Utilizing IDs to Accelerate Incremental View Maintenance[*]

Yannis Katsis
UC San Diego
ikatsis@cs.ucsd.edu

Kian Win Ong
UC San Diego
kianwin@cs.ucsd.edu

Yannis Papakonstantinou
UC San Diego
yannis@cs.ucsd.edu

Kevin Keliang Zhao
UC San Diego
kezhao@cs.ucsd.edu

ABSTRACT

Prior Incremental View Maintenance (IVM) algorithms specify the view tuples that need to be modified by computing diff sets, which we call *tuple-based diffs* since a diff set contains one diff tuple for each to-be-modified view tuple. *idIVM* assumes the base tables have keys and performs IVM by computing *ID-based diff* sets that compactly identify the to-be-modified tuples through their IDs.

This work makes the following contributions: (a) An ID-based IVM system for a large subset of SQL that includes the algebraic operators selection, join, grouping and aggregation, generalized projection involving functions, antisemijoin (and therefore negation/difference) and union. The system is based on a modular approach, allowing one to extend the supported language simply by adding one algebraic operator at-a-time, along with equations describing how ID-based changes are propagated through the operator. (b) An efficient algorithm that creates an IVM plan for a given view in four passes that are polynomial in the size of the view expression. (c) A formal analysis comparing the ID-based IVM algorithm to prior IVM approaches and analytically showing when one outperforms the other. (d) An experimental comparison of the ID-based IVM algorithm to prior IVM algorithms showing the superiority of the former in common use cases.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

materialized views; incremental view maintenance

1. INTRODUCTION

Materialized views are widely used to speed up query evaluation by storing the results of commonly asked queries. Being materialized, these views have to be brought up to date when the underlying data change. This is typically done through *Incremental View Maintenance (IVM)*. Abstracting out the details of different IVM approaches, a typical IVM algorithm takes as input three diff tables $\mathcal{D}_R^+$, $\mathcal{D}_R^-$ and $\mathcal{D}_R^u$ per base relation R, containing the tuples that were inserted, deleted and updated in R and computes the corresponding diff tables $\mathcal{D}_V^+$, $\mathcal{D}_V^-$ and $\mathcal{D}_V^u$ for the view V, containing the changes that have to be performed on V to bring it up to date.

In prior IVM work, each diff table $\mathcal{D}_V^+$, $\mathcal{D}_V^-$ and $\mathcal{D}_V^u$ contains one diff tuple for each view tuple that has to be inserted, deleted and updated, respectively. This is why we refer to such diffs as *tuple-based diffs* (in short *t-diffs*). In this work we show that if the base tables contain keys, one can represent the view modifications in a much more compact way through a novel type of diffs, called *ID-based diffs* (in short *i-diffs*), which identify the to-be-modified view tuples through their IDs. In contrast to t-diff tuples, a single i-diff tuple can represent modifications to *multiple* view tuples. This difference is crucial, as i-diffs are more efficient to compute than t-diffs, requiring in general fewer base table accesses as we will explain next. This leads to more efficient ID-based IVM algorithms, under common assumptions. The following example demonstrates the difference between t-diffs and i-diffs. To differentiate between the two types of diffs, in the rest of the paper we will be using the standard symbol Δ to refer to the newly introduced i-diffs and the symbol $\mathcal{D}$ to refer to traditional t-diffs.

```
devices(did, category)
parts(pid, price)
devices_parts(did, pid)
```

(a) Database schema

```
CREATE VIEW V AS
SELECT did, pid, price
FROM parts NATURAL JOIN
  devices_parts NATURAL JOIN
  devices
WHERE category = "phone"
```

(b) View definition

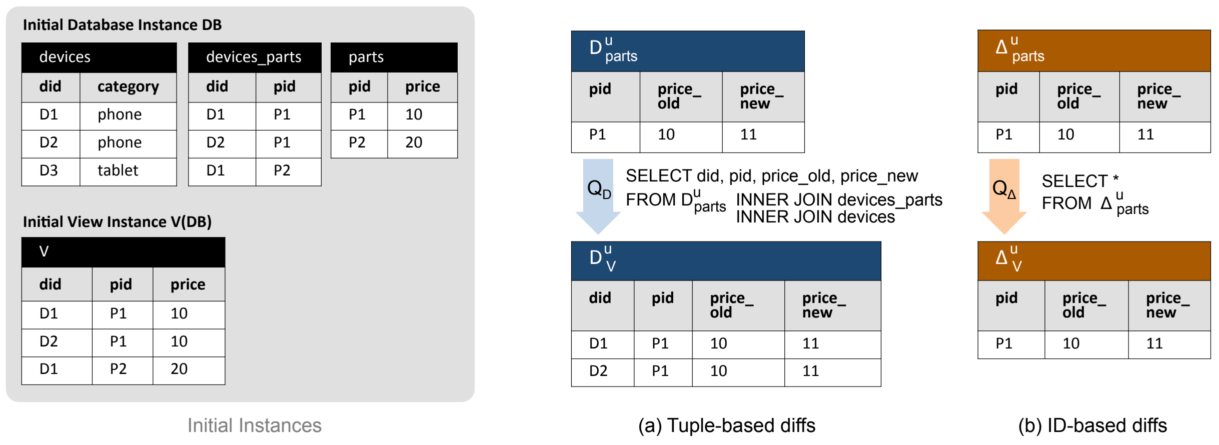
Figure 1: Database schema and view for running example

EXAMPLE 1.1. *Consider the database of an electronic device manufacturer, storing a list of devices and their parts. Figure 1a shows the respective database schema, consisting of the relations* ***devices***, ***parts*** *and* ***devices_parts****. The key attributes of each relation are shown underlined. Also consider the view* ***V*** *of Figure 1b returning the list of parts for devices of type 'phone'.*

Figure 2 shows an example of tuple-based and ID-based incremental maintenance of ***V****. The initial database and view instance are shown on the left and t-diffs and i-diffs for a sample change in relation* ***parts*** *on the right in Figures 2a and 2b, respectively. Consider the action of updating the price of part "P1" from $10 to $11. This modification is represented through identical t-diff and i-diff tuples, as seen in the t-diff table $\mathcal{D}^u_{parts}$ and i-diff table Δ^u_{parts}, respectively. However, this is no longer true when we look at the diffs that represent the resulting updates that have to happen to the view. Note that the change of the "P1" price in* ***parts*** *has to be propagated as updates of all "P1" tuples in the view (which in this case are the first two tuples). While the t-diff table $\mathcal{D}^u_V$ describes these changes by two diff tuples, each describing an entire view tu-*

[*]Supported by NSF III-1018961 and NSF III-1219263, PI'd by Prof. Papakonstantinou, who has commercial interest in this research. Also supported by NSF III-1117527.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2750546.



devices	
did	category
D1	phone
D2	phone
D3	tablet

devices_parts	
did	pid
D1	P1
D2	P1
D1	P2

parts	
pid	price
P1	10
P2	20

V		
did	pid	price
D1	P1	10
D2	P1	10
D1	P2	20

D^{u}_{parts}		
pid	price_old	price_new
P1	10	11

Q_D

```
SELECT did, pid, price_old, price_new
FROM D^u_parts INNER JOIN devices_parts
               INNER JOIN devices
```

D^{u}_{V}			
did	pid	price_old	price_new
D1	P1	10	11
D2	P1	10	11

Δ^{u}_{parts}		
pid	price_old	price_new
P1	10	11

Q_Δ

```
SELECT *
FROM Δ^u_parts
```

Δ^{u}_{V}		
pid	price_old	price_new
P1	10	11

Figure 2: Example of tuple-based and ID-based IVM

ple that has to be updated, the i-diff table Δ_V^u describes the same modifications through a single diff tuple (which intuitively instructs updating all "P1" tuples in the view).

Obviously ID-based diffs are more compact than their tuple-based counterparts. More importantly though i-diffs are in general more efficient to compute than t-diffs. Intuitively, the performance gains come from the fact that in contrast to t-diffs, i-diffs do not need to recreate the entire view tuples to be modified and thus can avoid some base table accesses. We use this observation to design an ID-based IVM algorithm, which is shown both analytically and experimentally to perform in most cases fewer base-table and view accesses than prior tuple-based approaches.

EXAMPLE 1.2. *Queries $Q_{\mathcal{D}}$ and $Q_\triangle$ show how the diffs for the view can be computed from the base tables and the diff for base relation **parts**. While computing the t-diff requires joining $\mathcal{D}^u_{parts}$ with the base tables **devices_parts** and **devices** (see $Q_{\mathcal{D}}$) to find all devices containing part "P1", producing the i-diffs simply amounts to finding the modified parts (and not the devices in which they are contained) and can therefore be accomplished by accessing only Δ^u_{parts} and avoiding all joins with the base relations (see $Q_\triangle$).*

Note that the fewer base table accesses of i-diff computations are not, just by themselves, an absolute proof of superior performance of the i-diffs, as maintaining the view should also count the cost of applying the i-diffs to the view. While both the ID-based and tuple-based approaches will have to join the resulting view diffs with the view, the ID-based approach has the drawback of potentially trying to join more diff tuples by creating *dummy* i-diff tuples, i.e., i-diff tuples describing changes for tuples that do not even exist in the view. For example, assume that **parts** included a tuple (P3, 20) and Δ^u_{parts} included a change of P3's price. Then Δ^u_V would include a dummy P3 tuple, i.e., the system would pay the price of attempting to update the P3's in V, albeit there would be no P3 in V. We call this effect *overestimation*. Nevertheless, our theoretical and experimental analysis show that under common circumstances the i-diff approach is indeed superior. Furthermore, we show that the advantage of ID-based IVM grows as the queries become more complex.

Contributions. This paper makes the following contributions:

(a) An ID-based IVM system, called *idIVM*, applicable when the base relations have primary keys. The *idIVM* is based on a modular, algebraic approach, allowing one to extend the supported view definition language simply by adding one relational algebra operator at-a-time and providing *i-diff propagation equations* describing how ID-based changes are propagated through it.

(b) A set of i-diff propagation equations for a large subset of SQL (denoted by Q_{SPJADU}) that includes the algebraic operators select, project, join, grouping and aggregation with associative functions, generalized projection involving functions, antisemijoin[1] and union. Although the framework can be easily extended to more expressive view definition languages as described above, our analytical and experimental results focus on Q_{SPJADU}, as it is expressive enough to cover a large number of practical use cases.

(c) An efficient 4-pass algorithm that creates an IVM plan for a given algebraically-expressed view and a given set of modification types in four passes that are polynomial in the size of the view expression: The first pass computes the IDs of intermediate results. The second pass instantiates the operator IVM equations to the specifics of the view's operators. The third pass composes individual equations into the queries of the IVM plan. Finally, the fourth pass applies minimization and other optimizations particular to the IVM problem. Unlike general purpose minimization the considered minimization is polynomial.

(d) An algorithm that given a view expression decides what types of i-diffs should be mined from the modification log or captured from triggers. The problem is non-trivial since, as we will see, the number of types of i-diffs that are applicable, given a base schema and a view schema, is exponential in the size of the schemas. The presented algorithm uses the view definition to decide the much smaller number of sufficient and efficient i-diffs.

(e) A formal analysis proving that for Q_{SPJADU} views in many use cases the ID-based IVM with the i-diff propagation equations described in the paper is more efficient than tuple-based IVMs. The analysis is based on a fine-grained cost model counting data accesses and includes a discussion under the specific conditions under which tuple-based IVMs can perform better.

(f) An experimental evaluation of the proposed IVM system for Q_{SPJADU} views indicating that in most cases it significantly outperforms traditional tuple-based approaches. The experimental results show speedups of 2 to more than 50 over tuple-based IVMs.

Note that ID-based IVM optimization is orthogonal and can be combined with many of the other IVM issues studied in the literature [12, 8], such as materialized view selection [3, 25, 19], self maintenance [5, 11] and compilation into code [2]. We briefly describe these prior IVM works and their synergies in Section 8.

Outline. The paper is structured as follows: Section 2 defines ID-based diffs (i-diffs). Section 3 presents the architecture of *idIVM*. It consists of two main parts: The first transforms base table mod-

[1]Therefore capturing queries with negation. The difference operator is a special case of antisemijoin.

ifications to i-diffs and the second, given a set of i-diffs, creates a DML script for maintaining the view. For ease of exposition, we present them in the reverse order, i.e. Section 4 describes the algorithm for the DML script generation and Section 5 describes the transformation of changes to i-diffs. Sections 6 and 7 compare analytically and experimentally the efficiency of the generated script to those produced by tuple-based IVM approaches. Finally, Sections 8 and 9 discuss related work and conclude the paper, respectively.

2. ID-BASED DIFFS

For the following discussion we consider a relational database DB whose base tables contain keys and a relational view $V(\bar{I}, \bar{A})$ over DB, containing a set of key attributes (which we will refer to as IDs) $\bar{I}$ and a set of non-key attributes $\bar{A}$.

EXAMPLE 2.1. *The view V of our running example contains IDs $\bar{I}$ = {did, pid} and non-ID attributes $\bar{A}$ = {price}. In the following we will be using the initial instance of V of Figure 2.*

View definition language. Although, as we will see, the framework can be easily extended to more expressive view definition languages, unless otherwise stated, we consider views from the language Q_{SPJADU}, which contains all SQL queries that can be formulated using the algebraic operators Selection, Projection (involving functions), Join (with arbitrary join conditions), Aggregation with associative functions sum, avg and count, Antisemijoin (and thus Difference), and Union[2].

General Structure of an i-diff. Let $V(\bar{I}, \bar{A})$ be a view with IDs $\bar{I}$ and non-ID attributes $\bar{A}$. An *ID-based diff* (in short *i-diff*) of type $t \in \{+, -, u\}$ for relation V is in its most general form a relation $\Delta^t_V(\bar{I}', \bar{A}'_{pre}, \bar{A}''_{post})$ satisfying the following properties:

- It contains a subset $\bar{I}'$ of the view's IDs $\bar{I}$. These are used to identify the tuples to be modified.
- It may contain two sets $\bar{A}'_{pre}$ and $\bar{A}''_{post}$ of attributes, such that $\bar{A}', \bar{A}''$ are sets of non-ID attributes of V. An attribute A_{pre} and A_{post} intuitively stores the pre-state value (i.e., initial value before the change) and respectively post-state value (i.e., new value after the change) of attribute A of V.

Depending on their type, i-diffs may not contain both pre-state and post-state attributes. In particular, insert i-diffs (i.e, of type $t = +$), do not contain pre-state attributes, since they represent insertions of tuples that did not exist before. Similarly, delete i-diffs (i.e., of type $t = -$) do not contain post-state attributes. Only update diffs (i.e., of type $t = u$) may contain both old and new attribute values. We next describe the semantics for each i-diff type:

Update i-diff. An update i-diff instance Δ^u_V for view $V(\bar{I}, \bar{A})$ is a relation instance with schema $\Delta^u_V(\bar{I}', \bar{A}'_{pre}, \bar{A}''_{post})$, where $\bar{I}'$ is a subset of the IDs $\bar{I}$ of V and $\bar{A}'$, $\bar{A}''$ are potentially different subsets of the non-ID attributes $\bar{A}$ of V (with $\bar{A}'$ being potentially the empty set).

Intuitively, each tuple $(\bar{i}', \bar{a}'_{pre}, \bar{a}''_{post})$ in Δ^u_V specifies that all tuples in V with values $\bar{i}'$ for their $\bar{I}'$ attributes should have the values of their $\bar{A}''$ attributes updated to $\bar{a}''_{post}$. Formally, applying Δ^u_V on an instance I_V of V is equivalent to applying the following DML statement on I_V:

APPLY Δ^u_V: UPDATE V
SET $\bar{A}''$ = $\bar{A}''_{post}$
FROM Δ^u_V
WHERE $V.\bar{I}' = \Delta^u_V.\bar{I}'$ [3]

In the rest of the paper, the instance I_V will be implied from the context and therefore for simplification we will simply refer to a diff as being applied on the view V.

Note, that although not affecting its semantics, an update i-diff may also contain pre-state values of some non-ID attributes of V. As we will see later, this additional information is leveraged by the IVM algorithm to reduce the number of accesses to the database.

EXAMPLE 2.2. *Applying the following update i-diff*

Δ^u_V	pid	price$_{pre}$	price$_{post}$
	P1	10	11

leads to the update of the price of both tuples in V with pid = "P1" from 10 to 11.

Remark. In the following we consider only i-diffs where $\bar{I}'$ forms a primary key of the i-diff. The reason is that if $\bar{I}'$ is not a key, then update i-diffs are not well-defined and insert i-diff applications may lead to primary key violations.

Insert i-diff. An insert i-diff instance Δ^+_V for a view $V(\bar{I}, \bar{A})$ is a relation instance with schema $\Delta^+_V(\bar{I}, \bar{A}_{post})$, or in other words a relation containing the post-state values for all attributes of the view and no pre-state values.

Intuitively, an insert i-diff instance Δ^+_V contains a set of tuples that should be inserted into V. Formally, applying Δ^+_V has the same effect as applying the following DML statement on V:

APPLY Δ^+_V: INSERT INTO V
SELECT $\bar{I}$, $\bar{A}_{post}$ AS $\bar{A}$ FROM Δ^+_V
WHERE ROW($\bar{I}$, $\bar{A}_{post}$) NOT IN
(SELECT $\bar{I}$, $\bar{A}$ FROM V)

EXAMPLE 2.3. *Applying the following insert i-diff*

Δ^+_V	did	pid	price$_{post}$
	D3	P2	20
	D4	P3	30

inserts tuples <D3, P2, 20> and <D4, P3, 30> in V.

Remark. The WHERE clause in the above DML statement ensures that an attempt is made to insert a tuple into V only if it is does not already exist in V in the exact same form. This allows multiple insert i-diffs to try to insert the same tuple.

Delete i-diff. A delete i-diff instance Δ^-_V for a relation $V(\bar{I}, \bar{A})$ is a relation instance with schema $\Delta^-_V(\bar{I}', \bar{A}'_{pre})$, where $\bar{I}'$ is a subset of the IDs $\bar{I}$ of V and $\bar{A}'$ is a potentially empty subset of the non-IDs $\bar{A}$ of V.

Intuitively, Δ^-_V specifies the tuples that should be deleted from V based on the values of the $\bar{I}'$ attributes. Formally, applying Δ^-_V has the same effect as applying the following DML statement on V:

APPLY Δ^-_V: DELETE FROM V
WHERE ROW($\bar{I}'$) IN (SELECT $\bar{I}'$ FROM Δ^-_V)

Note that, similarly to update i-diffs, a delete i-diff may also specify the pre-state values of the deleted tuples, which are used to create more efficient IVM solutions.

[2]To maintain the IDs for the bag union, we employ a special union all operator, outputting a special attribute **b**, denoting which child branch (b = 0/1 for left and right, resp.) a tuple came from.

[3]Note that for conciseness the UPDATE statement is written using PostgreSQL's special UPDATE FROM syntax. However, it could be equivalently written using standard SQL syntax.

EXAMPLE 2.4. *Applying the following delete i-diff*

Δ_V^-	pid	price$_{pre}$
	P1	10

leads to the deletion of both tuples with pid = "P1" from V.

Effective i-diff instances. Given a set of i-diff instances $\bar{\Delta}$ for a relation V, applying them on V leads in general to different results depending on the order of application. However, in this work we only look at sets of i-diffs where any order of applying them on V yields the same result. To this end, we define the notion of *effective* i-diff instances. Given the pre-state V^{pre} and post-state V^{post} of a relation V, an i-diff instance Δ_V^t is said to be *effective* w.r.t. V^{pre} and V^{post} if for each value of a tuple of V it reflects its final value. Formally, it is effective iff it satisfies the following properties:

- If Δ_V^t is an insert i-diff: Every tuple inserted by the i-diff exists in the post-state (i.e., $\Delta_V^+ \subseteq V^{post}$).
- If Δ_V^t is a delete i-diff over schema $\Delta_V^-(\bar{I}', \bar{A}'_{pre})$: Every tuple deleted by the i-diff does *not* exist in the post-state relational instance (i.e., $\pi_{\bar{I}'}\Delta_V^- \cap \pi_{\bar{I}'}V^{post} = \emptyset$).
- If Δ_V^t is an update i-diff over schema $\Delta_V^u(\bar{I}', \bar{A}'_{pre}, \bar{A}''_{post})$: Every tuple updated by Δ_V^u that exists in the post-state instance, has all updated attributes $\bar{A}''_{post}$ set to the corresponding values in that instance (i.e., $\pi_{\bar{I}',\bar{A}''_{post}}\Delta_V^u \ltimes_{\bar{I}'} V^{post} \subseteq \pi_{\bar{I}',\bar{A}''}V^{post}$).

It can be shown that a set of effective i-diffs lead to the same result regardless of the order in which they are applied. In the following the i-diff instances we consider are assumed to be effective. We will discuss in Sections 4 and 5 how *idIVM* makes sure that it always operates on effective i-diff instances.

i-diff schemas. It should be obvious that a single modification could be represented through i-diffs of different schemas. In particular, one can include pre-state or post-state values for different sets of attributes. More importantly, different base table i-diffs may lead to IVM solutions of different efficiencies. For instance, an update of a t tuple of relation $R(\bar{I}, A_1, A_2)$ on attribute A_1 can be represented by either an i-diff that contains only the post-state of A_1, or both the post-state of A_1 and A_2 (even though the value of A_2 did not change). However, the first i-diff will in general lead to a more efficient solution, since for the second i-diff the IVM algorithm will have to account also for the change of A_2, although this is not needed. This generates a novel challenge of selecting which base tables i-diff schemas to create, as explained next.

IDs and functional dependencies. As discussed earlier, the set $\bar{I}$ of ID attributes of a view V forms a key of that view. Moreover, any i-diff Δ_V for the view V identifies the tuples of V to be updated, deleted or inserted through a subset of that key. However, this cannot be an *arbitrary* subset of the key. The key $\bar{I}$ of a view is split into components $(\bar{I}_1, \bar{I}_2, \dots \bar{I}_n)$, such that for each component of the key there is a functional dependency $\bar{I}_i \rightarrow \bar{A}_i$ from this component $\bar{I}_i$ to a subset $\bar{A}_i$ of non-key attributes of the view. The i-diff can identify the tuples of the view to be modified through a subset of these key components.

EXAMPLE 2.5. *For instance, in our running example, V has ID/key $\bar{I} = \{did, pid\}$, which can be decomposed into two components $\bar{I}_1 = did$ and $\bar{I}_2 = pid$, since there is the functional dependency pid $\rightarrow$ price (in a more general view V' containing also attributes of the* **devices** *relation there would also be a functional dependency from did to those attributes). Thus the tuples of V can be identified by an i-diff either through did or through pid (as is the case in our example).*

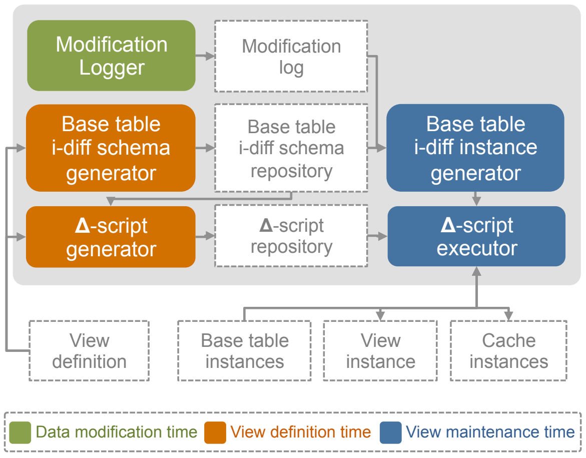


Figure 3: *idIVM* architecture

3. SYSTEM ARCHITECTURE

Figure 3 depicts the architecture of *idIVM*; an ID-based IVM system based on i-diffs and built on top of a relational DBMS. The modules of the system are shown as rounded boxes, while the system's data structures are depicted as white rectangles. *idIVM* can be setup to maintain the view either eagerly (i.e., whenever the base data change, known as *eager IVM* [6, 7, 10]) or lazily at some later point in time (known as *deferred* IVM [9, 15, 17]). In either case, the modification logging module of the *idIVM* remains the same. Furthermore, the only part of the architecture that is substantially different in the two approaches is the i-diff propagation rules and cache maintenance rules (see Figure 4). This paper describes the deferred IVM rules. *idIVM* contains modules executed at three different times (shown through color-coding in Figure 3): (a) when the views are defined (orange), (b) whenever the data in the underlying database change (green), and (c) whenever the views are maintained (blue). We present next briefly each of these stages:

View definition time. The most interesting and novel computations happen when a view is added to the system. At that point *idIVM* precomputes in the form of DML scripts how to translate i-diffs on the base tables to view updates. This computation happens through the synergy of two components: First, it employs a *base table i-diff schema generator* to decide which i-diff schemas to generate for the base tables. As we have discussed in Section 2, this is a non-trivial problem, as the same update could be modelled through i-diffs of different schemas. Once the base table i-diff schemas have been decided, *idIVM* invokes the Δ-*script generator* creating a DML script that accesses the generated i-diffs, the base tables and the potential caches (which as we will see can be used to speedup the IVM) to maintain the view. The resulting Δ-script is stored in a repository to be used at view maintenance time.

Data modification & view maintenance time. Given this offline computation, the system's online component is simple: When the base data are modified, a *modification logger* logs these changes for later use. When the time comes to maintain the view, the *base table i-diff instance generator* consults the modification log and converts it to instances of the base table i-diff schemas precomputed at view definition time. A Δ-*script executor* then retrieves the Δ-script corresponding to the view from the Δ-script repository and executes it to propagate the changes represented by the base table i-diff instances to the view instance.

We next describe the Δ-script generation, leaving the discussion of how to convert base table modifications to i-diffs for Section 5.

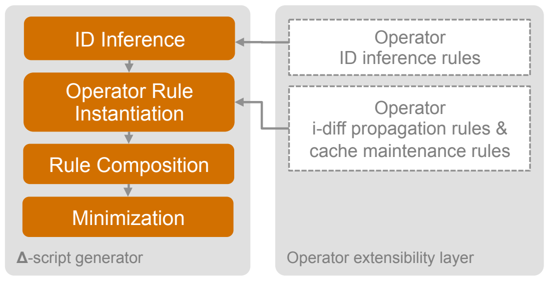
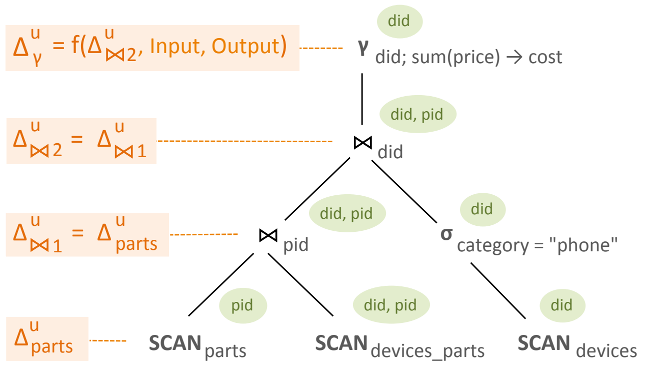

ID Inference
Operator Rule Instantiation
Rule Composition
Minimization
Δ-script generator
Operator ID inference rules
Operator i-diff propagation rules & cache maintenance rules
Operator extensibility layer

Figure 4: Δ-script generator architecture



(a) Algebraic plan (annotated by the Δ-script generator)

```
CREATE VIEW V’ AS
SELECT did, sum(price) AS cost
FROM parts NATURAL JOIN
     devices_parts NATURAL JOIN
     devices
WHERE category = "phone"
GROUP BY did
```

(b) View definition

Figure 5: View definition and plan for extended running example

4. Δ-SCRIPT GENERATION ALGORITHM

Given a view definition and a set of base table i-diff schemas, the Δ-script generation algorithm creates a DML script that includes (a) queries over the base table i-diffs, the base tables and the auxiliary caches (which as we will see are used by *idIVM* to speed up the IVM) that compute the corresponding view i-diffs and (b) UPDATE / INSERT / DELETE statements of the form described in Section 2 that apply these i-diffs on the view.

The Δ-script generator is based on the algebraic IVM approach [21, 10, 22]: Each relational operator type (e.g., selection, projection, join, etc.) is annotated with a set of *rules*, describing how to transform an (effective) i-diff over its input schema to an (effective) i-diff over its output schema. Given this information, the Δ-script for a view V can be composed from the individual rules for each operator that appears in an algebraic plan of V. Intuitively, the algorithm is computing how to maintain the entire view by first computing how to maintain all intermediate subviews in the algebraic plan. This approach enables a modular implementation in which the supported view definition language can be easily extended by adding rules for additional relational algebra operators. In this work we present the rules for all operators included in Q_{SPJADU} (i.e., selection, join, generalized projection involving functions, grouping with aggregation, antisemijoin, and union). Similarly to prior algebraic IVM approaches, we assume that the algebraic plan of the view on which the algorithm operates is given as input.

Operator	Output ID attributes
$SCAN(R)$	$key(R)$
$\sigma_\phi(R)$	$ID(R)$
$\pi_{\bar{D}}(R)$	$ID(R)$
$R \times S$	$ID(R) \cup ID(S)$
$R \bowtie_\phi S$	$ID(R) \cup ID(S)$
$R \bar{\ltimes}_\phi S$	$ID(R)$
bag union $R \cup S$	$ID(R) \cup ID(S) \cup \{b\}$
group by $\gamma_{\bar{G}, f(\bar{M})\dots}(R)$	$\bar{G}$

Table 1: Operator ID inference rules

EXAMPLE 4.1. *To showcase the algorithm, we extend the view of our running example to also perform an aggregation, returning the total cost of the parts for each device. Figures 5b and 5a show the view definition V' and a corresponding algebraic plan, respectively. The shaded components are annotations inserted by the algorithm, which we explain below.*

idIVM performs the 4 efficient passes of Figure 4.

Pass 1: Inferring ID information for intermediate views. Since i-diffs determine the view tuples that have to be modified through their IDs, the view and all intermediate subviews should contain as part of their output schema a set of ID attributes that form a key of the corresponding view. *idIVM* determines the ID attributes that should be contained in the output schema of each subview through the use of *ID inference rules*. An ID inference rule is supplied for each operator type supported by the system and describes how the IDs of the view rooted at an operator p can be computed from the IDs of the subviews rooted at p's children. Table 1 shows the ID inference rules for operators in Q_{SPJADU}.[4] *idIVM* uses these rules to perform a postorder traversal of the plan checking at each operator whether the IDs inferred by these rules exist in the operator's output schema. If this is not the case, *idIVM* automatically extends the plan to include the required ID attributes.

EXAMPLE 4.2. *Figure 5a shows the set of IDs for each operator in a shaded oval on the top right side of the operator.*

Note that extending the view with additional ID attributes simply increases the width of the view instance (i.e., the number of columns) but does not affect its cardinality (i.e., the number of tuples). In particular, if V_{orig} is an original view with attributes $\bar{A}$ and V_{ID} is the view inferred by the ID-inference algorithm, then for all instances of the base tables the original view can be computed from the original view simply by projecting out the additional attributes introduced by the ID-inference algorithm (i.e., $V_{orig} = \pi_{\bar{A}} V_{ID}$). Given that the number of additional ID attributes is usually small compared to the number of attributes already in the view, we do not expect the extension of the view schema with ID attributes to significantly affect the query evaluation performance. Importantly, the above observations hold not only for operators in Q_{SPJADU} but *for any* SQL operator (for the reader wondering how this can be the case for the duplicate elimination operator δ, given that in general $\delta(\pi_{A,B}(R))$ is different from $\delta(\pi_{ID,A,B}(R))$, consider an implementation where the duplicate elimination operator δ above is replaced by the group by operator $\gamma_{A,B}$).

Pass 2: Instantiating rules for each intermediate operator. To construct the Δ-script for the view, the algorithm employs operator rules that describe how each operator can propagate i-diffs over its input to i-diffs over its output.

Operator rules. An operator describes how to transform an i-diff Δ^t_{input} over one of its input schemas to an i-diff Δ^t_{output} over its

[4] Union refers to the *union all* operator described in Section 2.

output schema through a set of queries known as *i-diff propagation rules*. These queries can access (a) the operator's input i-diff Δ^t_{input} and (b) the data corresponding to the subview rooted at the operator or at one of its child operators. The latter is the way in which *idIVM* allows operator rules to access data from the base tables. Since an operator does not have knowledge of the exact place in the query plan where it appears to ask for a query result over the base tables, it can access the base table data only indirectly by asking for the subview rooted at one of its children through the use of the $Input_{i=l,r}$ (standing for left and right input, resp. for binary operators) or for the subview rooted at itself using the $Output$ keyword, respectively. The input subviews can be requested either in their pre-state form (i.e., using the instances of the base tables before the diffs were applied to them) or in the post-state (i.e., using the final instances of the base tables after the diffs were applied to them). An i-diff propagation rule can specify which of the two versions of the input it needs through the subscripts pre and $post$. The output is always provided in pre-state.

EXAMPLE 4.3. *For instance, a general grouping and aggregate operator* $V = \gamma_{\bar{G},f(\bar{X})\rightarrow c}(Input)$ *contains among others the i-diff propagation rule:* $\Delta^u_V = \gamma_{\bar{G},f(\bar{X})\rightarrow c}(Input_{post} \ltimes_{\bar{G}} \Delta^+_{Input})$, *which semijoins the post-state of the subview rooted at the operator's child with an input insert i-diff to find all tuples that belong to groups affected by the insertions and use them to recompute the value of the aggregate function for these groups. The* $Input_{post}$ *keyword is the way in which the operator asks for the (post-state of) some base data (in this case the base data defined by the subview rooted at the operator's child, which for the aggregate operator of Figure 5a is the subview* ***parts*** $\bowtie_{pid}$ ***devices_parts*** $\bowtie_{did}$ $\sigma_{category=\text{"phone"}}$ ***devices****).*

There are two different classes of operators in *idIVM*: The first consists of operators which can produce an output effective i-diff by looking at one input i-diff at a time. These operators are called *non-blocking* operators, in contrast to *blocking* operators which need to inspect the entire set of input i-diffs before creating an effective output i-diff. The operator type affects how the operator's i-diff propagation rules are expressed. For non-blocking operators, each rule is expressed over a single input i-diff, while for blocking operators, a rule is expressed over all input i-diffs.

EXAMPLE 4.4. *The general aggregate operator* γ *of Example 4.3 is a non-blocking operator, since it can decide how to propagate an input insert i-diff without looking at other input i-diffs (e.g., delete or update i-diffs). The reason is that for each insert i-diff tuple it recomputes the entire affected group from the base data thus reflecting indirectly also the changes incurred by other input i-diffs. On the other hand, imagine a more efficient aggregate operator designed specifically for the SUM aggregate function. This operator avoids recomputing entire groups by combining all input i-diffs to figure out the amount by which the aggregate value of each group has changed. While it avoids some base table accesses, it requires knowledge of all input i-diffs and is thus a blocking operator.*

Tables 4-13 show the i-diff propagation rules for the operators considered in this work, including join, union, generalized projection involving functions, antisemijoin and aggregation. Rules for aggregation are provided in four different versions (see Tables 7, 9, 11, and 12); one for general aggregation functions and others for specialized functions, such as SUM, COUNT and AVG.

Some operator rules may also benefit from special caches to speed up IVM. For instance, an aggregate AVG operator in the presence of a COUNT and SUM cache can incrementally maintain its output without accessing the base tables. To accomodate such cases, *idIVM* allows operators to declare special *operator caches* and associated *cache maintenance rules*, describing how to compute the i-diffs that maintain the caches. The i-diff propagation rules can then be expressed also over the operator cache schemas and the operator cache i-diffs. Table 12 shows the cache maintenance rules and i-diff propagation rules for the AVG operator.

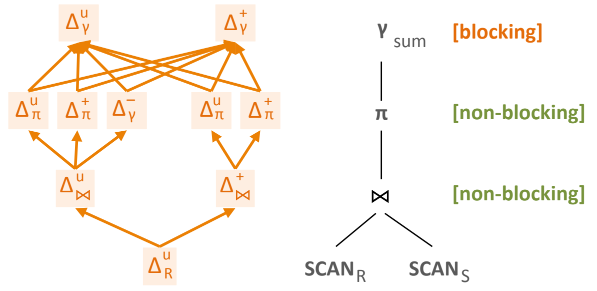


Figure 6: Rule DAG structure

Rule instantiation. In its second pass, the Δ-script generator algorithm employs the predefined operator rules to compute how each base table i-diff is propagated from operator to operator in the view plan. This is done as follows: For each base table i-diff schema Δ^t_R, the algorithm starts from the scan operator of the corresponding base table R and in a bottom-up fashion instantiates the rules for all operators in the path from the scan operator to the root of the plan.[5] The rule instantiation simply consists in selecting from all rules for the particular operator the ones that apply in the particular case (based on the input i-diff schema and other conditions) and replacing the abstract schema used in the rules with the concrete schema of the particular operator instance (e.g., for an operator $V = \pi_x(R)$ the general projection i-diff propagation rule $\Delta^+_V = \pi_{\bar{a},f(\bar{X})\rightarrow c}\Delta^+_R$ becomes $\Delta^+_V = \pi_x\Delta^+_R$).

EXAMPLE 4.5. *Consider an update i-diff schema* $\Delta^u_{parts}(pid, price_{pre}, price_{post})$ *modeling updates on the price attribute of table* ***parts*** *of our running example. Figure 5a shows on the left the corresponding instantiated rules generated by the algorithm. The exact rule for the aggregate operator is omitted due to lack of space. However, it is important to note that it is a rule that mentions the input i-diff and the input and the output of the operator, respectively.*

Note that for a single input i-diff an operator may create multiple output i-diffs. For instance, an update i-diff going through a selection operator may lead to insert, update and delete i-diffs, depending on whether a tuple satisfied the condition before and after the change. Whenever the rules of an operator create multiple output i-diffs, the above computation continues conceptually in parallel for each generated i-diff schema. This leads to a directed *rule DAG*, whose nodes are instantiated rules and whose edges point from a rule to all rules that were created using its output schema. Figure 6 shows such a structure. Note how blocking rules convert the structure that would otherwise be a tree into a DAG.

Pass 3: Composing operator-level instantiated rules into a Δ-script. Each rule in the DAG is a query expressed over the output schema of its parent rules (note that the DAG in Figure 6 is shown inverted with its root shown at the bottom). Thus each i-diff for the view (which corresponds to a leaf) can be computed by composing the instantiated rules of its ancestors. The exact order in which these compositions are performed does not matter, since all considered i-diffs are effective. This is guaranteed by the fact that (a) the base table diff instance generator creates effective diffs (as we will discuss in Section 5) and (b) i-diff propagation rules transform effective input i-diffs to effective output i-diffs.

[5]If the base table R appears with multiple aliases, this process is repeated for every scan operator of R.

```
$\Delta^u_{Cache} = \Delta^u_{parts}$;
APPLY $\Delta^u_{Cache}$;
$\Delta^u_{V'} = \pi_{did,cost\rightarrow cost_{pre},cost+cost_\Delta\rightarrow cost_{post}}(V' \bowtie$
        $\gamma_{did,sum(price_\Delta)\rightarrow cost_\Delta}($
        $\pi_{did,pid,(price_{post}-price_{in})\rightarrow price_\Delta}($
        $\Delta^u_{Cache} \bowtie \pi_{price\rightarrow price_{in}} Cache)))$;
APPLY $\Delta^u_{V'}$;
```

Figure 7: Δ-script for running example

To make the generated plan more efficient, *idIVM* employs also additional caching, other than the caching used internally by operators. In particular, for aggregate operators, whose rules typically ask for the base data corresponding to their input/output (through the $Input_i$ and $Output$ keywords, resp.), *idIVM* attempts[6] to create an *intermediate* cache in which it materializes this result. This cache is treated as any other view and maintained during the IVM process. In particular, *idIVM* first composes all rules that create the i-diffs for the cache and then using them as input, composes the rest of the rules up to the next cache until it reaches the view.

EXAMPLE 4.6. *For instance, as we have seen in Figure 5a, the instantiated rule for the aggregation mentions both the input and the output of the operator. Thus,* idIVM *tries to generate two intermediate caches; one before the aggregate and another after the aggregate. Since however the output of the aggregate coincides with the view (which is already materialized),* idIVM *creates only the first cache and utilizes the already existing view as the second.*

The result of this composition is a Δ-script, containing queries that compute i-diffs for an intermediate cache/view and APPLY operators that use the DML statements corresponding to each i-diffs type (shown in Section 2) to apply these i-diffs to the cache/view.

EXAMPLE 4.7. *In our running example* idIVM *employs an intermediate cache below the aggregate operator. Thus, it composes the rules up to that point, updates the cache and then uses it as input to compose the rules up to the view, which is subsequently updated. This leads to the Δ-script of Figure 7.*

Pass 4: Optimizing the generated Δ-script. As a last step, *idIVM* optimizes the Δ-script by performing semantic optimization, which minimizes each individual query included in the plan. In contrast to general minimization, this minimization takes into account the special semantics of i-diff tables. As described in Section 2, given a base table $R(\bar{I}, \bar{A})$ in its post-state and i-diffs $\Delta^+_R(\bar{I}, \bar{A}_{post})$, $\Delta^-_R(\bar{I}, \bar{A}'_{pre})$, and $\Delta^u_R(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$ over this table, the following constraints hold: (a) $C_1 : \Delta^+_R \subseteq R$, (b) $C_2 : \pi_{\bar{I}}\Delta^-_R \cap \pi_{\bar{I}}R = \emptyset$, and (c) $C_3 : \pi_{\bar{I},\bar{A}''_{post}}\Delta^u_R \ltimes_{\bar{I}} R \subseteq \pi_{\bar{I},\bar{A}''}R$. *idIVM* minimizes w.r.t. constraints C_1 - C_3 by employing on top of the standard relational rewrite rules also the rewrite rules presented in Figure 8. Semantic minimization is crucial in eliminating inefficiencies introduced by composing individual operator rules, improving in some cases performance by more than 50%.

Designing operator i-diff propagation rules. The efficiency of the Δ-script obviously depends on the provided i-diff propagation rule definitions. Reasoning about the efficiency of individual rules is hard, as rules affect each other (e.g., a rule avoiding base table accesses may not bring in some information that could be used by rules later in the plan, thus leading to higher access cost later).

[6]Intermediate caches are not generated when they are expected to contain multi-valued dependencies (for instance due to a many-to-many join), since in that case reading the result from the cache would incur more tuple accesses than simply recomputing it on the fly from the base tables. *idIVM* exploits foreign key constraints to infer the absence of multi-valued dependencies.

For semijoin
$\Delta^+_R \ltimes_{R.\bar{I}} \sigma_{\phi(\bar{X})} R \rightarrow \sigma_{\phi(\bar{X}_{post})}\Delta^+_R$
$R \ltimes_{R.\bar{I}} \sigma_{\phi(\bar{Y})} \Delta^+_R \rightarrow \pi_{\bar{I},\bar{A}_{post}\rightarrow\bar{A}}\sigma_{\phi(\bar{Y})}\Delta^+_R$
$\Delta^u_R \ltimes_{R.\bar{I}} \sigma_{\phi(\bar{X})} R \rightarrow \sigma_{\phi(\bar{X}_{post})}\Delta^u_R$, if $\bar{X} \subseteq \bar{A}''$
$R \ltimes_{R.\bar{I}} \sigma_{\phi(\bar{Y})} \Delta^u_R \rightarrow \pi_{\bar{I},\bar{A}''_{post}\rightarrow\bar{A}}\sigma_{\phi(\bar{Y})}\Delta^u_R$, if $\bar{A}'' = \bar{A}$
$\Delta^-_R \ltimes_{R.\bar{I}} \sigma_{\phi(\bar{X})} R \rightarrow \emptyset$
$R \ltimes_{R.\bar{I}} \sigma_{\phi(\bar{X})} \Delta^-_R \rightarrow \emptyset$

For antisemijoin
$\Delta^+_R \overline{\ltimes}_{R.\bar{I}}\sigma_{\phi(\bar{X})} R \rightarrow \sigma_{\neg\phi(\bar{X}_{post})}\Delta^+_R$
$\Delta^u_R \overline{\ltimes}_{R.\bar{I}}\sigma_{\phi(\bar{X})} R \rightarrow \sigma_{\neg\phi(\bar{X}_{post})}\Delta^u_R$, if $\bar{X} \subseteq \bar{A}''$
$\Delta^-_R \overline{\ltimes}_{R.\bar{I}}\sigma_{\phi(\bar{X})} R \rightarrow \Delta^-_R$
$R \overline{\ltimes}_{R.\bar{I}}\sigma_{\phi(\bar{X})}\Delta^-_R \rightarrow R$

For join
$\Delta^+_R \bowtie_{R.\bar{I}} R \rightarrow \Delta^+_R$
$\Delta^u_R \bowtie_{R.\bar{I}} R \rightarrow \Delta^u_R$
$\Delta^-_R \bowtie_{R.\bar{I}} R \rightarrow \emptyset$
* *up to renaming*

Figure 8: Rewrite rules for semantic optimization

However, in this work we show that we do not have to get into this reasoning process. Simply creating rules that individually avoid accessing the base tables when possible leads to efficient Δ-scripts, as shown by our analytical and experimental results. To avoid data accesses, the rules are even allowed to *overestimate*, i.e. skip some filtering that would require base table accesses and propagate to their output i-diffs dummy tuples that do not affect the view.

EXAMPLE 4.8. *For instance, the selection operator allows a delete input i-diff to pass through the operator unmodified. However, this means that the output i-diff will also instruct the deletion of tuples that do not satisfy the selection conditions and thus do not exist in the view. Although this is an overestimated i-diff, it does not affect the correctness of the generated Δ-script, since the latter will simply try to delete some tuples from the view that do not exist. On the other hand, this rule locally minimizes the base table accesses, as it avoids accessing the base tables to filter out the tuples that do not satisfy the selection condition.*

5. FROM MODIFICATIONS TO I-DIFFS

We saw above how given a set of base table i-diffs, *idIVM* maintains the view. In this section we explain how these base table i-diffs are generated from base table modifications. This is a non-trivial problem, since as explained in Section 2, a single modification can be represented through i-diffs of different schemas, each leading potentially to Δ-scripts of different efficiencies.

idIVM solves the i-diff generation problem through the synergy of three components shown in Figure 3: (a) a modification logger recording the base table modifications at data modification time, (b) a base table i-diff *schema* generator deciding at view definition time which base table i-diff schemas to generate, and (c) a base table i-diff *instance* generator, translating at view maintenance time the modifications recorded in the log to instances of the pre-computed i-diff schemas. Logging changes to the base tables can be easily performed through known techniques, such as DBMS log inspections, timestamp queries or triggers (currently used by *idIVM*). Therefore we focus next on the other two components.

Generating i-diff schemas. Given a view definition V, *idIVM* generates suitable base-table i-diff schemas for all base tables mentioned in V. Insertions and deletions are straightforward cases: Consider a base table $R(\bar{I}, \bar{A})$ with key attributes $\bar{I}$ and non-key attributes $\bar{A}$. For each such table, the i-diff schema generator creates a single insert i-diff schema $\Delta^+_R(\bar{I}, \bar{A}_{post})$ (containing all attributes of R) and a single delete i-diff schema $\Delta^-_R(\bar{I}, \bar{A}_{pre})$ (containing all non-ID attributes of R in pre-state form). This is based on the observation that pre-state values can lead only to a more efficient Δ-script as they may reduce overestimation and the respective view index lookups. For instance, as shown in Table 6 with blue, a selection operator can exploit pre-state attributes to filter out the tuples of an incoming delete i-diff that do not satisfy the condition.

The same does not hold though for post-state attributes included in update i-diffs. Including more post-state attributes in an update i-diff schema leads to a generally less efficient Δ-script, as it has to account also for changes in these attributes. Creating one update i-diff schema for each subset of attributes of each base table is obviously not an option, due to the exponentiality involved.

In *idIVM* we solve this problem by observing that the base table attributes can be divided into sets of attributes whose updates lead to the same Δ-script and can thus be grouped together in a single i-diff schema. For each operator *op* in the algebraic view plan, let C_{op}, be the set of (non-key) base table attributes involved in any condition (e.g., selection, join etc)[7]. We refer to C_{op} as the *set of conditional attributes for op*. The set of (non-key) base table attribute not included in any C_{op} for any operator *op* in the view's plan is referred to as the *set of non-conditional attributes* NC. Non-conditional attributes may still affect the view (since they could be included in the view's output), but intuitively they do not affect the generated Δ-script (up to projections). Updates on each set of conditional attributes C_{op} on the other hand may lead to a different Δ-script, since the updated values may affect whether the i-diffs make it past *op*'s condition. Therefore for each base table $R(\bar{I}, \bar{A})$, the i-diff schema generation algorithm creates (a) for each set C_{op} an update i-diff $\Delta^u_R(\bar{I}, \bar{A}_{pre}, \bar{A}'_{post})$, s.t. $\bar{A}' = \bar{A} \cap C_{op}$ and (b) an additional update i-diff $\Delta^u_R(\bar{I}, \bar{A}_{pre}, \bar{A}''_{post})$, containing the non-conditional attributes of R (i.e., $\bar{A}'' = \bar{A} \cap NC$).

Populating i-diff instances. Every time *idIVM* is invoked to maintain the view, the i-diff instance generator simply populates the i-diff tables created at view definition time. This is done by extracting the changes since the last view maintenance from the modification log and adding them as diff-tuples to all i-diff tables that contain at least one of the modified attributes (in the case of updates) and to the single insert and delete i-diff tables (in the case of inserts and deletes, respectively). Note, that when extracting the modifications from the log, the algorithm combines multiple modifications to the same tuple to a single modification, so as to generate *effective* diffs. As discussed in Sections 2 and 4, this is crucial for the algorithm's correctness.

6. PERFORMANCE ANALYSIS

We next analytically compute the speedup ratio of ID-based over tuple-based IVM (i.e., the ratio $\frac{\text{tuple-based cost}}{\text{ID-based cost}}$). A speedup ratio greater than 1 signifies that the ID-based approach has a lower cost than the tuple-based approach and thus is more efficient than the latter, while a speedup ratio lower than 1 signifies the opposite.

To compute the speedup we first compute the individual cost of each IVM approach. The cost of the ID-based/tuple-based approach is measured in the combined number of tuple accesses and index lookups incurred by the Δ/$\mathcal{D}$-script, generated by the corresponding approach. For the purposes of this analysis, we assume that both approaches have access to view indices on the view IDs and additionally the tuple-based IVM has access to appropriate base table indices (which are not required by the ID-based approach). We also try to be as general as possible regarding the query plan that the DBMS might choose to execute a particular Δ/$\mathcal{D}$-script. However, since DBMSs employ complex optimizations that cannot be comprehensively accounted for in an analytical model, the computed cost and the associated speedups reported in this Section should only be used as rough estimates of the actual cost of performing IVM that illustrate the difference in performance between the two approaches. For an experimental comparison of the two IVM approaches, please refer to Section 7.

We next present the speedup for two representative cases: (a) SPJ views, which by default do not involve intermediate caches and (b) Aggregate views involving grouping and associative functions, which (by default) are supported by caches. For a detailed analysis explaining how this speedup was computed, please refer to Appendix A.

6.1 SPJ Views

Consider the SPJ view V_{spj}:

$$\texttt{SELECT } \bar{S} \texttt{ FROM } R, R_1, \ldots, R_n \texttt{ WHERE } c$$

whose FROM clause involves a single alias of a table R, (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R. [8]

Parameters affecting speedup. The speedup of the ID-based approach over the tuple-based approach can be expressed in terms of two parameters: the *i-diff compression factor* p and the *tuple-based computation cost per base table diff tuple* a. The i-diff compression factor $p = |\mathcal{D}_{V_{\mathrm{spj}}}|/|\Delta_{V_{\mathrm{spj}}}|$ is the ratio of the size of the tuple-based diff to the size of the ID-based diff for the view. p may be less than 1 (when i-diffs summarize the modifications to the view in a more compact way than t-diffs, as shown in Figure 2) but may also be greater than 1 (when i-diffs are overestimating and trying to modify tuples that do not exist in V_{spj}). The second parameter is the number of accesses a that the tuple-based approach has to perform on average to compute the t-diff tuples for the view that result from a given t-diff tuple for the base table. This cost will typically vary, depending on the plan chosen by the DBMS to evaluate the tuple-based $\mathcal{D}$-script.

Speedup ratio. The speedup ratio of the ID-based approach over the tuple-based approach is given by the following formula:

(a) if Δ_R/$\mathcal{D}_R$ is an update i-diff/t-diff on attributes of R that are not involved in selection or join conditions in V_{spj}, then

$$\text{A: Speedup ratio} = \tfrac{a+2p}{1+p}$$

(b) else (i.e., if Δ_R/$\mathcal{D}_R$ is any other update i-diff/t-diff or it is an insert or delete i-diff/t-diff)

$$\text{B: Speedup ratio} \geq min\left(\tfrac{a+2p}{1+p}, 1\right)$$

Discussion. Let us first explain why update diffs on attributes of R that are non-conditional may lead to a different speedup ratio than other types of diffs. Since the updates do not affect how a tuple behaves w.r.t. selections or joins, they are guaranteed to lead to updates (i.e., neither inserts, nor deletes) on the view. When this is the case (case (a) above), the ID-based IVM algorithm can simply propagate the base table i-diffs to the view without accessing the base tables, leading to a speedup ratio of $\frac{a+2p}{1+p}$. This speedup is in most practical cases greater than 1 (meaning that the ID-based is more efficient than the tuple-based approach). For the tuple-based approach to perform better, it should be the case that $a < 1 - p$, which can be satisfied only in the corner case when the following conditions simultaneously hold: (a) the tuple-based approach incurs $a < 1$ tuple accesses for each tuple in $\mathcal{D}^u_R$ on average (which can only happen if many tuples of $\mathcal{D}^u_R$ share the same join attribute values and thus the joined tuples can be retrieved once and reused for all of them) and (b) the ID-based approach is severely overestimating (i.e., $p \ll 1$). We have experimentally verified that by following this pattern it is possible to create contrived scenarios in which the tuple-based IVM outperforms the ID-based approach.

[7]Base table key attributes do not need to be considered for updates as they are immutable.

[8]Recall that we use the symbols Δ and $\mathcal{D}$ to represent i-diffs and t-diffs, respectively.

However, in all other cases, the ID-based approach performs better (with a difference that raises proportionally to p).

When the base table diffs are insert or delete diffs or they are updates on attributes involved in conditions (case (b) above), then they will lead in general to updates, inserts and/or deletes on the view. If they lead to updates and deletes only, then the resulting speedup is the same as in the first case (i.e., $\frac{a+2p}{1+p}$) and thus the ID-based approach is expected to perform better in most cases. However, if they lead to inserts, then the two approaches will perform identically and hence exhibit a speedup of 1. Finally, if the base table diffs lead to a combination of updates, deletes and inserts, then the speedup will be a linear combination of the above two speedups and thus will be greater than the smaller of the two (hence the use of inequality and the min function in the above formula).

Thus for SPJ queries the ID-based IVM will always (up to the corner case described above) perform at least as good as the tuple-based IVM and in most cases better then the latter. The two approaches will only perform identically if the IVM workload is heavy on modifications that lead to insertions to the view.

6.2 Aggregate Views

Consider the aggregate view V_{agg}:

SELECT $\bar{G}, f(\bar{X})$ AS g FROM $R, R_1, \ldots, R_n$
WHERE c GROUP BY $\bar{G}$

whose FROM clause involves a single alias of R, and f is an associative aggregation function such as sum (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R.

To ease exposition, we isolate the aggregation operator of the query, expressing it through the plan $V_{\text{agg}} = \gamma_{\bar{G}, f(\bar{X}) \to g} V_{\text{spj}}$, where V_{spj} is the plan for the SPJ query presented in Section 6.1. We study the interesting case, where the ID-based IVM has identified that an intermediate cache storing the input of the aggregate operator, which is the result of the SPJ query, is beneficial (since without cache both approaches would perform identically). The tuple-based approach does not use a cache, since it cannot benefit from it.

Both approaches operate in two two stages: They first compute the diff to maintain the SPJ subview V_{spj} and then use it to maintain the final aggregate view V_{agg}. The second step is the same in both cases. Thus the difference in performance comes from computing the diff for the V_{spj}, which in the case of the ID-based approach is also used to maintain the cache. Let a and p be defined as above for the subview V_{spj} (i.e., let $p = |\mathcal{D}_{V_{\text{spj}}}|/|\Delta_{V_{\text{spj}}}|$ and let a be the average cost incurred by the tuple-based diff to compute for each base table diff tuple the corresponding diff tuples for the view V_{spj}).

Speedup ratio. The speedup ratio of the ID-based approach over the tuple-based approach is given by the following formula:

(a) if $\Delta_R/\mathcal{D}_R$ is an update i-diff/t-diff on non-conditional attributes of R, then

$$\text{Speedup ratio} = \frac{a+x}{1+p+x}$$

(b) else

$$\text{Speedup ratio} \geq min\left(\frac{a+x}{1+p+x}, \frac{a+x}{a+k+x}\right)$$

where x is a cost related to grouping that is shared by both approaches and thus can be safely ignored for the purposes of our discussion and k is a parameter concerning insert diffs that we will explain later.

Discussion. Similarly to SPJ views, we differentiate between cases where the base table diffs lead to update or delete diffs on the view V_{spj} and cases where they lead to insert diffs on V_{spj}.

In the first case the speedup ratio is $s_1 = \frac{a+x}{1+p+x}$. This speedup is always going to be at least 1, meaning that the tuple-based approach

Relation	Tuples
User	1M
FriendList	100M
Microblog (i.e. tweets)	20M
Retweets	4M (#tweets $\times$ 10% $\times$ 2 retweets per tweet)
Mentions	8M (#tweets $\times$ 20% $\times$ 2 mentions per tweet)
Rel_Event_Microblog	16M (#tweets $\times$ 40% $\times$ 2 events per tweet)

(a) BSMA relation sizes

Query	Description
Q7	Mentioned users within a time range
Q10	Users who are retweeted within a time range
Q11	Pair of retweeting users, grouped by retweeting times
Q15	Users talking about events within a time range
Q18	Pairwise count of mentions
Q*1	Aggregate of friends of friends within the same city
Q*2	Aggregate of retweeters for every user
Q*3	Aggregate of users who tweet about topics

(b) BSMA queries and additional Queries

Figure 9: Configuration of social analytics experiments

can never perform better than the ID-based approach. This happens because the cost a incurred by the tuple-based approach for each diff tuple in $\mathcal{D}_R$ is at least $1 + p$, since for each such tuple it will have to incur at least one index access (to find the tuple of the other relations it joins with) and p tuple accesses (to read the tuples it joins with to create the corresponding p tuples in the view). Note that these are lower bounds that apply when the view V_{spj} contains only one join. If it contains more joins, the speedup ratio and thus the performance benefit of ID-based IVM will be even higher.

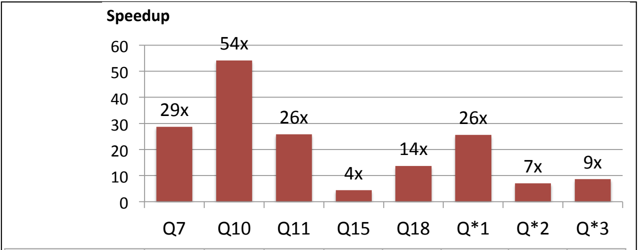
On the other hand when base table diffs lead to insert diffs on the view V_{spj}, the ID-based approach will be performing the same plan with the tuple-based approach but will also be inserting tuples into the cache. Thus if k is the number of tuples created in V_{spj} on average as a result of a single diff in $\mathcal{D}_R$, then the speedup will be $s_2 = \frac{a+x}{a+k+x}$. This speedup is less than 1 (meaning that the tuple-based approach will be performing better), but now the loss is bounded, as it is always 1 per tuple inserted into V_{spj}.

Similarly to SPJ views, when the base table diffs are updates on non-conditional attributes (case (a) above), the generated diffs on the view V_{spj} are guaranteed to be update or delete diffs and thus the speedup ratio will be equal to s_1. In any other case (case (b)), the generated view diffs will in general be combinations of update, delete, and insert diffs and thus the speedup will be a linear combination of s_1 and s_2 (and thus bounded by the smaller of them).

To summarize, for all base table diffs that do not lead to inserts in the cache, the ID-based approach is guaranteed to perform not worse (and the more complex the query the better) than the tuple-based approach. It only performs worse in workloads heavy on modifications that lead to insertions to the view, because it has to maintain a cache, so that the update and delete diffs can benefit from it. However, even this loss is bounded and we expect it to not be significant in practice. Moreover, this loss will be balanced out by the speedup on diffs that lead to deletes and updates in the view, which benefit from the cache.

7. EXPERIMENTAL EVALUATION

To compare the performance of ID-based and tuple-based IVM, we ran two sets of experiments. In our first set of experiments we studied the performance of both approaches on a diverse workload of views, by applying them on views commonly used in social networks. In our second set of experiments we studied the effect of varying different parameters on both the view and the data (such as selectivity, fanout, number of joins and base-table diff size).



	Q7	Q10	Q11	Q15	Q18	Q*1	Q*2	Q*3
ID IVM (sec)	0.07	0.07	0.09	0.73	0.16	14.24	0.64	0.43
Tuple IVM (sec)	2.04	3.98	2.26	3.23	2.15	364.50	4.56	3.75

Figure 10: Speedup & IVM time for extended set of BSMA queries

In all cases we used *idIVM* to generate the Δ-script and $\mathcal{D}$-script (the latter was produced using our implementation of *idIVM* with tuple-based diff propagation rules instead of the standard ID-based propagation rules). We then measured the times to run each script in PostgreSQL. All experiments were run using Ubuntu LTS 12.04, OpenJDK JRE 7 and PostgreSQL 9.1, on top of an Amazon Web Services (AWS) m1.large dedicated instance configured with 1,200 input/output operations per sec (IOPS) for 16KB blocks. The configured IOPS provide predictable throughput for random block accesses. Each experiment was run with cold PostgreSQL page buffers and Linux disk buffers, which is the common case when large number of views need to be maintained.

7.1 IVM in social analytics

To study the relative performance of ID-based and tuple-based IVM on a diverse set of real queries, we applied both IVM approaches to a workload of analytics views over social media. Maintaining analytics over social media is a primary use case for IVM techniques because: (a) large base tables are produced by social media such as Twitter and Facebook, (b) rapid, frequent updates occur on the base tables, and (c) analytic views that monitor metrics and trends need to be updated continuously.

To generate the workload, we utilized the Benchmark for Social Media Analytics (BSMA) [26]. Figure 9a shows the size of the relations generated, while Figure 9b provides a summary of the workload, which comprises:

- 100 update diffs on the **User** table for attributes $tweetsnum$ (i.e number of tweets) and $favornum$ (i.e. number of favorites)
- Views corresponding to queries Q7, Q10, Q11, Q15 and Q18 from BSMA, which exhibit join chains and aggregates, hence resulting in high cost for view re-computation. These queries are also minimally extended to: (a) extend the `SELECT` clause with attributes $tweetsnum$ and $favornum$ (b) remove the `ORDER BY` and `LIMIT` and the ID parameter in the WHERE clause in order to create larger views where the benefit of the IVM becomes apparent.
- An additional 3 aggregate views over the BSMA schema, labeled as Q*1, Q*2 and Q*3. Whereas queries Q7, Q10, Q11, Q15 and Q18 include aggregation, this aggregation is not affected by the updated attributes. Since, as we have discussed in Section 6 the ID-based and tuple-based approaches behave differently in the presence of aggregates affected by the updates, we designed views Q*1, Q*2 and Q*3 to include such aggregates.

Figure 10 shows the speedup ratio of the ID-based over the tuple-based approach for each of the views. The speedup varies widely between the 8 different views and its value does not seem to be determined by whether a view contains an aggregate affected by an update or not. From the reported views, it is interesting to look at the extreme cases with either very high or low speedup. Queries Q10 and Q*1 create a huge benefit for the ID-based IVM due to the fact that the tuple-based IVM has to incur a large number of data accesses, which can be avoided by the ID-based IVM. Interestingly, this need for data accesses is created in different ways by each of the two queries. In Q10 it is created by a long join chain (Q10 joins 4 relations), while in Q*1 it is created by a combination of a long join chain with a high selectivity that appears at the end of the join chain (so that the tuple-based has to perform a lot of data accesses before it can decide that a tuple will be dropped). Finally, Q15 displays a relatively low speedup because the resulting view and the number of tuples that need to be updated in the view is very large. This makes the view update time component (which is shared by both the ID-based and the tuple-based approaches) dominate the IVM cost, thus leading to a relatively small speedup. However, it should be noted that even in this case the ID-based approach outperforms the tuple-based approach by a factor of 4.

Relation	Tuples	Size
parts	5M	170MB
devices	5M	170MB
devices_parts	50M	3GB

(a) Relation sizes

Parameter	Defaults
d: Diff size	200
s: Selectivity	20%
f: Fanout	10
j: Joins	2

(b) Parameters

$$\Delta^u_{parts} = \mathcal{D}^u_{parts}(\underline{pid}, price_{pre}, price_{post})$$

(c) Base-table Diff

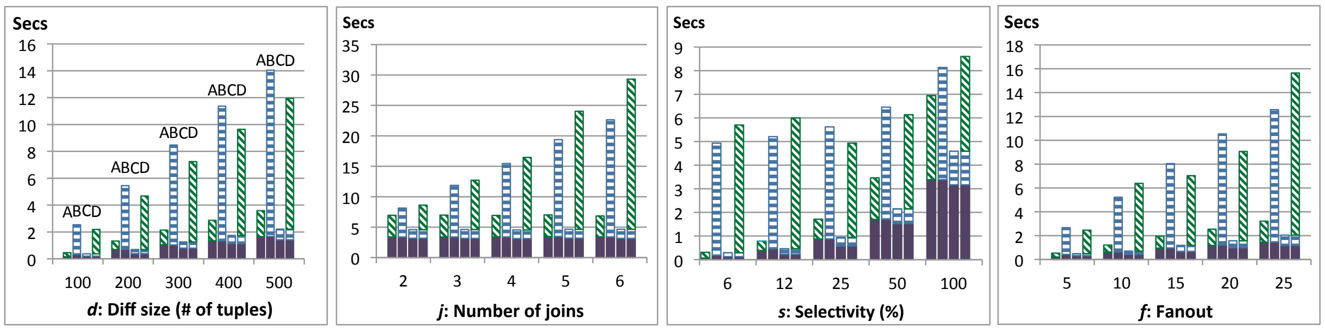
Figure 11: Configuration of varying parameter experiments

7.2 Effect of data & query parameters

To study in a more controlled fashion how the structure of the view and the data affects the ID-based and tuple-based IVM, we next ran a second set of experiments, in which we picked a single view and measured the performance of both ID-based and tuple-based IVM on maintaining the view, while varying different parameters of both the view definition and the underlying data. For ease of exposition we employed the view used in our running example and shown in Figure 5a.

Experimental setup. Figure 11 shows the properties of the dataset and the view used in the experiments. Figure 11a shows the sizes of the relations, Figure 11c presents the employed base-table diff (which captures updates on the prices of parts) and Figure 11b lists the parameters that we varied in the experiments and their default values. We varied four parameters: (a) The size d of the base-table diff (i.e., the number of price updates that happened), (b) the number of joins performed by the view (as we will explain later we extended the view that by default performs two joins with additional joins), (c) The selectivity s of the selection condition *category="phone"* (i.e., the percentage of **devices** tuples that satisfy the condition), and (d) the fanout f from **parts** to **devices_parts**, i.e. the number of parts for each device. (Note that the fanout from association table **devices_parts** to entity table **devices** is always 1.) For each experiment varying a parameter, we used for all other parameters their default values shown in Figure 11b.

Figure 12 shows the view maintenance times for ID-based IVM versus tuple-based IVM and the resulting speedup of ID-based over tuple-based IVM. The cost of each approach is broken down to its components. Column A represents ID-based IVM, with the top stack (diagonally striped) corresponding to cache update time (recall that the input of an aggregate is materialized as an intermediate cache), and the bottom stack (solid colored) corresponding to view update time. No stack is shown for the cache/view diff computation time as both are negligible. Column B represents tuple-based IVM, with the top stack (horizontally striped) corresponding to view diff computation time, and the bottom stack (solid colored) correspond-



d	100	200	300	400	500
Speedup	5.5	4.1	3.9	4.0	3.9

(a) Varying Diff Size

j	2	3	4	5	6
Speedup	1.2	1.7	2.2	2.8	3.3

(b) Varying Number of Joins

s	6	12	25	50	100
Speedup	15.9	6.6	3.3	1.9	1.2

(c) Varying Selectivity

f	5	10	15	20	25
Speedup	5.0	4.3	4.1	4.1	3.9

(d) Varying Fanout

A ID-based IVM B Tuple-based IVM C SDBT-fixed D SDBT-streams ▨ Cache Update ☰ View Diff Computation ■ View Update

Figure 12: View maintenance time of ID-based IVM vs tuple-based IVM and two DBToaster-inspired systems for varying parameters

ing to view update time. No stack is shown for cache update time, since as we have explained the tuple-based approach does not use a cache as it cannot benefit from it. Columns C and D correspond to simulations of DBToaster [2], which we discuss in Section 7.3. We next explain the effect of varying each of the parameters.

Varying the base-table diff size. Figure 12a illustrates the effects of varying the diff size d linearly from 100 to 500 tuples[9]. As shown on the figure, the speedup stays within 4-5. The experiment also shows a slight downward trend on the speedup. This is because as d increases, the chance also increases for reading a block of the **devices** table (out of its 20k blocks) from PostgreSQL's page buffers. In the running example, this buffering benefits tuple-based IVM, but not ID-based IVM.

Varying the number of joins. Figure 12b illustrates the effect of joins on the speedup. For this experiment we consider more complex views by augmenting the original view definition with j additional joins as follows: (i) **devices_parts** is joined with tables $R_1 \dots R_j$, such that each join is 1-to-1 on ($\underline{\text{pid, did}}$). This simulates joins across vertically-decomposed tables, which is common practice in data warehousing. (ii) The selection $\sigma_{\text{category=“phone”}}$ is disabled in order to focus on the effects of each additional join. Figure 12b shows that the total running time of ID-based IVM is unaffected by linearly varying j between 2 (i.e., the original view with no additional joins) to 6 (the view with four additional joins). On the other hand, the total running time of the tuple-based IVM increases with each additional join, making the speedup of ID-based IVM arbitrarily high, as the complexity of the query increases. This happens due to the fact that tuple-based IVM has to perform all joins in order to compute the entire view tuples that have to be modified, in contrast to the ID-based IVM that can simply propagate the base-table diff to the view and avoid performing any joins.

Varying the selectivity of the selection condition. Figure 12c shows how the speedup is affected by the selectivity s (i.e., the number of **devices** tuples that satisfy the condition). We vary s on a log scale from 6% to 100%. Allowing more tuples to pass through the selection adversely affects the performance of ID-based IVM. The reason is that more tuples of the **devices** table join with the other tables and thus the size of the intermediate cache employed by the ID-based approach increases. This in turn leads to a higher cost of updating the cache. It should be noted however that even at 100% selectivity, ID-based IVM is faster than tuple-based IVM, albeit at a lower speedup of 1.2. Thus, ID-based IVM is at least on par with tuple-based IVM, and performs better in the common case where the selection filters a subset of the base table tuples.

Varying the fanout. Finally, Figure 12d illustrates the effects of varying the fanout f of the join (**parts**, **devices_parts**) linearly between 5 to 25. For all values of the fanout, the ID-based IVM performs better than the tuple-based IVM by a factor of 4-5 times.

We highlight that the ID-based approach consistently outperforms the tuple-based approach. This is the case even though the experimental conditions were designed to explicitly benefit the tuple-based IVM. In particular, (a) we assumed the existence of appropriate base table indices to speedup tuple-based joins (without counting the associated index maintenance cost) and (b) we did not use a cache for the tuple-based IVM, to avoid the cache maintenance cost, since tuple-based maintenance of associative aggregate functions does not benefit from caches. When these optimizations are inadmissible, ID-based IVM will exhibit an even higher speedup.

7.3 Comparison to the state of the art

Finally, we compared *idIVM* to DBToaster [2]; the current state of the art IVM system, which, while being essentially a tuple-based system, has been shown to significantly outperform prior IVM approaches. DBToaster's performance is the result of five major optimizations: (a) performing IVM one diff tuple at a time (which leads to reducing $\mathcal{D}$-script joins with a diff table into selections), (b) compiling the $\mathcal{D}$-script into code, instead of SQL statements, (c) utilizing an in-memory implementation, (d) aggressively pushing aggregations down, and (e) materializing a large number of intermediate views (i.e., caches) that are used to maintain the original view and each other. On the other hand, *idIVM* benefits most from using (a) ID-based diffs and (b) update diffs (in contrast to DBToaster, where updates are simulated through inserts and deletes).

These differences make the two systems not directly comparable. Since *idIVM* could in principle also benefit from DBToaster's optimizations a-d, we next focus on comparing *idIVM* to the intermediate view materialization strategy used by DBToaster. To this end, we designed a DBToaster-inspired implementation (denoted as Simulated DBToaster or SDBT) that runs on top of a DBMS and uses the same intermediate views as the original DBToaster implementation (up to aggregation push-down). We then executed SDBT on all scenarios considered in Section 12. Since, in contrast to *idIVM*, DBToaster creates different intermediate views depending on the types of allowed base-table diffs, we ran two different versions of SDBT: one assuming that only diffs to the **parts** table are

[9]Similar trends can be observed for diff sizes up to 15,000 tuples. This is the point where it is beneficial to recompute the view rather than apply IVM, as discussed in prior work [13].

possible (referred to as SDBT-fixed) and one assuming that all base tables may change (referred to as SDBT-streams). Columns C and D in Figure 12 show the times of SDBT-fixed and SDBT-streams, respectively. We observe that *idIVM* in all cases significantly outperforms SDBT-streams, while it is in most cases slightly slower then SDBT-fixed. It should be noted however that we allowed both SDBT-fixed and SDBT-streams to employ update t-diffs. Had they simulated updates through inserts and deletes, as is the case in DBToaster, their performance would have been worse.

Note that SDBT captures only one of the optimizations used in DBToaster. Furthermore, SDBT (alike *idIVM* and tuple-based IVM) operates on large data, residing in secondary storage and managed via a database (PostgreSQL in this case). Due to the mix of optimizations involved and its main memory orientation, DBToaster behaves differently from SDBT. Experiments we conducted with DBToaster showed for instance that the compilation to code and in-memory implementation lead to 50-300 times faster execution than the PostgreSQL-based SDBT-fixed. On the other hand, the in-memory execution severely limits DBToaster's scalability (allowing it to scale only up to 2% of the data size used in our experiments when diffs are allowed on all base tables). Moreover, the lack of set-processing makes DBToaster's performance deteriorate much faster than SDBT with increasing diff sizes (e.g, DBToaster's speedup over SDBT-fixed drops from 300x for a diff size of 100 tuples to 50x when the diff size becomes 500).

8. RELATED WORK

IVM is a long studied problem with a lot of influential works [6, 5, 7, 21, 13]. *idIVM* falls under the category of IVM works that employ the *algebraic* [21, 10, 22, 18] approach. Due to the vast amount of related work in IVM, we focus next on approaches that are particularly related to the main aspects of our work, which are: (a) exploiting primary key information together with the associated (b) overestimation and (c) caching. Note that we cover all works in these areas, even if they do not follow the algebraic approach. For comprehensive surveys on IVM, the reader is referred to [12, 8].

Exploiting primary key constraints. The idea of exploiting primary key constraints to speed up IVM was first presented in [11, 23]. However, in contrast to our work, [11, 23] study only self-maintenance (potentially together with some auxiliary views) and not general view maintenance where some data from the base relations may be required to maintain the view. Furthermore, they are limited to maintenance of SPJ (including outer-join) views and their algorithms are not easily extensible to more general classes of queries as they operate by looking holistically at the view definition, in contrast to our modular algebraic approach. The first work that exploited primary keys in an extensible algebraic setting and introduced the notion of partial diffs, is [16]. However, the partial diffs of [16] always contain the entire primary key of the view. Thus, they are not true ID-based diffs, but instead (relaxed) tuple-based diffs, that may lack some of the (non-key) attributes of the view but will still incur the same number of accesses as tuple-based approaches. Finally, primary key information has also been used to optimize the rules for maintaining the output of particular operators (e.g., outer-join in [18]) within a tuple-based approach. However, these approaches do not look at exploiting the keys to avoid tuple-based diffs altogether, as done in this work.

Overestimation. Our definition of overestimation is similar to *safe overestimation* described in [4] and *ineffective updates* in [16]. While overestimation in these works appears only because of selection conditions, *idIVM* exploits also overestimation that arises because of joins, which do not appear in the former, since they are both (relaxed) tuple-based approaches.

Caching. Several works looked at the problem of materializing additional results to speed up IVM. These can be classified into two broad categories. The first category includes approaches where the cached results are operator-specific. Examples of such works include the IVM of aggregation under the assumption that previous aggregation results are available [22, 20] and of top-k results by caching additional view tuples that are beyond the top k in order to reduce the frequency of accessing the base tables [27]. These caches correspond to our notion of operator caches and can thus be incorporated in our framework as part of an operator definition. The second category contains approaches where the cached results are not tied to a particular operator, but are additional views that are then exploited holistically during the IVM of the original view [24, 19, 23, 2]. In contrast to our work that uses only caches that correspond to subplans of the original plan, these works benefit by employing caches that may not be subplans of a single plan. This aggressive materialization allows more efficient IVM, though at the cost of maintaining an increased number of intermediate views. A prime example of such approaches is DBToaster [2], which is discussed extensively in Section 7.3. However, by not being ID-based such approaches always access at least one materialized view, in contrast to our approach, which in some cases can avoid accessing base tables or cached views altogether. Finally, a related area to caching in IVM is view selection, consisting of works that decide which views to materialize to speed up query evaluation [1, 14]. Such approaches can be used in the context of *idIVM* to decide which intermediate caches to materialize.

9. CONCLUSIONS AND FUTURE WORK

We have shown how to exploit IDs towards an IVM algorithm that is more efficient than existing tuple-based approaches under common assumptions. An extension of this work involves minimizing base table accesses for insert i-diffs. Although in this work insert i-diffs incur the same base table accesses as tuple-based approches, more elaborate rules for i-diff's avoid base table accesses by instead utilizing data that potentially already exist in the view. However, in contrast to the current ID-based approach where one can know statically based on the i-diff schema whether base table accesses will be needed, the extended (for insert i-diff) version of the algorithm has to find out dynamically at run-time whether accesses are needed, depending on the i-diff instance.

10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Gaurav Saxena, Alok Singh, and Romain Vernoux for their detailed comments on prior versions of this paper. We also thank Christoph Koch, Mohammad Dashti, and Oliver Kennedy for extensive private communications on DBToaster.

11. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, 1997.

[4] A. Behrend and T. Jörg. Optimized incremental etl jobs for maintaining data warehouses. In *IDEAS*, pages 216–224, 2010.

[5] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[6] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[7] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[8] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[9] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[10] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[11] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.

[12] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 1995.

[13] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166. ACM Press, 1993.

[14] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[15] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *ICDE*, pages 106–117, 2005.

[16] T. Jörg and S. Deßloch. View maintenance using partial deltas. In *Datenbanksysteme für Business, Technologie und Web*, 2011.

[17] H. A. Kuno and G. Graefe. Deferred maintenance of indexes and of materialized views. In *DNIS*, pages 312–323, 2011.

[18] P.-Å. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, pages 56–65, 2007.

[19] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[20] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, pages 802–813, 2002.

[21] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[22] D. Quass. Maintenance expressions for views with aggregation. In *VIEWS*, pages 110–118, 1996.

[23] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *4th International Conference on Parallel and Distributed Information Systems*, 1996.

[24] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pages 447–458, 1996.

[25] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, 1998.

[26] F. Xia, Y. Li, C. Yu, H. Ma, and W. Qian. BSMA: A benchmark for analytical queries over social media data. *PVLDB*, 7(13), 2014.

[27] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

APPENDIX

A. DETAILED PERFORMANCE ANALYSIS

We next describe the detailed performance analysis of the ID-based and tuple-based approaches that led to the equations of Section 6. The analysis covers two representative cases: (a) SPJ views, which by default do not involve intermediate caches and (b) Aggregate views involving grouping and associative functions, which (by default) are supported by caches. In the following we assume that both approaches have access to view indices on the view IDs and additionally the tuple-based IVM has access to appropriate base table indices (which are not required by the ID-based approach).

A.1 SPJ Views

Consider (a) the SPJ view V_{spj}:

$$\texttt{SELECT } \bar{S} \texttt{ FROM } R, R_1, \ldots, R_n \texttt{ WHERE } c$$

whose FROM clause involves a single alias of a table R, (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R on R. We distinguish between two cases, depending on the type of the diff $\mathcal{D}_R/\Delta_R$.

A.1.1 Update diffs on non-conditional attributes

We first study base table update diffs on attributes of R that do not participate in any join or selection condition in V_{spj}. Consider such an update t-diff/i-diff on attributes $\bar{A}''$ of R:

$$\mathcal{D}^u_R = \Delta^u_R(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$$

where $\bar{I}$ is the key of R.

Since we are interested in the IVM of an update on R, we decompose the condition c into 3 subconditions c_R, c_{rest}, c_{R-rest} in conjunctive normal form, s.t. every one of their conjuncts involves only attributes of R, only attributes of $R_1, \ldots, R_n$ and both attributes of R and $R_1, \ldots, R_n$, respectively. It is easy to see that V_{spj} can be computed through the algebraic expression $\pi_S(\sigma_{c_R} R \bowtie_{c_{R-rest}} E)$, where $E = \sigma_{c_{rest}}(R_1 \times \ldots \times R_n)$.

In general, the i-diff Δ^u_R (resp. t-diff $\mathcal{D}^u_R$) will lead to $\Delta^u_{V_{\text{spj}}}$, $\Delta^+_{V_{\text{spj}}}$ and $\Delta^-_{V_{\text{spj}}}$ in the ID-based approach (resp. $\mathcal{D}^u_{V_{\text{spj}}}$, $\mathcal{D}^+_{V_{\text{spj}}}$ and $\mathcal{D}^-_{V_{\text{spj}}}$ in the tuple-based approach). However, since the updated attributes $\bar{A}''$ of R do not participate in the join condition c_{R-rest} or the selection condition c_R, the update diff on R will only lead to an update diff on V_{spj}. Furthermore, since the selection σ_{c_R} simply filters out tuples of $\mathcal{D}^u_R$, it has the same effect as using a smaller initial diff, and is therefore ignored in the rest of the analysis.

$\mathcal{D}/\Delta$-script. The scripts returned by the IVM algorithms are:

ID-based approach	Tuple-based approach
$\Delta^u_{V_{\text{spj}}} = \mathcal{D}^u_R$ APPLY $\Delta^u_{V_{\text{spj}}}$	$\mathcal{D}^u_{V_{\text{spj}}} = \pi_{\bar{S}} \mathcal{D}^u_R \bowtie_{c_{R-rest}} E$ APPLY $\mathcal{D}^u_{V_{\text{spj}}}$

The ID-based approach simply propagates the base table diff by exploiting the fact that no tuples need to be inserted or deleted from the view, since the modified attributes $\bar{A}''$ do not participate in the join condition.

Cost analysis. Both the ID-based IVM cost and the tuple-based IVM cost are the sum of the *diff computation cost* of $\Delta^u_{V_{\text{spj}}}$ (respectively $\mathcal{D}^u_{V_{\text{spj}}}$) and the *view modification cost*, which is the cost of applying the modifications dictated by $\Delta^u_{V_{\text{spj}}}$ (respectively $\mathcal{D}^u_{V_{\text{spj}}}$) on the materialized view. We measure both costs in terms of block accesses to indices and tuples. Employing common assumptions on the index structures[10], the cost of retrieving (using an index) the m tuples whose $\bar{X}$ attributes have given $\bar{x}$ values can be approximated by $1 + m$ (i.e., 1 index lookup and m tuple accesses). We next analyze each of the cost components.

Diff computation cost. Since the ID-based approach simply propagates Δ^u_R to the view as is, its diff computation cost is zero. On the other hand, the cost of the tuple-based IVM varies widely depending on the computation of $\mathcal{D}^u_{V_{\text{spj}}} = \pi_{\bar{S}} \mathcal{D}^u_R \bowtie_{c_{R-rest}} E$.

We consider the common case where the join condition c_{R-rest} is a conjunction of equalities of the form $R.J = R_i.J_i$. Furthermore, we assume that the database is optimized for tuple-based IVM, having all necessary indices for the efficient computation of $\mathcal{D}^u_{V_{\text{spj}}} = \mathcal{D}^u_R \bowtie_{c_{R-rest}} E$. Since the diff-table $\mathcal{D}^u_R$ is considered in the IVM literature to be smaller than the base tables, the DBMS will typically execute the above query through a *diff-driven loop* plan: For each tuple t of $\mathcal{D}^u_R$ it executes the subplan $\sigma_{c'_{R-rest}} E$, where c'_{R-rest} is an instantiation of the c_{R-rest} condition where the attributes of R have been replaced with their values in t. Let us name a the average number of accesses performed for each tuple of $\mathcal{D}^u_R$, i.e., the average number of accesses in each execution of $\sigma_{c'_{R-rest}} E$. Then the diff computation cost of the tuple-based approach is $|\mathcal{D}^u_R| a$. The DBMS may also choose to evaluate the

[10] We assume that indices satisfy the following conditions:
1. An index is either a hash index, or a B-tree with leaf nodes in secondary storage and non-leaf nodes in memory.
2. The retrieved tuples, if any, are not clustered together.
3. Caching of index leaves and/or tuples has minimal effects on the overall cost, as the cache is significantly smaller than the database.

Costs	ID-based	Tuple-based	
		Diff-driven loop plan	Other plan
Diff computation	0	$\|\mathcal{D}_R^u\|a$	E
View index lookups	$\|\mathcal{D}_R^u\|$	$\|\mathcal{D}_R^u\|p$	$\|\mathcal{D}_R^u\|p$
View tuple accesses	$\|\mathcal{D}_R^u\|p$		

Table 2: Costs of ID-based and tuple-based IVM on V_{spj}

query with a plan other than a diff-driven loop, but this is expected to happen only when the diff tables are very large, when the use of an IVM approach becomes questionable.

View modification cost. To apply $\Delta^u_{V_{\text{spj}}}$ (resp. $\mathcal{D}^u_{V_{\text{spj}}}$) to the view, the DBMS will typically utilize the view index to locate the view tuples that need to be modified. In either of the approaches there will be as many view index lookups as tuples in the view diff (i.e., $|\Delta^u_{V_{\text{spj}}}| = |\mathcal{D}^u_R|$ lookups for the ID-based and $|\mathcal{D}^u_{V_{\text{spj}}}|$ lookups for the tuple-based approach, respectively). Once the to-be-modified view tuples have been identified (which are in both cases equal to $|\mathcal{D}^u_{V_{\text{spj}}}|$), both approaches will incur $|\mathcal{D}^u_{V_{\text{spj}}}|$ view tuple accesses to update them. Table 2 shows the view index lookups and view tuple accesses for each approach utilizing the i-diff compression factor $p = |\mathcal{D}^u_{V_{\text{spj}}}|/|\Delta^u_{V_{\text{spj}}}|$.[11]

Discussion. Table 2 summarizes the costs for the tuple-based and ID-based approach. Combining them leads to the following speedup ratio of the ID-based over the tuple-based approach (assuming a diff-driven loop plan for the tuple-based approach):

$$\text{Speedup ratio for } V_{\text{spj}} = \frac{|\mathcal{D}^u_R|(a+p+p)}{|\mathcal{D}^u_R|(1+p)} = \frac{a+2p}{1+p} \qquad (1)$$

The relative performance difference between the two approaches varies depending on the value of the compression factor p. When $p \geq 1$, the ID-based approach is guaranteed to be more efficient with its absolute gain (i.e., the difference of accesses from the tuple-based approach) raising proportionally to p. When $0 < p < 1$, the ID-based approach is also more efficient in the typical case when the tuple-based approach has to do at least one access for each tuple in $\mathcal{D}^u_R$ (which means that $a > 1$). For the tuple-based approach to perform better it should be the case that $a < 1 - p$, which can be satisfied only if the following conditions simultaneously hold: (a) the tuple-based approach incurs $a < 1$ tuple accesses for each tuple in $\mathcal{D}^u_R$ on average (which can only happen if many tuples of $\mathcal{D}^u_R$ share the same join attribute values and thus the joined tuples can be retrieved once and reused for all of them) and (b) the ID-based approach is severely overestimating (i.e., $p \ll 1$).

A.1.2 Other diffs

Consider now insert and delete i-diffs/t-diffs on the base relation R, as well as update i-diffs/t-diffs on attributes of R that are involved in some join or selection condition in V_{spj}. Such base table diffs will lead in general to a combination of $\Delta^u_{V_{\text{spj}}}$, $\Delta^+_{V_{\text{spj}}}$ and $\Delta^-_{V_{\text{spj}}}$ in the ID-based approach (resp. $\mathcal{D}^u_{V_{\text{spj}}}$, $\mathcal{D}^+_{V_{\text{spj}}}$ and $\mathcal{D}^-_{V_{\text{spj}}}$ in the tuple-based approach). For the cases in which a base table diff is translated into an update or delete i-diff/t-diff on the view, the ID-based and tuple-based algorithms will behave as described in Section A.1.1 and thus the speedup ratio will be $\frac{a+2p}{1+p}$. On the other hand, in the case when a base table diff is translated into an insert i-diff/t-diff on the view, the ID-based and tuple-based algorithm will produce identical scripts, leading to a speedup of 1 (i.e., the ID-based algorithm will degenerate to the tuple-based algorithm but will not behave worse than the latter).

[11] In the unlikely case, where the DBMS chooses to identify the to-be-modified tuples by performing a full scan of the view instead of using the index, the view modification cost becomes the same for both approaches. In this case, the difference between the two approaches reduces to the difference between their computation costs.

A.2 Aggregate Views

Consider (a) the aggregate view V_{agg}:

```
SELECT $\bar{G}, f(\bar{X})$ AS $g$ FROM $R, R_1, \ldots, R_n$
WHERE $c$ GROUP BY $\bar{G}$
```

whose FROM clause involves a single alias of a table R, and f is an associative aggregation function such as `sum`, (b) a t-diff $\mathcal{D}_R$ on R and (c) a corresponding i-diff Δ_R on R.

To ease exposition, we isolate the aggregation operator of the query, expressing it through the plan $V_{\text{agg}} = \gamma_{\bar{G}, f(\bar{X}) \to g} V_{\text{spj}}$, where V_{spj} is the algebraic plan for the SPJ query presented in Section A.1.

We consider the case where the ID-based algorithm has determined that it is beneficial to create an intermediate cache storing the result of the SPJ subview V_{spj}, since otherwise both approaches will be performing identically. The tuple-based does not employ a cache, as it cannot benefit from it.

Similarly to the case of SPJ views, we distinguish between two cases depending on the type of the base table diff $\mathcal{D}_R/\Delta_R$.

A.2.1 Update diffs on non-conditional attributes

Consider an update t-diff/i-diff on attributes $\bar{A}''$ of R that are not involved in any condition in V_{agg}:

$$\mathcal{D}^u_R = \Delta^u_R(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$$

where $\bar{I}$ is the key of R.

Cache diff computation / modification cost. The ID-based approach maintains an intermediate cache, which is equivalent to V_{spj}. The cache incurs *cache diff computation* cost and *cache modification cost*, which are also equivalent to the diff computation and view modification cost of V_{spj}. There is no intermediate cache for the tuple-based approach.

View diff computation cost. We consider the case where f is *incrementally computable*. That is, there is an incremental function $f_{\mathcal{D}}$ that inputs $\mathcal{D}^u_{V_{\text{spj}}}(\bar{S}, \bar{X}_{pre}, \bar{X}_{post})$ where $\bar{S}$ is the key of V_{spj}, and outputs $\mathcal{D}^u_{V_{\text{agg}}}(\bar{G}, g_{pre}, g_{post})$. For example, when f is the `sum` function, $f_{\mathcal{D}}$ is also f, since `sum` is an associative aggregation function. Given $f_{\mathcal{D}}$, the tuple-based approach computes $\mathcal{D}^u_{V_{\text{agg}}} = \gamma_{\bar{G}, f_\Delta(\bar{X}) \to g} \mathcal{D}^u_{V_{\text{spj}}}$.

Given that $|\mathcal{D}^u_R|$ is much smaller than base tables, the number of groups in $\mathcal{D}^u_{V_{\text{agg}}}$ will be smaller than the number of groups in V_{agg}. The efficient implementation for γ is thus hash aggregation with in-memory buckets, which can be pipelined. Due to pipelining, no additional block accesses are incurred for γ. Thus, the tuple-based approach has the same diff computation cost for V_{spj} and V_{agg}.

As an optimization, the ID-based approach uses the UPDATE RETURNING statement to update the cache and return the result of the update in a single step. Thus, $\Delta^u_{V_{\text{spj}}}$ is obtained without additional accesses over cache modification costs. Similar to the tuple-based approach, $\Delta^u_{V_{\text{agg}}} = \gamma_{\bar{G}, f_\Delta(\bar{X}) \to g} \Delta^u_{V_{\text{spj}}}$, and the γ uses pipelined hash aggregation. Thus, the ID-based approach also has the same diff computation cost for V_{spj} and V_{agg}.

View modification cost. To apply $\Delta^u_{V_{\text{agg}}}$ (resp. $\mathcal{D}^u_{V_{\text{agg}}}$) to the view, both approaches will incur an index lookup and a tuple access per tuple in the i-diff (resp. t-diff). We denote the grouping compression factor $g = |\mathcal{D}^u_{V_{\text{agg}}}|/|\mathcal{D}^u_{V_{\text{spj}}}|$ in Table 3. **Discussion.** For V_{agg}, Table 3 summarizes the costs for both ID-based and tuple-based approaches. Combing them leads to the following speedup ratio of the ID-based over the tuple-based approach (assuming a diff-driven loop plan for the tuple-based approach):

$$\text{Speedup ratio for } V_{\text{agg}} = \frac{a+2pg}{1+p+2pg} \qquad (2)$$

Costs	ID-based	Tuple-based	
		Diff-driven loop plan	Other plan
Cache diff computation	0	–	
Cache index lookups	$\lvert\mathcal{D}_R^u\rvert$	–	
Cache tuple accesses	$\lvert\mathcal{D}_R^u\rvert p$	–	
View diff computation	0	$\lvert\mathcal{D}_R^u\rvert a$	E
View index lookups	$\lvert\mathcal{D}_R^u\rvert pg$		
View tuple accesses	$\lvert\mathcal{D}_R^u\rvert pg$		

Table 3: Costs of ID-based and tuple-based IVM on V_{agg}

For the tuple-based approach to perform better on V_{agg}, it should be the case that $\frac{a+2pg}{1+p+2pg} < 1$, which implies that $a < 1 + p$. However, we will show that this is never possible. The average cost a spent by the tuple-based IVM for each diff tuple in $\mathcal{D}_R^u$ will always be at least $1+p$, as for each such tuple t the algorithm would have to perform (a) at least one index access to check whether t joins with some of the other relations in the FROM clause and (b) at least p tuple accesses to retrieve the tuples it joins with from the other relations to create p diff tuples in $\mathcal{D}_{V_{\text{spj}}}^u$. Note that these are the lower bounds for a that happen when the query contains just a single join $R \bowtie R_1$. For longer join chains the tuple-based IVM will have to perform additional index and tuple accesses, yielding even worse performance compared to the ID-based approach.

Note that the above analysis exploits the absence of multivalued dependencies in V_{spj} (which is a necessary condition for *idIVM* to create a cache). If there was a multivalued dependency, multiple tuples in $\mathcal{D}_R^u$ could share the same computation (and thus it would not be the case that each of them would incur at least $1 + p$ accesses).

A.2.2 Other diffs

Similarly to the SPJ views, an insert and delete base table diff or an update diff on an attribute of R involved in a condition, might lead to insert, update and delete diffs on V_{spj}. For the cases that lead to deletes and updates on V_{spj} the cost will be the same as the one outlined in Section A.2.1.

On the other hand when base table diffs lead to insert diffs on the view V_{spj}, the ID-based approach will be performing the same plan with the tuple-based approach but will also be inserting tuples into the cache. Let k be the number of tuples created in V_{spj} on average as a result of a single diff in $\mathcal{D}_R$. Then the cost of the tuple-based and ID-based approach is $|\mathcal{D}_R|(a+2pg)$ and $|\mathcal{D}_R|(k+a+2pg)$, respectively. Thus the speedup ratio is $\frac{a+2pg}{a+k+2pg}$. This speedup is less than 1 (meaning that the tuple-based approach will be performing better). However, this loss is bounded, as it is always 1 per tuple inserted into V_{spj}.

B. I-DIFF PROPAGATION RULES

For $\Delta^+_{Input_l}(\bar{I}, \bar{A}''_{post})$
$\Delta^+_V = \Delta^+_{Input_l} \times Input_r^{post}$
For $\Delta^+_{Input_r}(\bar{I}, \bar{A}''_{post})$
$\Delta^+_V = Input_l^{post} \times \Delta^+_{Input_r}$
For $\Delta^-_{Input_l}(\bar{I}, \bar{A}'_{pre})$ ($\Delta^-_{Input_r}$ is symmetric)
$\Delta^-_V = \Delta^-_{Input_l}$
For $\Delta^u_{Input_l}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$ ($\Delta^u_{intpu_r}$ is symmetric)
$\Delta^u_V = \Delta^u_{Input_l}$

Table 4: Rules for $\times$

For $\Delta^+_{Input_l}(\bar{I}, \bar{A}_{post})$ (For $\Delta^+_{Input_r}$ replace 0 by 1)
$\Delta^+_V = \pi_{*,b\to 0}\Delta^+_{Input_l}$
For $\Delta^-_{Input_l}(\bar{I}, \bar{A}'_{pre})$ (For $\Delta^-_{Input_r}$ replace 0 by 1)
$\Delta^-_V = \pi_{*,b\to 0}\Delta^-_{Input_l}$
For $\Delta^u_{Input_l}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$ (For $\Delta^u_{Input_r}$ replace 0 by 1)
$\Delta^-_V = \pi_{*,b\to 0}\Delta^u_{Input_l}$

Table 5: Rules for $\cup$

For $\Delta^+_{Input}(\bar{I}, \bar{A}_{post})$
$\Delta^+_V = \sigma_{\phi(\bar{X})}\Delta^+_{Input}$
For $\Delta^-_{Input}(\bar{I}, \bar{A}'_{pre})$
$\Delta^-_V = \sigma_{\phi(\bar{X}_{pre})}\Delta^-_{Input}$
For $\Delta^u_{Input}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$

if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
    $\Delta^u_V = \sigma_{\phi(\bar{X}_{pre})}\sigma_{\phi(\bar{X})}\Delta^u_{Input}$
else
    $\Delta^u_V = \Delta^-_{Input}$

if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
    Δ^+_V = not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
    $\Delta^+_V = Input^{post} \ltimes_{\bar{I}} \sigma_{\neg\phi(\bar{X}_{pre})}\sigma_{\phi(\bar{X})}\Delta^u_{Input}$
else
    $\Delta^+_V = \sigma_{\neg\phi(\bar{X}_{pre})}\sigma_{\phi(\bar{X})}(Input^{post} \ltimes \Delta^u_{Input})$

if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
    Δ^-_V = not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
    $\Delta^-_V = \pi_{\bar{I},\bar{A}'_{pre}}\sigma_{\phi(\bar{X}_{pre})}\sigma_{\neg\phi(\bar{X})}\Delta^u_{Input}$
else
    $\Delta^-_V = \pi_{\bar{I},\bar{A}'_{pre}}\sigma_{\phi(\bar{X}_{pre})}\sigma_{\neg\phi(\bar{X})}Input^{post} \ltimes \Delta^u_{Input}$

Blue portion applies when pre-state attributes present.

Table 6: Rules for $\sigma_{\phi(\bar{X})}$

For $\Delta^-_{Input}(\bar{I}, \bar{A}'_{pre})$ where $\bar{I} \subseteq \bar{G}$
$\Delta^-_V = \Delta^-_{Input}$
For any $\Delta^t_{Input}(\bar{I}, \bar{A}')$ and f, we can recompute groups

if $\bar{G} \subseteq (\bar{I} \cup \bar{A}')$ then
    $\Delta^u_V = \gamma_{\bar{G},f(\bar{X})\to c}(\Delta^t_{Input} \rtimes_{\bar{G}} Input^{post})$
else
    $\Delta^u_V = \gamma_{\bar{G},f(\bar{X})\to c}(\Delta^t_{Input} \rtimes_{\bar{I}} Input^{post} \rtimes_{\bar{G}} Input^{post})$

(Do not handle group creation/deletion)

Table 7: Rules for $\gamma_{\bar{G},f(\bar{X})\to c}$

For $\Delta^+_{Input}(\bar{I}, \bar{A}_{post})$
$\Delta^+_V = \pi_{\bar{D},f(\bar{X})\to c,\bar{I}}\Delta^+_{Input}$
For $\Delta^-_{Input}(\bar{I}, \bar{A}'_{pre})$

if $\bar{X} \subseteq \bar{I} \cup \bar{A}'_{pre}$ then
    $\Delta^-_V = \pi_{(\bar{D}\cap(\bar{I}\cup\bar{A}'_{pre}))\cup f(\bar{X}_{pre})\to c_{pre},\bar{I}}\Delta^-_{Input}$
else
    $\Delta^-_V = \pi_{\bar{D}\cap(\bar{I}\cup\bar{A}'_{pre}),\bar{I}}\Delta^-_{Input}$

For $\Delta^u_{Input}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$
if $(\bar{I} \cup \bar{A}''_{post}) \cap \bar{X} = \emptyset$ then $\Delta^u_V = \sigma_{isupd}\pi_{\bar{D}',\bar{I}}\Delta^u_{Input}$ else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then $\Delta^u_V = \sigma_{isupd}\pi_{\bar{D}',f(\bar{X})\to c_{post},f(\bar{X}_{pre})\to c_{pre},\bar{I}}\Delta^u_{Input}$ else $\Delta^u_V = \sigma_{isupd}\pi_{\bar{D}',f(\bar{X})\to c_{post},f(\bar{X}_{pre})\to c_{pre},\bar{I}}$ $(Input^{post} \ltimes_{\bar{I}} \Delta^u_{Input})$ where σ_{isupd} selects tuples corresponding to actual updates (i.e., where $c_{pre} \neq c_{post}$ or $a_{pre} \neq a_{post}$ for some attribute a), $\bar{D}' = (\bar{D} \cap (\bar{I} \cup \bar{A}''_{post})) \cup (\bar{D}_{pre} \cap \bar{A}'_{pre})$, and $\bar{D}_{pre}$ is the pre-state counterpart of $\bar{D}$.
Blue portion applies when pre-state attributes present.

Table 8: Rules for $\pi_{\bar{D},f(\bar{X})\to c}$

For $\Delta^u_{Input}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$, and $\bar{G} \cap \bar{A}''_{post} = \emptyset$
$\Delta^i_1 = \pi_{\bar{I}, x_{post} - x_{in} \rightarrow x_\Delta}(\Delta^u_{Input} \bowtie \pi_{x \rightarrow x_{in}} Input^{pre})$
For $\Delta^-_{Input}(\bar{I}, \bar{A}'_{pre})$
$\Delta^j_2 = \pi_{\bar{I}, 0 - x_{in} \rightarrow x_\Delta}(\Delta^-_{Input} \bowtie \pi_{x \rightarrow x_{in}} Input^{pre})$
For $\Delta^+_{Input}(\bar{I}, \bar{A}''_{post})$
$\Delta^k_3 = \pi_{\bar{I}, x \rightarrow x_\Delta}(\Delta^+_{Input} \bar{\ltimes} Input^{pre})$
For converting Δ to output update i-diffs
Happens after all $\Delta^i_1, \Delta^j_2, \Delta^k_3$ are computed.
$\Delta^u_V = \pi_{\bar{G}, c \rightarrow c_{pre}, c + c_\Delta \rightarrow c_{post}}(Output \bowtie$
$\gamma_{\bar{G}, sum(x_\Delta) \rightarrow c_\Delta}(\Delta^i_1 \cup \Delta^j_2 \cup \Delta^k_3))$

(Do not handle group creation/deletion)

Table 9: Rules for $\gamma_{\bar{G}, sum(\bar{X}) \rightarrow c}$

For $\Delta^+_{Input_l}(\bar{I}, \bar{A}_{post})$
$\Delta^+_V = \Delta^+_{Input_l} \bowtie_{\phi(\bar{X})} Input^{post}_r$
For $\Delta^+_{Input_r}(\bar{I}, \bar{A}_{post})$
$\Delta^+_V = Input^{post}_l \bowtie_{\phi(\bar{X})} \Delta^+_{Input_r}$
For $\Delta^-_{Input_l}(\bar{I}, \bar{A}'_{pre})$ ($\Delta^-_{Input_r}$ is symmetric)
$\Delta^-_V = \sigma_{\phi(\bar{X}_{pre})} \Delta^-_{Input_l}$
For $\Delta^u_{Input_l}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$ ($\Delta^u_{Input_r}$ is symmetric)
if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\Delta^u_V = \sigma_{\phi(\bar{X}_{pre})} \sigma_{\phi(\bar{X})} \Delta^u_{Input_l}$
else
$\Delta^u_V = \Delta^u_{Input_l}$
if $\bar{I} \cap \bar{A}''_{post} = \emptyset$ then
$\Delta^+_V =$ not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\Delta^+_V = (Input^{post}_l \ltimes_{\bar{I}} \sigma_{\neg\phi(\bar{X}_{pre})} \sigma_{\phi(\bar{X}_{post})} \Delta^u_{Input_l})$
$\bowtie_{\phi(\bar{X})} Input^{post}_r$
else
$\Delta^+_V = \pi_{Input_l, Input_r} \sigma_{\neg\phi(\bar{X}_{pre})} \sigma_{\phi(\bar{X}_{post})}$
$(Input^{post}_l \bowtie \Delta^u_{Input_l} \bowtie_{\phi(\bar{X})} Input^{post}_r)$
if $\bar{I} \cap \bar{A}''_{post} = \emptyset$ then
$\Delta^-_V =$ not triggered
else if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\Delta^-_V = \pi_{\bar{I}, \bar{A}'_{pre}} \sigma_{\phi(\bar{X}_{pre})} \sigma_{\neg\phi(\bar{X}_{post})} \Delta^u_{Input_l}$
else
$\Delta^-_V = \pi_{\bar{I}, \bar{A}'_{pre}} \sigma_{\phi(\bar{X}_{pre})} \sigma_{\neg\phi(\bar{X}_{post})} Input^{post}_l$
$\bowtie \Delta^u_{Input_l} \bowtie_{\phi(\bar{X})} Input^{post}_r$
Blue portion applies when pre-state attributes present.

Table 10: Rules for $\bowtie_{\phi(\bar{X})}$

For $\Delta^u_R(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$, and $\bar{G} \cap \bar{A}''_{post} = \emptyset$
$\Delta^i_1 = \emptyset$
For $\Delta^-_R(\bar{I}, \bar{A}'_{pre})$
$\Delta^j_2 = \pi_{\bar{I}, -1 \rightarrow x_\Delta}(\Delta^-_R \bowtie \pi_{x \rightarrow x_{in}} Input)$
For $\Delta^+_R(\bar{I}, \bar{A}''_{post})$
$\Delta^k_3 = \pi_{\bar{I}, 1 \rightarrow x_\Delta}(\Delta^+_R \bar{\ltimes} Input)$
For converting Δ to output update i-diffs
Happens after all $\Delta^i_1, \Delta^j_2, \Delta^k_3$ are computed.
$\Delta^u_V = \pi_{\bar{G}, c \rightarrow c_{pre}, c + c_\Delta \rightarrow c_{post}}(Output \bowtie$
$\gamma_{\bar{G}, sum(x_\Delta) \rightarrow c_\Delta}(\Delta^i_1 \cup \Delta^j_2 \cup \Delta^k_3))$

(Do not handle group creation/deletion)

Table 11: Rules for $\gamma_{\bar{G}, count(\bar{X}) \rightarrow c}$

Operator cache schemas:
$Cache_{sum}(\bar{G}, c_{sum}), Cache_{count}(\bar{G}, c_{count})$
Cache maintenance rules:
For $\Delta^u_{Cache_{sum}}$: Use rules of $\gamma_{\bar{G}, sum(\bar{X}) \rightarrow c}$ (Table 9)
For $\Delta^u_{Cache_{count}}$: Use rules of $\gamma_{\bar{G}, count(\bar{X}) \rightarrow c}$ (Table 11)
i-diff propagation rules:
$\Delta^u_V = \pi_{\bar{G}, c^{sum}_{pre}/c^{count}_{pre} \rightarrow c_{pre}, c^{sum}_{post}/c^{count}_{post} \rightarrow c_{post}}($
$\Delta^u_{Cache_{count}} \bowtie_{\bar{G}} \Delta^u_{Cache_{sum}})$

Table 12: Rules for $\gamma_{\bar{G}, avg(\bar{X}) \rightarrow c}$

For $\Delta^+_{Input_l}(\bar{I}, \bar{A}''_{post})$
$\Delta^+_V = \Delta^+_{Input_l} \bar{\ltimes}_\phi Input^{post}_r$
For $\Delta^-_{Input_l}(\bar{I}, \bar{A}'_{pre})$
$\Delta^-_V = \Delta^-_{Input_l}$
For $\Delta^u_{Input_l}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$
$\Delta^u_V = \Delta^u_{Input_l}$
if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
$\Delta^+_V =$ not triggered
if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\Delta^+_V = Input^{post}_l \ltimes_{\bar{I}_{Input_l}} (\Delta^u_{Input_l} \bar{\ltimes}_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r)$
else
$\Delta^+_V = (Input^{post}_l \ltimes_{\bar{I}_{Input_l}} \Delta^u_{Input_l}) \bar{\ltimes}_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r$
if $\bar{X} \cap \bar{A}''_{post} = \emptyset$ then
$\Delta^-_V =$ not triggered
if $\bar{X} \subseteq \bar{I} \cup \bar{A}''_{post}$ then
$\Delta^-_V = \pi_{\bar{I}}(\Delta^u_{Input_l} \ltimes_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r)$
else
$\Delta^-_V = \pi_{\bar{I}}((Input^{post}_l \ltimes_{\bar{I}_{Input_l}} \Delta^u_{Input_l})$
$\ltimes_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r)$
For $\Delta^+_{Input_r}(\bar{I}, \bar{A}''_{post})$
$\Delta^-_V = \pi_{\bar{I}}(Input^{post}_l \ltimes_{\phi(\bar{X}_{post}, \bar{Y})} \Delta^+_{Input_r})$
For $\Delta^-_{Input_r}(\bar{I}, \bar{A}'_{pre})$
if $\bar{Y} \subseteq \bar{I} \cup \bar{A}'_{pre}$ then
$\Delta^+_V = (Input^{post}_l \ltimes_{\phi(\bar{X}_{pre}, \bar{Y})} \Delta^-_{Input_r})$
$\bar{\ltimes}_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r$
else
$\Delta^+_V = (Input^{post}_l \ltimes_{\phi(\bar{X}_{pre}, \bar{Y})} (Input^{pre}_r \ltimes \Delta^-_{Input_r}))$
$\bar{\ltimes}_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r$
For $\Delta^u_{Input_r}(\bar{I}, \bar{A}'_{pre}, \bar{A}''_{post})$
Treat input update as combination of insert and delete
if $\bar{Y} \cap \bar{A}''_{post} = \emptyset$ then
$\Delta^-_V =$ not triggered
else
$\Delta^-_V = \pi_{\bar{I}}(Input^{post}_l \ltimes_{\phi(\bar{X}_{post}, \bar{Y})} (Input^{post}_r \ltimes \Delta^u_{Input_r}))$
if $\bar{Y} \cap \bar{A}''_{post} = \emptyset$ then
$\Delta^+_V =$ not triggered
else if $\bar{Y} \subseteq \bar{I} \cup \bar{A}'_{pre}$ then
$\Delta^+_V = (Input^{post}_l \ltimes_{\phi(\bar{X}_{pre}, \bar{Y})} \Delta^-_{Input_r})$
$\bar{\ltimes}_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r$
else
$\Delta^+_V = (Input^{post}_l \ltimes_{\phi(\bar{X}_{pre}, \bar{Y})} (Input^{pre}_r \ltimes \Delta^-_{Input_r}))$
$\bar{\ltimes}_{\phi(\bar{X}_{post}, \bar{Y})} Input^{post}_r$

Table 13: Rules for $\bar{\ltimes}_{\phi(Input_l.\bar{X}, Input_r.\bar{Y})}$