# Immediate Materialized Views with Outerjoins

Anisoara Nica
Sybase iAnywhere
Waterloo, Ontario, Canada
anica@sybase.com

## ABSTRACT

Queries using outerjoins appear very frequently in traditional applications such as data warehousing. Lately, they have been widely used in newly emerged systems such as Object-Relational Mapping (**ORM**) tools, schema integration and information exchange systems, and probabilistic databases. Materialized views using outerjoins are allowed in many database management systems, but without support for their incremental maintenance. In this paper we present the algorithms used in SQL Anywhere **RDBMS** for the incremental maintenance of materialized views with outerjoins. The algorithms achieve the following improvements over the previous work with respect to the class of materialized outerjoin views which can be incrementally maintained, and with respect to the performance of the view updates:
(1) Relax the requirement for the existence of the primary key attributes in the select list of the view to only some of the relations (namely only the relations referenced as a preserved side in an outerjoin predicate).
(2) Relax the null-intolerant property requirement for only some predicates used in the view definition (namely, those outerjoin predicates referencing relations which can be null-supplied by another nested outerjoin).
(3) The maintenance of outerjoin views is implemented by using exactly one update statement per view for each relation referenced in the view.

Another main characteristic of the algorithms is that they allow the design and implementation of the incremental maintenance of materialized views with outerjoins to be easily integrated into the SQL Anywhere Optimizer by relying on the normalized join tree representation used for optimizing queries with outerjoins.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; H.2.7 [**Database Management**]: Database Administration—*Data warehouse and repository*

## General Terms

Algorithms,Design,Theory



## Keywords

Outerjoin Views, Incremental Maintenance, View Maintenance, Materialized Views, Query Optimization, **SQL MERGE** Statement, SQL Anywhere

## 1. INTRODUCTION

SQL Anywhere [1][1] is an ANSI SQL-compliant RDBMS designed to run on a variety of platforms from server-class installations to mobile devices using the Windows Mobile operating system. SQL Anywhere is a self-managing RDBMS with high reliability, high performance, synchronization capabilities, small footprint, and a full range of SQL features across a variety of 32- and 64-bit platforms.

SQL Anywhere first introduced materialized views in SQL Anywhere 10.0 by supporting manual materialized view which can only be refreshed by complete recomputation but which can be defined by any complex query. SQL Anywhere 11.0 added support for incremental maintenance of the materialized views for **GROUP-SELECT-PROJECT-JOIN** views. In SQL Anywhere , materialized views which can be incrementally maintained are called *immediate Materialized Views* (iMVs). The algorithms presented in this paper are designed for and implemented in SQL Anywhere 12.0 which supports an extended class of immediate materialized views, namely outerjoin views with and without aggregation.

Outerjoin queries are used more and more frequently in new systems and external tools where **DBAs** or experienced database developers are not at hand to fine-tune the generated **SQL** statements. The SQL Anywhere Optimizer [7, 8, 9] has sophisticated techniques for processing outerjoin queries from semantics transformations to view matching using outerjoin views. It is then a necessity to extend our support to incremental maintenance of materialized views with outerjoins, as this can speed up many of the applications using the SQL Anywhere **RDBMS**. The goals for an efficient support of immediate materialized views with outerjoins are multifold. Firstly, the new algorithms must be easy to integrate in the current design of the SQL Anywhere Optimizer and the existing support for immediate materialized views. The update of iMVs is achieved using internally generated triggers, one for each relation referenced in an iMV. Internally generated triggers contain update statements which, given a $\Delta T$ update relation, perform updates to the affected iMVs. The update statements are **SQL** statements internally generated

[1]Sybase and SQL Anywhere are trademarks of Sybase Inc. Other company or product names referenced in this paper are trademarks and/or servicemarks of their respective companies.

which are processed and optimized by SQL Anywhere server just like any other query taking full advantage of query optimization techniques present in the SQL Anywhere Optimizer. Hence, the incremental maintenance of iMVs with outerjoins must be done in a similar fashion which we achieve via **MERGE**, **INSERT**, **DELETE**, and **UPDATE** statements [5] with embedded update statements aka DML-derived-tables. Secondly, given the current landscape of the applications using our server, the restrictions imposed to the class of the outerjoin views which can be immediately maintained have to be kept to a minimum. We find that the previous solutions were very restrictive with respect to the content of the select list and the null-intolerant properties imposed to the predicates. Thirdly, the extra information needed by the new generation algorithms for outerjoin update statements has to be efficiently added to the SQL Anywhere Optimizer internal representation of the outerjoin query blocks.

The $\mathcal{PSNS}$() (*P*reserved *S*ide / *N*ull-supplied *S*ide) Algorithm presented in this paper achieves exactly that.

The rest of the paper is organized as follows. Section 2 introduces the main definitions and notations used in later sections. Section 3 describes the algorithms for building the $\mathcal{PSNS}$-annotated normalized join tree, and how these normalized trees are used to derive maintenance formulas for materialized views with outerjoins. We discuss the current implementation of the incremental maintenance of materialized views with outerjoins in SQL Anywhere **RDBMS** in Section 4. Section 5 presents related work while Section 6 concludes the paper.

## 2. PRELIMINARY

We denote by $Schema(T)$ all the attributes of a relation $T$. A tuple $t$ over the $Schema(T)$ is an assignment of values to attribute names of $Schema(T)$. For a tuple $t$ defined over the $Schema(T_1, \ldots, T_n)$ we will use the notation $t[T_i]$ to denote the values of the tuple $t$ for the attributes of the relation $T_i$. We will use the notation $n(t[T])$ to denote that *all* attributes in the $Schema(T)$ have $\mathcal{N}ull$ in $t$, while $nn(t[T])$ denotes that *at least one* attribute in the $Schema(T)$ is not $\mathcal{N}ull$. $t = (\ldots, n(T), \ldots, nn(R), \ldots)$ indicates that the tuple $t$ is $\mathcal{N}ull$ on $T$ and not-null on $R$.

A predicate $p$ is defined over a set of attributes identified by $Schema(p)$. We will use the notation $p(T_1, \ldots, T_n)$ to denote a predicate referencing some attributes of the relations $T_1, \ldots, T_n$, i.e., $Schema(p) \subseteq \bigcup_{i=1,n} Schema(T_i)$, and $Rels(p) = \{T_1, \ldots, T_n\}$. A predicate $p$ applied to the tuples $t_1 \in T_1, \ldots, t_n \in T_n$ can have three values $False$, $True$ or $Unknown$.

The *outerunion* of the two relations $T_1$ and $T_2$ is denoted by $T_1 \uplus T_2$, and it is computed by first padding the tuples of each relation $T_i, i = 1, 2$ with $\mathcal{N}ull$ for the attributes in $(Schema(T_1) \bigcup Schema(T_2)) \setminus Schema(T_i)$, and then computing the union of the resulting sets [2]. For two relations $T_1$ and $T_2$ the join, left, right, and full outerjoins are defined as follows, respectively:

$R \bowtie_{p(R,T)} T = \{(r, t) | r \in R, t \in T, p(r, t)\}$

$R \overset{lo}{\bowtie}_{p(R,T)} T = (R \bowtie_{p(R,T)} T) \uplus (R \triangleright_{p(R,T)} T)$

$R \overset{ro}{\bowtie}_{p(R,T)} T = T \overset{lo}{\bowtie}_{p(R,T)} R$

$R \overset{fo}{\bowtie}_{p(R,T)} T = (R \bowtie_{p(R,T)} T) \uplus (R \triangleright_{p(R,T)} T) \uplus (T \triangleright_{p(R,T)} R)$

We call a tuple for which all attributes of a relation $T$ are null-padded or null-extended, a *$T$-null-supplied* tuple, and a tuple where $T$ is not null-supplied a *$\neg T$-null-supplied* tuple. We will use the notion of a tuple $t$ *dominating* a tuple $r$ if they are defined on the same schema, and $t[A] = r[A], \forall A \in Schema(t)$ for which $r[A]$ is not null. If $\forall A \in Schema(t)$, $t[A]$ is not distinct from $r[A]$, i.e., they are both $\mathcal{N}ull$ or they are equal, then we say that $t$ is a *duplicate* of $r$. We will use the definition of the *best match operator* $\beta$ as defined in [10] to be $\beta(R) = \{r | r \in R, r$ is not dominated or duplicated by any tuple in $R$ and it has at least one non null value $\}$, and $\delta(R)$ for the *duplicate elimination operator*.

*Definition 1.* A predicate $p(T_1, \ldots, T_n)$ is *$\mathcal{NS}$-intolerant* on the relation $T \in \{T_1, \ldots, T_n\}$ if the predicate doesn't evaluate to $True$ for tuples which are *$T$-null-supplied*.

We want to differentiate between a $\mathcal{NS}$-intolerant (*Null-Supplied*-intolerant) predicate and a *strong* predicate defined in previous work [6, 3] which is very restrictive. Namely, a strong predicate $p$ is required to be null-intolerant on any attribute in $Schema(p)$. For example, $T.X$ **IS NOT DISTINCT FROM** $R.X$[2] **AND** $rowid(T)$ **IS NOT NULL** is not a strong predicate according to [6, 3] but it is $\mathcal{NS}$-intolerant on the relation $T$ according to our definition. In our experience, a large class of customer queries do not use strong predicates even in the **ON** clauses. For example, a typical **ON** clause for **SQL** statements generated by an Object-Relational Mapping system is a null tolerant predicate of the form **ON ([PR8].[CID] = [EX1].[CID]) OR (([PR8].[CID] IS NULL) AND ([PR2].[CID] IS NULL))**. When the property of a predicate to be $\mathcal{NS}$-intolerant for a certain relation $T_i$ is important, we will denote this property by underlying the relation $T_i$, e.g., $p(T_1, \ldots, \underline{T_i}, \ldots, T_n)$.

An *outerjoin* query is a query which contains left and full outerjoins (the right outerjoins are transformed into left outerjoins), and inner joins. An outerjoin query is represented by a join operator tree whose internal nodes are joins and the leaves are relations. For a join node of type left outerjoin, we say that the outerjoin *null-supplies* the relations from its right hand side, while for a full outerjoin node, the outerjoin null-supplies the relations from its both sides.

*Definition 2.* For any relation $T$ in an outerjoin query represented by an operator tree, we define the *direct outerjoin* of $T$ to be the first ancestor node of type left outerjoin or full outerjoin which null-supplies $T$. Any other outerjoin which also null-supplies $T$ is called an *indirect outerjoin* of $T$.

If a relation has a direct outerjoin, it is null-supplied by its direct outerjoin, and by any other indirect outerjoin for which the direct outerjoin is nested in its null-supplying side.

In this paper we assume that the materialized view definitions are outerjoin queries with and without aggregations, and the predicate of any join $\mathcal{J}$ must be a $\mathcal{NS}$-intolerant predicate only on a relation which has a direct outerjoin and this direct outerjoin is different than $\mathcal{J}$. $\mathcal{NS}$-intolerant property imposed on some of the outerjoin predicates assures the null-supplying rippling effect: if a relation $T$ is null-supplied in a tuple $t$ by its direct outerjoin, then any other outerjoin whose predicate $p$ references $T$ must also null-supply its null-supplying side in $t$ since the predicate $p$ doesn't evaluate to $True$. For example, for the outerjoin query $V_2 = (R \overset{lo}{\bowtie}_{p(R,T)} T) \overset{fo}{\bowtie}_{p(\underline{T},S)} S$ we require that only the predicate $p(\underline{T}, S)$ of the full outerjoin $\overset{fo}{\bowtie}_{p(\underline{T},S)}$ is $\mathcal{NS}$-intolerant on the relation $T$ which can be null-supplied by

[2]The **IS NOT DISTINCT FROM** predicate was introduced in ANSI SQL:1999 [5] and it is equivalent to the predicate $T.X = R.X$ $OR$ ($T.X$ **IS NULL AND** $R.X$ **IS NULL**).

its direct outerjoin, namely $R \overset{lo}{\bowtie}_{p(R,T)} T$. The predicate $p(R,T)$ doesn't have to be $\mathcal{NS}$-intolerant as both relations referenced by $p(R,T)$ have $\overset{lo}{\bowtie}_{p(R,T)}$ as their direct outerjoin. We also assume that the materialized view has a unique index with nulls not distinct on attributes which maybe null[3]. Extra requirements for the immediate materialized views with outerjoins will be discussed in the Section 3.

## 3. PRESERVED SIDE / NULL-SUPPLIED SIDE ($\mathcal{PSNS}$) ALGORITHM

For a given outerjoin query defining an immediate materialized view $V$, we want to find a representation of the null-supplying properties of a base relation $T$ which will give us the correct formula for an update statement for the view $V$ after update operations on the relation $T$. The main goal is to impose as few restrictions as possible to the view definition, and also view update statements to be very efficient.

We assume that the relation $T$ being updated is null-supplied by at least one outerjoin in the view definition[4]. Let $V$ be a table expression containing outerjoins which references a set of relations $T_1, \ldots, T_n$ and the relation $T$.For the rest of the paper, the assumption is that only the relation $T$ is updated and all other relations $T_1, \ldots, T_n$ referenced by the view are left unchanged. Hence, we will use, whenever possible, the simplified notations where the fix relations are not mentioned, e.g., $V(T, T_1, \ldots, T_n) = V(T)$. In general, a view definition $\overline{V}$ projects a select list $(v_1, \ldots, v_k)$ where each $v_i$ is an expression over the attributes of the relations $T, T_1, \ldots, T_n$, i.e., $\overline{V(T)} = \pi_{v_1,\ldots,v_k} V(T) = \{(v_1(t), \ldots, v_k(t)) | t \in V(T)\}$. Unless the overline notation is used (i.e., $\overline{V}$), the assumption is that the table expression projects all the attributes of the referenced relations.

*Definition 3.* The set of all tuples in the instance $V(T)$ which are not null-supplying the relation $T$ ($\neg T$*-null-supplied*) is defined as $V(nn(T)) = \{t | t \in V(T), nn(t[T])\}$. $V(nn(T))$ can be computed by using the original view definition where direct and indirect outerjoins null-supplying $T$ are transformed into inner, left or right outerjoin such that $T$ is no longer null-supplied. For example, if $V_3 = (R \overset{fo}{\bowtie}_{p(R,T)} T) \overset{fo}{\bowtie}_{p(\underline{T},S)} S$, $V_3(nn(T)) = (R \overset{ro}{\bowtie}_{p(R,T)} T) \overset{lo}{\bowtie}_{p(\underline{T},S)} S$.

*Definition 4.* The set of all $T$*-null-supplied* tuples in the instance $V(T)$ is denoted $V(n(T))$.

*Definition 5.* The *maximum set* of the $T$*-null-supplied* tuples in $V$, denoted by $\mathcal{N}ull\,(T,V)$, is the set of all possible $T$*-null-supplied* tuples in any instance of the view $V$ computed over the fix relations $T_1, \ldots, T_n$ and any instance of the relation $T$. I.e.,
$\mathcal{N}ull\,(T,V) = \delta(\bigcup_{\textbf{any instance of } T} V(n(T))))$.

$\mathcal{N}ull\,(T,V)$ can be obtained by using the original view definition where the relation $T$ is replaced by empty set. I.e., $\mathcal{N}ull\,(T,V) = V(T \to \emptyset)$. For example, $\mathcal{N}ull\,(T,V_3) = (R \overset{fo}{\bowtie}_{p(R,T)} \emptyset) \overset{fo}{\bowtie}_{p(\underline{T},S)} S$. $\mathcal{N}ull\,(T,V)$ contains all possible $T$*-null-supplied* tuples which can be present in an instance of $V(T)$. In other words, for any instance of the relation $T$, any $T$*-null-supplied* tuple in $V(T)$ must be present in $\mathcal{N}ull\,(T,V)$ as long as the content of all other base relations is unchanged, i.e., $V(n(T)) \subseteq \mathcal{N}ull\,(T,V)$. In our example $V_3$, if $R = \{r_0, r_1\}, S = \{s_0\}$, any $T$*-null-supplied* tuple that can exist in $V_3$ must be one of the following tuples $\mathcal{N}ull\,(T,V_3) = \{(r_0, \mathcal{N}ull, \mathcal{N}ull), (r_1, \mathcal{N}ull, \mathcal{N}ull), (\mathcal{N}ull, \mathcal{N}ull, s_0)\}$.

For incremental maintenance of the materialized views with outerjoins, we know how to compute, for each insert or delete operation on the relation $T$ using the set $\Delta T$, the set of not null-supplied tuples to be inserted into or deleted from $V$. This set, denoted by $V(nn(\Delta T))$ can be obtained by using $\Delta T$ in the definition of $V(nn(T))$ instead of $T$. However, each such operation can have a side effect related to the $T$*-null-supplied* tuples. For a delete operation on $T$, deleting tuples from $V$, namely, $V(nn(\Delta T))$, may leave the view $V$ in need of new $T$*-null-supplied* tuples. Furthermore, inserting new tuples in $V$, namely $V(nn(\Delta T))$, may leave some spurious old tuples in $V$ which are now dominated by new tuples in $V(nn(\Delta T))$.

*Definition 6.* The $\mathcal{NS}$*-compensation* operation (*Null-Supplying-compensation*) is the update operation on the view $V$ which inserts or deletes $T$*-null-supplied* tuples, after $V$ was updated using $V(nn(\Delta T))$.

Our goal is to design an algorithm having the following properties: (1) for each tuple $t \in V(nn(\Delta T))$, compute, in the same time with the computation of $V(nn(\Delta T))$, the set of potential $T$*-null-supplied* tuples corresponding to the tuple $t$, $\mathcal{N}ull\,(t,T,V)$. $\mathcal{N}ull\,(t,T,V)$ will be used for $\mathcal{NS}$-compensation after the insert, delete, or update operations; (2) for each tuple $t' \in \mathcal{N}ull\,(t,T,V)$, decide, using the view $V$, if $t'$ must be deleted (for insert and update operations) or inserted (for delete and update operations) into $V$ to $\mathcal{NS}$-compensate for the inserted and deleted tuples from $V(nn(\Delta T))$; (3) there is no need to save partial deltas into temporary tables, hence a single statement should be used for both the update operation using $V(nn(\Delta T))$ and $\mathcal{NS}$-compensation operation.

In order to achieve these goals, each immediate materialized view definition is internally represented as an annotated normalized operator tree, a $\mathcal{PSNS}$-annotated $n\mathcal{T}$, which contains all the metadata necessary to generate SQL update statements used to incrementally maintain the view after any referenced table is updated. Given a general view definition containing outerjoins, $\overline{V} = \pi_{v_1,\ldots,v_k} V$, its $\mathcal{PSNS}$-annotated normalized tree $n\mathcal{T}\,(\overline{V})$ is built in two steps: (1) Algorithm 1 builds the normalized operator tree $n\mathcal{T}$ for $\overline{V}$; (2) Algorithm 2 annotates $n\mathcal{T}$ with preserved side/null-supplied side information necessary to generate the $\mathcal{NS}$-compensation operations.

A normalized join $n\mathcal{J} = (Dn\mathcal{J}, DRels, n\mathcal{J}s, p)$ (Algorithm 1) has a direct join parent $Dn\mathcal{J}$, $DRels$ all relations having $n\mathcal{J}$ as their direct outerjoin, and $n\mathcal{J}s$ the nested outerjoins which can be directly null-supplied by $n\mathcal{J}$. All the normalized joins are null-supplying except the root join. For example, a simple join operator tree
$V_4 = (R_1 \overset{lo}{\bowtie}_{p(R_1,T,S)} (S \bowtie_{p(T,S)} T)) \bowtie_{p(R_1,R_2)} R_2$ has $n\mathcal{T}\,(V_4) = n\mathcal{J}_0 = (\mathcal{N}ull, \{R_1, R_2\}, \{n\mathcal{J}_1\}, p(R_1, R_2))$ where $n\mathcal{J}_1 = (n\mathcal{J}_0, \{S, T\}, \emptyset, (p(R_1, T, S) \wedge p(T, S)))$ is of type left outerjoin. For another example, in Figure 1, the normalized join $n\mathcal{J}_1^{fo}$ representing the table expression

[3] In a unique index with nulls not distinct defined on the attributes $(A, B)$, two tuples $(a, \mathcal{N}ull)$ and $(a, \mathcal{N}ull)$ are considered equals and cannot exist in the same time in the view.

[4] For relations which cannot be null-supplied by an outerjoin, the view update statements are similar to the formulas for innerjoin immediate materialized views.

$((R_1 \bowtie_{p(R_1,R_2)} R_2) \overset{fo}{\bowtie}_{p(R_1,T_1)} (T_1 \bowtie_{p(T_1,T_2)} T_2))$, can be all null-supplied by the full outerjoin $\overset{fo}{\bowtie}_{p(\underline{R_1},\underline{X_1},\underline{X_2},\underline{Y_2})}$, and hence it is nested in the $n\mathcal{J}^{fo}_0$. The main property of a normalized join $n\mathcal{J}$ is that all its relations must be null-supplied together in a tuple of $V$. Algorithm 1 also checks the $\mathcal{NS}$-intolerant properties of the predicates of the original operator tree $\mathcal{T}$ which must be imposed on the immediate view definition to assure the null-supplying rippling effect on any tuple of $V$.

**Algorithm 1** $n\mathcal{T}\,(\mathcal{T},\, n,\, n\mathcal{J}\,)$

**Procedure:** Build a normalized operator tree from $\mathcal{T}$
**Input:** $\mathcal{T}$, a node $n$, and a direct parent $n\mathcal{J} = (Dn\mathcal{J}\,, DRels = \{T_1, \ldots, T_n\}, n\mathcal{J}s = \{n\mathcal{J}_1, \ldots, n\mathcal{J}_m\}, p)$
  $Rels(n\mathcal{J}\,) \stackrel{def}{=} DRels \cup_i Rels(n\mathcal{J}_i)$
All semantic transformations, such as join elimination, outerjoin to innerjoin transformations, predicate push-down were already applied to the operator tree $\mathcal{T}$
$n\mathcal{J}_L = \mathcal{N}ull$, $n\mathcal{J}_R = \mathcal{N}ull$, $n\mathcal{J}_F = \mathcal{N}ull$
**if** $n$ is a relation $T$ with predicate $p$: **then**
  $n\mathcal{J}.p = n\mathcal{J}.p \wedge p$, $n\mathcal{J}.DRels \cup = \{T\}$, $T.Dn\mathcal{J} = n\mathcal{J}$
**return**
**if** $n$ is an inner join with predicate $p$: **then**
  $n\mathcal{J}.p = n\mathcal{J}.p \wedge p$
  call $n\mathcal{T}\,(\mathcal{T}$, left child of $n$, $n\mathcal{J}\,)$
  call $n\mathcal{T}\,(\mathcal{T}$, right child of $n$, $n\mathcal{J}\,)$
  goto *check_predicates*
**if** $n$ is a left outerjoin with predicate $p$: **then**
  call $n\mathcal{T}\,(\mathcal{T}$, left child of $n$, $n\mathcal{J}\,)$
  $n\mathcal{J}_L$ = new normalized join of type left outerjoin
  $n\mathcal{J}_L.(Dn\mathcal{J}\,, p) = (n\mathcal{J}\,, p)$, $n\mathcal{J}.n\mathcal{J}s \cup = \{n\mathcal{J}_L\}$
  call $n\mathcal{T}\,(\mathcal{T}$, right child of $n$, $n\mathcal{J}_L)$
  goto *check_predicates*
**if** $n$ is a full outerjoin with predicate $p$: **then**
  $n\mathcal{J}_F$ = new normalized join of type full outerjoin
  $n\mathcal{J}_F.(Dn\mathcal{J}\,, p) = (n\mathcal{J}\,, p)$, $n\mathcal{J}.n\mathcal{J}s \cup = \{n\mathcal{J}_F\}$
  $n\mathcal{J}_L$ = new normalized join of type outerjoin
  $n\mathcal{J}_L.Dn\mathcal{J} = n\mathcal{J}_F$
  $n\mathcal{J}_R$ = new normalized join of type outerjoin
  $n\mathcal{J}_R.Dn\mathcal{J} = n\mathcal{J}_F$
  $n\mathcal{J}_F.n\mathcal{J}s = \{n\mathcal{J}_L, n\mathcal{J}_R\}$
  call $n\mathcal{T}\,(\mathcal{T}$, left child of $n$, $n\mathcal{J}_L)$
  call $n\mathcal{T}\,(\mathcal{T}$, right child of $n$, $n\mathcal{J}_R)$
  goto *check_predicates*
*check_predicates*: /* (P1) $p$ must refer to both left and right child of $n$, and $Rels(p) \subseteq Rels(n\mathcal{J}\,)$
(P2) $p$ must be $\mathcal{NS}$-intolerant on $T \in Rels(p)$ iff $n$ is not the direct outerjoin of $T$ */
**for** $T \in Rels(p)$ : **do**
**Require:** If $Dn\mathcal{J}\,(T) \notin \{n\mathcal{J}\,, n\mathcal{J}_L, n\mathcal{J}_R\}$ then $p$ must be $\mathcal{NS}$-intolerant on $T$.

$Union\mathcal{NS}\,(n\mathcal{J}\,, T)$

/* If $T$ is null-supplied then $n\mathcal{J}$ is also null-supplied */
**for** $(ps, ns) \in \mathcal{PSNS}\,(R), T \in ns, Rels(n\mathcal{J}\,) \cap ps = \emptyset$ **do**
  $ns = ns \bigcup Rels(n\mathcal{J}\,)$

The $\mathcal{PSNS}$ Algorithm 2 computes sets of preserved side/null-supplied side pairs based on the view definition. For any referenced relation $T$, each pair $(ps_i, ns_i) \in \mathcal{PSNS}\,(T)$ de-

**Algorithm 2** $\mathcal{PSNS}\,(n\mathcal{J}\,)$

**Procedure:** Compute $\mathcal{PSNS}\,()$ (*P*reserved *S*ide / *N*ull-supplied *S*ide) sets for the normalized join
$n\mathcal{J} = (Dn\mathcal{J}\,, DRels, n\mathcal{J}s\,, p) \in n\mathcal{T}\,(\overline{V})$
$\mathcal{PSNS}\,(T) \stackrel{def}{=} \mathcal{PSNS}\,(Dn\mathcal{J}\,(T))$
**RULE 0:** /* If $n\mathcal{J}$ is null-supplied in a tuple $t$, then all its relations must be null-supplied in $t$ */
$\mathcal{PSNS}\,(n\mathcal{J}\,) = \{(\emptyset, Rels(n\mathcal{J}\,))\}$
**for** $n\mathcal{J}\,' \in n\mathcal{J}s\,(n\mathcal{J}\,)$ **do**
  $\mathcal{PSNS}\,(n\mathcal{J}\,')$
**for** $T \in Rels(p)$ **do**
  **if** $n\mathcal{J}$ type is left outerjoin **then**
    **RULE 1:** /* $(T...) \overset{lo}{\bowtie}_{p(T,...)} (n\mathcal{J}\,)$ : $T$ is in the preserved side of the left outerjoin $n\mathcal{J}$. If $T$ is null-supplied in a tuple $t$ then $n\mathcal{J}$ must be all null-supplied in $t$ as well since the predicate $p$ cannot evaluate to $True$ (rippling effect): */
    **if** $T \notin Rels(n\mathcal{J}\,)$ **then**
      $Union\mathcal{NS}\,(n\mathcal{J}\,, T)$
    **RULE 2:** /* $\ldots \overset{lo}{\bowtie}_{p(T,...)} (T...)$ : $T$ is in the null-supplying side of the left outerjoin $n\mathcal{J}$. The relations in $Rels(p) \setminus Rels(n\mathcal{J}\,)$ must be preserved if $n\mathcal{J}$ is null-supplied: */
    **if** $T \in Rels(n\mathcal{J}\,)$ **then**
      **for** $(ps, ns) \in \mathcal{PSNS}\,(n\mathcal{J}\,)$ **do**
        $ps \bigcup = Rels(p) \setminus Rels(n\mathcal{J}\,)$
  **if** $n\mathcal{J}$ type is full outerjoin **then**
    /* $n\mathcal{J}s\,(n\mathcal{J}\,) = \{n\mathcal{J}_L, n\mathcal{J}_R\}$ */
    **if** $T \in Rels(n\mathcal{J}_L)$ **then**
      $(n\mathcal{J}_1, n\mathcal{J}_2) = (n\mathcal{J}_L, n\mathcal{J}_R)$
    **if** $T \in Rels(n\mathcal{J}_R)$ **then**
      $(n\mathcal{J}_1, n\mathcal{J}_2) = (n\mathcal{J}_R, n\mathcal{J}_L)$
    **RULE 3:** /* If $T$ is null-supplied by an outerjoin other than this full outerjoin, then $n\mathcal{J}_2$ must be null-supplied as well since the predicate $p$ cannot evaluate to $True$ (rippling effect): */
    **for** $j = Dn\mathcal{J}\,(T), j \neq n\mathcal{J}\,, j = Dn\mathcal{J}\,(j)$ **do**
      $Union\mathcal{NS}\,(n\mathcal{J}_2, j)$
    **RULE 4:** /* The relations in $n\mathcal{J}_2$ must be preserved if $n\mathcal{J}_1$ is null-supplied: */
    $ns = \{n\mathcal{J}_1\}$
    $ps = (Rels(n\mathcal{J}_2) \bigcap Rels(p))$
    **for** $j = Dn\mathcal{J}\,(T), j \neq n\mathcal{J}\,, j = Dn\mathcal{J}\,(j)$ **do**
      $\mathcal{PSNS}\,(j) \bigcup = \{(ps, ns)\}$
/* *check_selectlist*: Check restrictions on the select list of the view $\overline{V} = \pi_{v_1,\ldots,v_k} V$: (P3) Any relation in a $ps_i$ set must have a primary key, and the primary key attributes must be among the select list expressions (P4) For any normalized join $n\mathcal{J}$ in a $ns_i$ set, the predicates $nn(n\mathcal{J}\,)$ and $n(n\mathcal{J}\,)$ must be generated using the select list expressions. */
**for** $R \in Rels(n\mathcal{T}\,(\overline{V})), (ps, ns) \in \mathcal{PSNS}\,(R)$ : **do**
  **for** $T \in ps$ : **do**
**Require:** $T$ has a primary key $T.pk = \{C_1, \ldots, C_j\} \subseteq \{v_1, \ldots, v_k\}$ /* The predicate $\wedge_{1<=i<=j}(\overline{V}.C_i = \overline{V}(\Delta T).C_i)$ is needed. */
  **for** $n\mathcal{J} \in ns$ : **do**
**Require:** $\exists S \in DRels(n\mathcal{J}\,)$ and $S.C = C \in \{v_1, \ldots, v_k\}$ a not-null attribute /* The predicates $n(\overline{V}.C)$ and $nn(\overline{V}.C)$: $\overline{V}.C$ IS [NOT] NULL are needed. */

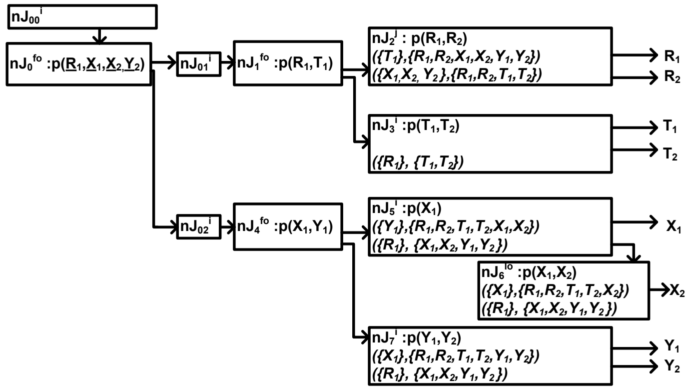


**Figure 1:** $V_1 = ((R_1 \bowtie_{p(R_1,R_2)} R_2) \overset{fo}{\bowtie}_{p(R_1,T_1)} (T_1 \bowtie_{p(T_1,T_2)} T_2)) \overset{fo}{\bowtie}_{p(\underline{R_1},\underline{X_1},\underline{X_2},\underline{Y_2})} ((\sigma_{p(X_1)} X_1 \overset{lo}{\bowtie}_{p(X_1,X_2)} X_2) \overset{fo}{\bowtie}_{p(X_1,Y_1)} (Y_1 \bowtie_{p(Y_1,Y_2)} Y_2))$**: $\mathcal{PSNS}$-annotated $n\mathcal{T}$**

$$V_1 = ((R_1 \bowtie_{p(R_1,R_2)} R_2) \overset{fo}{\bowtie}_{p(R_1,T_1)} (T_1 \bowtie_{p(T_1,T_2)} T_2)) \overset{fo}{\bowtie}_{p(\underline{R_1},\underline{X_1},\underline{X_2},\underline{Y_2})} ((\sigma_{p(X_1)} X_1 \overset{lo}{\bowtie}_{p(X_1,X_2)} X_2) \overset{fo}{\bowtie}_{p(X_1,Y_1)} (Y_1 \bowtie_{p(Y_1,Y_2)} Y_2))$$

| | |
|---|---|
| $\mathcal{PSNS}(R_1) = \mathcal{PSNS}(R_2) =$ | $\{(\{T_1\},\{R_1,R_2,X_1,X_2,Y_1,Y_2\}),(\{X_1,X_2,Y_2\},\{R_1,R_2,T_1,T_2\})\}$ |
| $\mathcal{PSNS}(T_1) = \mathcal{PSNS}(T_2) =$ | $\{(\{R_1\},\{T_1,T_2\})\}$ |
| $\mathcal{PSNS}(X_1) =$ | $\{(\{Y_1\},\{R_1,R_2,T_1,T_2,X_1,X_2\}),(\{R_1\},\{X_1,X_2,Y_1,Y_2\})\}$ |
| $\mathcal{PSNS}(X_2) =$ | $\{(\{X_1\},\{R_1,R_2,T_1,T_2,X_2\}),(\{R_1\},\{X_1,X_2,Y_1,Y_2\})\}$ |
| $\mathcal{PSNS}(Y_1) = \mathcal{PSNS}(Y_2) =$ | $\{(\{X_1\},\{R_1,R_2,T_1,T_2,Y_1,Y_2\}),(\{R_1\},\{X_1,X_2,Y_1,Y_2\})\}$ |
| (P2) $\mathcal{NS}$-intolerant: | $p(\underline{R_1},\underline{X_1},\underline{X_2},\underline{Y_2})$ on $R_1,X_1,X_2,Y_2$ |
| (P3) $pk$ in select list: | $R_1,T_1,X_1,X_2,Y_1,Y_2$ |
| (P4) $nn(),n()$ in select list: | $R_1,T_1,X_1,X_2,Y_1$ |

**Figure 2: The $\mathcal{PSNS}$-annotated $n\mathcal{T}(V_1)$ Example**

scribes exactly one $T$-*null-supplied* tuple, $\mathcal{N}ull\,(i,t,T,V)$, which can be obtained from the tuple $t \in V(nn(\Delta T))$, and provides the condition used to check the need for using the tuple $\mathcal{N}ull\,(i,t,T,V)$ for $\mathcal{NS}$-compensation of $V$. The set $ns_i$ contains all the relations in $Rels(V)$ which have to be null-supplied together with $T$ in the tuple $\mathcal{N}ull\,(i,t,T,V)$ ( $T \in ns_i$). The set $ps_i$ represents all the relations which must be preserved in the tuple $\mathcal{N}ull\,(i,t,T,V)$ ($ns_i \cap ps_i = \emptyset$), hence $ps_i$ relations cannot be null-supplied in the original tuple $t$. The relations in $Rels(V) \setminus (ps_i \cup ns_i)$ are *don't care* relations: their values are left unchanged in the tuple $\mathcal{N}ull\,(i,t,T,V)$ (i.e., they can be null-supplied or not in the tuple $t$). Note that two relations $T$ and $R$ having the same direct outerjoin $n\mathcal{J}$ have $\mathcal{PSNS}(T) = \mathcal{PSNS}(R)$ which we denote by $\mathcal{PSNS}(n\mathcal{J})$. Let's consider **RULE 1** of Algorithm 2. If the table $T$ is not in the null-supplying side of a left outerjoin $n\mathcal{J}$ but it is referenced in the predicate $p$ of $n\mathcal{J}$, the predicate must be $\mathcal{NS}$-intolerant on $T$ (according to the property (P2) defined by Algorithm 1) if $T$ is null-supplied by any other outerjoin. In this case, the predicate $p$ cannot evaluate to $True$, and all relations in $n\mathcal{J}$ must also be null-supplied. Hence, all the joins null-supplying $T$ must have in their $ns$ sets $Rels(n\mathcal{J})$. To explain **RULE 3** and **RULE 4**, let's consider the following example for the view $V_2 = (R \overset{lo}{\bowtie}_{p(R,T)} T) \overset{fo}{\bowtie}_{p(\underline{T},S)} S$. If the tuple $(r,t,s) \in V_2$ is deleted as a result of the tuple $(t) \in T$ being deleted, two potential $T$-*null-supplied* tuples must $\mathcal{NS}$-compensate $V_2$ : $(r,\mathcal{N}ull,\mathcal{N}ull)$ by **RULE 3**, and $(\mathcal{N}ull,\mathcal{N}ull,s)$ by **RULE 4** (in this example, $n\mathcal{J}_2 = \{S\}$ and $n\mathcal{J}_1 = \{R,T\}$). Furthermore, **RULE 3** and **RULE 4** must also take care of the rippling effect if $T$ is in a nested outerjoin. Note that each full outerjoin introduces a new pair of $(ps', ns')$ which intuitively corresponds to the fact that each tuple $t = (l,r)$ in the cross product $L \times R$ of the left and right hand sides of a full outerjoin $L \overset{fo}{\bowtie}_{p(L,R)} R$ may potentially produce two tuples in the result set of the full outerjoin, namely $(l,\mathcal{N}ull)$ and $(\mathcal{N}ull,r)$. Figures 2 and the table below describe more examples which contain the $\mathcal{PSNS}$ sets, and also the restrictions imposed by Algorithm 1 ((P2) property) and Algorithm 2 ((P3) and (P4) properties).

For a given tuple $t \in V(nn(\Delta T))$, if the relations in $ps_i$ are not null-supplied in $t$, the tuple $\mathcal{N}ull\,(i,t,T,V)$ is obtained as follows: (1) all attributes of $T$ and the relations in $ns_i$ are replaced with $\mathcal{N}ull$. (2) the values for the don't care relations $Rels(V) \setminus (ps_i \cup ns_i)$, and for the $ps_i$ relations are left unchanged. Any tuple $\mathcal{N}ull\,(i,t,T,V)$ is dominated by the tuple $t$, hence they cannot both exist in the view $V$. The formal definitions for the sets $\mathcal{N}ull\,(i,t,T,V)$, $\mathcal{N}ull\,(t,T,V)$, and $\mathcal{N}ull\,(\Delta T,T,V)$ when $nn(t[ps_i])$ are given below:

$$\textit{don't care} = Rels(V) \setminus (ps_i \cup ns_i)$$
$$\mathcal{N}ull\,(i,t,T,V) = (t[\textit{don't care}], t[ps_i], n(ns_i))$$

$$\mathcal{N}ull\,(t,T,V) = \bigcup_{(ps_i,ns_i) \in \mathcal{PSNS}(T)} \mathcal{N}ull\,(i,t,T,V)$$

$$\mathcal{N}ull\,(\Delta T,T,V) = \bigcup_{t \in V(nn(\Delta T))} \mathcal{N}ull\,(t,T,V)$$

Given the $\mathcal{PSNS}$-annotated normalized operator tree $n\mathcal{T}(\overline{V})$ of a view $\overline{V} = \pi_{v_1,\ldots,v_k} V(T,T_1,\ldots,T_n)$ the general view update operation after $T$ is updated using $\Delta T$ is as follows:

Step 1: Compute, in the same time, $V(nn(\Delta T))$ and $\mathcal{N}ull\,(\Delta T, T, V)$ using $\Delta T$, $T_1, \ldots, T_n$ relations. The formula for $V(nn(\Delta T))$ is derived by traversing the normalized join tree $n\mathcal{T}\,(\overline{V})$ changing the direct and indirect outerjoins of $T$ such that $T$ is no longer null-supplied as described in Definition 3.
Step 2: Apply $\overline{V(nn(\Delta T))} = \{v_1(t), \ldots, v_k(t) | t \in V(nn(\Delta T))\}$ to $\overline{V}$
Step 3: Perform $\mathcal{NS}$-compensation using $\mathcal{N}ull\,(\Delta T, T, V)$. The predicates used to check the need to $\mathcal{NS}$-compensate a tuple in $\mathcal{N}ull\,(\Delta T, T, V)$ are generated using $ps_i$ and $ns_i$ sets as described in Algorithm 2 in the restrictions (P3) and (P4). If a tuple $t \in \mathcal{N}ull\,(\Delta T, T, V)$ needs to $\mathcal{NS}$-compensate $\overline{V}$, the tuple $t' = (v_1(t), \ldots, v_k(t)) \in \overline{\mathcal{N}ull\,(\Delta T, T, V)}$ will be inserted into or deleted from $\overline{V}$.

| $V_2 = (R \overset{lo}{\bowtie}_{p(R,T)} T) \overset{fo}{\bowtie}_{p(\underline{T},S)} S$ | |
|---|---|
| $\mathcal{PSNS}\,(R) =$ | $\{(\{S\}, \{R, T\})\}$ |
| $\mathcal{PSNS}\,(T) =$ | $\{(\{R\}, \{T, S\}), (\{S\}, \{R, T\})\}$ |
| $\mathcal{PSNS}\,(S) =$ | $\{(\{T\}, \{S\})\}$ |
| (P2) $\mathcal{NS}$-intolerant: | $p(\underline{T}, S)$ on $T$ |
| (P3) *pk* | $R, S, T$ |
| (P4) $nn(), n()$ | $R, S, T$ |

| $V_3 = (R \overset{fo}{\bowtie}_{p(R,T)} T) \overset{fo}{\bowtie}_{p(\underline{T},S)} S$ | |
|---|---|
| $\mathcal{PSNS}\,(R) =$ | $\{(\{T\}, \{R\})\}$ |
| $\mathcal{PSNS}\,(T) =$ | $\{(\{R\}, \{T, S\}), (\{S\}, \{R, T\})\}$ |
| $\mathcal{PSNS}\,(S) =$ | $\{(\{T\}, \{S\})\}$ |
| (P2) $\mathcal{NS}$-intolerant: | $p(\underline{T}, S)$ |
| (P3) *pk* | $T, R, S$ |
| (P4) $nn(), n()$ | $T, R, S$ |

| $V_4 = (R_1 \overset{lo}{\bowtie}_{p(R,T,S)} (S \bowtie_{p(T,S)} T)) \bowtie_{p(R_1,R_2)} R_2$ | |
|---|---|
| $\mathcal{PSNS}\,(R_1) = \mathcal{PSNS}\,(R_2) =$ | $\emptyset$ |
| $\mathcal{PSNS}\,(T) = \mathcal{PSNS}\,(S) =$ | $\{(\{R_1\}, \{T, S\})\}$ |
| (P2) $\mathcal{NS}$-intolerant: | none |
| (P3) *pk* | $R_1$ |
| (P4) $nn(), n()$ | only one of $T, S$ |

| $V_5 = R \overset{lo}{\bowtie}_{p(R,S)} (S \overset{lo}{\bowtie}_{p(T,S)} T)$ | |
|---|---|
| $\mathcal{PSNS}\,(R) =$ | $\emptyset$ |
| $\mathcal{PSNS}\,(S) =$ | $\{(\{R\}, \{T, S\})\}$ |
| $\mathcal{PSNS}\,(T) =$ | $\{(\{S\}, \{T\})\}$ |
| (P2) $\mathcal{NS}$-intolerant: | none |
| (P3) *pk* | $R, S$ |
| (P4) $nn(), n()$ | $S, T$ |

| $V_6 = (R \overset{lo}{\bowtie}_{p(R,S)} S) \overset{lo}{\bowtie}_{p(R,T)} T)$ | |
|---|---|
| $\mathcal{PSNS}\,(R) =$ | $\emptyset$ |
| $\mathcal{PSNS}\,(S) =$ | $\{(\{R\}, \{S\})\}$ |
| $\mathcal{PSNS}\,(T) =$ | $\{(\{R\}, \{T\})\}$ |
| (P2) $\mathcal{NS}$-intolerant: | none |
| (P3) *pk* | $R$ |
| (P4) $nn(), n()$ | $S, T$ |

| $V_7 = (R \overset{fo}{\bowtie}_{p(R,T)} T) \overset{fo}{\bowtie}_{p(\underline{T},S)} (S \overset{lo}{\bowtie}_{p(S,W)} W)$ | |
|---|---|
| $\mathcal{PSNS}\,(R) =$ | $\{(\{T\}, \{R\})[$*don't care* $= \{S, W\}]\}$ |
| $\mathcal{PSNS}\,(T) =$ | $\{(\{R\}, \{T, S, W\})[$*don't care* $= \emptyset]$, $(\{S\}, \{R, T\})[$*don't care* $= \{W\}]\}$ |
| $\mathcal{PSNS}\,(S) =$ | $\{(\{T\}, \{S, W\})[$*don't care* $= \{R\}]\}$ |
| $\mathcal{PSNS}\,(W) =$ | $\{(\{S\}, \{W\})[$*don't care* $= \{R, T\}]\}$ |
| (P2) $\mathcal{NS}$-intolerant: | $p(\underline{T}, S)$ |
| (P3) *pk* | $T, R, S$ |
| (P4) $nn(), n()$ | $T, R, S, W$ |

For an example, let's consider the view $V_7$ from above (the *don't care* sets are also shown), and the relation $T$ being updated with $\Delta T$. For a tuple $x \in V_7(nn(\Delta T))$ it is known that $x[T]$ must be not null-supplied ($V_7(nn(\Delta T))$ contains only $\neg T$-*null-supplied* by Definition 3). Let's assume that $x = (r, t, s, w)$ with $x[T] = (t)$ is not null-supplied, but $x[R] = (r)$, $x[S] = (s)$, and $x[W] = (w)$ maybe null-supplied. The set $\mathcal{PSNS}\,(T)$ was computed to be $\{(\{R\}, \{T, S, W\})$, $(\{S\}, \{R, T\})\}$ in the $n\mathcal{T}\,(V_7)$. Intuitively, the first $(ps_1, ns_1) = (\{R\}, \{T, S, W\})$ represents the $T$-*null-supplied* tuples where the relation $R$ must be preserved and $T$ is null-supplied. As $T$ is null-supplied in $\mathcal{N}ull\,(1, x, T, V_7)$, and the predicate $p(\underline{T}, S) <> True$ for the full outerjoin $\overset{fo}{\bowtie}_{p(\underline{T},S)}$, then the whole other side of the full outerjoin must be null-supplied as well besides $T$. This explains why $ns_1 = \{T, S, W\}$ contains the relations $S$ and $W$. If $x[R] = (r)$ is not null-supplied, then $\mathcal{N}ull\,(1, x, T, V_7) = (r, \mathcal{N}ull\,, \mathcal{N}ull\,, \mathcal{N}ull\,)$. Similarly, if $x[S] = (s)$ is not null-supplied, $\mathcal{N}ull\,(2, x, T, V_7) = (\mathcal{N}ull\,, \mathcal{N}ull\,, s, w)$. Note that the value $t[W] = w$ maybe be null-supplied or not, it will be left unchanged in $\mathcal{N}ull\,(2, x, T, V_7)$. If the update operation on the relation $T$ is an insert, if any of the tuples $\overline{\mathcal{N}ull\,(1, x, T, V_7)}$ and $\overline{\mathcal{N}ull\,(2, x, T, V_7)}$ exists in the $V_7$ then it must be deleted as it is a spurious tuple being dominated by the new tuple $x$. If the update operation is a delete, the tuple $\overline{\mathcal{N}ull\,(1, x, T, V_7)}$, where $ps_1 = \{R\}$, can be used for $\mathcal{NS}$-compensation of $V_7$ if, after applying $\overline{V_7(nn(\Delta T))}$, no tuple has the values $x[R] = (r)$ in $V_7$, i.e., $\nexists t' \in V_7, t'[R] = x[R]$. Similar argument holds for the tuple $\overline{\mathcal{N}ull\,(2, x, T, V_7)}$.

In the next Section 4 we will discuss how $\mathcal{PSNS}$-annotated normalized join trees are used to generate the **SQL** update statements for immediate materialized views.

# 4. IMPLEMENTATION

Materialized views which can be incrementally maintained or used in the query optimization process are represented internally as $\mathcal{PSNS}$-annotated normalized join operator trees as described in Algorithms 1 and 2, the same representation used by the cost-based SQL Anywhere Optimizer [7, 9] for the query optimization process. Both the view matching algorithm and the generation of the update statements for the incremental maintenance of the materialized views are using $n\mathcal{T}$ representing the definition of a view.

For an immediate materialized view, the $\mathcal{PSNS}$-annotated $n\mathcal{T}$ is built at the first reference to the view since the server was started. If a relation $T$ is updated, an internally generated trigger is created containing all the update statements for any immediate materialized view referencing the relation $T$. This section describes some of the update statements generated for these internal triggers for iMVs with outerjoins. Each update statement is a **SQL** statement which can be an **INSERT**, **UPDATE**, or **MERGE** [5] statement. The update statements are generated from the $\mathcal{PSNS}$-annotated $n\mathcal{T}$ representation of the materialized views. The execution of an internally generated trigger is done after any update operation on the relation $T$ when $\Delta T$ is passed on to the trigger. Each generated update statement is processed like any other **SQL** query, hence all the optimizations supported in the SQL Anywhere Optimizer such as exploitation of foreign key constraints, outer and inner join elimination are applied to find efficient execution plans. The generation algorithm is designed to produce correct update statements which can be efficiently optimized by the SQL Anywhere Optimizer.

Sections 4.1 4.2 describe **SQL** statements generated for an immediate materialized view with outerjoins $\overline{V} = \pi_{v_1,\ldots,v_k} V$ for the insert and update operations, respectively, using the $\mathcal{PSNS}$-annotated $n\mathcal{T}(\overline{V})$.

## 4.1 INSERT Operation

For an insert operation, the computation of the set $\mathcal{N}ull\,(\Delta T, T, V)$ can be done before $\overline{V(nn(\Delta T))}$ is applied, as the spurious *T-null-supplied* tuples maybe present in the view before the insert operation. Algorithm 3 describes the logical steps for computing and applying (i.e., delete) $\overline{\mathcal{N}ull\,(i,t,T,V)}$ for insert operations. The **SQL MERGE** statement computes $\overline{V(nn(\Delta V))}$ (lines 6-7) in the same time with the $\mathcal{N}ull\,(\Delta T, T, V)$ (lines 8-14). For efficiency, only rowids of the tuples found in $\overline{V} \bigcap \overline{\mathcal{N}ull\,(\Delta T, T, V)}$ are passed to be processed by the **WHEN MATCHED** clause. The **MERGE** statement processes first the spurious tuples which are deleted by the first **WHEN MATCHED** clause (line 21) (this is the $\mathcal{NS}$-compensation operation). The rows from $\overline{V(nn(\Delta T))}$ are inserted by the next **WHEN NOT MATCHED** clause (lines 22-23).

**Algorithm 3** Compute and apply $\overline{\mathcal{N}ull\,(i,t,T,V)}$ for insert operations

1: **for** $t \in V(nn(\Delta T))$ **do**
**Ensure:** $nn(t[T])$
2: **for** $i$ such that $(ps_i, ns_i) \in \mathcal{PSNS}\,(T), ps_i \neq \emptyset nn(t[ps_i])$ **do**
3: /* Generate *T-null-supplied* tuple in $\mathcal{N}ull\,(i,t,T,V)$ */
4: $\mathcal{N}ull\,(i,t,T,V) =$
5: $= (t.[Rels(V) \setminus ns_i], n(ns_i))$
6: $\overline{\mathcal{N}ull\,(i,t,T,V)} =$
7: $= (v_1(\mathcal{N}ull\,(i,t,T,V)), \ldots, v_k(\mathcal{N}ull\,(i,t,T,V)))$
/* $\overline{\mathcal{N}ull\,(i,t,T,V)} \in \overline{V}$ iff $\overline{V}.ps_i = t[ps_i]$**AND** $n(\overline{V}.ns_i)$ */
8: **if** $\overline{\mathcal{N}ull\,(i,t,T,V)} \in \overline{V}$ **then**
9: $\overline{V} = \overline{V} \setminus \{\overline{\mathcal{N}ull\,(i,t,T,V)}\}$ / / delete $\overline{\mathcal{N}ull\,(i,t,T,V)}$

1: $\boxed{\textbf{MERGE}}$ **INTO** $\overline{V}$ **USING**
2: **( WITH** $V(nn(\Delta T))$ *AS*
3: **(SELECT *,** $\Delta T$.DML_type **FROM** $V(nn(\Delta T))$ )
4: **SELECT DT.***
5: **FROM** $V(nn(\Delta T))$, **LATERAL (**
/* $\neg$*T-null-supplied* : $\overline{V(nn(\Delta T))}$ **to be inserted** */
6: **SELECT** $v_1(V(nn(\Delta T))), \ldots, v_k(V(nn(\Delta T)))$,
7: $V(nn(\Delta T))$.DML_type **AS DML_type,**$\mathcal{N}ull$ **AS** *d_rowid*
8: **UNION ALL**
/* *T-null-supplied* : $\mathcal{N}ull\,(1,t,T,V)$ $\mathcal{NS}$**-compensation(delete)** */
9: **SELECT** $\mathcal{N}ull$ **AS** $v_1, \ldots, \mathcal{N}ull$ **AS** $v_k$,
10: $V(nn(\Delta T))$.DML_type$*(-1)$**AS DML_type,rowid(**$\overline{V}$**)** *d_rowid*
11: **FROM** $\overline{V}$ **WHERE** $\overline{V}.ps_1 = V(nn(\Delta T)).ps_1$ **AND** $n(\overline{V}.ns_1)$
12: **UNION ALL**
13: ...
14: **UNION ALL**
/* *T-null-supplied* : $\mathcal{N}ull\,(m,t,T,V)$ $\mathcal{NS}$**-compensation(delete)** */
15: **SELECT** $\mathcal{N}ull$ **AS** $v_1, \ldots, \mathcal{N}ull$ **AS** $v_k$,
16: $V(nn(\Delta T))$.DML_type$*(-1)$**AS DML_type, rowid(**$\overline{V}$**)** *d_rowid*
17: **FROM** $\overline{V}$ **WHERE** $\overline{V}.ps_m = V(nn(\Delta T)).ps_m$ **AND** $n(\overline{V}.ns_m)$
18: ) **AS DT**
19: ) **AS** $\Delta V(\Delta T)$
20: **ON** DML_type = *-1* **AND** *rowid*$(\overline{V}) = \Delta V(\Delta T)$.*d_rowid*
21: **WHEN MATCHED THEN** $\boxed{\textbf{DELETE}}$
22: **WHEN NOT MATCHED AND** DML_type = *+1*
23: **THEN** $\boxed{\textbf{INSERT}}$

## 4.2 UPDATE Operation

For the delete and update operations $\overline{V(nn(\Delta T))}$ is applied by using the unique index with nulls not distinct which the materialized view must have. The condition $\overline{V}.ui$ **IS NOT DISTINCT FROM** $\overline{V(nn(\Delta T))}.ui$ (line 24) is the conjunction of all **IS NOT DISTINCT FROM** predicates applied to the columns of the unique index.

The computation of the set $\mathcal{N}ull\,(\Delta T, T, V)$ must see the view data after $\overline{V(nn(\Delta T))}$ is applied, such that a tuple $\mathcal{N}ull\,(i,t,T,V)$ is generated only if it is needed for the $\mathcal{NS}$-compensation operation. Hence, an embedded[5] **MERGE** statement (lines 19-28) first computes (line 20) and applies (line 25-28) $\overline{V(nn(\Delta T))}$. The outer $\boxed{\textbf{MERGE}}$ statement will see the modified view when the conditions for $\mathcal{NS}$-compensation are checked (lines 9, 14). The set $\overline{\mathcal{N}ull\,(\Delta T, T, V)}$ is processed by the **WHEN [NOT] MATCHED** clauses (lines 31-33).

1: $\boxed{\textbf{MERGE}}$ **INTO** $\overline{V}$ **USING**
2: **( WITH** $V(nn(\Delta T))$ *AS*
3: **(SELECT *,** $\Delta T$.DML_type**FROM** $V(nn(\Delta T))$ )
4: **SELECT** $v_1(DT), \ldots, v_k(DT)$, $DT$.DML_type **FROM (**
5: **SELECT DISTINCT** *DT_nulls*.$*$ **FROM** $V(nn(\Delta T))$,
6: **LATERAL (**
/* *T-null-supplied* : $\overline{\mathcal{N}ull\,(1,t,T,V)}$ $\mathcal{NS}$**-compensation(insert)** */
7: **SELECT** $v_1(\mathcal{N}ull\,(1,t,T,V)), \ldots, v_k(\mathcal{N}ull\,(1,t,T,V))$
8: $V(nn(\Delta T))$.DML_type$*(-1)$**AS DML_type,**$\mathcal{N}ull$ **AS** *d_rowid*
9: **WHERE** $nn(V(nn(\Delta T)).ps_1)$ **AND** DML_type = *+1* **AND**
10: **NOT EXISTS ( SELECT 1 FROM** $\overline{V}$ **WHERE**
11: $\overline{V}.ps_1 = V(nn(\Delta T)).ps_1$)
12: **UNION ALL**
/* *T-null-supplied* : $\overline{\mathcal{N}ull\,(1,t,T,V)}$ $\mathcal{NS}$**-compensation(delete)** */
13: **SELECT** $\mathcal{N}ull$ **AS** $v_1, \ldots, \mathcal{N}ull$ **AS** $v_k$
14: $V(nn(\Delta T))$.DML_type$*(-1)$**AS DML_type,rowid(**$\overline{V}$**) AS** *d_rowid*
15: **FROM** $\overline{V}$ **WHERE** DML_type = *-1* **AND**
16: $\overline{V}.ps_1 = V(nn(\Delta T)).ps_1$ **AND** $n(\overline{V}.ns_1)$
17: **UNION ALL**
18: ...
19: ) **AS** *DT_nulls*
20: ) **AS** *DT*,
21: ,(**SELECT FIRST 1 FROM (** $\boxed{\textbf{MERGE}}$ **INTO** $\overline{V}$
22: **USING (**
23: **select** $v_1(DT), \ldots, v_k(DT)$, $DT$.DML_type **FROM (**
/* $\neg$*T-null-supplied* : $\overline{V(nn(\Delta T))}$ **update, insert, or delete** */
24: **SELECT** $*$ **FROM** $V(nn(\Delta T))$
25: )**AS DT**
26: )**AS** $\overline{V(nn(\Delta T))}$
27: **ON** $\overline{V}.ui$ **IS NOT DISTINCT FROM** $\overline{V(nn(\Delta T))}.ui$
28: **WHEN MATCHED AND DML_type=***+1* **THEN** $\boxed{\textbf{UPDATE}}$
29: **WHEN MATCHED AND DML_type=***-1* **THEN** $\boxed{\textbf{DELETE}}$
30: **WHEN NOT MATCHED AND DML_type=***+1* **THEN** $\boxed{\textbf{INSERT}}$
31: )**REFERENCING( )) AS ZZ**
32: ) **AS** $\overline{\mathcal{N}ull\,(\Delta T, T, V)}$
33: **ON** DML_type = *-1* **AND** *rowid*$(\overline{V}) = \overline{\mathcal{N}ull\,(\Delta T, T, V)}$.*d_rowid*
34: **WHEN MATCHED THEN** $\boxed{\textbf{DELETE}}$
35: **WHEN NOT MATCHED AND** DML_type = *+1*
36: **THEN** $\boxed{\textbf{INSERT}}$

[5]The embedded update statements (aka *DML-derived-tables*) are executed first, hence the rest of the query sees the modified view after $\overline{V(nn(\Delta T))}$ was applied.

## 5. RELATED WORK

A large body of work exists on the subject of query optimization with outerjoins. Two representations for outerjoin queries are proposed in [3] and [10], and provide a strong intuition for understanding how null-supplied tuples are generated by outerjoin expressions. Galindo-Legaria [3] introduces the powerful join-disjunctive normal form which represents an outerjoin query as a sequence of minimum unions of different joins. Rao et al. [10] introduce a canonical abstraction for queries with left outerjoins, based on the analysis of the predicates and the query join operator tree. If a query contains only left outerjoins, our $\mathcal{PSNS}\,(T)$ has exactly one pair $(ps, ns)$ (Algorithm 2). The $ns$ set contains all relations which are null-supplied together with $T$, and, it can be proven, it can be obtained from the nullification sets $NS_X$ defined in [10] by finding relations which have common nullification predicates with $T$, i.e., $ns = \{R | R \in Rels(V), NS_T \subseteq NS_R\}$.

Both representations [3] and [10] allow the search space generation algorithm to consider extra join sequences, hence better final execution plans can be found.

Two seminal papers describing incremental maintenance of materialized views with outerjoins are [4] and [6].

The Griffin and Kumar paper [4] gives maintenance expressions for propagation of updates through semijoins and outerjoins. As described in Larson and Zhou [6], these maintenance expressions become inefficient when large number of view rows are affected by a base relation update.

The Larson and Zhou algorithms are based on the join-disjunctive normal form introduced by Galindo-Legaria in [3]. Their incremental maintenance algorithm consists of a series of steps: one step for computing and applying the primary delta, then a set of subsequent steps for applying secondary deltas to delete or insert null-supplied tuples. The primary delta is saved and reused in the computation of the secondary deltas. This computation may need to access again the base relations in order to correctly compute the null-supplied tuples. Their proposed solution uses a separate **SQL** statement to implement each of the needed steps. For example, for the view $V_1$ depicted in Figures 1 and 2, for the relation $X_2$ the view update algorithm will consist of five steps, each implemented by a separate **SQL** statement: computing and applying the primary delta, and computing and applying four secondary deltas corresponding to the join-disjunctive normal form terms $X_1Y_1Y_2$, $X_1$, $R_1R_2T_1T_2$, and $R_1R_2$. By comparison, $\mathcal{PSNS}\,(X_2)$ (Figure 2) has two tuples $(\{X_1\}, \{R_1, R_2, T_1, T_2, X_2\})$ and $(\{R_1\}, \{X_1, X_2, Y_1, Y_2\})$.
The tuple $(\{X_1\}, \{R_1, R_2, T_1, T_2, X_2\})$, for which the don't care set is $\{Y_1, Y_2\}$, can be interpreted as a compact representation for the two normal form terms $X_1Y_1Y_2$ and $X_1$; while the tuple $(\{R_1\}, \{X_1, X_2, Y_1, Y_2\})$, with the don't care set $\{R_2, T_1, T_2\}$, can be interpreted as a compact representation for the other two normal form terms $R_1R_2T_1T_2$, and $R_1R_2$.

## 6. CONCLUSIONS

In this paper, we presented the $\mathcal{PSNS}\,()$ (*P*reserved *S*ide / *N*ull-supplied *S*ide) Algorithm, which builds $\mathcal{PSNS}$-annotated normalized join trees, designed and implemented in SQL Anywhere **RDBMS** for incremental maintenance of materialized views with outerjoins.
Firstly, the algorithm allows us to generate just a single maintenance update statement for each materialized outerjoin view. This in turn allows powerful optimizations to be applied while processing the update statements, in order to achieve better performance.
Secondly, the computation of $V(nn(\Delta T))$ ($\neg\Delta T$-*null-supplied* tuples) in the same time with $\mathcal{N}ull\,(\Delta T, T, V)$ ($\Delta T$-*null-supplied* tuples) allows us to support an extended class of immediate materialized views, due to fewer restrictions being imposed on the definitions of the views.
Thirdly, since only one update statement is used, no intermediate temporary tables need to be saved during a view update operation.

## 7. REFERENCES


[1] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. SQL Anywhere: A holistic approach to database self-management. In *Proceedings, ICDE Workshops (Self-Managing Database Systems)*, pages 414–423, Istanbul, Turkey, Apr. 2007. IEEE Computer Society Press.

[2] E. F. Codd. Extending the database relational model to capture more meaning. ACM *Transactions on Database Systems*, 4(4):397–434, Dec. 1979.

[3] C. Galindo-Legaria. Outerjoins as disjunctions. In ACM SIGMOD *International Conference on Management of Data*, pages 348–358, Minneapolis, Minnesota, May 1994. Association for Computing Machinery.

[4] T. G. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. ACM SIGMOD *Record*, 27(3):22–27, 1998.

[5] International Standards Organization. *(ANSI/ISO) 9075-2,* SQL *Foundation*, July 2008.

[6] P.-Å. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *Proceedings, 23rd* IEEE *International Conference on Data Engineering*, pages 56–65, 2007.

[7] A. Nica. System and methodology for generating bushy trees using a left-deep tree join enumeration algorithm. US Patent 7,184,998, Feb. 2007.

[8] A. Nica. Query optimization using materialized views in database management systems. US Patent 7,606,827, Oct. 2009.

[9] A. Nica, D. S. Brotherston, and D. W. Hillis. Extreme visualisation of the query optimizer search spaces. In ACM SIGMOD *International Conference on Management of Data*, pages 1067–1070, Providence, Rhode Island, June 2009.

[10] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In ACM SIGMOD *International Conference on Management of Data*, pages 671–682, Paris, France, June 2004. Association for Computing Machinery.