and without constraints. Liu et al. [36] proposed the view update analysis in a pure XML context. Cong et al. [13], [14] studied the time complexities of various versions of view update.

Regarding filtering, Luo et al. [38] proposed a filter for the view maintenance problem. However, view maintenance is the "inverse" problem of view update. Specifically, the former propagates updates of source databases to views, whereas the latter propagates updates of views to source databases.

TABLE 1
Table of frequently used notations

I	database instance	R or R_i	relation
$\mathcal{V}$	view definition	$J_{\mathcal{V}}$ (J_R)	join tree (rooted at R)
V=$\mathcal{V}(I)$	view	G	database graph
t_v	a tuple in view	G^+	source graph
t_i	segment of t_v for R_i	G^-	negative graph

There are some studies of view update with lineage tracing (sometimes referred to as data provenance) in the literature. Lineage describes the origins of data and/or its processing history in databases [9], [17]. Lineage has been studied in various application contexts (*e.g.* data warehouse [17], uncertainty data [1], [46], scientific data [6]). Specifically, Cui et al. [17] proposed techniques to trace the lineage of view tuples and their transformations in data warehouses. However, [17] focuses on lineage computation and does not study view update. The TRIO project [46] aims at integrating the management of data, data accuracy and its lineage. A prototype of TRIO can be found in ULDB [1]. However, TRIO and ULDB focus on uncertain data. More importantly, [1], [17] determine exact lineage. As motivated, if adopted, exact side-effect detection can be inefficient. In contrast, we propose an estimator that can be tuned to be inexact. Green et al. [25] proposed ORCHESTRA that uses lineage to support data sharing across large communities and propagates updates from one peer to another. In contrast, we study propagating updates of views to source databases. Bhagwat et al. [2] proposed an annotation management system. It associates an annotation to each cell of a relation, which is termed as *where*-provenance in [10]. However, *where*-provenance misses the join information of view tuples, as the join attributes may be projected out. In comparison, the view update problem studied in this paper requires *why*-provenance [10]. Cui et al. [16] and Buneman et al. [10] used *why*-provenance to support view deletions. As presented in [16], [41], it is not clear how lineage to support insertions, as tuples to-be-inserted (and hence and their lineage) in general do not exist in the views. In comparison, we support both insertions and deletions. Due to space restriction, we refer the interested readers to [41] for details of lineage.

In relational databases, selectivity estimation is one of the key steps in query optimization. Query optimizers rely on accurate result counts of (sub-)queries to quantify evaluation times of query plans. Classical techniques include building histograms of data and utilizing statistical assumptions, such as uniform distribution and independence of join predicates [43]. Recently, Getoor et al. [22] proposed a uniform framework to estimate the count of select-and-join queries, derived from Bayesian networks. Bayesian networks minimize overall errors, whereas side-effect estimation intuitively focuses on small errors only and large errors are simply irrelevant.

Up-to-date, the support of updates through views in commercial relational DBMSs (e.g., [27], [39], [42]) is provided for rather restricted views. In contrast, this work proposes an optimization of side-effect estimation to extend DBMSs with the capability to support practical view updates.

3 Preliminaries and Problem Statement

This section presents the preliminaries and the problem statement of this paper.

We use $\oplus$ to denote an application of an update, *e.g.*, $V' = u \oplus V$ denotes the view after updating V with u.

Syntax for view definition. We present our techniques with an important class of view definitions, namely *select*, *project* and *join* queries (SPJ queries). We remark that any SPJ query can be converted into the following normal form in linear time:

$$\mathcal{V} = \pi_A(\sigma_P(R_1 \bowtie R_2 ... \bowtie R_n)),$$

where A is the set of projection attributes of the view, P is the selection predicates and R_is are relations for $i \in \{1,...,n\}$. Hence, the technical discussions of this paper assume this normal form. We mostly focus on side-effect detection with join queries, as they are the most technically challenging. Then we present extensions to support projections. In this paper, we assume the joins form a tree. Join trees are common in practice and can simplify our side-effect detection. Note that each relation in $\mathcal{V}$ can serve as the root of a join tree and different join trees lead to different estimation accuracies.

Source of view tuple. A view tuple t_v can be decomposed into n *segments*, t_1,..., t_n, where $t_i = \pi_{A_i}(t_v)$ and A_i denotes the attributes of R_i in $\mathcal{V}$. We call $\cup_{i=1...n}$ $\{t_i \mid t_i$ is a segment of t_v and $t_i \in R_i\}$ *the source of* t_v. For example, $\{s_2, p_3, o_3, a_1\}$ is the source of the view tuple v_1 in Fig. 1. We remark that in general, t_i may not always exist in the source database.

View update problem. The view update problem can be presented as follows: *Given a view definition $\mathcal{V}$ of a relational database instance I, where the view V is $\mathcal{V}(I)$, and an update u on the view V, find a translated update u' on I such that $u \oplus \mathcal{V}(I) = \mathcal{V}(I \oplus u')$.*

An update u having a possible update translation u' is called a *translatable update*. Otherwise, u is called an *untranslatable update*. Untranslatable view updates are untranslatable often because they lead to certain side effects [29]. We recall the definition of side effects as follows.

Side effects. Given an update u on a view $V = \mathcal{V}(I)$ and its translation u' on I, *side effects* of u' are the changes on the updated view $\mathcal{V}(I \oplus u')$ that are not specified by u.

Much previous work on view updates (*e.g.,* [12]) involves rejecting untranslatable view updates as early as possible, proposed as an optimization of update translation.

This paper proposes a side-effect detector, which may be further tuned to be an efficient estimator. Our side-effect detector has two types of errors. (i) False positives denote that the view updates do not lead to side effects but the detector declares side effects and rejects the updates. (ii) False negatives denote that the view updates cause side effects in fact but the detector declares no side effect. False positives are the *true error* of our view updater, whereas false negatives are a performance issue as they will be detected in the update translation. False positives and false negatives will be defined formally in the next section.

4 Quality of Side-effect Detector

This section presents the notion of errors used in our proposed detector. A side-effect detector can be considered as a special form of a probability distribution estimator. Sec. 4.1 presents the relationship between the Kullback-Leibler (KL) divergence of probability distribution estimation and the expected error of our side-effect estimation. First, not surprisingly, the expected side-effect estimation error is bounded by the KL divergence. Specifically, if the KL divergence of the probability distribution estimation tends to zero, the expected side-effect estimation error tends to zero as well. Second, in the context of updates, it is not feasible to enumerate the infinitely-many future updates. Sec. 4.2 presents a revised notion of errors on a finite set of updates. In the analysis, we do not consider alternate sequences of updates where errors depend on the choice of update sequence not detectors. For presentation simplicity, we present our analysis with insertions, unless otherwise specified, as deletions are obviously bounded by the view.

4.1 Analysis with KL Divergence

To describe the analysis of the estimation quality, we first recall the definition of KL divergence (KL), which is used to measure the distance between the real and estimated distributions. Let $P_D(\mathbb{X})$ and $P_E(\mathbb{X})$ be the real distribution and the estimated distribution of a *random variable* $\mathbb{X}$, respectively. Then, we have the following:

$$KL(P_D(\mathbb{X})||P_E(\mathbb{X})) = \sum_{X \in \mathbb{X}} P_D(X) \log \frac{P_D(X)}{P_E(X)}$$

We use random variables $\mathbb{S}$ and $\mathbb{J}$ to capture possible selections and joins on attributes of all possible views $\mathbb{V}$ on a database instance I. We use a random variable $\mathbb{M}$ to denote possible insertions. (An insertion event $M \in \mathbb{M}$ is modeled as (S, J), where $S \in \mathbb{S}$ and $J \in \mathbb{J}$.) For simplicity, we skip projections in this analysis, as detectors can be extended with projections (detailed in Sec. 8). We remark that an insertion may involve attributes besides S and J. The analysis of these attributes is trivial and hence omitted. Let $|V|$ denote the size of view $V \in \mathbb{V}$ before an insertion and let $f_{real}(M, V)$ and $f_{est}(M, V)$ denote the real and estimated view sizes after an insertion, respectively. We may omit M and V when they are clear from the context.

As discussed at the end of Sec. 3, a side-effect detector has two kinds of errors: false positives E_+ and false negatives E_-. Let θ be a user-tunable parameter of acceptable estimation errors, where $0 \leq \theta \leq 1$. The formal definitions of E_+ and E_- are given below.

Definition 4.1: False positives $E_+(M,V)$: Given an insertion M on a view V, $E_+(M,V)$ is defined as follows:

- $E_+(M,V)$=1, if f_{real}=$|V|$+1 and $f_{est} \geq f_{real}+\theta$; and
- $E_+(M,V)$=0, otherwise. □

Definition 4.2: False negatives $E_-(M,V)$: Given an insertion M on a view V, $E_-(M,V)$ is defined as follows:

- $E_-(M,V)$=1, if $f_{real}>|V|+1$ and $f_{est}<|V|+1+\theta$; and
- $E_-(M,V)$=0, otherwise. □

We remark that E_+ and E_- are two binary random variables. The expected error can be expressed as the integration of all possible insertions $\mathbb{M}$ on all possible views over their *probability distribution* $P(\mathbb{M}, \mathbb{V})$:

$$EXP(E_+(\mathbb{M}, \mathbb{V})) = \sum_{M \in \mathbb{M}} \sum_{V \in \mathbb{V}} P(M, V) \times E_+(M, V).$$

We highlight that a side-effect detector is not simply a classical estimator of cardinalities. Classical works on selectivity estimation focus on *overall* accuracies of estimated cardinalities. However, side-effect estimation requires only an accurate estimation of cardinalities that signifies the *boundary between the absence or presence of side effects*.

Example 4.1: To illustrate the main difference between side-effect estimation and cardinality estimation, we present two detectors, namely $J_{lineitem}$ and J_{order}, on a simplified Q7 of TPC-H. While the details of these detectors are presented in Sec. 10, they can now be understood as detectors using different summaries of the result of Q7 and the source database. We set θ to 1. That is, a view insertion with side effects changes the view size by a number greater than or equal to 2. (An illustration of the experimental result of estimation is shown in Fig. 18 in Appendix B.) We tried 1800 insertions, *all* with side effects. We report that the error of *cardinalities* of $J_{lineitem}$ was 3.7 times smaller than that of J_{order}. In contrast, a side-effect detector requires to distinguish (i) the insertions that change the view size by 1 and (ii) those greater than 1. The exact change in cardinalities that is larger than 1 is not important. With this notion of errors, our experiment found that $J_{lineitem}$ exhibited 79 incorrect side-effect estimations while J_{order} did not produce any incorrect estimation.

While false positives and false negatives of a side-effect detector are defined with an error threshold θ, the quality of a detector still exhibits a close relationship with the KL divergence. These are summarized in Propositions 4.1 and 4.2.

Proposition 4.1: *When* KL *divergence of the estimated distribution of insertions tends to 0, i.e.,* $KL(P_D(\mathbb{M},\mathbb{V}||P_E(\mathbb{M},\mathbb{V})) \to 0$, *the expected false positives of the detector tends to 0, i.e.,* $EXP(E_+(\mathbb{M},\mathbb{V})) \to 0$. □

Proof idea: We model the possible views and their updates by random variables. The expected value of false positives is expressed in terms of these variables. In the arithmetic derivations, we apply the Markov's inequality and Pinsker's inequality to obtain a bound that consists of KL divergence. Then, we can easily show that as KL divergence tends to zero, so does the expected values. Please refer to the full arithmetic derivations in Appendix A.1

Proposition 4.2: *When* KL *divergence of the estimated distribution of insertions tends to 0, i.e.,* $KL(P_D(\mathbb{M},\mathbb{V})||P_E(\mathbb{M},\mathbb{V})) \to 0$, *the expected false negatives of the detector tends to 0, i.e.,* $EXP(E_-(\mathbb{M},\mathbb{V})) \to 0$. □

Proof idea: Similar to that of Proposition 4.1.

4.2 Revised Notion of Errors

Propositions 4.1 and 4.2 show that it makes sense to construct a side-effect detector by minimizing the KL divergence between the detector and the actual data. The unique problem in a side-effect detector is that the KL divergences are defined on all possible future *insertions*, which can be infinitely-many. However, the *relative qualities* of detectors only depend on *certain attributes* of the database, and more importantly, a *finite set* M' of insertions.

Let t_v denote the view tuple to-be-inserted and $t_v.A$ denote the value of A in t_v. Let $dom(A)$ and $adom(A)$ denote the domain and active domain of attribute A, respectively. Next, we present the attributes and insertions that are irrelevant to the relative qualities of the detectors.

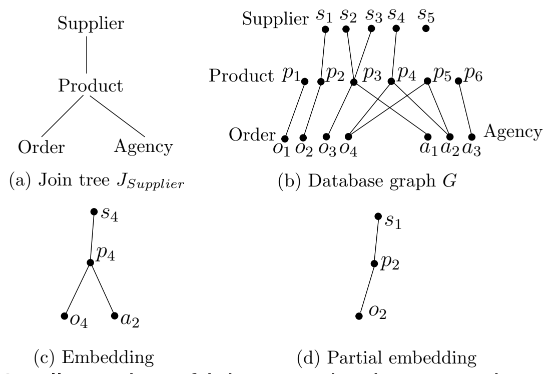


Fig. 3. Illustration of join tree, database graph, embedding and partial embedding

(i) Suppose attribute A is projected out. If $|dom(A)|$ is infinite, or $|dom(A)|$ is finite but $|adom(A)|<|dom(A)|$, A is irrelevant to the qualities of detectors as an update translation can always find a new value of A without causing side effects.

(ii) On the other hand, if A is in a view definition, regardless of its domain. If $t_v.A$ is new, $t_v.A$ does not cause side effects and is irrelevant to the detector's quality. Otherwise, $t_v.A$ exists in the database and the number of its possible values is bounded by $|adom(A)|$.

Based on the observations above, in Definition 4.3, we formalize the effective insertions that affects detectors' qualities. We remark that Definition 4.3 is independent of any error metrics. Moreover, this notion of errors is useful in building summaries for side-effect estimation/detection.

Definition 4.3: *Effective insertions* of a view $\mathcal{V}(I)$ are a finite set M', where M' is an enumeration of insertions on:

- the active domains of the attributes on $\mathcal{V}$; and
- the domains of the attributes of finite domains in the relations participating in $\mathcal{V}$.

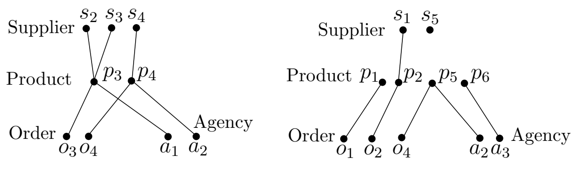
The *revised error* of a detector is its error on M'.

5 Join Cardinality Summary (JCard)

In this section, we propose a *join cardinality summary* (`JCard`) for estimating side effects. `JCard` is specially designed for joins since joins are technically challenging in SPJ views. For illustration purposes, we present `JCard` with insertions, unless otherwise specified. `JCard` has two components. (i) The first one is the summary of the dangling tuples of a database (*i.e.*, those do not currently form any view tuple). (ii) The second one is *candidate tuples* and they capture how close dangling tuples may form view tuples, under random insertions. When a view tuple is inserted, both components are used to estimate the change of the view size. For simplicity, in this section, we assume that the primary keys of the tuples are present in the views (*i.e.*, SJ views). We remove this assumption in Sec. 8.

5.1 Terminologies and Notations

We first give the notations needed to present `JCard`. Suppose the view $\mathcal{V}$ involves R_1,...,R_n relations. We construct a join tree $J_{\mathcal{V}}$ as follows. Each relation R_i forms a node in $J_{\mathcal{V}}$, also denoted by R_i as the meaning is often clear from the context. If there is a join between R_i and R_j in $\mathcal{V}$, we create an edge (R_i, R_j) in $J_{\mathcal{V}}$. The join tree rooted at R_i is denoted as J_{R_i}.



(a) Source graph G^+ (b) Negative graph G^-

Fig. 4. Source graph and negative graph

Join trees of different roots may have different accuracies in side-effect estimation.

We use a *database graph* $G(V, E)$ to represent the database I. A node $t{\in}V$ denotes a tuple t in I and an edge $(t, t'){\in}E$ denotes that t and t' are joinable w. r. t. $\mathcal{V}$.

Example 5.1: Consider the view shown in Fig. 1. Fig. 3(a) presents the join tree $J_{Supplier}$ rooted at `Supplier`. Fig. 3(b) shows the database graph G.

Next, we introduce the terminologies to discuss the tuples summarized in `JCard`.

Embeddings and partial embeddings. Suppose S is a set of tuples in G where each tuple belongs to a distinct relation in $J_{\mathcal{V}}$. If the tuples in S form a view tuple, S is called *an embedding* of the join tree $J_{\mathcal{V}}$. Otherwise, S may form *a partial embedding* of $J_{\mathcal{V}}$, defined in Definition 5.1.

Definition 5.1: Given a set of tuples S in a database graph G and a join tree $J_{\mathcal{V}}$, S is a *partial embedding* of $J_{\mathcal{V}}$, if

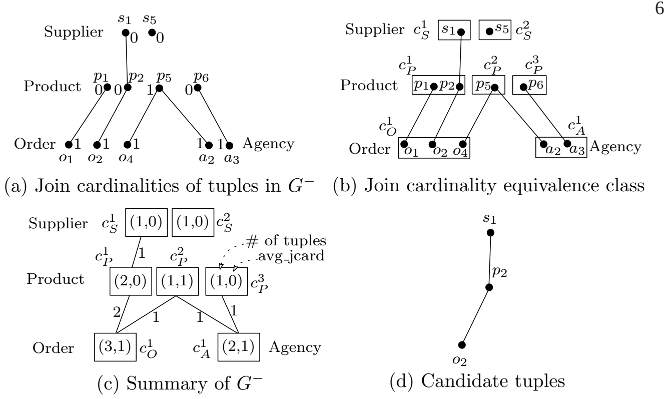
(i) the tuples in S together with the tuples in G do not form a view tuple;
(ii) each tuple in S belongs to a distinct relation;
(iii) the tuples in S are joinable; and
(iv) S is maximal, *i.e.*, no tuple can be added to S and (i)-(iii) are true. □

Example 5.2: Continue with Example 5.1. Fig. 3(c) shows an embedding of $J_{Supplier}$ as the tuples are of different relations and they form the view tuple v_3 in Fig. 1. $\{s_1, p_2, o_2\}$ shown in Fig. 3(d) is a partial embedding. However, $S{=}\{p_4, o_4, a_2\}$ is not a partial embedding as S and s_4 form a view tuple. $S'{=}\{p_2, o_2\}$ is not a partial embedding as s_1 can be added to S' and Conditions (i)-(iii) of Definition 5.1 are true. $\{s_5, p_6, a_3\}$ is also not a partial embedding as the tuples are not joinable.

Dangling tuples and extended dangling tuples. Dangling tuples are the tuples that do not form an embedding. Dangling tuples may become non-dangling (*i.e.*, form view tuples) after an insertion. The estimation of side effects is more accurate with not only dangling tuples but also *extended dangling tuples* (Definition 5.2). The intuition is that given an insertion, some dangling tuples may become non-dangling and the extended dangling tuples may cause additional view tuples, *i.e.*, side effects. Thus, we summarize both kinds of tuples. Finally, we remark that extended dangling tuples may join with some other tuples and unlike dangling tuples, they may appear on the view.

Definition 5.2: *Extended dangling tuples* comprise (i) dangling tuples; and (ii) the tuples that can form, together with some dangling tuples, a partial embedding of any subtree of the join tree $J_{\mathcal{V}}$. □

Example 5.3: In Fig. 3(b), p_2 is an extended dangling tuple as it is a dangling tuple (Definition 5.2(i)). o_4 is an extended dangling tuple as it forms a partial embedding $\{o_4, a_2, p_5\}$ with the dangling tuples a_2 and p_5 (Definition 5.2(ii)).



(a) Join cardinalities of tuples in G^- (b) Join cardinality equivalence class

(c) Summary of G^- (d) Candidate tuples

Fig. 5. Illustration of `JCard` and candidate tuples

Similar to tuples in a database, the dangling and extended dangling tuples may be represented by graphs.

Definition 5.3: Given a source database I of a view V, the database graph obtained from the extended dangling tuples in I is called the *negative graph* G^-. The database graph obtained from the source tuples of the view V is called the *source graph* G^+. □

Example 5.4: Fig. 4(a)-(b) show the source graph G^+ and the negative graph G^- of the database graph in Fig. 3(b), respectively.

We remark that G^+ contains all the embeddings in G and G^- contains all the partial embeddings in G. G^+ and G^- may be overlapping due to the extended dangling tuples but $G^+{\cup}G^-{=}G$ holds. G^+ and G^- together are an exact representation of the database I, in the sense that any side effect of updates on I can be determined from them. Obviously, the sizes of G^+ and G^- is $O(|I|)$, which can be too large for efficient estimations. Therefore, we summarize them in the next subsection.

5.2 JCard Definition

`JCard` handles G^+ and G^- differently. G^+ contains the embeddings, which are the tuples of the view, and can be easily maintained and indexed. Therefore, we focus on the summary of G^- in this subsection.

The summarization of G^- is derived from a notion of *join cardinality equivalence* between nodes, denoted as $t_1 \approx t_2$, w. r. t. a join tree $J_{\mathcal{V}}$. To define the join cardinality equivalence, we first define the join cardinality of tuples.

Definition 5.4: Given a join tree J_{R_i} (rooted at R_i of $\mathcal{V}$) and a negative graph G^-, let t be a tuple in $R \in \mathcal{V}$, the *join cardinality* of t, denoted as $t.jcard$, is defined as follows.

(i) If R is a leaf relation in J_{R_i}, t.`jcard`=1;
(ii) Otherwise t.`jcard`=$\prod_{R'\in\Re}$ ($\sum_{(t,t')\in G^-, t'\in R}$ (t'.`jcard`)), where $\Re$ is the set of child relations of R in J_{R_i}. □

The intuition of t.`jcard` is that we traverse the join tree J_{R_i} bottom-up and join the relations visited, and t.`jcard` is the number of intermediate join results containing t when the bottom-up traversal reaches R.

Example 5.5: Consider the negative graph G^- in Fig. 4(b) and the join tree $J_{Supplier}$ in Fig. 3(a). Fig. 5(a) shows the join cardinalities of tuples in G^-. The `jcard`s of the tuples in `Order` and `Agency` are 1 by Definition 5.4(i). Tuples in `Product` and `Supplier` are the product of the `jcard`s of their child relations, respectively, by Definition 5.4(ii). In particular, p_1.`jcard`=1×0=0, where o_1.`jcard` is 1 as p_1 joins with o_1

in `Order`; and 0 is `jcard` from `Agency` as p_1 does not join with any tuple in `Agency`.

Definition 5.5: Given a join tree J_{R_i} (rooted at R_i of $\mathcal{V}$) and a negative graph G^-, t_1 and t_2 are *cardinality equivalent* (denoted as $t_1 \approx t_2$) if the following statements hold:

(i) t_1 and t_2 are in the same relation R;
(ii) t_1.`jcard` = t_2.`jcard`; and
(iii) the child relations of R, in which t_1 has joinable tuples, are identical to those of t_2. □

Side-effect estimation using G^- is accurate due to the following reason. `JCard` (G^-) summarizes the tuples by the equivalence classes. All partial embeddings in G^- are compactly represented by equivalence classes. Upon an insertion of a tuple, the tuple is joined with the equivalence classes and the estimation is to determine the *average joinable tuples* in a class. When the insertion produces a large or moderate number of new embeddings (side effects), the average count is sufficient to signify the presence of side effects.

Example 5.6: Fig. 5(b) shows the join cardinality equivalence classes corresponding to the join cardinalities of tuples shown in Fig. 5(a). We remark that even though p_1 and p_6 have the same `jcard` 0, they are of different classes as `Order` is the joinable child relation of p_1, but that of p_6 is `Agency` (Definition 5.5(iii)).

JCard structure and its construction. `JCard` is a graph of supernodes, where each supernode represents an equivalent class in G^-. The supernode of the equivalent class c is also denoted by c as the meaning is clear from the context. A supernode is a binary tuple ($|c|$, $c.est$), where $|c|$ is the number of tuples in c and $c.est$ is the estimated average `jcard` of the tuples in c. `JCard` is constructed by the rules below.

1) For each leaf relation R in J_{R_i}, create a supernode (an equivalence class) for all tuples in R, as all tuples in R are cardinality equivalent, by Definition 5.5.
2) Construct `JCard` *bottom-up*:
   a) For each non-leaf node R in J_{R_i}, partition the tuples in R by Definition 5.5;
   b) For each equivalence class, create a supernode c to represent that class and add an edge (c, c') if there exists tuples in c that join with tuples in c'; and
   c) Determine the weight $|E_{c,c'}|$ of (c, c') as the number of joins between the tuples in c and those in c', where $|E_{c,c'}|$ will be used in estimation.

Example 5.7: Fig. 5(c) presents the `JCard` summary of G^- in Fig. 4(b). Each node in Fig. 5(c) summarizes an equivalent class in Fig. 5(b). In particular, the node $c^3_P(1, 0)$ means that the class c^3_P contains one tuple and the average join cardinality of tuple in c^3_P is 0. $|E_{c^1_A,c^3_P}|$=1 means that there is one join between c^1_A and c^3_P.

There are two remarks on `JCard` worth-noting. First, for simplicity, Definition 5.5 defines the equivalence based on identical count. In general, we may define an equivalence using *similar counts*, which leads to even smaller summary graphs. Our experiments show that by using Definition 5.5, we obtain small summaries of our benchmark datasets and we do not further reduce the summary size. Second, since we always analyze summary graphs, *we overload the notations G^- and G^+ to refer to the summaries, unless otherwise specified.*

Candidate view tuples. `JCard` summarizes tuples in G^- based on (join) cardinalities. Certainly, the cardinalities are directly relevant to the number of view tuples generated from insertions. However, it does not consider how *likely* dangling tuples may form view tuples. Suppose view insertions are random. The dangling tuples that form an embedding with just fewer new tuple segments will form view tuples easier than other dangling tuples. Therefore, `JCard`'s second structure is sets of *candidate view tuples*. Each set of *candidate view tuples* is simply a set of tuples that forms a partial embedding to the join tree. The rank of each tuple set is defined to be the number of new tuple segments needed to form an embedding. Each individual candidate tuple will be split from its original equivalence class and form an individual equivalence class in G^- by itself (*i.e.* no summarization).

Example 5.8: Continuing with Example 5.7, if we are given a budget to take one candidate tuple set, we will select $\{s_1, p_2, o_2\}$ or $\{o_4, p_5, a_2\}$ as they will form embeddings by one additional segment, respectively. Specifically, $\{s_1, p_2, o_2\}$ needs a new segment of `Agency` joinable to p_2 and $\{o_4, p_5, a_2\}$ needs one of `Supplier` joinable to p_5, whereas other two partial embeddings need two segments.

6 Side-effect Estimation with JCard

As illustrated in Example 4.1, traditional join cardinality estimations may not be suitable for side-effect estimation as they minimize overall errors on cardinalities. This section presents our side-effect estimation algorithm on `JCard`. Due to space limitations, an analysis of the causes of errors, which reveals the design issues of `JCard`, is presented in Appendix D.

Our side-effect estimation of a view insertion t_v consists of two steps. The first step is to join t_v with the source tuples of V in G^+. If the join result is not empty, then t_v forms additional view tuple(s) (*i.e.*, side effects) and hence rejected. Otherwise, the second step estimates the join cardinality of t_v with the extended dangling tuples in G^-. This section focuses on the second step as it is more technically involved.

Fig. 6 presents the overall estimation algorithm. The inputs of `estimate_side_effects` are the `JCard`, the view definition in the form of join tree J_R, the view tuple to-be-inserted t_v and a parameter θ on the error threshold. `estimate_side_effects` estimates the number of new view tuples generated by inserting t_v. We recall that `estimate_side_effects` declares a side-effect free insertion when the estimated number of new view tuples is smaller than 1 + θ; otherwise, it declares side effects.

The details of `estimate_side_effects` can be described as follows. Given a view tuple t_v to-be-inserted, we decompose it into n segments $t_1, \ldots, t_n$, where n is the number of relations in J_R, by `decompose_view_tuple` in Line 01. Each segment t_i corresponds to a relation R_i in J_R. t_1,..., t_n form at least an embedding on J_R for a valid insertion. We update `JCard` via `update_equiv_class` for a more accurate estimation. In a nutshell, `update_equiv_class` creates a new equivalence class in G^- for each new segment. If t_i exists in G^-, we split t_i from its original equivalence class

```
Procedure estimate_side_effects
Input: view tuple to-be-inserted $t_v$, join tree $J_R$ of view,
       JCard $G^-$ and error threshold $\theta$
Output: true if it estimates side effects; false otherwise
01 $\mathcal{C}$ = decompose_view_tuple($t_v$, $J_R$)
02 $G^{-\prime}$ = update_equiv_class($J_R$, $G^-$, $\mathcal{C}$)
03 for each equivalence class $c$ of the root relation $R$ of $J_R$
04    propagate($c$, $G^{-\prime}$, $J_R$)
05 let $C$ to be the classes of $R$ that form embeddings with $t_v$
//let $a$ denote the number of embeddings in $G^-$ without ins.
06 return ($\sum_{c \in C}(c$.est$\times|c|)$ - $a > 1 + \theta$)
```

Fig. 6. Procedure `estimate_side_effects`

```
Procedure propagate
Input: an equivalence class $c$, JCard $G^-$ and join tree $J_R$
Output: the estimated average jcard of tuples in $c$: $c$.est
01 if $c$ is a leaf node in $G^-$
02    return $c$.est = 1
03 else //suppose $c$ is of relation $R'$ in $J_R$
04    for each child relation $R''$ of $R'$ in $J_R$
         //let $c'$ denote a child of $c$ in $G^-$
05       for each $c$'s child $c'$ of $R''$
06          if !$c'$.visited then propagate($c', G^-, J_\mathcal{V}$)
07       $c$.est.$R''$ = $\sum_{c' \text{ of } R''}(c'$.est$\times|E_{c,c'}|/|c|)$
08    return $c$.est = $\prod_{\text{child } R'' \text{ of } R' \text{ in } J_R}(c$.est.$R'')$
```

Fig. 7. Procedure `propagate`

and update the edge weights of `JCard` correspondingly. Due to space limitations, we present its details in Appendix C.

The essence of Procedure `propagate` is the *propagation logic* of estimation of counts (Line 04). Lines 05-06 simply sum up all the estimation counts of the classes that may form embedding(s) with t_v.

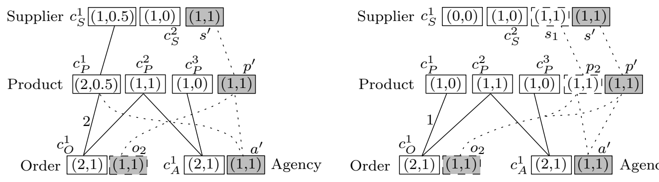
Propagation of estimation counts. Procedure `propagate` is a recursive procedure that estimates the number of new view tuples by inserting the view tuple t_v (Fig. 7). The recursion is simple: `propagate` traverses G^- top-down from the equivalence classes of the root relation of J_R (Line 06). The estimated count of a leaf equivalence class is 1 (Line 02) and that of an internal class is computed by the formulas in Lines 07 and 08. Specifically, given a class c, Line 07 sums up the counts propagated from the equivalence classes of a child relation of c. This formula assumes that the values of the join attributes have the same probability in participating the join. Therefore, one tuple in c obtains $|E_{c,c'}|$ / $|c|$ of c'.`est` on average, where weight $|E_{c,c'}|$ of the edge (c, c') in G^- denotes the number of joinable pairs of tuples between c and c'. Line 08 multiplies the counts from all child relations.

A restriction of `JCard` is that it assumes join trees, as propagation always terminates. In practice, acyclic joins are common, *e.g.*, all TCP-H queries, except one, are acyclic [44].

Example 6.1: Consider the view V and the view update u_2 of inserting (S_5, FL, P_6, C_3, A_4) as shown in Fig. 1. Fig. 8 illustrates the propagation logic on the `JCard` of V (Fig. 5(c)). Fig. 8(a) is the propagation without candidate tuples. The gray boxes are the classes for the segments of the insertion u_2, the dashed boxes are the classes split from existing classes and the dotted lines denote the join related to the insertion. (The edge weights that equal to 1 are omitted for presentation brevity.)

Firstly, u_2 is decomposed into four segments: s'=(S_5, FL) for `Supplier`, p'=(S_5, P_6) for `Product`, a'=(A_4, P_6) for `Agency` and o_2=(C_3, P_6) for `Order`, where s', p' and a' are new segments and o_2 is an existing tuple.

In propagation without candidate tuples (Fig. 8(a)), we construct an equivalent class for each segment. The classes are



Supplier c_S^1 (1,0.5) (1,0) (1,1)
c_S^2 s'
c_P^1 c_P^2 c_P^3 p'
Product (2,0.5) (1,1) (1,0) (1,1)
2
c_O^1 o_2 a'
Order (2,1) (1,1) c_A^1 (2,1) (1,1) Agency
Supplier c_S^1 (0,0) (1,0) (1,1) (1,1)
c_S^2 s_1 s'
c_P^1 c_P^2 c_P^3 p_2 p'
Product (1,0) (1,1) (1,0) (1,1) (1,1)
1
c_O^1 o_2 a'
Order (2,1) (1,1) c_A^1 (2,1) (1,1) Agency

(a) without candidate tuples (b) with candidate tuples

Fig. 8. Illustration of propagation of join cardinality

also denoted as s', p', o_2 and a', respectively, if it is clear from the context. Since class o_2 is split from c_O^1, the size of c_O^1 is reduced by 1. Since a' is joinable with p_2 in c_P^1, we add an edge (a', c_P^1) into `JCard` and $|E_{a',c_P^1}| = 1$. Initially, o_2.`est` and a'.`est` are 1 as they are of leaf relations. Next, we estimate the join cardinality of internal nodes (Line 11 of Fig. 7). For example, $c_P^1.\texttt{est}=\frac{c_O^1.\texttt{est}\times|E_{c_O^1,c_P^1}|}{|c_P^1|}\times\frac{a'.\texttt{est}\times|E_{a',c_P^1}|}{|c_P^1|}=\frac{1\times2}{2}\times\frac{1\times1}{2}$=0.5. Since the number of embeddings in G^- without insertion is 0, we finally obtain the estimated number of new embeddings from the root relation as $c_S^1.\texttt{est}\times|c_S^1|+s'.\texttt{est}\times|s'|$ =0.5+1=1.5. This number is smaller than the real number 2, because of the averaging at c_P^1.

Fig. 8(b) is propagation with candidate tuples. The propagation with candidate tuples is more accurate (Fig. 8(b)). Suppose we have one set of candidate tuples $\{s_1, p_2, o_2\}$ in Fig. 5(d). We split s_1 and p_2 from c_S^1 and c_P^1, respectively, and construct an equivalent class for each of them (shown in dashed blocks); o_2 keeps in the gray block as it is an insertion segment. After the split, $|c_S^1| = 0$, $|c_P^1| = 1$, $|E_{c_O^1,c_P^1}| = 1$ and $|E_{c_P^1,c_S^1}| = 0$. After the propagation, we obtain the estimated number of new embeddings as $s_1.\texttt{est}\times|s_1|$ $+s'.\texttt{est}\times|s'|$=1+1=2 and the side effect is detected.

7 Deletions and Replacements

To complete the discussion on updates, we present the support of deletions and replacements with `JCard`.

7.1 Deletions

In this subsection, we extend `JCard` to support deletions. The side-effect detection on SJ views using `JCard` is exact and runs in PTIME [13], [14].

Recall that G^+ captures all the source tuples of a view. We can use G^+ to determine the side effect of deletions as below.

1) The detection uses the values of primary keys in t_v to locate its segments from the relations in G^+;
2) For each segment, if it occurs in V multiple times, this segment is not deletable;
3) If all segments are not deletable, t_v is not translatable; Otherwise, we can delete any deletable segment as a translation of t_v.

If the deletion is side-effect free and translatable, we will delete it from G^+ and update both G^+ and G^-.

7.2 Replacements

The main idea of our technique to support replacements is to transform a replacement to a deletion followed by an insertion. Subsequently, we can adopt our techniques of deletions and insertions to support replacements. Our method is developed based on the following observation, which holds as the primary keys of relations are present in the SJ views.

Proposition 7.1: *Given a replacement replacing t_v to t'_v on a SJ view, the replacement is side-effect free, iff the deletion of t_v is side-effect free and the subsequent insertion of t'_v is also side-effect free.* □

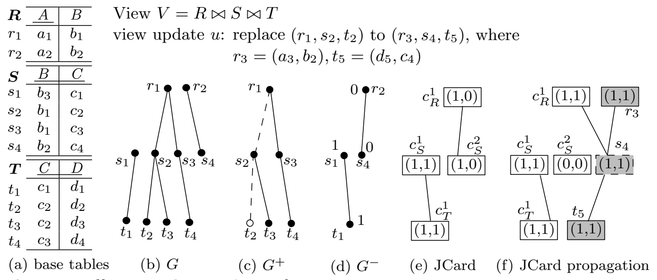

View $V = R \bowtie S \bowtie T$
view update u: replace (r_1, s_2, t_2) to (r_3, s_4, t_5), where
$r_3 = (a_3, b_2), t_5 = (d_5, c_4)$
R: A B; r_1 a_1 b_1; r_2 a_2 b_2
S: B C; s_1 b_3 c_1; s_2 b_1 c_2; s_3 b_1 c_3; s_4 b_2 c_4
T: C D; t_1 c_1 d_1; t_2 c_2 d_2; t_3 c_2 d_3; t_4 c_3 d_4
c_R^1 (1,0) (1,1) (1,1) r_3
c_S^1 c_S^2 (1,1) (1,0) (0,0) (1,1) s_4
c_T^1 (1,1) t_5
(a) base tables (b) G (c) G^+ (d) G^- (e) JCard (f) JCard propagation

Fig. 9. Illustration of replacement

Since the side-effect detection of deletions is exact, the estimation error of replacement equals to that of the insertion; and the replacement time is dominated by the insertion time.

Example 7.1: Fig. 9 illustrates the main steps of replacement. The view is a join of relations R, S and T (Fig. 9(a)), where the primary keys are underlined. Figs. 9(b)-(e) are the database graph G, source graph G^+, negative graph G^-, `JCard` before insertion, respectively. Fig. 9(f) shows the propagation on the `JCard`. Suppose we replace a view tuple v_1 that joins r_1, s_2 and t_2 (shown in dashed lines in Fig. 9(c)) by another view tuple v_2 that joins r_3, s_4 and t_5, where r_3 and t_5 are new.

We first delete v_1 from G^+. In its segments (*i.e.*, r_1, s_2 and t_2), only t_2 is deletable (shown by the circle in Fig. 9(c)) as it just occurs once in V. Hence, the deletion has no side effect. Second, we insert v_2 into V. Following the technique proposed in Sec. 6, we (i) find it does not join with any existing tuple in G^+ and then (ii) insert it into G^-. Fig. 9(f) shows that the estimated number of new embeddings after insertion is c_R^1.est+r_3.est=2 and the side effect is detected.

8 Projection

It has been known that if the view definitions involve projections, their view update problems often become NP-complete [13], [14]. The main reason is that the attributes projected out (*a.k.a.*, the missing attributes) can be primary keys, foreign keys, or *join* attributes. This section presents an extension of `JCard` to support views with projections. (We discuss with insertions with missing join attributes as they are more technically involved.) First, we fill in the feasible values for the missing attributes. Among many feasible fillings, we propose a *greedy method* to fill in values that may cause the fewest side effects. Second, the cardinality equivalence is extended with attribute values and the estimation algorithm is adjusted accordingly to estimate side effects.

8.1 Filling in Missing Attributes

Due to projection, the tuple segments of a view *insertion* contain missing (join) attributes which must be filled in, prior to estimation of side effects. We call the tuple segments with the missing attributes filled in a *filling*. There is a spectrum of approaches for determining a filling without side effects. For instance, one may directly employ a potentially costly heuristic algorithm to determine a side-effect free filling, *e.g.*, [12]. Another extreme is to fill in these attributes randomly and estimate the side effects of the filling. This is repeated until a filling with zero side-effect estimate is obtained or the insertion is simply rejected. In this subsection, we propose a greedy approach for determining a filling.

Definition 8.1: A filling is *feasible* if the following holds:

1) the filling does not violate integrity constraints and referential constraints of the source database; and

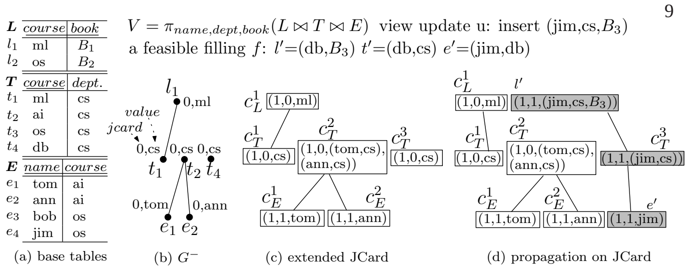

$V = \pi_{name,dept,book}(L \bowtie T \bowtie E)$ view update u: insert (jim,cs,B_3)
a feasible filling f: l'=(db,B_3) t'=(db,cs) e'=(jim,db)
L: course book; l_1 ml B_1; l_2 os B_2
T: course dept.; t_1 ml cs; t_2 ai cs; t_3 os cs; t_4 db cs
E: name course; e_1 tom ai; e_2 ann ai; e_3 bob os; e_4 jim os
value
jcard
0,ml
0,cs 0,cs 0,cs
0,tom 0,ann
c_L^1 (1,0,ml) (1,1,(jim,cs,B_3))
c_T^1 (1,0,cs) c_T^2 (1,0,(tom,cs),(ann,cs)) c_T^3 (1,0,cs) (1,1,(jim,cs))
c_E^1 (1,1,tom) c_E^2 (1,1,ann) (1,1,jim)
l' e'
(a) base tables (b) G^- (c) extended JCard (d) propagation on JCard

Fig. 10. Illustration of projection

2) the filling satisfies the selection and joins of the view.

Let s be a tuple segment of a relation R, which is a relation in the view definition $\mathcal{V}$. The joins are on primary keys and foreign keys. Feasible fillings can be greedily determined by the following rules:

Case 1. The projected attribute A is part of the primary key of R.

1) Suppose there is some value p in $dom(A)$ that is not present in R yet, we fill in s with the value p. Note that p is new and does not have any existing joining tuples in other relations.

2) If no new primary key is available, we check the tuples t_s in R that are consistent with s. The tuple in t_s with fewer joining neighbouring tuples (edges in database graph G) is selected earlier.

Case 2. The projected attribute A is part of the foreign key referencing to a relation R. We fill in the missing value of A such that it has the fewest number of joining tuples.

Feasible fillings are generated one by one according to the above rules and are passed to the side-effect estimation. Furthermore, users may specify a bound k on the number of feasible fillings passed to the side-effect estimation, with a trade-off on estimation accuracy.

8.2 Extension of JCard and Its Algorithm

Projections are defined with set semantics, where duplicate values are "removed". Due to the cardinality equivalence of `JCard` proposed in Sec. 5, tuples with different values may be placed in the same equivalence class. Subsequently, the estimation algorithm may over-estimate side effects. Therefore, we propose a refinement on the notion of cardinality equivalence. In addition to the conditions in Definition 5.5, we introduce a condition that t_1 and t_2 are *value-cardinality* equivalence if they are cardinality equivalent and *they have the same values on the projection attributes*. We then construct *extended* `JCard` *by using value-cardinality equivalence*.

The estimation algorithm `estimate_side_effects` (Fig. 6) is adjusted to incorporate with the *value-cardinality* equivalence. Specifically, if the count of new embeddings is larger than 1 + θ, then the estimator declares side effects. Otherwise, the estimator declares no side effect. However, the logic of the count propagation remains the same.

Example 8.1: Consider a view on a database of three relations, `Library` L, `Teaching` T and `Enrollment` E (Fig. 10(a)). The primary keys of the relations are underlined. Suppose that $L.course$ refers to $T.course$ and $E.course$ also refers to $T.course$. The tuple to-be-inserted is (jim, ie, B_3). It can be decomposed into segments l'=(X, B_3), t'=(Y, ie) and e'=(jim, Z), where X, Y and Z are the missing attributes to-be-filled in. Suppose the domain of $T.course$ is simply {db, os, ai, ml}. A feasible filling f is: X=Y=Z=db. By using f, t'=t_4

```
Procedure candidate_tuples
Input: join tree $J_{\mathcal{V}}$, negative graph $G^-$, the number of
       candidate sets $k$, the join attributes $\mathcal{A}$ of $J_{\mathcal{V}}$
Output: a set of candidate tuples
01 $(t_u, t_v)$.capacity = $\infty$, $\forall$ $(t_u, t_v) \in G^-.E$
02 for each $(t_u, t_v) \notin G^-.E$
03    if $t\_u \in R_u$, $t\_v \in R_v$ and $(R_u, R_v) \in J_{\mathcal{V}}.E$
04       add $(t_u, t_v)$ into $G^-$
05       set $(t_u, t_v)$.capacity = $log$(max_prob($(t_u, t_v)$))
06 add a single sink and a single source to connect
      the sink(s) and source(s) of $J_{\mathcal{V}}$, respectively
07 add a sink and a source to connect the sink(s)
      the source(s) of each connected subgraph in $G^-$
08 set the capacity of the edges connecting to the sink or
      the source to infinity
09 $\mathcal{C}$ = $\emptyset$ and $G' = G^-$            // initialization
10 while $|\mathcal{C}| < k$ and $G' \neq \emptyset$
11    $g$ = max_flow ($G'$)
12    $\mathcal{G}$ = extend($g$)
13    $\mathcal{C}$ = $\mathcal{C} \cup \{t \mid t \in g', g' \in \mathcal{G}\}$
14    for each $g' \in \mathcal{G}$    $G' = G' - g'$
15 return $\mathcal{C}$
```

Fig. 11. Procedure `candidate_tuples`

and the number of joining tuples between T and E is 1, as t_4 joins with e'. The number of joining tuples between T and L is 1, as t_4 joins with l'. There are 2 joining tuples in total. If X=Y=Z=ai, the number of joining tuples is 4; 3 for ml; and 4 for os. Hence, the greedy algorithm analyzes f first.

Fig. 10(b) shows the G^- and the join cardinalities extended with values. Fig. 10(c) shows the extended `JCard` before insertion and the propagation on the `JCard` is illustrated in Fig. 10(d). `propagate` estimates that the view size after inserting the filled tuples and returns 1 new embedding. Hence, the insertion has no side effect.

Remark. For deletions, we use lineage technique (*e.g.*, [16], [23]) as a blackbox. We compute the lineage of the view tuple to-be-deleted and then apply `JCard` to detect the side effects.

9 Optimization Problems in JCard

There are two important optimization problems in the construction of `JCard`. In Sec. 9.1, we show that selecting the candidate tuples of `JCard` for accurate side-effect estimation is equivalent to Minimum Set Cover (MSC) and illustrate how approximation algorithms for MSC can be adopted to solve the selection problem. In Sec. 9.2, we address the selection of the representation of join trees, which is a crucial input to side-effect estimation.

9.1 Candidate Tuples Selection

To begin with, we formalize the problem of selection of candidate tuples and investigate its hardness.

Definition 9.1:(Selection of Candidate Tuples (SCT)) Given a space budget B, a SPJ view $\mathcal{V}$ and the G^- summary of the source database, select a set of candidate to minimize side-effect estimation errors of `propagate`.

Theorem 9.1: *SCT is NP-complete.*

Next, we present a reduction from an SCT instance to an MSC instance. For each possible insertion u, we create an element u in the universe $\mathcal{U}$. For the tuple s_i for selection, we create a clause C_i. For each s_i whose selection leads to the insertions $\mathcal{U}$ accurate, we ensure that $u \in C_i$, where $u \in \mathcal{U}$. With such a reduction, it is straightforward that the optimal solution of the MSC instance from this reduction is the optimal solution for the SCT problem as well. Most importantly, the approximation ratio of heuristics for MSC is trivially preserved in such a simple reduction.

The MSC problem has known to be an NP-complete problem that greedy algorithms work well with reasonable bounds. Hence, we propose a greedy algorithm, namely Procedure `candidate_tuples` (shown in Fig. 11), which is equivalent to a greedy algorithm of MSC, whose approximation ratio is known to be OPT $\times$ $lg(\mathcal{U})$.

We make two observations on `candidate_tuples`. Firstly, we do not require completely reducing an SCT instance to an MSC instance in `candidate_tuples`. Secondly, there are admittedly many heuristics for the MSC problem. We propose Procedure `candidate_tuples` in the style of a well-known greedy algorithm for ease of analysis.

The main idea of `candidate_tuples` is to convert the negative graph G^- into a graph with a single source and single sink (Lines 01-08). For the joinable tuples (edges) that are not in G^-, we introduce them into G^-. The capacity of such an edge, denoted as (t_u, t_v), is the logarithm of the probability of tuples with the join attribute values, where we assume the values in the domain exhibit *uniform probability*. A subtle point is that the tuple segments in G^-, by definition, do not form an embedding (a view tuple). We simply apply a maximum flow from the source to the sink (Line 11). The flow is the sum of logarithm of probabilities, which is simply proportional to the product of probabilities. This is equivalent to picking the tuple segments that are most probable to form a view tuple. Minor details include (i) extending the path of maximum flow into a partial embedding, which is one set of candidate tuples (Line 12) and iteratively selecting the embeddings from the negative graph until k embeddings are selected (Lines 10-14), where k is a user-defined parameter.

The complexity of `candidate_tuples` is simply the complexity of maximum flow multiplied by k.

9.2 Optimal Join Tree Selection

The join cardinality summary `JCard` G^- defined by Sec. 5 assumes a particular join tree $J_{\mathcal{V}}$ of a given view definition $\mathcal{V}$, denoted as $G^-(J_{\mathcal{V}})$. However, given $\mathcal{V}$, there are $|\mathcal{V}|$ join tree alternatives, whose accuracies may differ from each other. In this section, we present a selection algorithm, that is an adoption of simple sampling technique, to determine the optimal join tree $J_{\mathcal{V}}^{opt}$ with respect to its error produced by the estimation algorithm `propagate`.

A naive method to select an optimal join tree (*i.e.*, with the smallest error) is to enumerate all possible insertions of join trees. Given a `JCard` of a join tree $G^-(J_{\mathcal{V}})$, one may determine the set M of possible insertions by Definition 4.3. Given a particular insertion $m \in M$, its error can be determined by calling `propagate` with $G^-(J_{\mathcal{V}})$ and m and comparing the result with a side-effect detection. However, determining the true error of $G(J_{\mathcal{V}})$ requires calling `propagate` with all permutations of M.

We propose to simplify the computation on errors of join trees for practical solutions. We make two assumptions on the problem. First, we assume that insertions are equally probable. Second, each insertion is independent. With these assumptions, we can estimate the proportion of falsely estimated insertions

TABLE 2
Characteristics of datasets

		# of tables	max table size	avg. table size	avg. fan-out of G
TPC-H	200M	6	1,200,000	288,671	131.75
	400M	6	2,400,000	577,338	262.81
	600M	6	3,600,000	866,005	393.86
	800M	6	4,800,000	1,154,671	524.98
	1G	6	6,000,000	1,443,338	659.06
DBLP		6	245,888	73,489	4.32
SYNTHETICDB		6	30,000	18,000	2.03

by using sampling. The sample size can be determined by *estimation of proportion*. In a nutshell, without information about future insertions, we may exploit the maximum variance of samples to estimate the true error. The classical result is that the error bound can be determined by $4\sqrt{0.25/|S|}$, where S is a sample. For example, the error bound is 5% when the sample size is 400. Furthermore, if the relative accuracies between join trees cannot be distinguished due to errors of sampling, more insertions can always be sampled.

Finally, the details with deletions are similar and deletions are always bounded by the view size.

10 Experimental Evaluation

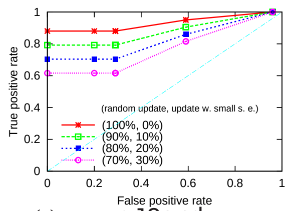
This section presents a comprehensive experimental evaluation that verifies the efficiency and effectiveness of our techniques.

Experimental settings. We ran our experiments on a PC with a Quad-core 2.4GHz CPU running Ubuntu 11.04. Our implementation was written in C++, using MySQL 5.1. The maximum memory for our C++ program was set to only 500M bytes. In this experiment, the memory representation of the largest test dataset could not fit into 500M byte memory. Moreover, our algorithms are independent of graph storage which is a research topic in and of itself.

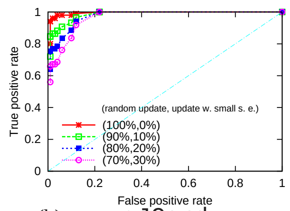
Benchmark datasets. We use two publicly available datasets TPC-H [44] and DBLP [40], and one synthetic dataset SYNTHETICDB that is implemented by ourselves. There is no dangling tuple in TPC-H and DBLP. Hence, we randomly sample subsets of tuples from their relations, respectively, to obtain the benchmark datasets. Regarding TPC-H, we sampled five test datasets from TPC-H of scaling factor 4.0. They are of the sizes 200M, 400M, 600M, 800M and 1G, respectively. We use the 1G dataset by default, unless otherwise specified. Regarding DBLP, we sampled half tuples from the full DBLP [40] as our test dataset. Regarding SYNTHETICDB, the generator is tunable with four parameters: relation number, relation size, primary and foreign key join direction (*e.g.*, $R_1.FK$ referred to $R_0.PK$ denoted a join from R_0 to R_1) and the maximum tuple fan-out (*e.g.*, fan-out of a tuple in R_0 is the number of tuples in R_1 joinable with it). Some characteristics of the three datasets are reported in Table 2.

Error metrics. Let M be the set of insertions tested. Denote S_+ and S_- be the real-positive and real-negative insertions in M, respectively. In this experiment, we set $|S_+| = |S_-|$. Let A_+ be the estimated positive insertions in S_- and A_- the estimated negative insertions in S_+. We define the false negative (FN) to be $\frac{|A_-|}{|S_+|}$ and the false positive (FP) to be $\frac{|A_+|}{|S_-|}$.

Query workload. Regarding TPC-H, the view is a simplified $Q7$ in TPC-H [44]. For illustration purposes, we focus on the joins in $Q7$. We tested *all* join queries in [44] and obtained

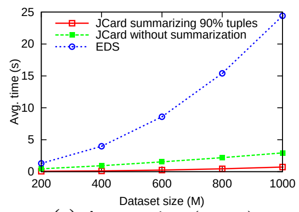


(a) ROC of `JCard` (TPC-H)

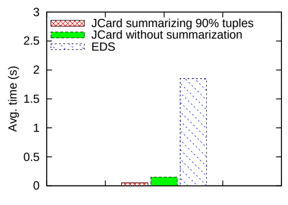


(b) ROC of `JCard` (DBLP)

Fig. 12. ROC curves of `JCard` on TPC-H and DBLP



(a) Average time (TPC-H)



(b) Average time (DBLP)

Fig. 13. Comparison with EDS on TPC-H and DBLP

similar results (detailed in Appendix E). We use the view on the full TPC-H to obtain possible insertions.

Regarding DBLP, we generated a set of views randomly. The last query in Fig. 20 shows the view template. We also use the view on the full DBLP [40] to obtain possible insertions.

Regarding SYNTHETICDB, we use the query joining all tables as our view. The insertions were generated by the generator with the same parameters.

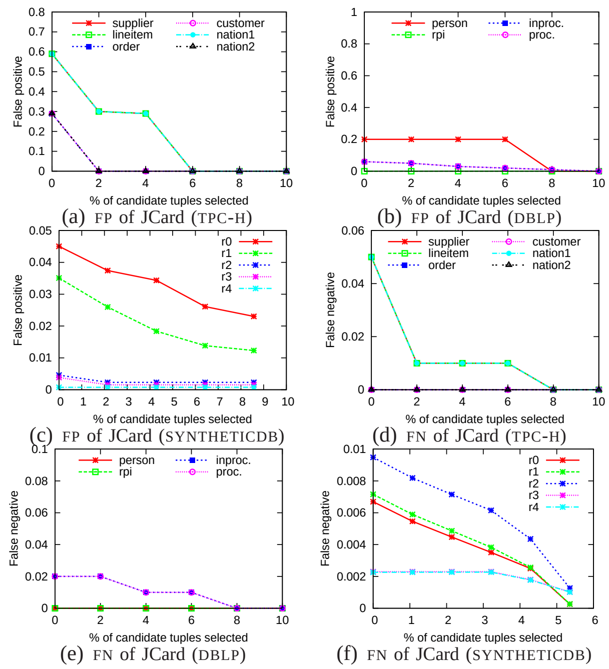
In this experiment, we analyze the performance of our techniques on FP, FN and the estimation time, respectively. *The reported performances are averaged performances on 1,000 view updates.* We often plot the performances of various *join trees* (though they may be overlapping) to show our technique is robust against join trees selection.

Experiment A: ROC curve of JCard.

Figs. 12(a)-(b) show the ROC curves of `JCard` on TPC-H and DBLP, respectively. To illustrate the performances of `JCard` in various scenarios, we generate artificial updates that have tiny side effects, which are hard to estimate, mixed with random updates. The `JCard` has no candidate tuple. Fig. 12(a) shows the ROC curves and verifies that our side-effect estimation performs very well. For instance, the AUC of `JCard` is 0.74 on workloads with 30% hard updates. From the figure, as expected, the fewer the hard updates, the larger the AUC. We observe similar results from DBLP (see Fig. 12(b)). We remark that the hard updates are carefully generated by examining the joining tuples from the datasets, which are rare if all updates are considered equally probable. Hence, we focus on random updates in the remaining experiments.

Experiment B: Comparison with EDS

Next, we compare the performances of `JCard` with our implementation of the latest related work EDS technique [20]. Since EDS does not produce errors, we report its runtime in this experiment. For a fair comparison, we report the runtime of an exact `JCard` (*i.e.*, no tuple is summarized). In addition, we observe that the `JCard` summarizing 90% tuples can still comfortably attain no error. Hence, we report the runtime of such a `JCard`, as a reference. Figs. 13(a)-(b) present the runtime on TPC-H and DBLP, respectively. In Fig. 13(a), the x- and y-axes are the dataset size and the runtime, respectively. From Fig. 13(a), we note that the exact `JCard` is already at least one order of magnitude faster than EDS and the



(a) FP of JCard (TPC-H) (b) FP of JCard (DBLP) (c) FP of JCard (SYNTHETICDB) (d) FN of JCard (TPC-H) (e) FN of JCard (DBLP) (f) FN of JCard (SYNTHETICDB)

Fig. 14. Side-effect estimation error of `JCard`

performance gap increases as the dataset size increases. We observe similar results in Fig. 13(b).

A possible reason is that EDS is designed for the `XML` views, whose advantages cannot be fully observed from relational views. More specifically, in EDS, the values of EDS attributes of tuples can be updated without causing any side effect. An attribute is EDS if (I) its values do not appear in the `XML` view (*i.e.*, the attribute is projected out); or (II) its values appear in the view only once and they are not accessed elsewhere in the `XML` view definition. `XML` uses subtrees to naturally model one-to-many relationships such as the relationship between `Person` and `InProceedings` in DBLP. In contrast, when encoded in relational views, both person and proceeding entities appear multiple times in the views. Due to Condition (II), most (if not all) attributes of the relational views are not EDS. As a result, costly update analysis is needed.

From Figs. 13(a)-(b), we also note that the `JCard` summarizing 90% tuples is even faster.

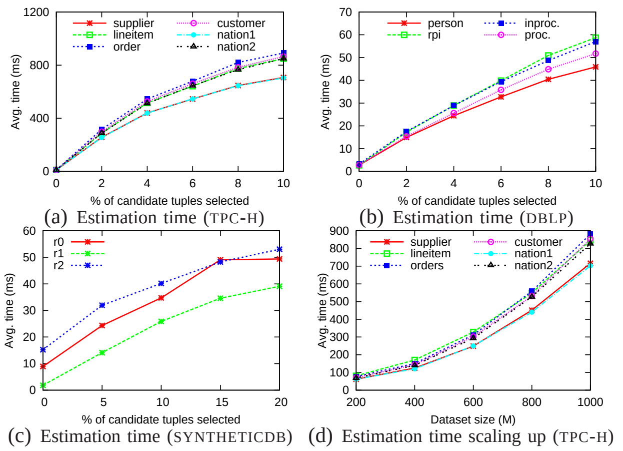
Experiment C: Overall performance of JCard

Estimation error. We select $x\%$ candidate tuples and study the estimation error of `JCard`. We use the three datasets and set θ=0.4. Fig. 14 reports the result. Figs. 14(a)-(c) show FPs (*i.e.*, the real error of `JCard`). Figs. 14(a)-(c) show that the selection of candidate tuples is effective in reducing the estimation error. In particular, FP reduces as selecting more candidate tuples. After a certain small percentage (*e.g.*, 6% in Fig. 14(a), 10% in Fig. 14(b) and 2% in Fig. 14(c)), FP approaches to zero. We observe similar results of FN in Figs. 14(d)-(f).

In addition, we observe that if the optimal join tree is used, even no candidate tuple is selected, the estimation error of `JCard` can be zero as shown in Figs. 14(b), (d) and (e).

Finally, this experiment verifies that the accuracies of `JCard` of different join trees are different, which is more notable on SYNTHETICDB as shown in Figs. 14(c) and (f).

Detection time vs. estimation time. We then compare the detection time of an exact `JCard` (*i.e.*, no tuple is summarized)



(a) Estimation time (TPC-H) (b) Estimation time (DBLP) (c) Estimation time (SYNTHETICDB) (d) Estimation time scaling up (TPC-H)

Fig. 15. Side-effect estimation time of `JCard`

and the estimation time of `JCard`s summarizing $1-x\%$ tuples. The average detection time of TPC-H, DBLP and SYNTHETICDB is about 3s, 0.18s and 60ms, respectively. Their average estimation time is reported in Figs. 15(a)-(c), respectively. Figs. 15(a)-(c) show that our estimation is much faster than the detection. In particular, on TPC-H, when 6% candidate tuples are selected, FP is zero (Fig. 14(a)), but the estimation time is about 700ms (Fig. 15(a)), which is about 4 times smaller than the detection time. This is because that the `JCard` summarizes about 90% non-candidate tuples in equivalence classes, which can clearly save the propagation time of join cardinalities. We observe similar results on DBLP (Fig. 14(b) and Fig. 15(b)) and SYNTHETICDB (Fig. 14(c) and Fig. 15(c)).

Moreover, the estimation time increases roughly linearly with the percentage of candidate tuples selected. The "slope" is about 80ms, 5ms and 2ms per 1% candidate tuples selected on TPC-H, DBLP and SYNTHETICDB, respectively.

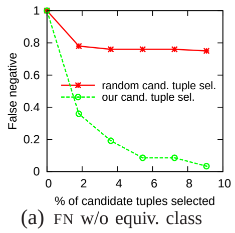
Scalability test. We tested the scalability of the `JCard` using TPC-H. In this experiment, we tune the `JCard` to be error free and focus on its estimation time. Specifically, we select 10% candidate tuples and set $\theta = 0.4$. (From Figs. 14(a) and (d), we note that the `JCard` is error free at such a setting.) The result is reported in Fig. 15(d). From Fig. 15(d), we observe that the join trees have similar estimation time and the growth of time is almost linear as the dataset size increases. However, it is always much faster than the detection time as discussed.

Experiment D: Optimizations on JCard

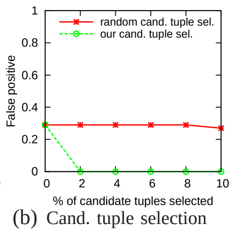
In this experiment, we focus on TPC-H as other datasets exhibit similar performance characteristics.

Effectiveness of equivalence classes. Previous experiments verify the importance of candidate tuples and this experiment shows the importance of equivalence classes, by skipping them. Fig. 16(a) reports the result. Consistent with the estimation with equivalence classes (Fig. 14(d)), the error reduces as we select more candidate tuples. However, when comparing Fig. 14(d) and Fig. 16(a), we note that the equivalence classes sometimes offer more than an order of magnitudes improvement on accuracies.

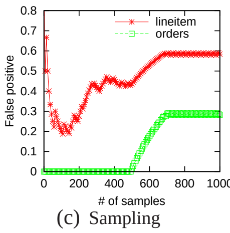
This experiment can be modified to show the effectiveness of candidate tuple selection. Fig. 16(a) further shows that our candidate tuple selection outperforms a random method. In particular, when 10% candidate tuples are selected, our FN is close to zero. When x=9%, our FN is 3%, whereas that of the



(a) FN w/o equiv. class



(b) Cand. tuple selection



(c) Sampling

Fig. 16. Effectiveness of optimizations on TPC-H

random method is 75%. Fig. 16(a) does not show FP as FP does not occur when equivalence classes are skipped.

Candidate tuple selection. To show the effectiveness of our candidate tuple selection approach, we compare its FPs (the real error) with the FPs of a random approach. Both approaches use equivalence classes. Fig. 16(b) shows that our technique significantly outperforms the random approach. In particular, when $x = 2\%$, our FP is zero, whereas FP of the latter is 0.29. We obtained similar comparison results for FNs.

Join tree selection. We then tested the sampling-based join tree selection technique presented in Sec. 9. Since FP is the real error of JCard, we present the accumulated FP of 1,000 real-negative insertions in Fig. 16(c). In this experiment, we select no candidate tuple and set $\theta = 0.4$. First, with reference to Fig. 14(a), we note that our sampling technique produced the optimal join trees. Second, we note that the estimation error converges quickly, *i.e.*, after 700 sample insertions.

Experiment E: Deletions and replacements.

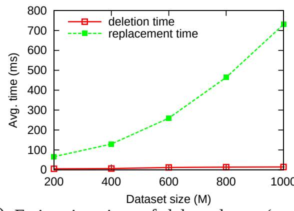
In this experiment, we tested the performance of the support of deletions and replacements as reported in Fig. 17(a). Since the error of replacements is identical to that of insertions (as discussed in Sec. 7), we focused on the estimation time here. We selected 10% candidate tuples. Fig. 17(a) shows that the time overhead of deletions is tiny (*e.g.*, 14.3ms for the dataset of 1G bytes); and the replacement time is almost the same as the insertion time (Fig. 13(a)).

Experiment F: Projections

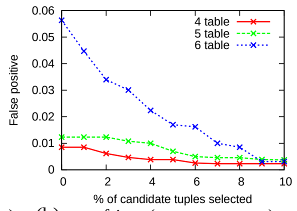
Next, we show the results on views with projections. We use on SYNTHETICDB as it is easier to control. We project out some attributes of relations randomly, where the primary keys and the join attributes may be projected out. As remarked in Sec 8, when needed, we adopt lineage technique (*e.g.*, [16], [23]) as a blackbox to trace the tuples to-be-updated.

Estimation error. To study the performances of incorporating lineage technique into JCard, we vary the number of relations of views. Figs. 17(b)-(c) present FPs of insertions and deletions of our extended JCard, respectively. Figs. 17(b)-(c) show that with more joins, FP increases rapidly, as it is harder to estimate join cardinalities accurately with more joins [28]. However, FPs are well controlled under 6%. There is no FN due to the set semantics of projections.

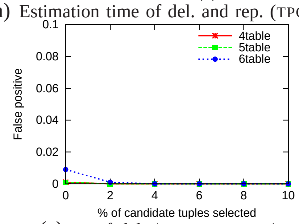
Estimation time and lineage computation time. Figs. 17(d)-(e) report the estimation time of JCards with $x\%$ candidate tuples on insertions and deletions, respectively. From Fig. 17(d), we observe that the estimation on views with projections takes longer time than that without projections (Fig. 15(c)). For example, when the views contains six relations and $x = 0$, JCards for views without projections are roughly 300 times faster than those with projections. It is not surprising because propagating tuple values is more time-consuming than propagating counts. Fig. 17(d) also shows the estimation time of JCards on views having four and five relations. As expected, the estimation time increases rapidly as the number of relations increased. Importantly, the side-effect estimation times are significantly smaller than the translation time.



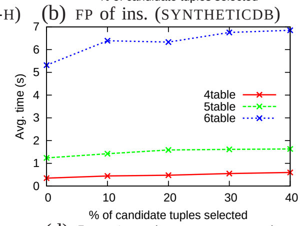
(a) Estimation time of del. and rep. (TPC-H)



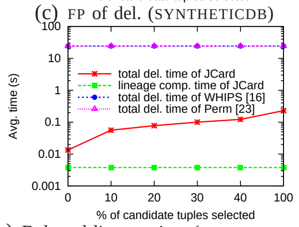
(b) FP of ins. (SYNTHETICDB)



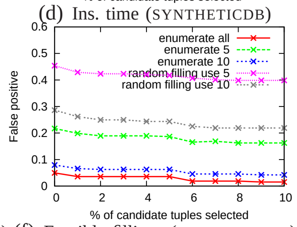
(c) FP of del. (SYNTHETICDB)



(d) Ins. time (SYNTHETICDB)



(e) Del. and lineage time (SYNTHETICDB)



(f) Feasible fillings (SYNTHETICDB)

Fig. 17. Estimation time of replacements and performance results of JCard on views with projection

Further, we illustrate a simple performance breakdown of total deletion time. (We do not show view insertions as they are not supported by lineage.) We set the view with four tables for simplicity and vary the percentage of candidate tuples. Fig. 17(e) shows that deletions always take less than 0.5s. In addition, the lineage computation accounts for a small fraction of the total time (*e.g.*, 6% of total time when $x = 10\%$).

[16], [23] are capable of exact view deletions. Deletion translation time dominated the time of lineage computation [16] or retrieval [23]. Fig. 17(e) reports their time (24.3s for [16] and 24.2s for [23]). It is clear that JCard is significantly more efficient. Even when x = 100% (where no tuple is summarized and no error is produced by JCard), JCard is about two orders of magnitude faster than [16] and [23].

Feasible filling. Finally, we compare our greedy approach with a random filling approach as shown in Fig. 17(f). We also show FP of enumerating all possible fillings, as a reference. Fig. 17(f) shows that when the number of feasible fillings k is fixed (*e.g.*, k=5 or k=10), our approach is clearly more accurate than the random approach.

11 Conclusions

In this paper, we proposed a data-oriented approach to provide practical support for the view update problem. Specifically, we proposed a side-effect detector for SPJ views that estimates or detects whether a view update causes side effects and rejects untranslatable updates early, in turn avoiding costly update translations. The core of the detector was a novel structure — the update filter. In this paper, the update filter is a join cardinality summary JCard that consists of structures that summarize (extended) dangling tuples and the source of view tuples. JCard is derived from a notion of cardinality equivalence. We proposed an estimation algorithm on `JCard`, and we extended `JCard` to support projections. We presented optimizations to construct an efficient and accurate `JCard`. Extensive experiments demonstrated that `JCard` could be tuned to be accurate on TPC-H, DBLP and our synthetic dataset. All proofs are available in the appendices.

With regard to future work, we are investigating a larger class of view definitions such as views with unions and selections with inequality.

Acknowledgement. This work is partly supported by GRF210510 and GRF210409.

References

[1] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. Uldbs: databases with uncertainty and lineage. VLDB, pages 953–964, 2006.

[2] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. VLDB, pages 900–911, 2004.

[3] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: A language for updatable views. In *PODS*, pages 42–67, 2006.

[4] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.

[5] I. Boneva, A.-C. Caron, B. Groz, Y. Roos, S. Tison, and S. Staworko. View update translation for xml. In *ICDT*, pages 42–53, 2011.

[6] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[7] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, 2004.

[8] R. Bryant, R. Katz, and E. Lazowska. Big-data computing: Creating revolutionary breakthroughs in commerce, science, and society. In *Computing Research Initiatives for the 21st Century*, 2008.

[9] P. Buneman, S. Khanna, and W. C. Tan. Data provenance: Some basic issues. FST TCS 2000, pages 87–93, 2000.

[10] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. PODS, pages 150–158, 2002.

[11] H. Chen and H. Liao. A comparative study of view update problem. In *DSDE*, pages 83–89, 2010.

[12] B. Choi, G. Cong, W. Fan, and S. D. Viglas. Updating recursive XML views of relations. In *ICDE*, pages 766–775, 2007.

[13] G. Cong, W. Fan, and F. Geerts. Annotation propagation revisited for key preserving views. CIKM '06, pages 632–641, 2006.

[14] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the complexity of view update analysis and its application to annotation propagation. *TKDE*, 24(3):506–519, 2012.

[15] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *J. ACM*, 31:742–760, 1984.

[16] Y. Cui and J. Widom. Run-time translation of view tuple deletions using data lineage. *Technical report, Standford University*, 2001.

[17] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB J.*, 12(1):41–58, 2003.

[18] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *VLDB*, pages 368–377, 1978.

[19] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, 1982.

[20] L. Fegaras. Propagating updates through xml views using lineage tracing. In *ICDE*, pages 309–320, 2010.

[21] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, pages 233–246, 2005.

[22] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, pages 461–472, 2001.

[23] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. ICDE, pages 174–185, 2009.

[24] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *TODS*, 13:486–524, 1988.

[25] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. VLDB, pages 675–686, 2007.

[26] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[27] IBM. *IBM DB2 Universal Database SQL Reference*. *http://www.ibm.com/software/data/db2/* .

[28] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, pages 268–277, 1991.

[29] A. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, 1985.

[30] A. Keller. The role of semantics in translating view updates. *Computer*, 19:63–73, January 1986.

[31] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates through views: A new hope. In *ICDE*, page 2, 2006.

[32] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on sparql views. In *WWW*, pages 655–664, 2011.

[33] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *PODS*, pages 49–55, 2003.

[34] J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *TODS*, 28:175–208, 2003.

[35] F. Li and Y. Ishikawa. Query processing with materialized views in a traceable p2p record exchange framework. In *WAIM*, 2010.

[36] J. Liu, C. Liu, T. Haerder, and J. X. Yu. Updating typical xml views. In *DASFAA*, pages 126–140, 2012.

[37] J. Liu, M. Roantree, and Z. Bellahsene. Optimizing xml data with view fragments. In *ADC*, pages 151–159, 2010.

[38] G. Luo and P. S. Yu. Content-based filtering for efficient online materialized view maintenance. In *CIKM*, pages 163–172, 2008.

[39] Oracle. *SQL Reference*. *http://www.oracle.com/technology/documentation/database10g.html*.

[40] Richard Cyganiak. DBLP. *http://www4.wiwiss.fu-berlin.de/bizer/d2rq/benchmarks/index01.html*.

[41] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.

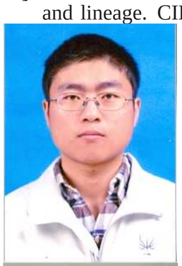
[42] SQL server. *Bayesian Network tools in Java*. *http://msdn.microsoft.com/library*.

[43] A. N. Swami and K. B. Schiefer. On the estimation of join result sizes. In *EDBT*, pages 287–300, 1994.

[44] TPC-H. TPC Benckmark H (Revision 2.14.2), 2011. *http://www.tpc.org/tpch/spec/tpch2.14.2.pdf*.

[45] L. Wang, M. Mulchandani, and E. Rundensteiner. Updating XQuery views published over relational data: A round-trip case study. In *Xsym*, pages 223–237, 2003.

[46] J. Widom. Trio: a system for integrated management of data, accuracy, and lineage. CIDR, 2005.

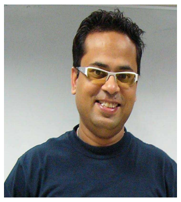


Yun Peng is a PhD student in the Department of Computer Science, Hong Kong Baptist University. He received his BSci and MPhil degrees in Computer Science from Shandong University in 2006 and Harbin Institute of Technology (HIT) in 2008, respectively. His research interests include graph-structured databases. He is a member of the Database Group at Hong Kong Baptist University (http://www.comp.hkbu.edu.hk/∼db/).

Byron Choi received the bachelor of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively. He is now an assistant professor in the Department of Computer Science at the Hong Kong Baptist University.

Jianliang Xu is an associate professor in the Department of Computer Science, Hong Kong Baptist University. He received his BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and his PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He held visiting positions at Pennsylvania State University and Fudan University.

Haibo Hu is a research assistant professor in the Department of Computer Science, Hong Kong Baptist University. Prior to this, he held several research and teaching posts at HKUST and HKBU. He received his PhD degree in Computer Science from the Hong Kong University of Science and Technology in 2005. His research interests include mobile and wireless data management, location-based services, and privacy-aware computing.



Sourav S Bhowmick is an Associate Professor in the School of Computer Engineering, Nanyang Technological University. He is a Visiting Associate Professor at the Biological Engineering Division, Massachusetts Institute of Technology. He held the position of Singapore-MIT Alliance Fellow in Computation and Systems Biology program (05'-12'). He received his Ph.D. in computer engineering in 2001.

Appendix A
Proofs

In this appendix, we provide the proofs of all propositions and theorems in the paper.

A.1 Proof of Proposition 4.1

Proposition 4.1: *When* KL *divergence of the estimated distribution of insertions tends to 0, i.e.,* $KL(P_D(\mathbb{M},\mathbb{V})||P_E(\mathbb{M},\mathbb{V})) \to 0$*, the expected false positives of the detector tends to 0, i.e.,* $EXP(E_+(\mathbb{M},\mathbb{V})) \to 0$. □

Proof: Recall that $\mathbb{M}$ and $\mathbb{V}$ are the random variables of insertions and views, respectively; θ is the user-defined parameter in the definition of false positives. Let $|V|$ be the size of a view $V \in \mathbb{V}$ before an insertion. Let $CP(M,V)$ denote the Cartesian product of relations involved in the view V after the insertion $M \in \mathbb{M}$. Then, the expected number of false positives is:

$$\begin{aligned}
& EXP(E_+(\mathbb{M},\mathbb{V})) \\
= \;& \textstyle\sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} P(M,V)\times E_+(M,V) \\
= \;& \textstyle\sum_{V\in\mathbb{V}} P\Big(f_{est}(\mathbb{M},V) \geq |V|+1+\theta, f_{real}(\mathbb{M},V) = |V|+1\Big) \\
\leq \;& \textstyle\sum_{V\in\mathbb{V}} P\Big(f_{est}(\mathbb{M},V) - f_{real}(\mathbb{M},V) \geq \theta\Big) \\
\leq \;& \textstyle\sum_{V\in\mathbb{V}} P\Big(|f_{est}(\mathbb{M},V) - f_{real}(\mathbb{M},V)| \geq \theta\Big) \\
< \;& \sum_{V\in\mathbb{V}} \frac{EXP(\big|f_{est}(\mathbb{M},V) - f_{real}(\mathbb{M},V)\big|)}{\theta} \\
& \qquad\qquad \text{by Markov's inequality} \\
= \;& \sum_{V\in\mathbb{V}} \frac{\sum_{M\in\mathbb{M}} P(M)\times \big|f_{est}(M,V) - f_{real}(M,V)\big|}{\theta} \\
\leq \;& \sum_{V\in\mathbb{V}} \frac{\sum_{M\in\mathbb{M}} \big|f_{est}(M,V) - f_{real}(M,V)\big|}{\theta} \\
= \;& \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{\big|f_{est}(M,V) - f_{real}(M,V)\big|}{\theta} \\
= \;& \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{\big||CP(M,V)|\times P_E(M,V) - |CP(M,V)|\times P_D(M,V)\big|}{\theta} \\
& \qquad\qquad \text{similar to Getoor et al. [22]} \\
= \;& \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{|CP(M,V)|\times \big|P_E(M,V) - P_D(M,V)\big|}{\theta} \\
\leq \;& |CP| \times \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{\big|P_E(M,V) - P_D(M,V)\big|}{\theta} \\
& \qquad\qquad \text{where } |CP| = \max\{|CP(M,V)|\} \\
\leq \;& \frac{|CP|}{\theta} \times KL(P_D(\mathbb{M},\mathbb{V})||P_E(\mathbb{M},\mathbb{V})) \\
& \qquad\qquad \text{by Pinsker's inequality}
\end{aligned}$$

Therefore the expected error is bounded by KL divergence. □

A.2 Proof of Proposition 4.2

Proposition 4.2: *When* KL *divergence of the estimated distribution of insertions tends to 0, i.e.,* $KL(P_D(\mathbb{M},\mathbb{V})||P_E(\mathbb{M},\mathbb{V})) \to 0$*, the expected false negatives of the detector tends to 0, i.e.,* $EXP(E_-(\mathbb{M},\mathbb{V})) \to 0$. □

Proof: The total number of false negatives is:

$$\begin{aligned}
& EXP(E_-(\mathbb{M},\mathbb{V})) \\
= \;& \textstyle\sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} P(M,V)\times E_-(M,V) \\
= \;& \textstyle\sum_{V\in\mathbb{V}} P\Big(f_{est}(\mathbb{M},V) < |V|+1+\theta, f_{real}(\mathbb{M},V) \geq |V|+2\Big) \\
\leq \;& \textstyle\sum_{V\in\mathbb{V}} P\Big(f_{real}(\mathbb{M},V) - f_{est}(\mathbb{M},V) \geq 1-\theta\Big) \\
\leq \;& \textstyle\sum_{V\in\mathbb{V}} P\Big(\Big|f_{real}(\mathbb{M},V) - f_{est}(\mathbb{M},V)\Big| \geq 1-\theta\Big) \\
\leq \;& \textstyle\sum_{V\in\mathbb{V}} \frac{EXP(\big|f_{real}(\mathbb{M},V) - f_{est}(\mathbb{M},V)\big|)}{1-\theta}, \\
& \qquad\qquad \text{by Markov's inequality} \\
= \;& \sum_{V\in\mathbb{V}} \frac{\sum_{M\in\mathbb{M}} P(M)\big|f_{real}(M,V) - f_{est}(M,V)\big|}{1-\theta} \\
\leq \;& \sum_{V\in\mathbb{V}} \frac{\sum_{M\in\mathbb{M}} \big|f_{real}(M,V) - f_{est}(M,V)\big|}{1-\theta} \\
= \;& \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{\big|f_{real}(M,V) - f_{est}(M,V)\big|}{1-\theta} \\
= \;& \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{\big||CP(M,V)|\times P_E(M,V) - |CP(M,V)|\times P_D(M,V)\big|}{1-\theta} \\
& \qquad\qquad \text{similar to Getoor et al. [22]} \\
= \;& \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{|CP(M,V)|\times \big|P_E(M,V) - P_D(M,V)\big|}{1-\theta} \\
\leq \;& |CP| \times \sum_{V\in\mathbb{V}}\sum_{M\in\mathbb{M}} \frac{\big|P_E(M,V) - P_D(M,V)\big|}{1-\theta} \\
& \qquad\qquad \text{where } |CP| = \max\{|CP(M,V)|\} \\
\leq \;& \frac{|CP|}{1-\theta} \times KL(P_D(\mathbb{M},\mathbb{V})||P_E(\mathbb{M},\mathbb{V})) \\
& \qquad\qquad \text{by Pinsker's inequality}
\end{aligned}$$

Similar to the derivation of Proposition 4.1, the total number of false negatives and the expected error (false negatives) are bounded by KL divergence. □

A.3 Proof of Theorem 9.1

Theorem 9.1: *SCT is NP-complete.* □

Proof: We establish the hardness of SCT by a reduction from MINIMUM SET COVER (MSC).

We first recall the definition of MSC: Given a universe $\mathcal{U}$ and a set $\mathcal{S}$ of subsets of $\mathcal{U}$, we want to find a subset $\mathcal{C}$ of $\mathcal{S}$ such that (i) $|\mathcal{C}| \leq B$; (ii) the maximum number of elements in $\mathcal{U}$ is covered.

Given an instance of MSC, we construct an instance of SCT that contains the scenario shown in Fig. 19(b).

The instance of SCT contains four parts. (i) We use possible insertions to encode the universe $\mathcal{U}$. (ii) We use selection predicates to define possible insertions to be exactly $\mathcal{U}$. (iii) A relation R_i is used to encode a subset $C_i \in \mathcal{S}$ of MSC. (iv) A segment s_i^j of a view tuple to-be-inserted t_i that already exists in G^- (R_j) is used to encode an element u_i in C_j. The SCT instance is encoding by composing such segments in G^- in a special way. Next, we elaborate the four parts below.

For simplicity, we refer the domain of a relation to denote the *domain of the join attribute of the relation* and the discussion always focuses on join attributes only, unless specified in (ii).

(i) For each u_i in $\mathcal{U}$, we define a view insertion, denoted as t_i.

(ii) To ensure that t_i's are the only possible insertions, we introduce the following selection clauses. Specifically, for each

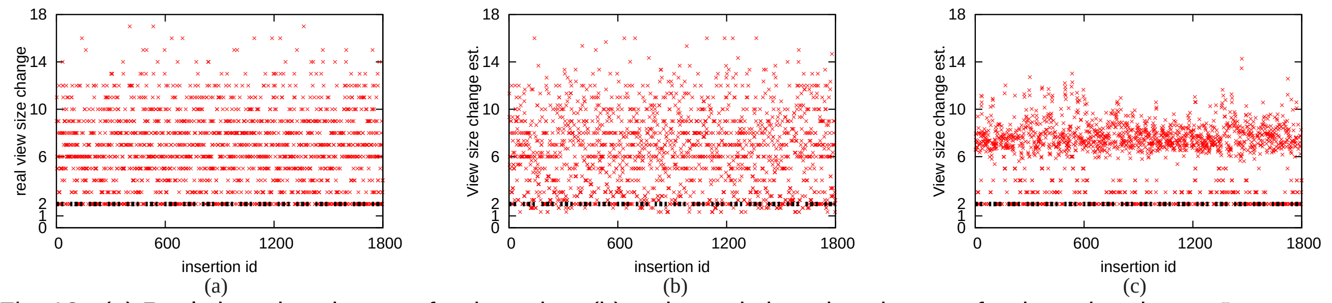


Fig. 18. (a) Real view size change after insertion; (b) estimated view size change after insertion due to $J_{lineitem}$; and (c) estimated view size change after insertion due to J_{order}

insertion t_i: (s_i^1, s_i^2, ... s_i^m), we include a selection in the view definition, σ_i: A_1 = s_i^1 $\wedge$ A_2 = s_i^2 $\wedge$ ... $\wedge$ A_m = s_i^m, where A_i's are simply attributes of s_i's.

(iii) Suppose the MSC instance contains m subsets, we create m relations. Each relation R_i encodes a subset C_i of the MSC instance. The join of the view definition is $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$.

(iv) This part encodes the membership of an element u_i in a subset C_j. Denote that t_i is composed by segments of tuples (s_i^1, s_i^2, ... s_i^j, ...). We can construct the G^- graph such that (iv.i) s_i^j is new *if* $u_i \notin C_j$ and (iv.ii) s_i^j exists in G^-, but not in the view, *if* $u_i \in C_j$. Moreover, for (iv.ii), s_i^j is placed in some large equivalence class; and the class contains one tuple $s_i^{j\prime}$ joinable with some neighbouring tuples, *e.g.*, s_i^{j+1} and s_i^{j-1}. This forms the scenario in Fig. 19(b). Then we can always set a threshold θ such that the estimation always report zero side effect for t_i. Suppose C_j = $\{u_{i_1}, u_{i_2}, ..., u_{i_k}\}$, we make $s^{j\prime}$ = $s_{i_1}^{j}{}'$ = $s_{i_2}^{j}{}'$ = ... = $s_{i_k}^{j}{}'$. Finally, to ensure that $s^{j\prime}$ is selected by our greedy algorithm, we add dummy tuples D^j s.t. $s^{j\prime} \cup D^j$ misses only one tuple to form a full embedding.

The error caused by Case (iv.ii) can be avoided by selecting the existing segments (*i.e.*, candidate tuples) from G^- and determine those side effects with them separately. The SCT problem is now to select B candidate tuples to reduce the most errors. Suppose that we selected $s^{j\prime}$ from G^-. When either of the insertions t_{i_1}, t_{i_2}, ..., t_{i_k} are encountered, the estimation algorithm will locate the additional path formed by $s^{j\prime}$, leads to a count at least 2.

Therefore, the SCT problem determines the least number of $s^{j\prime}$'s to be selected from G^- which estimates the insertions t_i's the most accurate with the budget B. It is straightforward that the selected s^j's are the subsets C_j's to-be-selected of the MSC problem. □

Appendix B Experiments with Side-effect Estimation and Classical Cardinality Estimation

This appendix provides supplementary information for the experiment discussed in Example 4.1 of Sec. 4. The experiment tested the estimation of two estimators, namely $J_{lineitem}$ and J_{order}, on the simplified Q7 of TPCH. We generated 1800 random insertions, *all with side effects*. The threshold for side-effect errors θ was 1. Fig. 18(a) shows the actual change of the size (*i.e.*, the number of tuples) of the view on the y-axis after the x-th insertion on the view. Each dot represents one insertion. Due to space constraint, it is not possible to derive the exact numbers from the figure. However, we highlight the change of 2, which indicates the boundary between the presence and absence of the side effect. For example, a dot with a y-value 14 means that the corresponding insertion causes 14 new tuples and therefore has side effects.

The estimated counts of the change of the view size reported by $J_{lineitem}$ and J_{order} after the insertions are shown on the y-axis of Fig. 18(b) and Fig. 18(c), respectively. Although it is not possible to derive the overall errors from the figures, we report that the errors of *cardinality* estimation of $J_{lineitem}$ were 3.7 times smaller than those of J_{order}. The figures showed that the distribution of dots of Fig. 18(a) was closer to that of Fig. 18(b) than that of Fig. 18(c). However, regarding side-effect estimation, it is important for $J_{lineitem}$ and J_{order} to estimate the number of side-effect free insertions, *i.e.*, dots that have a value below 2. In Fig. 18(b), the number of incorrect estimations of $J_{lineitem}$ was 79. In comparison, Fig. 18(c) showed that J_{order} did not estimate any side-effect free insertions. Therefore, J_{order} was a perfect side-effect estimator, with respect to the 1800 insertions.

Appendix C Details of Procedure `update_equiv_class`

In this appendix, we present the details of Procedure `update_equiv_class` (Line 02 of Fig. 6). `update_equiv_class` essentially splits the equivalence classes of `JCard` for higher estimation accuracies. (We omit its pseudo-code since it involves tedious details on manipulating nodes and edges.)

Suppose $\mathcal{C}$ is the the segments of the view tuple t_v to-be-updated, we make an *individual equivalence class c_s for each segment* t_s in $\mathcal{C}$. $\mathcal{C}$ will form at least *one* embedding due to c_ss. If the segment t_s exists (*i.e.*, t_s is in R_s of G^-), we split t_s from its original equivalence class and form an individual class c_s, like in Example 6.1. On the other hand, if t_s is new, we create a new equivalence class c_s for t_s. Since it is certain that t_s is directly relevant to the update, having c_s specially for t_s improves the accuracy of `propagate`. To update the weights of edges related to c_s, as they are required by the formula in Line 07 of `propagate`, we perform a local join between. t_s and the neighbouring relations of R_s in $J_{\mathcal{V}}$. . .

This update of equivalence classes often needs. relatively little computation in practice. Firstly, the existing segments

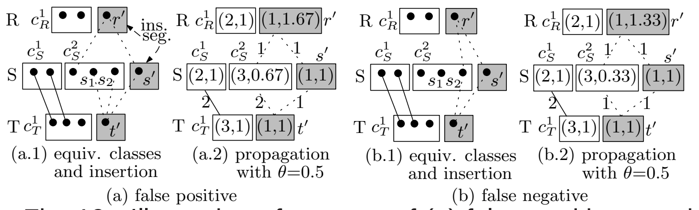

(a.1) equiv. classes and insertion
(a.2) propagation with θ=0.5
(b.1) equiv. classes and insertion
(b.2) propagation with θ=0.5
(a) false positive
(b) false negative
ins. seg.

Fig. 19. Illustration of reasons of (a) false positives; and (b) false negatives

of a view tuple to-be-inserted are directly retrieved from the `JCard`. Secondly, its new segments are locally joined with the existing `JCard`. These two steps are efficient.

Before ending this appendix, we note that if `propagate` estimates that t_v causes side effects, the splits and the edge weights are revoked. Otherwise, `propagate` declares t_v is side-effect free and if t_v can be translated and updated, we merge the split G^- with Definition 5.5. Moreover, as t_v is inserted, the extended dangling tuples related to t_v are removed from G^- and inserted into G^+.

APPENDIX D
ANALYSIS OF SOURCE OF ESTIMATION ERRORS

This appendix analyzes how errors are caused by `JCard` and the estimation algorithm in Fig. 6. Foremost, we present the scenario where errors may occur.

Proposition 4.1: *Given an insertion of view tuple t_v and the set T_v of segments of t_v, if the number of new embeddings after adding T_v into G^- is greater than 1, then (i) false positives and (ii) false negatives are possible.* □

Proof: The proof can be established by deriving two small examples.

Suppose the estimator declares side effects when the count returned by Procedure `propagate` is greater than or equal to 1 + θ. Otherwise, the estimator declares no side effect.

(i) We first illustrate false positives with a small example shown in Fig. 19(a). We suppose that the view definition is $R \bowtie S \bowtie T$. Suppose the segments of the view tuple t_v are illustrated on the RHS of Fig. 19(a) and the `JCard` of G^- is shown on the LHS of the figure. For simplicity, we assume the segment t_1 is new; and the segments (t_2 and t_3) of R and S are old. We sketch the equivalence classes in the `JCard` whose cardinalities are indicated for illustration only.

Suppose that the tuple t_3 can join with a tuple t_s of S and t'_s can be joined with t_1. Since t'_s is old and Procedure `propagate` uses `est` of the *equivalence class* of t'_s to conduct estimation (Line 08). While t'_s cannot join with t_3, another tuple t_s, in the same class with t_s, can join with t_3. By Procedure `propagate`, the count returned is 1 (due to the new embedding) + 1 × $|E_{c_1,c_s}|$ × 1 / $|c_s|$, where E_{c_1,c_s} is the number of joinable tuple pairs between the classes c_1 and c_s containing t_1 and t_s, respectively. Assume that the estimator declares side effect (positive) when the estimation is greater than or equal to 1 + θ, where θ is a user-defined threshold. Thus, there are false positives when $|c_s|$ / $|E_{c_1,c_s}| \leq 1$ / θ.

(ii) Similarly, we can construct a case to show possible false negatives, shown in Fig. 19(b). The insertion causes two new embeddings as before. The tuple segments are t_1, t_2 and t_3, where t_1 is new and t_2 and t_3 are old. t_1 joins with t_s and t_s in turn joins with t_3. Procedure `propagate` returns 1 + 1 × $|E_{c_1,c_s}|$× 1 / $|c_s|$. The estimator returns false negative when $|c_s|$ / $|E_{c_1,c_s}| > 1$ / θ. □

Discussions. Fig. 19(a) illustrates how the join cardinality is over-estimated due to the summarization of tuples, which causes false positives. The join cardinality is over-estimated in Fig. 19(a) as s_1 and s_2 are placed in the same equivalence class c_S^2 but they cannot join with r'. Assuming random insertions, the larger the class is, the higher the probability such over-estimation occurs.

Fig. 19(b) shows how the counts are "averaged out" among the equivalence classes during propagation, which causes false negatives. In Fig. 19(b), false negatives are directly proportional to the size of equivalence class c_S^2.

These observations show that reducing the size of equivalence classes reduces the errors. The selection of candidate view tuples from equivalence classes (Sec. 5.2) is a refinement of equivalence classes. One may be tempted to use bisections to refine them until each class is smaller than $|E_{t',c_S^2}|/\theta$. However, our preliminary experiments show that such a bisection-based method is not robust. In particular, the estimation accuracy is very sensitive to the values of θ. No improvement can be observed until a certain number of bisections are applied that led to a sharp increase of accuracies.

APPENDIX E
ADDITIONAL EXPERIMENTS

Due to space limitation, we could only highlight the representative results in Sec. 10. Firstly, Sec. 10 presents the experiments with $Q7$ from TPC-H. In this appendix, we present the experimental results of all join queries on TPC-H and show that the results are similar to those $Q7$ presented in Sec. 10. Secondly, while Sec. 10 presents the effectiveness of our optimization techniques with TPC-H, this appendix presents a supplementary experiment on the DBLP dataset. Finally, this appendix presents the performances of `JCard` on replacements on views with projections.

E.1 Benchmark with All Join Queries of TPC-H

In this experiment, we test `JCard` on *all* join queries of TPC-H [44] as listed in Fig. 20. In a nutshell, we extract the join queries from the TPC-H. $Q1$ and $Q6$ involve only one table and are thus omitted, as their side-effect detections are straightforward. For the only cyclic join query $Q5$, we broke the cycle by randomly removing a join when detecting side effects. We use the same settings as in Sec. 10. In particular, we select 10% candidate tuples and θ was 0.4. Similar to the experiments in Sec. 10, we present the false positive (FP) and false negative (FN). The result is shown in Table 3.

From Table 3, we observe that the FP and FN equal to zero for almost all queries. Table 3 also compares the side-effect estimation time and the exact detection time. We observe that our side-effect estimation is on average 8.3 times faster than the exact detection on average. This further verifies that side-effect detection is often efficient while the estimation can be

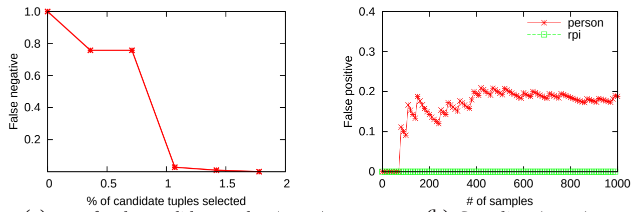
TABLE 3
Performances of insertions on TPC-H benchmark queries

Query	# of tables	# of joins	FP	FN	est. time (ms)	detect. time (ms)
$Q1$	1	0	-	-	-	-
$Q2$	5	4	0	0	59.1	300.8
$Q3$	3	2	0	0	140.6	1096.6
$Q4$	2	1	0	0	45.5	442.5
$Q5$	6	6	0	0	699.5	1910.6
$Q6$	1	0	-	-	-	-
$Q7$	6	5	0	0	716.1	2942.3
$Q8$	8	7	0	0	1767.8	5325.9
$Q9$	6	5	0	0	3087.1	6119.6
$Q10$	5	4	0.11	0	196.3	1104.5
$Q11$	3	2	0	0	5.2	57.7
$Q12$	2	1	0	0	42.9	505.5
$Q13$	2	1	0	0	10.3	93.4
$Q14$	2	1	0	0	66.1	720.2
$Q15$	2	1	0	0	16.2	427.4
$Q16$	2	1	0	0	6.6	68.8
$Q17$	2	1	0	0	66.1	720.2
$Q18$	3	2	0	0	140.6	1096.6
$Q19$	2	1	0	0	66.1	720.2
$Q20$	2	1	0	0	0.4	0.9
$Q21$	4	3	0	0	413.0	2244.1
$Q22$	2	1	0	0	10.3	93.4

easily tuned to be even more efficient and (at the same time) highly accurate.

E.2 Supplementary Experiments of Optimizations with DBLP

This experiment uses the same setting on DBLP as presented in Sec. 10. Similar to the experiments of optimizations on TPC-H (Sec. 10 (Experiment D)), we present the results of the optimizations by presenting the effectiveness of equivalence classes and the join tree selection.

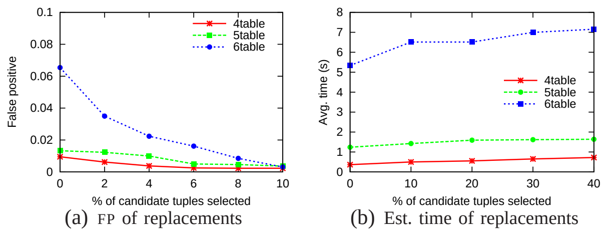


(a) FN of only candidate tuples (DBLP) (b) Sampling (DBLP)

Fig. 21. Performances of optimizations on DBLP

Effectiveness of equivalence classes. This experiment verifies that the equivalence classes are indeed important, by skipping the equivalence classes in the side-effect estimation. We only report the FN in Fig. 21(a) as the removal of the equivalence classes will not cause FP. We use the optimal join tree as shown in Fig. 14(b) and Fig. 14(e). Foremost, the results are consistent with those presented earlier – the more candidate tuples selected, the smaller the errors. We observe that the estimation errors could be large without the equivalence classes. When we compare the difference of errors with and without the equivalence classes (Fig. 14(e) and Fig. 21(a)), we note that the equivalence classes significantly improves the accuracies.

Join tree selection. We perform the sampling method to select optimal join trees as presented in Sec. 9. Since FP is the real error of the JCard, we present the accumulated FP of 1,000 negative insertions as shown in Fig. 21(b). In this experiment, we select no candidate tuple and set $\theta = 0.4$. We note that the estimated error converges quickly, *i.e.*, after 400 sample insertions. Considering with Fig. 14(b), we observe that our sampling technique can easily determine the optimal join trees.



(a) FP of replacements (b) Est. time of replacements

Fig. 22. JCard's error and runtime of replacements on views having projections on SYNTHETICDB

E.3 Supplementary Experiments of Replacements on Views with Projections

We test JCard's performances of replacements on views with projections. This experiment uses the same settings on SYNTHETICDB as presented in Sec. 10. Similar to the experiments of insertions and deletions (Sec. 10 (Experiment F)), we present JCard's performances of replacements with respect to both the side-effect estimation error and the estimation time.

Estimation error. Fig. 22(a) reports FP of replacements of our extended JCard on views with projections. Fig. 22(a) shows that with more joins, FP increases rapidly. This is consistent with the result that it is harder to estimate join cardinalities accurately with more joins [28]. However, FPs are well controlled under 7%. There is no FN due to the set semantics of projections.

Estimation time. Fig. 22(b) reports the estimation time of JCards with $x\%$ candidate tuples on replacements. Fig. 22(b) shows that replacements have almost identical estimation time with insertions (Fig. 17(d)) as the times of deletions are very small, *e.g.*, less than 30ms when $x = 0\%$ (Fig. 17(e)).

$Q2$
```
select *
from
    part,
    supplier,
    partsupp,
    nation,
    region
where
    p_partkey=ps_partkey
    and s_suppkey=ps_suppkey
    and s_nationkey=n_nationkey
    and n_regionkey=r_regionkey
```

$Q3$
```
select *
from
    customer,
    orders,
    lineitem,
where
    c_custkey=o_custkey
    and l_orderkey=o_orderkey
```

$Q4$
```
select *
from
    orders,
    lineitem
where
    and l_orderkey=o_orderkey
```

$Q5$
```
select *
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey=o_custkey
    and l_orderkey=o_orderkey
    and l_suppkey=s_suppkey
    and s_nationkey=n_nationkey
    and n_regionkey=r_regionkey
```

$Q7$
```
select *
from
    customer,
    orders,
    lineitem,
    supplier,
    nation n1,
    nation n2
where
    s_suppkey=l_suppkey
    and o_orderkey=l_orderkey
    and c_custkey=o_custkey
    and s_nationkey=n1.n_nationkey
    and c_nationkey=n2.n_nationkey
```

$Q8$
```
select *
from
    part,
    supplier,
    lineitem,
    orders,
    customer,
    nation n1,
    nation n2,
    region
where
    p_partkey=l_partkey
    and s_suppkey=l_suppkey
    and l_orderkey=o_orderkey
    and o_custkey=c_custkey
    and c_nationkey=n1.n_nationkey
    and n1.n_regionkey=r_regionkey
    and s_nationkey=n2.n_nationkey
```

$Q9$
```
select *
from
    part,
    supplier,
    lineitem,
    partsupp,
    orders,
    nation
where
    s_suppkey=l_suppkey
    and ps_suppkey=l_suppkey
    and ps_partkey=l_partkey
    and p_partkey=l_partkey
    and o_orderkey=l_orderkey
    and s_nationkey=n_nationkey
```

$Q10$
```
select *
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey=o_custkey
    and l_orderkey=o_orderkey
    and c_nationkey=n_nationkey
```

$Q11$
```
select *
from
    partsupp,
    supplier,
    nation
where
    ps_suppkey=s_suppkey
    and s_nationkey=n_nationkey
```

$Q12$
```
select *
from
    orders,
    lineitem
where
    l_orderkey=o_orderkey
```

$Q13$
```
select *
from
    customer,
    orders
where
    c_custkey=o_custkey
```

$Q14$
```
select *
from
    lineitem,
    part
where
    l_partkey=p_partkey
```

$Q15$
```
select *
from
    supplier,
    lineitem
where
    l_suppkey=s_suppkey
```

$Q16$
```
select *
from
    partsupp,
    part
where
    p_partkey=s_suppkey
```

$Q17$
```
select *
from
    lineitem,
    part
where
    l_partkey=p_partkey
```

$Q18$
```
select *
from
    customer,
    orders,
    lineitem,
where
    c_custkey=o_custkey
    and l_orderkey=o_orderkey
```

$Q19$
```
select *
from
    lineitem,
    part
where
    l_partkey=p_partkey
```

$Q20$
```
select *
from
    supplier,
    nation
where
    s_nationkey=n_nationkey
```

$Q21$
```
select *
from
    supplier,
    lineitem,
    orders,
    nation
where
    s_suppkey=l_suppkey
    and o_orderkey=l_orderkey
    and s_nationkey=n_nationkey
```

$Q22$
```
select *
from
    customer,
    order
where
    o_custkey=c_custkey
```

DBLP
```
select *
from
    Person,
    RelationPersonInProceeding,
    InProceeding,
    Proceeding
where
    PName='bob'
    and <join conditions>
```

Fig. 20. The list of all join queries from TPC-H and the form of views from DBLP