Utilizing IDs to Accelerate Incremental View Maintenance

Yannis Katsis† ikatsis@cs.ucsd.edu
Kian Win Ong† kianwin@cs.ucsd.edu
Yannis Papakonstantinou† yannis@cs.ucsd.edu
Kevin Keliang Zhao† kezhao@cs.ucsd.edu
†CSE Department, UC San Diego

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. \textit{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, antijoin (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.

1. INTRODUCTION
Business intelligence softwares relies on materialized views to speed up query evaluation by precomputing the results of commonly asked queries or subqueries. The views need to be kept up to date when the underlying data change. This is typically done through \textit{Incremental View Maintenance} (IVM) algorithms. A typical IVM algorithm takes as input three diff tables \( \Delta u, \Delta R \) and \( \Delta v \) per base relation \( R \), containing the tuples that were inserted, deleted and updated in \( R \) since the last run of the algorithm. Given the diff’s, it computes the corresponding diff tables \( \Delta \text{V}, \Delta \text{V}^+ \) and \( \Delta \text{V}^- \) for the view \( V \), containing the changes that have to be performed on the view to bring it up to date.

In prior IVM work, each diff tuple \( \Delta \text{V}^+, \Delta \text{V}^- \) and \( \Delta \text{V}^\Delta \) 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 \( \text{tuple-based diffs} \) (in short \( t\text{-diffs} \)). 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 \textit{ID-based diffs} (in short \( i\text{-diffs} \)), which identify the to-be-modified view tuples through their IDs. In contrast to \( t\text{-diffs} \), a single \( i\text{-diff} \) tuple can represent modifications to \textit{multiple} view tuples. This difference is crucial, as \( i\text{-diffs} \) are more efficient to compute than \( t\text{-diffs} \), 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 \( t\text{-diffs} \) and \( i\text{-diffs} \).

\begin{example}
Consider the database of an electronic device manufacturer storing the parts that make up each device. Figure 1a shows the respective database schema (underlined attributes correspond to keys). Also consider the view \( V \) of Figure 1b returning the list of parts of each phone.

Figure 1 shows an example of tuple-based and ID-based incremental maintenance of \( V \). The shaded box in the middle contains the initial database and view instance and the left and right columns of the figure contain the \( t\text{-diffs} \) and \( i\text{-diffs} \), respectively for the base table \( \text{parts} \) and for the view \( V \). Both the \( t\text{-diff} \) \( \Delta \text{parts} \) and the \( i\text{-diff} \) \( \Delta \text{parts} \) for \( \text{parts} \) contain a single tuple, representing an update of the price of part “P1” from $10 to $11. The difference between the two approaches becomes apparent in the diffs for the view \( V \). While the \( t\text{-diff} \) \( \Delta V \) contains two tuples, each representing an update of the price for a different “P1” tuple in the view, the \( i\text{-diff} \) \( \Delta V \) exploits the fact that \text{pid} is a key of the base table \( \text{parts} \) and represents the same set of view updates through a single \( i\text{-diff} \) tuple. This \( i\text{-diff} \) tuple intuitively specifies an update of the price for all “P1” tuples that exist in the view.

Obviously ID-based diffs are more compact than their tuple-based counterparts. More importantly, we show both analytically...
and experimentally that this compressed representation leads to an ID-based IVM algorithm that is most often more efficient in terms of base table and view accesses than prior tuple-based approaches. Intuitively, the performance gains come from the fact that ID-based diffs do not need to recreate the entire tuples to be modified and thus they can be computed by accessing less data from the base tables.

**Example 1.2.** Queries \(Q_\Delta\) and \(Q_{\Delta}\) show how the diffs for the view can be computed from the base tables and the diff for base relation \(\Delta_{\text{parts}}\). While computing the i-diff requires joining \(\Delta_{\text{parts}}\) with the base tables \(\text{devices}_{\text{parts}}\) and \(\text{devices}\) (see \(Q_\Delta\)), producing the i-diffs can be simply done by accessing only \(\Delta_{\text{parts}}\), and avoiding all joins with the base relations (see \(Q_{\Delta}\)).

![Diagram](image.png)

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

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. i-diffs have the drawback of potentially creating dummy i-diff tuples. For example, assume that \(\text{parts}\) included a tuple \((P3, 20)\) and \(\Delta_{\text{parts}}\) included a change of the P3 price. Then \(\Delta_{V}\) 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, our cost model correctly models the data accesses performed by IVM algorithms in practical use cases. The experimental results show that the ID-based algorithm achieves a typical speedup of 2-5 over traditional tuple-based IVM approaches.

**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 that includes the algebraic operators select, project, join, grouping and aggregation, generalized projection involving functions, antisemijoin1 and union. Although in general i-diff propagation equations can be written in different ways, we chose to implement equations that are individually minimal w.r.t. the required number of base table accesses, and may pay the (relatively smaller) price of overestimation.

(c) An efficient 4-pass algorithm that creates an IVM plan for a given algebraically-expressed view and a given type of modification 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 is given a view expression and decides what types of ID-based 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 ID-based 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 ID-based diffs.

(e) A formal analysis proving that in many use cases the ID-based IVM 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 indicating that our cost model correctly models the data accesses performed by IVM algorithms in practical use cases. The experimental results show that the ID-based algorithm achieves a typical speedup of 2-5 over traditional tuple-based IVM approaches.

The ID-based IVM optimization is orthogonal to prior IVM works [12][8] and can be combined with them. Such prior IVM works are materialized view selection [3][[25][19], self maintenance [5][11] and compilation into code [2]. We briefly describe these prior IVM algorithms. The idIVM is a special case of antisemijoin.

1Therefore capturing queries with negation. The difference operators

---

**References:**

1. [ID-based IVM](#)
2. [Tuple-based 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 modifications 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 3 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 IDs \( \bar{I} \) and a set of non-ID attributes \( \bar{A} \). The IDs of the view are typically (but not always) the keys of the base tables involved in the view definition and are used to identify the to-be-modified view tuples. We will see in Section 3 how idIVM makes sure that the view contains the required set of IDs.

Example 2.1. The view \( V \) of our running example contains IDs \( \bar{I} = \{ \text{did}, \text{pid} \} \) (which are the keys of the base relations devices) and non-ID attributes \( \bar{A} = \{ \text{price} \} \). In the following, we will be using the initial instance of \( V \) shown in Figure 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) for relation \( V \) is an annotated relation \( \Delta^\bar{V}_{\text{pre}}(\bar{I}, \bar{A}) \), whose schema satisfies the following conditions:

- It contains a subset \( \bar{I}' \) of the view’s IDs \( \bar{I} \). These are used to identify the tuples to be modified.
- It contains a set \( \bar{A}' \) of attributes, such that each attribute in the set is of the form \( A_{\text{pre}} \) or \( A_{\text{post}} \), where \( A \) is a non-ID attribute of \( V \). An attribute \( A_{\text{pre}} \) and \( A_{\text{post}} \) 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 \( V \).
- It contains a special annotation, called modification type that represents the type of modifications included in the i-diff. The modification type can be +, − or u, standing for insertion, deletion and update, respectively. Each i-diff contains modifications (represented as tuples of a single type only).

We use the notation \( \Delta^\bar{V}_{\text{pre}}(\bar{I}', \bar{A}'_{\text{pre}}, \bar{A}'_{\text{post}}) \) to represent an i-diff of type \( t \) that contains a subset \( \bar{I}' \) of \( \bar{I} \), the pre-state values for a set of attributes \( \bar{A}' \subseteq \bar{A} \) and the post-state values for a set of attributes \( \bar{A}' \subseteq \bar{A} \). We next describe the semantics for each type of i-diff:

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

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

\[
\text{UPDATE } V \\
\text{SET } A'_{\text{post}} = A'_{\text{post}}_{\text{pre}} \\
\text{FROM } \Delta^\bar{V}_{\text{pre}} \\
\text{WHERE } V.\bar{I}' = \Delta^\bar{V}_{\text{pre}}.\bar{I}' \\
\]

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

\[
\Delta^\bar{V}_{\text{pre}} \begin{array}{c|c|c} \\
| \text{pid} | \text{price}_{\text{pre}} | \text{price}_{\text{post}} \\
\hline \\
| P1 | 10 | 11 \\
\end{array}
\]

leads to the update of the price of both tuples in \( V \) with \( \text{pid} = \text{"P1"} \) 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 \( \Delta^\bar{V}_{\text{pre}} \) for a view \( V(\bar{I}, \bar{A}) \) is a relation instance with schema \( \Delta^\bar{V}_{\text{pre}}(\bar{I}, \bar{A}_{\text{post}}) \), 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 \( \Delta^\bar{V}_{\text{pre}} \) contains a set of tuples that should be inserted into \( V \). Formally, applying \( \Delta^\bar{V}_{\text{pre}} \) has the same effect as applying the following DML statement on \( V \):

\[
\text{INSERT INTO } V \\
\text{SELECT } I, \bar{A}_{\text{post}} \text{ AS } \bar{A} \text{ FROM } \Delta^\bar{V}_{\text{pre}} \\
\text{WHERE ROW(} \bar{I}, \bar{A}_{\text{post}} \text{) NOT IN} \\
\text{(SELECT } I, \bar{A} \text{ FROM } V) \\
\]

Example 2.3. Applying the following insert i-diff

\[
\Delta^\bar{V}_{\text{pre}} \begin{array}{c|c|c} \\
| \text{did} | \text{pid} | \text{price}_{\text{post}} \\
\hline \\
| D3 | P2 | 20 \\
| D4 | P3 | 30 \\
\end{array}
\]

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 does not already exist in \( V \) in the exact same form. This allows multiple insert i-diffs to have try to insert the same tuple.

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

Intuitively, \( \Delta^\bar{V}_{\text{pre}} \) specifies the tuples that should be deleted from \( V \) based on the values of the \( \bar{I}' \) attributes. Formally, applying \( \Delta^\bar{V}_{\text{pre}} \) has the same effect as applying the following DML statement on \( V \):

\[\text{DELETE FROM } V \text{ WHERE } \bar{I}' = \Delta^\bar{V}_{\text{pre}}.\bar{I}'\]

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.
DELETE FROM V
WHERE ROW(I') IN (SELECT I' FROM \(\Delta_v\))

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.

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

\[
\Delta_v \quad \text{pid} \quad P1
\]

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

**Effective i-diff instances.** Given a set of i-diff instances \(\overline{\Delta} \) over a database schema \(DB\), applying them on \(DB\) 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 \(DB\) yields the same result. To this end, we define the notion of effective i-diff instances. Given the pre-state \(I^\text{pre}_R\) and post-state \(I^\text{post}_R\) of a relation \(R\), an i-diff instance \(\Delta_R\) is said to be effective w.r.t. \(I^\text{pre}_R\) and \(I^\text{post}_R\) if for each value of a tuple of \(R\) it refers to it reflects its final value. Formally, it is effective iff it satisfies the following properties:

- If \(\Delta_R\) is an insert i-diff: Every tuple inserted by the i-diff exists in the post-state (i.e., \(\Delta_R \subseteq I^\text{post}_R\)).
- If \(\Delta_R\) is a delete i-diff over schema \(\Delta_R^- (\overline{I'}, \overline{A}_\text{pre})\): Every tuple deleted by the i-diff does not exist in the post-state relational instance (i.e., \(\pi_I \Delta_R^- \cap \pi_I (I^\text{post}_R) = \emptyset\)).
- If \(\Delta_R\) is an update i-diff over schema \(\Delta_R^u (\overline{I'}, \overline{A}_\text{pre}, \overline{A}_\text{post})\): Every tuple updated by the i-diff has all updated attributes \(\overline{A}_\text{post}\) set to the corresponding values in the post-state instance (i.e., \(\pi_{I'} \Delta_R^u (I^\text{post}_R) \subseteq \pi_{I'} \Delta_R^u (\overline{A}_\text{post})\)).

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 operates always 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 \(R(\overline{I}, A_1, A_2)\) 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.

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 falls in the class of deferred IVM approaches [9, 15, 17], maintaining the view lazily at periodic intervals, as well as when the view is queried. As such, it contains modules executed at 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 this 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 Delta-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 Delta-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 Delta-script executor then retrieves the Delta-script corresponding to the view from the Delta-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 Delta-script generation, leaving the discussion on all components used to convert base table modifications to i-diffs for Section 4.

4. **Delta-script generation algorithm**

Given a view definition and a set of base table i-diff schemas, the Delta-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 speedup the IVM) that compute the corresponding view i-diffs and (b) UP-
The $\Delta$-script generator is based on the algebraic IVM approach \cite{21} which consists of operators that can produce an output effective i-diff by transforming an (effective) i-diff over its input schema to an (effective) i-diff over its output schema. Given this information, the $\Delta$-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 selection, join, generalized projection involving functions, grouping with aggregation, antijoin 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.

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 5a and 5b show the view definition $V'$ and a corresponding algebraic plan,

Table 1: Operator ID inference rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Output ID attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SC\ AN(R)$</td>
<td>$key_{pre}(R)$</td>
</tr>
<tr>
<td>$\sigma_{\phi}(R)$</td>
<td>$ID(R)$</td>
</tr>
<tr>
<td>$\pi_{D}(R)$</td>
<td>$ID(R)$ when $ID(R) \subseteq D$</td>
</tr>
<tr>
<td>$R \times S$</td>
<td>$ID(R) \cup ID(S)$</td>
</tr>
<tr>
<td>$R \bowtie_{\phi} S$</td>
<td>$ID(R) \cup ID(S)$</td>
</tr>
<tr>
<td>$R \bowtie_{\phi}^c S$</td>
<td>$ID(R)$</td>
</tr>
<tr>
<td>bag union $R \cup S$</td>
<td>$ID(R) \cup ID(S) \cup {b}$</td>
</tr>
<tr>
<td>group by $\gamma_{G,f(\bar{S})}$ $(R)$</td>
<td>$G$</td>
</tr>
</tbody>
</table>

respectively. The shaded components are annotations inserted by the algorithm, which we explain below:

Pass 1: Inferring ID information for intermediate views. To enable the ID-based approach, the view and all intermediate subviews should contain IDs as part of their output schemas. The IVM algorithm ensures that this is the case, by performing a bottom up postorder traversal of the plan. It iteratively identifies the IDs of a view rooted at operator $p$ from the IDs of the views rooted at $p$'s children. The ID inference for each operator is done by employing an ID inference rule that is supplied with every operator type supported by the system. Table I shows the ID inference rules for the operators considered in this work. If during this process the algorithm fails to find the required IDs in an operator’s output schema, it explains the failure and if instructed, 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.

Pass 2: Instantiating rules for each intermediate operator. To construct the $\Delta$-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 $\Delta^u_{\text{input}}$ over one of its input schemas to an i-diff $\Delta^u_{\text{output}}$ over its output schema through a set of queries known as i-diff propagation rules. These queries can access (a) the operator’s input i-diff $\Delta^u_{\text{input}}$ and (b) the data corresponding to the inputs of the operator or to its output. This is facilitated through the use of the $Input_{i=1,r}$ (standing for left and right input, resp. for binary operators) and Output keywords, respectively. Each of the inputs can be requested either in pre-state or post-state by using the subscripts $pre$ and $post$, respectively. The output is always provided in pre-state.

Example 4.3. For instance, a general grouping and aggregate operator $V = \gamma_{G,f(\bar{S})}$ $(R)$ contains among others the i-diff propagation rule: $\Delta^u_{\text{V}} = \gamma_{G,f(\bar{S})} (\Delta^u_{\text{V}} \bowtie_{\phi}^c Input_{post})$, which semijoins the post-state of the operator’s input data 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.

There are two different classes of operators in idIVM: The first consists of operators which can produce an output effective i-diff by

3To maintain the IDs for 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.
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 $\gamma$ 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 reflects 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. Since in this process it looks at all input i-diffs, it is 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, antisemijoins and aggregation. Rules for aggregation are provided in four different versions (see Tables 10, 11, 12, 13); 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 accommodate 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 13 shows the cache maintenance rules and i-diff propagation rules for the AVG operator.

**Rule instantiation.** In its second pass, the $\Delta$-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 $\Delta \pi R$, 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. 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 $V = \pi_{\bar{x}}(\Delta \pi_{\bar{X}} R)$ the general projection i-diff propagation rule $\Delta \pi V' = \pi_{\bar{x}} f(V) \Rightarrow \Delta \pi_{\bar{x}} R$ becomes $\Delta \pi V' = \pi_{\bar{x}} \Delta \pi_{\bar{X}} R$).

**Example 4.5.** Consider an update i-diff schema $\Delta \pi parts\{pid, price\}_{pre, price\_{post}}$ modeling updates on the price attribute of table parts of our running example. Figure 5 shows the left-hand side of 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.

---

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

---

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 $\Delta$-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 of the tree) 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.

To make the generated plan more efficient, idIVM employs also additional caching, other than the caching used internally by the operators. In particular, whenever an operator rule asks for the base data corresponding to one of its inputs or to its output (through the Input and Output keywords, respectively), idIVM creates 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, the idIVM creates first 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 6a 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 is a $\Delta$-script, composed of queries that compute i-diffs for an intermediate cache/view and APPLY operators that use the DML statements corresponding to the various i-diffs types (shown in Section 3b) 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 \( \text{\textit{\$script}}} \) of Figure 2.

**Pass 4: Optimizing the generated \( \Delta \)-\textit{script}.** As a last step, \( \text{idIVM} \) optimizes the \( \Delta \)-\textit{script} produced in the previous step through two types of optimizations. The first is semantic optimization, which minimizes each individual query included in the plan. In contrast to general minimalizations, this is a minimization that requires knowledge of the IVM setting. Due to lack of space we omit the specifics of this step, explaining it instead through an example.

**Example 4.8.** As an example of semantic non-minimality, consider the plan \( \Delta R \equiv R \), which can be generated by the IVM algorithm. In this case \( R \) is a base table with ID attributes \( I \) and \( \Delta R \) is an update \( \Delta \)-\textit{diff} on \( R \). A general minimization algorithm cannot minimize this expression. However, given the knowledge that \( \Delta R \) is an update \( \Delta \)-\textit{diff} on \( R \), this expression can be minimized to \( \Delta R \).

The second type of optimization is a set of implementation-specific optimizations that improve the efficiency of the \( \Delta \)-\textit{script} by combining adjacent statements where possible. We next outline such an optimization through an example.

**Example 4.9.** The \( \Delta \)-\textit{script} of Figure 2 uses \( \Delta \)-\textit{script} twice: First to update the cache in line 2 and then to join it with the updated cache. To reduce base table accesses, \( \text{idIVM} \) combines these two accesses to the cache into one by using the \textit{UPDATE RETURNING} statement, which updates the cache and returns the result of the update in a single step.

**Designing operator \( \Delta \)-\textit{diff} propagation rules.** The efficiency of the \( \Delta \)-\textit{script} obviously depends on the provided \( \Delta \)-\textit{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).

However, in this work we show that we do not have to get into this reasoning process. Simply creating rules that perform a minimum number of accesses locally typically leads to efficient \( \Delta \)-\textit{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 \( \Delta \)-\textit{diff} dummy tuples that do not affect the view.

**Example 4.10.** For instance the selection operator allows a delete input \( \Delta \)-\textit{diff} to pass through the operator unmodified. However, this means that the output \( \Delta \)-\textit{diff} will also output deletion of tuples that do not satisfy the selection conditions and thus do not exist in the view. Although this is an overestimated \( \Delta \)-\textit{diff}, it does not affect the correctness of the generated \( \Delta \)-\textit{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 \( \Delta \)-\textit{diffs}, \( \text{idIVM} \) maintains the view. In this Section we explain how these base table \( \Delta \)-\textit{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 \( \Delta \)-\textit{diffs} of different schemas, each leading potentially to \( \Delta \)-\textit{scripts} of different efficiencies.

\( \text{idIVM} \) solves the \( \Delta \)-\textit{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 \( \Delta \)-\textit{diff} schema generator deciding at view definition time which base table \( \Delta \)-\textit{diff} schemas to generate. and (c) a base table \( \Delta \)-\textit{diff instance} generator, translating at view maintenance time the modifications recorded in the log to instances of the pre-computed \( \Delta \)-\textit{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 \( \text{idIVM} \)). Therefore we focus next on the other two components.

**Generating \( \Delta \)-\textit{diff schemas.** Given a view definition \( V \), \( \text{idIVM} \) generates suitable base-table \( \Delta \)-\textit{diff} schemas for all base tables mentioned in \( V \). Insertions and deletions are straightforward cases: Consider a base table \( R(I,A) \) with key attributes \( I \) and non-key attributes \( A \). For each such table, the \( \Delta \)-\textit{diff} schema generator creates a single insert \( \Delta \)-\textit{diff} schema \( \Delta R(I,A_{post}) \) (containing all attributes of \( R \)) and a single delete \( \Delta \)-\textit{diff} schema \( \Delta R(I,A_{pre}) \) (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 \( \Delta \)-\textit{script} as they may reduce overestimation and the respective 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 \( \Delta \)-\textit{diff} that do not satisfy the condition.

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

In \( \text{idIVM} \) we solve this problem by observing that the base table attributes can be divided into sets of attributes whose updates lead to the same \( \Delta \)-\textit{script} and can thus be grouped together in a single \( \Delta \)-\textit{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.) 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 \( \Delta \)-\textit{script} up to projections. Updates on each set of conditional attributes \( C_{op} \) on the other hand may lead to a different \( \Delta \)-\textit{script}, since the updated values may affect whether the \( \Delta \)-\textit{diffs} make it through \( op \)’s condition. Therefore for each base table \( R(I,A) \), the \( \Delta \)-\textit{diff} schema generation algorithm

![Figure 7: \( \Delta \)-\textit{script} for running example](image_url)

\[
\begin{align*}
1 & \Delta \text{\textit{script}}_{\text{cache}} = \Delta \text{\textit{script}}_{\text{parts}}; \\
2 & \text{APPLY } \Delta \text{\textit{script}}_{\text{Cache}}; \\
3 & \Delta \text{\textit{script}}_{\text{V}} = \pi \text{\textit{id},\textit{cost}} \rightarrow \text{\textit{cost}}_{\text{pre}}, \text{\textit{cost}}_{\text{post}} \Delta \rightarrow \text{\textit{cost}}_{\text{pre}}(V' \equiv \\
& \pi \text{\textit{id},\textit{sum}}(\text{\textit{price}}_{\text{pre}}) \rightarrow \text{\textit{price}}_{\text{pre}}, \text{\textit{price}}_{\text{post}} \rightarrow \text{\textit{price}}_{\text{post}}, \text{\textit{price}}_{\text{pre}})(\Delta \text{\textit{script}}_{\text{Cache}} \equiv \pi \text{\textit{price}}_{\text{post}} \rightarrow \text{\textit{price}}_{\text{post}}(\text{\textit{Cache}})); \\
4 & \text{APPLY } \Delta \text{\textit{script}}_{\text{V}}; \\
\end{align*}
\]
creates (a) for each set $C_{op}$ an update i-diff $\Delta^c_{op}(\bar{I}, \bar{A}_{pre}, \bar{A}_{post})$, s.t. $\bar{A}' = \bar{A} \cap C_{op}$ and (b) an additional additional update i-diff $\Delta^c_{op}(\bar{I}, \bar{A}_{pre}, \bar{A}'_{post})$, containing the non-conditional attributes of $R$ (i.e., $\bar{A}' = \bar{A} \setminus NC$).

**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 3 and 4 this is crucial for the algorithm’s correctness.

6. PERFORMANCE ANALYSIS

ID-based IVM outperforms tuple-based IVM under listed common assumptions. The following analysis defines two factors, named $i-diff$ compression and accesses per view i-diff tuple, which typically determine the performance ratio of ID-based IVM to tuple-based IVM. 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 necessitates access to the view modifications from the log, the algorithm combines multiple updates to the same tuple to a single modification, so as to generate effective diffs. As discussed in Sections 3 and 4 this is crucial for the algorithm’s correctness.

### 6.1 SPJ Views

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

`SELECT S FROM R, R_{1}, \ldots, R_{n} WHERE c`

whose FROM clause involves a single alias of a table $R$, (b) an update i-diff on attributes $\bar{A}'$ of $R$ and (c) a corresponding i-diff:

$$\Delta^c_{R} = \Delta_{R}(\bar{I}, \bar{A}_{pre}, \bar{A}'_{post})$$

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 $c_{pre}$, $c_{rest}$ 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 $\pi_{\bar{S}}(\sigma_{c_{pre}} \sigma_{c_{rest}} E)$, where $E = \sigma_{c_{rest}}(R_1 \times \ldots \times R_n)$.

In general, the i-diff $\Delta_{spj}$ (resp. t-diff $\Delta_{R}$) will lead to $\Delta^c_{V_{spj}}$, $\Delta^c_{V_{spj}}$ and $\Delta^c_{V_{spj}}$ in the ID-based approach (resp. $\Delta^c_{V_{spj}}$, $\Delta^c_{V_{spj}}$ and $\Delta^c_{V_{spj}}$ in the tuple-based approach). Due to space limitations, we present here the characteristic case where the attributes $\bar{A}'$ do not participate in the join condition $c_{rest}$ or the selection condition $c_{R}$. It is easy to see that in this case, the update diff on $R$ will only lead to an update diff on $V_{spj}$. Furthermore, since the selection $\sigma_{c_{rest}}$ simply filters out tuples of $\Delta_{R}$ it has the same effect as using an initial diff with fewer tuples, and is therefore ignored in the rest of the analysis.

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

<table>
<thead>
<tr>
<th>ID-based approach</th>
<th>Tuple-based approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta^c_{V_{spj}} = \Delta^c_{R}$</td>
<td>$\Delta^c_{V_{spj}} = \pi_{\bar{S}}(\sigma_{c_{rest}} E)$</td>
</tr>
<tr>
<td>APPLY $\Delta^c_{V_{spj}}$</td>
<td>APPLY $\Delta^c_{V_{spj}}$</td>
</tr>
</tbody>
</table>

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 $\Delta^c_{V_{spj}}$ (respectively $\Delta^c_{V_{spj}}$) and the view modification cost, which is the cost of applying the modifications dictated by $\Delta^c_{V_{spj}}$ (respectively $\Delta^c_{V_{spj}}$) on the materialized view. We measure both costs in terms of block accesses to indices and tuples. Employing common assumptions on the index structure, the cost of retrieving (using an index) the $m$ tuples whose $X$ attributes have given $x$ 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 $\Delta^c_{R}$ 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 $\Delta^c_{V_{spj}} = \pi_{\bar{S}}(\sigma_{c_{rest}} E)$. We consider the common case where the join condition $c_{rest}$ 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 $\Delta^c_{V_{spj}}$. Since the diff-table $\Delta_{R}$ 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 $\Delta_{R}$ it executes the subplan $\sigma_{c_{rest}} E$, where $\sigma_{c_{rest}} E$ is an instantiation of the $c_{rest}$ 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 $\Delta_{R}$, i.e., the average number of accesses in each execution of $\sigma_{c_{rest}} E$. Then the diff computation cost of the tuple-based approach is $|\Delta_{R}|a$. The DBMS may also choose to evaluate the 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 $\Delta^c_{V_{spj}}$ (resp. $\Delta^c_{V_{spj}}$) 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., $|\Delta^c_{V_{spj}}| = |\Delta^c_{R}|$ lookups for the ID-based and $|\Delta^c_{V_{spj}}|$ lookups for the tuple-based approach, respectively). Once the to-be-modified view tuples have been identified (which are in both cases equal to $|\Delta^c_{V_{spj}}|$), both approaches will incur $|\Delta^c_{V_{spj}}|$ view tuple accesses to update them. Table 3 shows the view index lookups and view tuple accesses for each approach utilizing the i-diff compression factor $p = |\Delta^c_{V_{spj}}|/|\Delta^c_{V_{spj}}|$. The diff computation cost is zero. On the other hand, the cost of the tuple-based IVM varies widely depending on the computation of $\Delta^c_{V_{spj}} = \pi_{\bar{S}}(\sigma_{c_{rest}} E)$. We consider the common case where the join condition $c_{rest}$ 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 $\Delta^c_{V_{spj}}$. Since the diff-table $\Delta_{R}$ 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 $\Delta_{R}$ it executes the subplan $\sigma_{c_{rest}} E$, where $\sigma_{c_{rest}} E$ is an instantiation of the $c_{rest}$ 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 $\Delta_{R}$, i.e., the average number of accesses in each execution of $\sigma_{c_{rest}} E$. Then the diff computation cost of the tuple-based approach is $|\Delta_{R}|a$. The DBMS may also choose to evaluate the 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 $\Delta^c_{V_{spj}}$ (resp. $\Delta^c_{V_{spj}}$) 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., $|\Delta^c_{V_{spj}}| = |\Delta^c_{R}|$ lookups for the ID-based and $|\Delta^c_{V_{spj}}|$ lookups for the tuple-based approach, respectively). Once the to-be-modified view tuples have been identified (which are in both cases equal to $|\Delta^c_{V_{spj}}|$), both approaches will incur $|\Delta^c_{V_{spj}}|$ view tuple accesses to update them. Table 3 shows the view index lookups and view tuple accesses for each approach utilizing the i-diff compression factor $p = |\Delta^c_{V_{spj}}|/|\Delta^c_{V_{spj}}|$. 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.
To ease exposition, we isolate the aggregation operator of the view, expressing it through the algebraic plan \( V_{agg} = \gamma_{\bar{G},f(X)} \rightarrow g \) where \( V_{spj} \) is the algebraic plan for the SPJ query presented in Section 6.4.

### 6.2 Aggregate Views

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

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

whose FROM clause involves a single alias of \( R \), and \( f \) is an associative aggregation function such as \( \sum \) (b) an update t-diff on attributes \( \bar{A} \) of \( R \), and (c) a corresponding i-diff:

\[
\Delta_{\bar{R}} = \Delta_{\bar{R}}(\bar{I}, A_{\text{pre}}, A_{\text{post}})
\]

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

To ease exposition, we isolate the aggregation operator of the query, expressing it through the algebraic plan \( V_{agg} = \gamma_{\bar{G},f(X)} \rightarrow g \) where \( V_{spj} \) is the algebraic plan for the SPJ query presented in Section 6.4.

### 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 \( f_\Delta \) that inputs \( \Delta V_{spj}(S, X_{\text{pre}}, X_{\text{post}}) \) where \( S \) is the key of \( V_{spj} \), and outputs \( \Delta V_{agg}(\bar{G}, g_{\text{pre}}, g_{\text{post}}) \). For example, when \( f \) is the \( \sum \) function, \( f_\Delta \) is also \( f \), since \( \sum \) is an associative aggregation function. Given \( f_\Delta \), the tuple-based approach computes \( \Delta V_{agg} = \gamma_{\bar{G},f_\Delta(X)} \rightarrow g \Delta V_{spj} \).

Given that \( |\Delta \bar{R}| \) is much smaller than base tables, the number of groups in \( \Delta V_{agg} \) will similarly be smaller than the number of groups in \( V_{agg} \). The efficient physical implementation for \( \gamma \) is thus hash aggregation with in-memory buckets, which can be pipelined. Due to pipelining, no additional block accesses are incurred for \( \gamma \). Thus, the tuple-based approach has the same diff computation cost for \( V_{spi} \) and \( V_{agg} \).

Recall from Example 4.9 that 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, \( \Delta V_{spi} \) is obtained without additional accesses over cache modification costs. Similar to the tuple-based approach, \( \Delta V_{agg} = \gamma_{\bar{G},f_\Delta(X)} \rightarrow g \Delta V_{spj} \), and the \( \gamma \) uses pipelined hash aggregation. Thus, the ID-based approach also has the same diff computation cost for \( V_{spi} \) and \( V_{agg} \).

### View modification cost.

To apply \( \Delta V_{agg} \) (resp. \( \Delta V_{agg} \)) 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 \( g = |\Delta V_{agg}| / |\Delta V_{spj}| \) in Table 3.

\[
\text{Speedup ratio for } V_{agg} = \frac{a + 2pg}{1 + p + 2pg}
\]

### Table 1: Summarizing the Costs for both ID-based and Tuple-based IVM on \( V_{agg} \)

<table>
<thead>
<tr>
<th>Costs</th>
<th>ID-based</th>
<th>Tuple-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diff computation</td>
<td>(</td>
<td>\Delta \bar{R}</td>
</tr>
<tr>
<td>View index lookups</td>
<td>(</td>
<td>\Delta \bar{R}</td>
</tr>
<tr>
<td>View tuple accesses</td>
<td>(</td>
<td>\Delta \bar{R}</td>
</tr>
</tbody>
</table>

### 7. Experimental Evaluation

To compare the performance of ID-based and tuple-based IVM in real-life and check whether the cost model of Section 6.3 correctly models reality, we used both systems to incrementally maintain multiple variants of the view of Figure 5a, created by varying different parameters of the view definition and the underlying data. For each configuration, we used idIVM to generate the corresponding \( \Delta \)-script and a modified version of it using the tuple-based rules to create the corresponding \( \Delta \)-script. We then measured the times to run each script. As we will see, the cost model predicts the effect
of each parameter. Furthermore, its estimated speedup ratio closely approximates the actual speedup exhibited in the experiments.

The experimental data comprises 5M parts tuples (170MB), 5M devices tuples (170 MB) and 50M devices_parts tuples (3GB). The respective fanouts \( f \) from entity tables devices / parts to association table devices_parts are both 10, i.e., each device has 10 parts, and each part is used in 10 devices. The selectivity \( s \) of \( \sigma_{\text{category} = \text{phone}} \) is 20%, i.e., the selection outputs 20% of devices tuples. We consider the view maintenance time when 200 parts tuples are updated with new prices. That is, we consider an \( i \)-diff and corresponding \( t \)-diff \( \Delta w^p \) s.t. \( \sigma^p \rightarrow \Delta \rightarrow \sigma^p \)

All experiments are 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 is run with cold PostgreSQL page buffers and Linux disk buffers, which is the common case for tasks that periodically update materialized views of data warehouses.

Figure 7 shows the ID-based \( \Delta \)-script, whereas Figure 8 shows the tuple-based \( \Delta \)-script. The latter has the following key differences from the former: (1) There is no intermediate cache, thus no cost is incurred for cache diff computation and cache update. (2) Computing the \( t \)-diff requires joins with base tables (line 1e) and selections (line 1d), which incurs costs for view diff computation. Figure 9 shows the view maintenance time for ID-based IVM versus tuple-based IVM, and their respective cost breakdowns. Column A (left) represents ID-based IVM, with the top stack (green) corresponding to cache update time, and the bottom stack (dark blue) corresponding to view update time. No stack is shown for the cache/view diff computation time as both are negligible. Column B (right) represents tuple-based IVM, with the top stack (light blue) corresponding to view diff computation time, and the bottom stack (dark blue) corresponding to view update time. All data points show that, as predicted by the cost model, the view update time is the same for both approaches.

Finally, Figure 9d illustrates the effects of varying the fanout \( f \) linearly between 5 to 25. By instantiating \( p = 0.2f \), \( a = 1 + 3f \) and \( g = 1 \), the estimated speedup simplifies to \( 1.66 f \), which approaches 5.66 as \( f \) increases. As predicted by the cost model, the actual speedup also converges, albeit at the lower bound of 4.0, due to buffering as explained above.

In summary, the ID-based approach performs in our experimental results consistently better than the tuple-based approach with a speedup that it closely approximated by the cost model. Note that this is the case even though the experimental conditions were designed to explicitly benefit the tuple-based IVM. In particular, (1) we assumed the existence of appropriate indices on the base tables to speedup the tuple-based joins (without counting the associated index maintenance cost) and (2) 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.

8. RELATED WORK

IVM is a long studied problem with a lot of influential works [6] [7] [21] [13]. idIVM falls under the category of IVM works that employ the algebraic [21] [10] [22] [18] and deferred [2] [15] [17] 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, regardless of whether they follow the algebraic or deferred approaches. For comprehensive surveys on IVM, the reader is referred to [1][2][8].

Exploiting primary key constraints. The idea of exploiting primary key constraints to speed up IVM was first presented in [1][23]. However, in contrast to our work [1][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 SPI (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 [3] 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 [23][29] 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 [26]. 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 [23][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 minimizes the cost of IVM for a single diff-tuple by materializing a large number of intermediate views. However, by not being ID-based this approach always accesses 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 the present paper insert i-diffs incur the same base table accesses as tuple-based approaches, 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. REFERENCES


Table 4: Rules for $V = R \times S$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta v_I^R(I, A_{post})$</td>
<td>$\Delta v = \pi_0 \rightarrow \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_I^S(I, A_{post})$</td>
<td>$\Delta v = \pi_{1-1} \Delta S$</td>
</tr>
<tr>
<td>$\Delta v_R(I, A_{pre})$</td>
<td>$\Delta v = \pi_0 \rightarrow \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_S(I, A_{pre})$</td>
<td>$\Delta v = \pi_{1-1} \Delta S$</td>
</tr>
<tr>
<td>$\Delta v_I^R(I, A_{pre}^\prime)$</td>
<td>$\Delta v_I^R = \Delta v_S^R$</td>
</tr>
<tr>
<td>$\Delta v_I^S(I, A_{pre}^\prime)$</td>
<td>$\Delta v_I^S = \Delta v_S^S$</td>
</tr>
<tr>
<td>$\Delta v_R(I, A_{pre}^\prime)$</td>
<td>$\Delta v_R = \pi_0 \rightarrow \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_S(I, A_{pre}^\prime)$</td>
<td>$\Delta v_S = \pi_{1-1} \Delta S$</td>
</tr>
</tbody>
</table>

Table 5: Rules for $V = R \cup S$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta v_I^R(I, A_{post})$</td>
<td>$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_I^S(I, A_{post})$</td>
<td>$\Delta v_I^S = \sigma_{\phi}(X_{pre}) \Delta S$</td>
</tr>
<tr>
<td>$\Delta v_R(I, A_{pre})$</td>
<td>$\Delta v_R = \sigma_{\phi}(X_{pre}) \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_S(I, A_{pre})$</td>
<td>$\Delta v_S = \sigma_{\phi}(X_{pre}) \Delta S$</td>
</tr>
</tbody>
</table>

if $X \subseteq I \cup A_{post}$ then
$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \sigma_{\phi}(X) \Delta R$
else
$\Delta v_I^R = \Delta R$

if $I \cap A_{post} = \emptyset$ then
$\Delta v_I^R = \emptyset$
else if $I \cup A_{post}^\prime$ is set of attributes of $R$ then
$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \sigma_{\phi}(X) \Delta R$
else if $X \subseteq I \cup A_{post}^\prime$ then
$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \sigma_{\phi}(X) \Delta R$
else
$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \sigma_{\phi}(X) \Delta R$

Blue portion applies when pre-state attributes present.

Table 6: Rules for $V = \sigma_{\phi}(X)(R)$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta v_I^R(I, A_{post})$</td>
<td>$\Delta v_I^R = \pi_{D_I(X_{pre})} \sigma_{\phi}(X) \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_I^S(I, A_{post})$</td>
<td>$\Delta v_I^S = \sigma_{\phi}(X_{pre}) \Delta S$</td>
</tr>
<tr>
<td>$\Delta v_R(I, A_{pre})$</td>
<td>$\Delta v_R = \sigma_{\phi}(X_{pre}) \Delta R$</td>
</tr>
<tr>
<td>$\Delta v_S(I, A_{pre})$</td>
<td>$\Delta v_S = \sigma_{\phi}(X_{pre}) \Delta S$</td>
</tr>
</tbody>
</table>

if $X \subseteq I \cup A_{pre}$ then
$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \sigma_{\phi}(X) \Delta R$
else
$\Delta v_I^R = \emptyset$

if $(I \cup A_{pre}) \cap X = \emptyset$ then
$\Delta v_I^R = \sigma_{\phi}(X_{pre}) \sigma_{\phi}(X) \Delta R$
else if $X \subseteq I \cup A_{pre}^\prime$ then
$\Delta v_I^R = \sigma_{\phi}(X) \sigma_{\phi}(X) \Delta R$
else
$\Delta v_I^R = \sigma_{\phi}(X) \sigma_{\phi}(X) \Delta R$

Blue portion applies when pre-state attributes present.

Table 7: Rules for $V = \pi_{D_I(X_{pre})} R$
Treat input update as combination of insert and delete

For $\Delta_R(I, A_{post})$

$\Delta_{V} = \Delta_{R} \cup \Delta_{S} S$

For $\Delta_R(I, A_{pre})$

$\Delta_{V} = \Delta_{R} \cup \Delta_{S} S$

For $\Delta_R(I, A_{pre}, A_{post})$

$\Delta_{V} = \Delta_{R} \cup \Delta_{S} S$

if $I \cup A_{pre} = \text{set of attributes of } R$ then

$\Delta_{V} = \pi_{R}(\Delta_{R} \cup \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$
else if $X \subseteq I \cup A_{pre}''$ then

$\Delta_{V} = R \times \pi_{R}(\Delta_{R} \cup \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$
else

$\Delta_{V} = R \times (\text{Inputpost} \times \Delta_{R} \times \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$

if $X \subseteq I \cup A_{pre}''$ then

$\Delta_{V} = \pi_{R}(\Delta_{R} \times \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$
else

$\Delta_{V} = \pi_{R}(\text{Inputpost} \times \Delta_{R} \times \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$

For $\Delta_S(I, A_{post})$

$\Delta_{V} = \pi_{R}(\Delta_{R} \times \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$

For $\Delta_S(I, A_{pre})$

if $Y \subseteq I \cup A_{pre}$ then

$\Delta_{V} = (R \times \phi_{(X_{post}, Y)} \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$
else

$\Delta_{V} = (R \times \phi_{(X_{post}, Y)} \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$

For $\Delta_S(I, A_{pre}, A_{post})$

$\Delta_{V} = \pi_{R}(\Delta_{R} \times \Delta_{S} S \circ \phi_{(X_{post}, Y)}) S$

Treat input update as combination of insert and delete

Table 8: Rules for $V = R \times \phi_{(R, X, Y)} S$

For $\Delta_R(I, A_{post})$

$\Delta_{V} = \Delta_{R} \times \Delta_{S} S$

For $\Delta_R(I, A_{post})$

$\Delta_{V} = \Delta_{R} \times \Delta_{S} S$

For $\Delta_R(I, A_{pre})$

$\Delta_{V} = \Delta_{R} \times \Delta_{S} S$

For $\Delta_R(I, A_{pre}, A_{post})$ ($\Delta_{S}$ is symmetric)

$\Delta_{V} = \phi_{(X_{pre}, \phi_{(X, Y)})} \Delta_{R}$

For $\Delta_R(I, A_{pre}, A_{post})$ ($\Delta_{S}$ is symmetric)

if $X \subseteq I \cup A_{pre}''$ then

$\Delta_{V} = \phi_{(X_{pre}, \phi_{(X, Y)})} \Delta_{R}$
else

$\Delta_{V} = \Delta_{R}$

if $I \cap A_{post} = \emptyset$ then

$\Delta_{V} = \emptyset$
else if $I \cap A_{post}'' = \text{set of attributes of } R$ then

$\Delta_{V} = \pi_{I \cup A_{pre}} \sigma_{\phi_{(X_{pre})}} \phi_{(X)} \Delta_{R} \times \phi_{(X)} S$
else if $X \subseteq I \cup A_{post}''$ then

$\Delta_{V} = \pi_{I \cup A_{pre}} \sigma_{\phi_{(X_{pre})}} \phi_{(X)} \Delta_{R} \times \phi_{(X)} S$
else

$\Delta_{V} = \pi_{I \cup A_{pre}} \sigma_{\phi_{(X_{pre})}} \phi_{(X)} \Delta_{R} \times \phi_{(X)} S$

if $I \cap A_{post} = \emptyset$ then

$\Delta_{V} = \emptyset$
else if $X \subseteq I \cup A_{post}''$ then

$\Delta_{V} = \pi_{I \cup A_{pre}} \sigma_{\phi_{(X_{pre})}} \phi_{(X)} \Delta_{R}$
else

$\Delta_{V} = \pi_{I \cup A_{pre}} \sigma_{\phi_{(X_{pre})}} \phi_{(X)} \Delta_{R}$

Blue portion applies when pre-state attributes present.

Table 9: Rules for $V = R \times \phi_{(X)} S$

For $\Delta_R(I, A_{pre}, A_{post})$ where $I \subseteq G$

$\Delta_{V} = \Delta_{R}$

For any $\Delta_R(I, A')$ and $f$, we can recompute groups

if $G \subseteq (I \cup A')$ then

$\Delta_{V} = \gamma_{G,f(X)}(\Delta_{R} \times \text{Inputpost})$
else

$\Delta_{V} = \gamma_{G,f(X)}(\Delta_{R} \times f \times \text{Inputpost})$

(Do not handle group creation/deletion)

Table 10: Rules for $V = \gamma_{G,f(X)}(R)$

For $\Delta_R(I, A_{pre}, A_{post})$, and $G \cap A_{post} = \emptyset$

$\Delta_{V} = \pi_{I \times \phi_{(X_{pre}, \phi_{(X, Y)})}} \Delta_{R} \times \phi_{(X_{pre}, \phi_{(X, Y)})}$

For $\Delta_R(I, A_{pre})$

$\Delta_{V} = \pi_{I \times \phi_{(X_{pre}, \phi_{(X, Y)})}} \Delta_{R} \times \phi_{(X_{pre}, \phi_{(X, Y)})}$

For $\Delta_R(I, A_{pre})$

$\Delta_{V} = \pi_{I \times \phi_{(X_{pre}, \phi_{(X, Y)})}} \Delta_{R} \times \phi_{(X_{pre}, \phi_{(X, Y)})}$

For converting $\Delta$ to output update i-diffs

Happens after all $\Delta_{1}, \Delta_{2}, \Delta_{3}$ are computed.

$\Delta_{V} = \gamma_{G,c,e \rightarrow e_{pre}, c_{e \rightarrow e_{post}}} \times \text{Output}$

(Do not handle group creation/deletion)

Table 11: Rules for $V = \gamma_{G,sum(x_{pre}) \rightarrow c_{e_{post}}} (R)$

For $\Delta_R(I, A_{pre}, A_{post})$

$\Delta_{V} = \Delta_{R} \times \Delta_{S}$

For $\Delta_R(I, A_{pre})$

$\Delta_{V} = \Delta_{R} \times \Delta_{S}$

For $\Delta_R(I, A_{pre})$

$\Delta_{V} = \Delta_{R} \times \Delta_{S}$

For converting $\Delta$ to output update i-diffs

Happens after all $\Delta_{1}, \Delta_{2}, \Delta_{3}$ are computed.

$\Delta_{V} = \gamma_{G,c,e \rightarrow e_{pre}, c_{e \rightarrow e_{post}}} \times \text{Output}$

(Do not handle group creation/deletion)

Table 12: Rules for $V = \gamma_{G,count(x_{pre}) \rightarrow c_{e_{post}}} (R)$

Operator cache schemas:

$\text{Cache}_{sum}(G, c_{2}), \text{Cache}_{count}(G, c_{1})$

Cache maintenance rules:

For $\Delta_{sum}^c$, Use rules of $\gamma_{G,sum(x_{pre}) \rightarrow c_{e_{post}}} (R)$ (Table 11)

For $\Delta_{count}^c$, Use rules of $\gamma_{G,count(x_{pre}) \rightarrow c_{e_{post}}} (R)$ (Table 12)

i-diff propagation rules:

$\Delta_{V} = \gamma_{G,c_{2}/c_{1} \rightarrow e_{pre}, c_{e \rightarrow e_{post}}}(c_{1} \rightarrow c_{2}) \times \text{Output}$

(Do not handle group creation/deletion)

Table 13: Rules for $V = \gamma_{G,avg(x_{pre}) \rightarrow c_{e_{post}}} (R)$