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 TPCH 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 $v_1$. 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 $v_1$ 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 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 updaters exploit data summaries of source databases, in addition...
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.

**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 Spyropoulos proposed the seminal work on view complements [15]. View updates could be translated without side effects if the updates resulted in unchanged view complements (aka 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.

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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>database instance</td>
</tr>
<tr>
<td>( V=\mathbb{V}(I) )</td>
<td>view definition</td>
</tr>
<tr>
<td>( t_v )</td>
<td>a tuple in view</td>
</tr>
<tr>
<td>( \mathbb{G} )</td>
<td>source graph</td>
</tr>
<tr>
<td>( \mathbb{G}^- )</td>
<td>negative graph</td>
</tr>
<tr>
<td>( \mathbb{G}^+ )</td>
<td>database graph</td>
</tr>
<tr>
<td>( R ) or ( R_I )</td>
<td>relation</td>
</tr>
<tr>
<td>( J_D )</td>
<td>join tree (rooted at ( R_I ))</td>
</tr>
</tbody>
</table>

Table 1: Table of frequently used notations
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:

$$V = \pi_A \sigma_P (R_1 \bowtie R_2 \ldots \bowtie R_n),$$

where $A$ is the set of projection attributes of the view, $P$ is the selection predicates and $R_i$s are relations for $i \in \{1, \ldots, n\}$.

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 $V$ 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, $t_1, \ldots, t_n$, where $t_i = \pi_{A_i}(t_v)$ and $A_i$ denotes the attributes of $R_i$ in $V$. We call $U_{i=1}^n \{t_i \mid t_i \text{ is a segment of } t_v \text{ and } t_i \in R_i \}$ 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 $V$ of a relational database instance $I$, where the view $V$ is $V(I)$, and an update $u$ on the view $V$, find a translated update $u'$ on $I$ such that $u \oplus V(I) = V(I \oplus u')$.

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 $V = V(I)$ and its translation $u'$ on $I$, side effects of $u'$ are the changes on the updated view $V(I \oplus u')$ 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 $S$ and $J$ to capture possible selections and joins on attributes of all possible views $V$ on a database instance $I$. We use a random variable $M$ 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 $f_{\text{real}}(M, V)$ and $f_{\text{est}}(M, V)$ 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 $\theta$ 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 $f_{\text{real}} - |V| + 1$ and $f_{\text{est}} \geq f_{\text{real}} + \theta$; 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 $f_{\text{real}} > |V| + 1$ and $f_{\text{est}} < |V| + 1 + \theta$; 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 $P(M, V)$:

$$\text{EXP}(E_+(M, V)) = \sum_{M \in \mathbb{M}} \sum_{V \in \mathbb{V}} P(M, V) \times E_+(M, V).$$
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 $J_{\text{inestem}}$ and $J_{\text{order}}$, 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 $\theta$ 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 $J_{\text{inestem}}$ was 3.7 times smaller than that of $J_{\text{order}}$. 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 $J_{\text{order}}$ exhibited 79 incorrect side-effect estimations while $J_{\text{inestem}}$ did not produce any incorrect estimation.

While false positives and false negatives of a side-effect detector are defined with an error threshold $\theta$, 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., $KL(P_D(M, V)||P_E(M, V)) \to 0$, the expected false positives of the detector tends to 0, i.e., $EXP(E_+(M, V)) \to 0$.

**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., $KL(P_D(M, V)||P_E(M, V)) \to 0$, the expected false negatives of the detector tends to 0, i.e., $EXP(E_-(M, V)) \to 0$.

**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 $t_o$ denote the view tuple to-be-inserted and $t_v.A$ denote the value of $A$ in $t_v$. Let $\text{dom}(A)$ and $\text{adom}(A)$ 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.

(i) Suppose attribute $A$ is projected out. If $|\text{dom}(A)|$ is infinite, or $|\text{dom}(A)|$ is finite but $|\text{adom}(A)|<|\text{dom}(A)|$, $A$ 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 $|\text{adom}(A)|$.

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 $V(I)$ are a finite set $M'$, where $M'$ is an enumeration of insertions on:

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

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

### 5 JOIN Cardinality Summary ($J_{\text{CARD}}$)

In this section, we propose a join cardinality summary ($J_{\text{CARD}}$) for estimating side effects. $J_{\text{CARD}}$ is specially designed for joins since joins are technically challenging in SPJ views. For illustration purposes, we present $J_{\text{CARD}}$ with insertions, unless otherwise specified. $J_{\text{CARD}}$ 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.
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. \( V \).

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

Next, we introduce the terminologies to discuss the tuples summarized in \( J_{\text{card}} \).

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

**Definition 5.1:** Given a set of tuples \( S \) in a database graph \( G \) and a join tree \( J_{V} \), \( S \) is a partial embedding of \( J_{V} \), 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 \( J_{\text{Supplier}} \) 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, \( S = \{p_4, o_1, o_2\} \) 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. \( \{s_5, p_6, o_3\} \) 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 \( J_{V} \).

**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 \{\( o_4, o_2, p_5 \)\} with the dangling tuples \( o_2 \) and \( p_5 \) (Definition 5.2(ii)).
in order; and 0 is $j\text{card}$ from $\text{Agency}$ as $p_1$ does not join with any tuple in $\text{Agency}$.

**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) $t_1.j\text{card} = t_2.j\text{card}$; 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. $\text{JCard}$ ($G^-$) 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 $j\text{card}$ 0, they are of different classes as $\text{Order}$ is the joinable child relation of $p_1$, but that of $p_6$ is $\text{Agency}$ (Definition 5.5(iii)).

**JCard structure and its construction.** $\text{JCard}$ 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 $([c], c.\text{est})$, where $[c]$ is the number of tuples in $c$ and $c.\text{est}$ is the estimated average $j\text{card}$ of the tuples in $c$. $\text{JCard}$ 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 $\text{JCard}$ 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 $\text{JCard}$ 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 $c^2_3(1,0)$ means that the class $c^3_3$ contains one tuple and the average join cardinality of tuple in $c^3_3$ is 0. $|E_{c_3_3,c_4}| = 1$ means that there is one join between $c^3_3$ and $c^4_3$.

There are two remarks on $\text{JCard}$ 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.** $\text{JCard}$ 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, $\text{JCard}$’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 $\{o_4, p_5, o_3\}$ as they will form embeddings by one additional segment, respectively. Specifically, $\{s_1, p_2, o_2\}$ needs a new segment of $\text{Agency}$ joinable to $p_2$ and $\{o_4, p_5, o_3\}$ needs one of $\text{Supplier}$ joinable to $p_3$, 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 $\text{JCard}$. Due to space limitations, an analysis of the causes of errors, which reveals the design issues of $\text{JCard}$, 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 $\text{estimate_side_effects}$ are the $\text{JCard}$, the view definition in the form of join tree $J_{R_i}$, the view tuple to-be-inserted $t_v$ and a parameter $\theta$ on the error threshold. $\text{estimate_side_effects}$ estimates the number of new view tuples generated by inserting $t_v$. We recall that $\text{estimate_side_effects}$ declares a side-effect free insertion when the estimated number of new view tuples is smaller than $1 + \theta$; otherwise, it declares side effects.

The details of $\text{estimate_side_effects}$ can be described as follows. Given a view tuple $t_v$ to-be-inserted, we decompose it into $n$ segments $t_1,...,t_n$, where $n$ is the number of relations in $J_{R_i}$, by $\text{decompose_view_tuple}$ in Line 01. Each segment $t_i$ corresponds to a relation $R_i$ in $J_{R_i}$, $t_1,...,t_n$ form at least an embedding on $J_{R_i}$ for a valid insertion. We update $\text{JCard}$ via $\text{update_equiv_class}$ for a more accurate estimation. In a nutshell, $\text{update_equiv_class}$ 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 \( C \) = decompose_view_tuple\((t_v, J_R)\)
02 \( G^- = update_equiv_class\((J_R, G^-, C)\)\)
03 for each equivalence class \( c \) of the root relation \( R \)
04 \( propagate(c, G^-, J_R) \)
05 let \( C \) be the classes of \( R \) that form embeddings with \( t_v \)
06 return \( \sum_{c|c \in C}(c.\text{est} \times |c| - a > 1 + \theta) \)

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 \( c \) : \( c.\text{est} \)
01 if \( c \) is a leaf node in \( G^- \)
02 return \( c.\text{est} = 1 \)
03 else if not \( c \) is of relation \( R' \) in \( J_R \)
04 for each child relation \( R'' \) of \( R' \) in \( J_R \)
05 \( c.\text{est} \times |c|\)
06 if \( c' \) visited then \( propagate(c', G^-, J_Y) \)
07 \( c.\text{est} \times |c|\)
08 return \( c.\text{est} = \prod_{c | c' \in \text{child}} c.\text{est} \times |c' | \)

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 \( c' \) 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 also denoted as \( s', p', o_2 \) and \( a' \) respectively, if it is clear from the context. Since class \( o_2 \) is split from \( c_2 \), the size of \( c_2 \) is reduced by 1. Since \( a' \) is joinable with \( p_2 \) in \( c_1 \), we add an edge \((a', c_2')\) into \( JCard \) and \( |E_{a',c_2'}(c) = 1 \). Initially, \( o_2 \) and \( a' \) are 1 as they are of leaf relations. Next, we estimate the join cardinality of internal nodes (Line 11 of Fig. 7). For example, \( c_1.\text{est} = \frac{c_2.\text{est} \times |E_{c_2,c_1}|}{|c_1|} \times \frac{a'.\text{est} \times |E_{a',c_2'}|}{|c_2'|} = 1 \times 2 \times 1 = 0.5 \times 1 = 0.5 \). 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 \( c_3.\text{est} \times |c_4| + s'.\text{est} \times |s'| = 0.5 + 1 = 1.5 \). This number is smaller than the real number 2, because of the averaging at \( c_1 \).

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 \( c_1 \) and \( c_2 \), 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, \( |c_1| = 0, |c_2| = 1, |E_{c_2,c_1}| = 1 \) and \( |E_{c_3,c_1}| = 0 \). After the propagation, we obtain the estimated number of new embeddings as \( c_3.\text{est} \times |c_4| + s'.\text{est} \times |s'| = 1 + 1 = 2 \) 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, if the deletion of \( t_v \) is side-effect free and the subsequent insertion of \( t'_v \) is also side-effect free.
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 graphs $G$, 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(e)) 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 (ii) insert it into $G^-$. Fig. 9(f) shows that the estimated number of new embeddings after insertion is $c_{1,t}^{est}+r_3^{est}=2$ 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 inference 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

Let $s$ be a tuple segment of a relation $R$, which is a relation in the view definition $V$. 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 it 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 $1 + \theta$, 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 $(\text{jam}, ie, B_3)$. It can be decomposed into segments $t'=(X, B_3)$, $t'=(Y, ie)$ and $t'=(\text{jam}, Z)$, 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 $f$, $t'=t_4$.
and the number of joining tuples between $T$ and $E$ is $1$, as $t_4$ joins with $e_i$. The number of joining tuples between $T$ and $L$ is $1$, as $t_4$ joins with $l_i$. There are 2 joining tuples in total. If $X=Y=Z=\text{all}$, the number of joining tuples is $4$; for $m$ 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 $J\text{Card}$ before insertion and the propagation on the $J\text{Card}$ is illustrated in Fig. 10(d). $\text{propagate}$ estimates 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 $\text{JCard}$ to detect the side effects.

## 9 Optimization Problems in JCard

There are two important optimization problems in the construction of $J\text{Card}$. In Sec. 9.1, we show that selecting the candidate tuples of $J\text{Card}$ 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 $V$ and the $G^-$ summary of the source database, select a set of candidate to minimize side-effect estimation errors of $\text{propagate}$.

**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 $U$. For the tuple $s_i$ for selection, we create a clause $C_i$. For each $s_i$ whose selection leads to the insertions $U$, we ensure that $u \in C_i$, where $u \in U$. 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 $\text{candidate_tuples}$ (shown in Fig. 11), which is equivalent to a greedy algorithm of MSC, whose approximation ratio is known to be $OPT \times \lg(U)$.

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

The main idea of $\text{candidate_tuples}$ 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 $\text{candidate_tuples}$ is simply the complexity of maximum flow multiplied by $k$.

### 9.2 Optimal Join Tree Selection

The join cardinality summary $JCard \quad G^-$ defined by Sec. 5 assumes a particular join tree $J_V$ of a given view definition $V$, denoted as $G^-(J_V)$. However, given $V$, there are $|V|$ 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 $J^*_{\text{opt}}$ with respect to its error produced by the estimation algorithm $\text{propagate}$.

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 $G^-(J_V)$, 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 $\text{propagate}$ with $G^-(J_V)$ and $m$ and comparing the result with a side-effect detection. However, determining the true error of $G(J_V)$ requires calling $\text{propagate}$ 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.

---

**Fig. 11. Procedure $\text{candidate_tuples}$**

<table>
<thead>
<tr>
<th>Procedure $\text{candidate_tuples}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> join tree $J_V$, negative graph $G^-$, the number of candidate sets $k$, the join attributes $A$ of $J_V$.</td>
</tr>
<tr>
<td><strong>Output:</strong> a set of candidate tuples</td>
</tr>
<tr>
<td><strong>01</strong> $(t_u, t_v).capacity = \infty \quad \forall \quad (t_u, t_v) \in G^- E$</td>
</tr>
<tr>
<td><strong>02</strong> for each $(t_u, t_v) \notin G^- E$</td>
</tr>
<tr>
<td><strong>03</strong> if $t_u \in R_u, t_v \in R_v \quad \text{and} \quad (R_u, R_v) \in J_V.E$</td>
</tr>
<tr>
<td><strong>04</strong> add $(t_u, t_v)$ into $G^-$</td>
</tr>
<tr>
<td><strong>05</strong> set $(t_u, t_v).capacity = \log(\max_{\text{prob}}(t_u, t_v))$</td>
</tr>
<tr>
<td><strong>06</strong> add a single sink and a single source to connect the sink(s) and source(s) of $J_V$, respectively</td>
</tr>
<tr>
<td><strong>07</strong> add a sink and a source to connect the $t$</td>
</tr>
<tr>
<td><strong>08</strong> set the capacity of the edges connecting to the sink or the source to infinity</td>
</tr>
<tr>
<td><strong>09</strong> $C = \emptyset$ and $G^* = G^-$ // initialization</td>
</tr>
<tr>
<td><strong>10</strong> while $</td>
</tr>
<tr>
<td><strong>11</strong> $g = \text{max}_\text{flow}(G^*)$</td>
</tr>
<tr>
<td><strong>12</strong> $G = \text{extend}(g)$</td>
</tr>
<tr>
<td><strong>13</strong> $C = C \cup {t \mid t \in g', g' \in G}$</td>
</tr>
<tr>
<td><strong>14</strong> for each $g' \in G \quad G = G^* - g'$</td>
</tr>
<tr>
<td><strong>15</strong> return $C$</td>
</tr>
</tbody>
</table>
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 dangerous 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., R1, FK referred to R0, PK denoted a join from R0 to R1) and the maximum tuple fan-out (e.g., fan-out of a tuple in R0 is the number of tuples in R1 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 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
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 \(\theta=0.4\). 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) and the estimation time of JCard 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
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 \( J\text{Card} \), we present the accumulated FP of 1,000 real-negative insertions in Fig. 16(a). 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 SYNTETICDB 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 \( J\text{Card} \), we vary the number of relations of views. Figs. 17(b)-(c) present FPs of insertions and deletions of our extended \( J\text{Card} \), 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 \( J\text{Cards} \) 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 \( x = 0 \), \( J\text{Cards} \) 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

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 \( J\text{Card} \) that consists of structures that summarize (extended) dangling tuples and the source of view tuples. \( J\text{Card} \) 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

[7] Y. Cui and J. Widom. Run-time translation of view tuple deletions using tuned to be accurate on...
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., \( KL(P_0(M, V)|P_e(M, V)) \to 0 \), the expected false positives of the detector tends to 0, i.e., \( EXP(E_+(M, V)) \to 0 \). □

**Proof:** Recall that \( M \) and \( V \) are the random variables of insertions and views, respectively; \( \theta \) 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:

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

\[
= \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} P(M, V) \times \left( f_{est}(M, V) + \theta \right)
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} P(M, V) \times \left( f_{est}(M, V) + \theta \right)
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} \frac{f_{est}(M, V) - f_{real}(M, V) + \frac{\theta}{\theta}}{\theta}
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} \frac{f_{est}(M, V) - f_{real}(M, V)}{\theta}
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} \frac{|CP(M, V)| \times P_e(M, V) - |CP(M, V)| \times P_D(M, V)}{\theta}
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} \frac{|CP(M, V)| \times P_e(M, V) - P_D(M, V)}{\theta}
\]

\[
\leq \frac{|CP|}{\theta} \times KL(P_D(M, V)||P_B(M, V))
\]

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.2 Proof of Proposition 4.2

**Proposition 4.2:** When KL divergence of the estimated distribution of insertions tends to 0, i.e., \( KL(P_0(M, V)|P_e(M, V)) \to 0 \), the expected false negatives of the detector tends to 0, i.e., \( EXP(E_-(M, V)) \to 0 \). □

**Proof:** The total number of false negatives is:

\[
EXP(E_-(M, V)) = \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} P(M, V) \times E_-(M, V)
\]

\[
= \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} P(M, V) \times \left( f_{est}(M, V) - f_{real}(M, V) \right)
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} P(M, V) \times \left( f_{est}(M, V) - f_{real}(M, V) \right)
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} \frac{f_{est}(M, V) - f_{real}(M, V)}{1 - \theta}
\]

\[
\leq \sum_{V \in \mathbb{V}} \sum_{M \in \mathbb{M}} \frac{|CP(M, V)| \times P_e(M, V) - P_D(M, V)}{1 - \theta}
\]

\[
\leq \frac{|CP|}{\theta} \times KL(P_D(M, V)||P_B(M, V))
\]

Therefore the expected error is 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 \( U \) and a set \( S \) of subsets of \( U \), we want to find a subset \( C \) of \( S \) such that (i) \(|C| \leq B\); (ii) the maximum number of elements in \( U \) 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 \( U \). (ii) We use selection predicates to define possible insertions to be exactly \( U \). (iii) A relation \( R_i \) is used to encode a subset \( C_i \) of MSC. (iv) A segment \( s_j \) of a view tuple to-be-inserted \( t_j \) that already exists in \( G^- \) (\( R_j \)) is used to encode an element \( u_{t_j} \) 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 \( U \), 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
insertion $t_i$; $(s_1^i, s_2^i, ..., s_n^i)$, we include a selection in the view definition, $\sigma_i$: $A_1 = s_1^i \land A_2 = s_2^i \land ... \land A_m = s_m^i$, 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 $R_1 \bowtie R_2 \bowtie ... \bowtie R_m$.

(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 $(s_1^i, s_2^i, ..., s_j^i, ...)$. We can construct the $G^-$ graph such that (iv.i) $s_1^i$ is new if $u_i \notin C_j$ and (iv.ii) $s_j^i$ exists in $G^-$, but not in the view, if $u_i \in C_j$. Moreover, for (iv.ii), $s_j^i$ is placed in some large equivalence class; and the class contains one tuple $s_j^{i'}$ joinable with some neighbouring tuples, e.g., $s_j^{i+1}$ and $s_j^{i-1}$. This forms the scenario in Fig. 19(b). Then we can always set a threshold $\sigma$ 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 $s_j^{i'} = s_1^{i_1} \cup s_2^{i_2} \cup ... \cup s_k^{i_k}$. Finally, to ensure that $s_j^{i'}$ is selected by our greedy algorithm, we add dummy tuples $D_j$ s.t. $s_j^{i'} \cup D_j$ 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 $s_j^{i'}$ from $G^-$. When either of the insertions $t_1, t_2, ..., t_i$ are encountered, the estimation algorithm will locate the additional path formed by $s_j^{i'}$, leads to a count at least 2.

Therefore, the SCT problem determines the least number of $s_j^{i'}$’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 $J_{\text{lineitem}}$ and $J_{\text{order}}$, on the simplified Q7 of TPCH. We generated 1800 random insertions, all with side effects. The threshold for side-effect errors $\theta$ 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 $J_{\text{lineitem}}$ and $J_{\text{order}}$ 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 $J_{\text{lineitem}}$ were 3.7 times smaller than those of $J_{\text{order}}$. 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 $J_{\text{lineitem}}$ and $J_{\text{order}}$ 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 $J_{\text{lineitem}}$ was 79. In comparison, Fig. 18(c) showed that $J_{\text{order}}$ did not estimate any side-effect free insertions. Therefore, $J_{\text{order}}$ 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 $C$ is 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 $C$. $C$ will form at least one embedding due to $c_s$’s. 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 $J_{\text{...}}$

This update of equivalence classes often needs relatively little computation in practice. Firstly, the existing segments
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 \texttt{propagate} estimates that \( t_v \) causes side effects, the splits and the edge weights are revoked. Otherwise, \texttt{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 \( G^+ \).

\section*{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.

\textbf{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.

\textit{Proof:} The proof can be established by deriving two small examples.

Suppose the estimator declares side effects when the count returned by Procedure \texttt{propagate} is greater than or equal to 1 + \( \theta \). 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 \texttt{propagate} uses \texttt{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 \texttt{propagate}, the count returned is 1 (due to the new embedding) \( 1 \times |E_{c_1,c_s}| \times 1 / |c_s| \), 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 + \( \theta \), where \( \theta \) is a user-defined threshold. Thus, there are false positives when \( |c_s| / |E_{c_1,c_s}| \leq 1 / \theta \).

(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 \texttt{propagate} returns \( 1 + 1 \times |E_{c_1,c_s}| \times 1 / |c_s| \). The estimator returns false negative when \( |c_s| / |E_{c_1,c_s}| > 1 / \theta \).

\textbf{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 \( c_2 \) 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 \( c_2^3 \).

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 \( |E_{c',c_3}| / \theta \). 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 \( \theta \). No improvement can be observed until a certain number of bisections are applied that led to a sharp increase of accuracies.

\section*{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.

\subsection*{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 \( \theta \) 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

<table>
<thead>
<tr>
<th>Query</th>
<th># of tables</th>
<th># of joins</th>
<th>FP</th>
<th>FN</th>
<th>est. time (ms)</th>
<th>defect. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>59.1</td>
<td>300.8</td>
</tr>
<tr>
<td>Q2</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>140.6</td>
<td>1096.6</td>
</tr>
<tr>
<td>Q3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>45.3</td>
<td>442.5</td>
</tr>
<tr>
<td>Q4</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>699.5</td>
<td>1910.8</td>
</tr>
<tr>
<td>Q5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1104.5</td>
<td>200</td>
</tr>
<tr>
<td>Q6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>716.1</td>
<td>2942.3</td>
</tr>
<tr>
<td>Q7</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1767.8</td>
<td>5325.9</td>
</tr>
<tr>
<td>Q8</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1087.1</td>
<td>2019.5</td>
</tr>
<tr>
<td>Q9</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>106.3</td>
<td>141</td>
</tr>
<tr>
<td>Q10</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>42.9</td>
<td>505.5</td>
</tr>
<tr>
<td>Q11</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>141.4</td>
<td>595.6</td>
</tr>
<tr>
<td>Q12</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>66.1</td>
<td>720.2</td>
</tr>
<tr>
<td>Q13</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>54.2</td>
<td>242.4</td>
</tr>
<tr>
<td>Q14</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>66.6</td>
<td>48.5</td>
</tr>
<tr>
<td>Q15</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>40.6</td>
<td>156.6</td>
</tr>
<tr>
<td>Q16</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>63.1</td>
<td>210.2</td>
</tr>
<tr>
<td>Q17</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>413.0</td>
<td>2234.1</td>
</tr>
<tr>
<td>Q18</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10.3</td>
<td>93.4</td>
</tr>
</tbody>
</table>

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.

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.

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 JCard’s performances of replacements on views with projections. 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)).

![Fig. 22. JCard’s error and runtime of replacements on views having projections on SYNTHETICDB](image-url)