Incremental View Maintenance of Spatial Joins *

Glenn S. Iwerks and Hanan Samet†
Computer Science Department,
Center for Automation Research, Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland 20742
{iwerks,hjs}@cs.umd.edu

Abstract

A new approach to incremental view maintenance of spatial joins is presented. To update materialized views after a transaction it is often more efficient to recompute the query in terms of changes to the base relations instead of recomputing the query from scratch. Changes to the base relations are stored in separate differential tables during a transaction. The differential tables are used by an incremental view maintenance algorithm to update the view at the end of the transaction. In the case of a join, a join operator is applied multiple times to a combination of differential tables and base relations. There are many ways to implement the join operator used within an incremental view maintenance algorithm. For spatial joins, a join operator algorithm created specifically for the task can be more efficient than a conventional join algorithm.

In this paper a new spatial join operator algorithm designed for use within an incremental view maintenance algorithm is presented. Other algorithms used as join operators within the incremental view maintenance algorithm are described for purposes of comparison. The incremental view maintenance algorithm in which these join operators are used is presented along with a rigorous proof of its correctness. The performance of these join operator algorithms, as used within the incremental view maintenance algorithm, is evaluated via experiments.

Keywords: incremental view maintenance, spatial join, spatial data

1 Introduction

In a relational database a *relation* is a table of values where each column represents an attribute of a particular data type or domain (e.g. integers, strings, points, etc.). Each row in a relation is a set of related values over the attribute domains called a *tuple*. The *schema* is the set of names associated with each column of a relation.

A spatial database is a database in which spatial attributes can be stored. A spatial database also stores non-spatial data. For instance, spatial data types may consist of points, lines, and polygons. Numbers and character strings are examples of non-spatial data.

Among the operations available in a database is the join operation. A join involves two input relations and a join condition. The result is a subset of the cross product of the input relations for which the join condition holds. A spatial join uses a condition defined on spatial attributes.

*Technical Report CS-TR-4175, CAR-TR-952, University of Maryland, College Park, MD, August 2000

†The support of the National Science Foundation under Grant EIA-99-00268 and IRI-97-12715 is gratefully acknowledged.

For example, let $r(R)$ and $s(S)$ be relations. The schemas of relations r and s are R and S, respectively. Let attribute $\alpha \in R$ and $\beta \in S$ be point attributes. Let d be a nonnegative scalar value. The following SQL query finds all the tuples in $r \times s$ where $r.\alpha$ and $s.\beta$ are within distance d of each other.

```
SELECT *
FROM r, s
WHERE Distance(r.α, s.β) ≤ d
```

Relations are updated during transactions. A transaction is a set of changes to a database state such that the database is left in a consistent state when the transaction completes. The database state may not change by any other means. If a transaction fails to complete, then the database reverts to its state just before the attempted transaction.

A *view* is a virtual relation defined by a database query expression. When the base relations upon which a view is defined are updated through a transaction, the view is recomputed to reflect the change. This virtual relation can then be treated the same as base relations in their use in subsequent queries. A *materialized view* is a view that is stored to disk.

The *heuristic of inertia* [7] states that updates to relations involve only a small fraction of tuples found in a relation. Assuming this heuristic, many researchers have developed techniques to update materialized views incrementally. Only the portion of the view affected by the change is recomputed. This is known as *incremental view maintenance.*

There are several strategies for incremental view maintenance including *periodic* update, *deferred* update, and *immediate* update. Periodic updates are performed on *snapshots* [2, 12] only occasionally. This approach allows for fast queries and updates, but can only be used in systems that can tolerate stale data in the query results. Deferred updates [1, 5] delay updating views until the view is needed in a query. This allows faster updates but slower query processing times. Immediate updates are performed at the end of each transaction that modifies a base relation upon which a view is defined. Algorithms for immediate incremental view maintenance updates are presented in [4, 7]. A comparison of immediate, deferred and query modification [15] is discussed in [8] along with a performance analysis by experiments. A system that uses immediate, deferred, and periodic update strategies in the same environment is described in [6].

Most, if not all, previous research on view maintenance has focussed on incremental maintenance of views involving non-spatial data. In this paper we compare some different approaches to view maintenance of spatial join queries. A new approach specifically designed for spatial join view maintenance is presented. This is compared with two other view maintenance approaches for spatial joins. A non-incremental method is also compared.

Section 1.1 defines the notation used in this paper. The incremental view maintenance algorithm for joins is presented in Section 1.2. A rigorous proof of the algorithm presented in Section 1.2 is given in Section 1.3. The algorithms for spatial join operators are described in Section 1.4 including the new algorithm. Section 1.5 presents the results of our experimental evaluation of the different spatial join operator algorithms. Conclusions are drawn in Section 2.

1.1 Notation

A schema is a set of attribute names denoted by an uppercase letter. The ordering of attributes within the schema is not significant. For example, schema $S = \{\alpha, \beta\} = \{\beta, \alpha\}$ where α and β are

attribute names. Relation names are denoted by lower case letters. To show that relation s is a collection of related tuple values over the domain of schema S we write $s(S)$. For a given relation, the name of its schema is the same as the name of the relation, except that it is an uppercase letter, unless otherwise stated. If $S = \{\alpha_0, \alpha_1, ..., \alpha_{n-1}\}$ where each attribute α_i $(0 \leq i < n)$ is an attribute name, then we express the domain of α_i as D_{α_i}. The domain of S is $D_S = D_{\alpha_0} \times D_{\alpha_1} \times ... \times D_{\alpha_{n-1}}$ and thus $s \subseteq D_S$. For some tuple $\tau_s \in s$ we denote the value of attribute α_0 in tuple τ_s as $\tau_s[\alpha_0]$.

The join of two relations $q(Q)$ and $r(R)$ is denoted $q \bowtie_{\mathsf{P}} r$ where P is a predicate defined on the schema attributes $Q \cup R$. Sometimes we write $q \bowtie r$ where P is understood. The concatenation of two tuples $\tau_q \in q$ and $\tau_r \in r$ is written $\tau_q\tau_r$. The domain of tuple $\tau_q\tau_r$ is $D_Q \times D_R$. Without loss of generality, we do not consider the order of attributes within a tuple to be significant, and thus $\tau_q\tau_r = \tau_r\tau_q$. If the schemas $Q \cap R \neq \emptyset$, then we assume that attributes are renamed as appropriate to avoid ambiguity when a tuple $\tau_q\tau_r$ is joined.

Let $r(R)$ and $s(R)$ be relations. Let the function $\mathsf{insert}(r, s)$ return all the tuples from relation r and relation s as one relation (duplicates allowed). Let the function $\mathsf{delete}(r, s)$ return the tuples found in relation r that have no equivalent counterpart among the tuples of relation s. In the case of duplicates, only one duplicate tuple is deleted for r for each match found in s. As a means of shorthand, let $\mathsf{insert}(r, s) \equiv r \uplus s$ and $\mathsf{delete}(r, s) \equiv r - s$. The insertion of a tuple τ_r into relation s is written $s \uplus \tau_r$. The size or number of tuples contained in a relation r is denoted as $|r|$. For example, suppose relation $r = \{(\text{a,b}),(\text{a,b})\}$ and relation $s = \{(\text{a,b})\}$ then $r \uplus s = \{(\text{a,b}),(\text{a,b}),(\text{a,b})\}$, $r - s = \{(\text{a,b})\}$, and $|r| = 2$.

A differential table is an auxiliary relation associated with a base relation or intermediate query result. A differential table contains all the tuples inserted into, or all the tuples deleted from, a relation during a transaction. If s is the state of a relation before some transaction Φ, then s' denotes the state of a relation after transaction Φ. Symbol i_s denotes the relation of tuples inserted into s during transaction Φ, and d_s is the relation of tuples deleted during the transaction. Relations i_s and d_s are the differential tables for relation s.

1.2 Incremental Update of Spatial Join Views

The view maintenance algorithm is described in terms of insert, delete and join operations on relations. Once the correctness of the incremental view maintenance algorithm is established, different algorithms used for the join operation are described. These are compared experimentally and the results presented.

Here are some assumptions. A tuple can be deleted and then reinserted during the same transaction. A join operation is a binary operation in that it joins two input relations. Joins of more than two relations can be achieved by nesting operations, e.g. $((r \bowtie s) \bowtie t)$. It is assumed that no attempt will be made to delete a tuple from a base relation that is not already in the relation, in other words $(d_s \subseteq s)$. Relations may contain duplicate tuples. If a relation has no specified key or discriminator, then the value of the whole tuple is assumed to be the tuple disciminator. It is assumed that the old version of each base relation (e.g. state of joined tables the last transaction) is available. It is also assumed that differential tables i_s and d_s are available for each input relation.

The differential tables used here are similar to *hypothetical relations* [16]. For some relation r, the intersections $i_r \cap r$ and $i_r \cap d_r$ are not necessarily empty. This is in contrast to the disjoint property of the tables described in [4]. The difference between hypothetical relations and our differential tables is that no extra attributes are required. The schema of each differential table is the same as its associated base relation. Another characteristic of this algorithm is that it does not require the use of keys, or any addition bookkeeping information to accompany base relations or query results.

Base relation update: Let relation i_s be the tuples inserted into relation s during transaction Φ. The insertion update to s is expressed as $s' = s \uplus i_s$. Let relation d_s be all tuples deleted from relation s during the same transaction. The deletion update to s is expressed as $s' = s - d_s$. By combining these two expressions we get $s' = (s \uplus i_s) - d_s$. The parentheses show proper precedence needed in the case that a tuple is inserted and deleted during the same transaction.

View update: Now, consider all views v in a database system. An update v resulting in v' can be expressed in terms of differential tables $v' = (v \uplus i_v) - d_v$. After the base relations in a system are updated by a transaction, the differential tables are then computed for each view. Views depending only on base relations are computed first followed by views depending on other views where their dependencies have already been computed. The dependencies must not be circular. After all differential tables are computed for all views, the views are updated by inserting tuples contained in the i_v differential tables followed by deleting the tuples contained in the d_v differential tables. Once this is finished, the differential tables are cleared and the system is ready for another transaction. This process allows for incremental updates of nested views.

Join update: Let $j' = l' \bowtie r$ be a join operation update after relation l is updated during some transaction Φ. By substitution this expression can be rewritten in terms of the relation l's differential tables and the state of l before the transaction as $j' = ((l \uplus i_l) - d_l) \bowtie r$. The join operation is distributive over the $\uplus$ and $-$ operations, so this expression can be rewritten as $j' = ((l \bowtie r) \uplus (i_l \bowtie r)) - (d_l \bowtie r)$. Substituting j for $(l \bowtie r)$ the expression becomes $j' = (j \uplus (i_l \bowtie r)) - (d_l \bowtie r)$. The $(i_l \bowtie r)$ term is the expression for the insert differential table of relation j, and the term $(d_l \bowtie r)$ is the expression for the delete differential table for relation j. This gives us a simple expression for the case when only one of the joined relations changes during a transaction.

The case when both of the joined relations change during a transaction is more complicated. Here we derive an expression to compute the differential tables involving changes in both of the joined relations. To create an incremental view maintenance algorithm for join, the following expressions are used.

Exp. 1.2.1 $\quad l' = ((l \uplus i_l) - d_l)$
Exp. 1.2.2 $\quad r' = ((r \uplus i_r) - d_r)$
Exp. 1.2.3 $\quad j' = l' \bowtie r'$
Exp. 1.2.4 $\quad j' = ((l \uplus i_l) - d_l) \bowtie ((r \uplus i_r) - d_r)$

Expressions 1.2.1 and 1.2.2 represent the update of two relations after a transaction. Expression 1.2.3 represents a join of those relations in terms of the state of the joined relations after a transaction. By substitution of expression 1.2.1 and 1.2.2 into expression 1.2.3, the join operation can be expressed in terms of the states of the joined relations before the transaction, and the differential tables. This is shown in expression 1.2.4. To make expression 1.2.4 useful, the joins are distributed over the relation addition ($\uplus$) and subtraction ($-$) operations to produce expression 1.2.5. A proof of the correctness of expression 1.2.5 is given in Section 1.3.

Exp. 1.2.5

$$j' = (j \uplus (((((i_l \bowtie r) \uplus (l \bowtie i_r)) \uplus (i_l \bowtie i_r)) - (i_l \bowtie d_r)) - (i_r \bowtie d_l))) \\ - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

The subexpressions of expression 1.2.5 form the basis for an algorithm to calculate the join view

differential tables i_j and d_j (see expressions 1.2.6 and 1.2.7 below). Section 1.4 explains the algorithm derived from these subexpressions.

Exp. 1.2.6

$$i_j = ((((i_l \bowtie r) \uplus (l \bowtie i_r)) \uplus (i_l \bowtie i_r)) - (i_l \bowtie d_r)) - (i_r \bowtie d_l)$$

Exp. 1.2.7

$$d_j = (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

1.3 Correctness

In this section we prove the correctness of expression 1.2.5. Formal definitions of the addition, subtraction and join of relations are presented. Following the definitions, we present six rewrite rules used in the proof of expression 1.2.5.

Definition 1.3.1 *Addition of relations*: Given relations $s(S)$ and $t(S)$, $s \uplus t = \{\tau : \tau \in s \vee \tau \in t\}$, duplicates allowed.

Definition 1.3.2 *Subtraction of relations*: Given relations $s(S)$ and $t(S)$, $s - t = \{\tau : \tau \in s \wedge \tau \notin t\}$ where only one tuple element of s is removed for each matching element in t.

Definition 1.3.3 *Join of relations*: Given relations $s(S)$ and $t(T)$ and predicate P defined on a subset of $S \cup T$, $s \bowtie_{\mathsf{P}} t = \{\tau_s\tau_t : \tau_s \in s \wedge \tau_t \in t \wedge \mathsf{P}(\tau_s, \tau_t)\}$.

In the following, rewrite rules s, t and v are relations and $\bowtie$, $-$, and $\uplus$ are binary operations on relations. A sample proof of a rewrite rule is found in Appendix 3.

Rule 1: *Distribution of join over subtraction*
$(s - t) \bowtie v \longleftrightarrow (s \bowtie v) - (t \bowtie v)$

Rule 2: *Distribution of join over addition*
$(s \uplus t) \bowtie v \longleftrightarrow (s \bowtie v) \uplus (t \bowtie v)$

Rule 3: *Commutativity of join*
$s \bowtie t \longleftrightarrow t \bowtie s$

Rule 4: *Associativity of subtraction*
$s - (t \uplus v) \longleftrightarrow (s - t) - v$

Rule 5: *Associativity of addition*
$(s \uplus t) \uplus v \longleftrightarrow s \uplus (t \uplus v)$

Rule 6: *Associativity of subtraction of subset relations*
$(s \uplus t) - v \longleftrightarrow s \uplus (t - v)$ iff $v \subseteq t \vee s \cap v = \emptyset$

Rule 7: *Commutativity of addition*
$s \uplus t \longleftrightarrow t \uplus s$

By successively applying rules 1, 2 and 3 to expression 1.2.4, the join operators are pushed down to the lowest possible level. Rules are then applied to rearrange the join terms. The rewrite rules used for each step are indicated by the numbers inside curly braces at the end of each line. For example, {3,5} indicates one or more applications of rule 3 followed by one or more applications of rule 5.

Proof:

$$j' = ((l \uplus i_l) - d_l) \bowtie ((r \uplus i_r) - d_r)$$

{expression 1.2.4}

$$= ((l \uplus i_l) \bowtie ((r \uplus i_r) - d_r)) - (d_l \bowtie ((r \uplus i_r) - d_r))$$

{1}

$$= (((r \uplus i_r) \bowtie (l \uplus i_l)) - (d_r \bowtie (l \uplus i_l))) \\ - (((r \uplus i_r) \bowtie d_l) - (d_r \bowtie d_l))$$

{3,2}

$$= (((r \bowtie (l \uplus i_l)) \uplus (i_r \bowtie (l \uplus i_l))) - ((l \bowtie d_r) \uplus (i_l \bowtie d_r))) \\ - (((r \bowtie d_l) \uplus (i_r \bowtie d_l)) - (d_r \bowtie d_l))$$

{3,2}

$$= ((((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))) \\ - ((l \bowtie d_r) \uplus (i_l \bowtie d_r))) - (((r \bowtie d_l) \uplus (i_r \bowtie d_l)) - (d_r \bowtie d_l))$$

{3,2}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))) \\ - (((l \bowtie d_r) \uplus (i_l \bowtie d_r)) \uplus (((r \bowtie d_l) \uplus (i_r \bowtie d_l)) - (d_r \bowtie d_l)))$$

{4}

- $d_r \subseteq r$ is true by assumption.
- $d_r \subseteq r \Rightarrow (d_r \bowtie d_l) \subseteq (r \bowtie d_l)$
  $\Rightarrow (d_r \bowtie d_l) \subseteq ((r \bowtie d_l) \uplus (i_r \bowtie d_l))$

so

$$j' = (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))) \\ - ((((l \bowtie d_r) \uplus (i_l \bowtie d_r)) \uplus ((r \bowtie d_l) \uplus (i_r \bowtie d_l))) - (d_r \bowtie d_l))$$

{6}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))) \\ - (((i_l \bowtie d_r) \uplus ((l \bowtie d_r) \uplus ((r \bowtie d_l) \uplus (i_r \bowtie d_l)))) - (d_r \bowtie d_l))$$

{7,5}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))) \\ - (((i_l \bowtie d_r) \uplus ((i_r \bowtie d_l) \uplus ((r \bowtie d_l) \uplus (l \bowtie d_r)))) - (d_r \bowtie d_l))$$

{7,5}

$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$
$\quad - ((((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \uplus ((r \bowtie d_l) \uplus (l \bowtie d_r))) - (d_r \bowtie d_l))$

{5}

- $d_r \subseteq r$ is true by assumption.
- $d_r \subseteq r \Rightarrow (d_r \bowtie d_l) \subseteq (r \bowtie d_l)$
  $\Rightarrow (d_r \bowtie d_l) \subseteq ((r \bowtie d_l) \uplus (l \bowtie d_r))$

so
$j' = (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$
$\quad - (((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \quad \uplus (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l)))$

{6}

$= ((((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$
$\quad - ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$

{4}

$= (((l \bowtie r) \uplus ((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))))$
$\quad - ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$

{5}

- $d_l \subseteq l \wedge d_r \subseteq r$ is true by assumption.
- $d_l \subseteq l \Rightarrow (i_r \bowtie d_l) \subseteq (i_r \bowtie l)$
- $d_r \subseteq r \Rightarrow (i_l \bowtie d_r) \subseteq (i_l \bowtie r)$
- $(i_r \bowtie d_l) \subseteq (i_r \bowtie l) \wedge (i_l \bowtie d_r) \subseteq (i_l \bowtie r)$
  $\Rightarrow ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \subseteq ((i_l \bowtie r) \uplus (l \bowtie i_r))$
  $\Rightarrow ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \subseteq ((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$

so
$j' = ((l \bowtie r) \uplus (((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$
$\quad - ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$

{6}

$= ((l \bowtie r) \uplus ((((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$
$\quad - (i_l \bowtie d_r)) - (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$

{4}

$j' = ((l \bowtie r) \uplus (((((i_l \bowtie r) \uplus (l \bowtie i_r)) \uplus (i_l \bowtie i_r))$
$\quad - (i_l \bowtie d_r)) - (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$

{5}

= expression 1.2.5

{substitution of j for $(l \bowtie r)$}

□

1.4 Spatial Join Operator Algorithms

Converting expression 1.2.5 into an algorithm is straightforward. Given below is the function Incremental_View_Update(). It computes the differential relations for an incremental join update after a transaction Φ has taken place. The resulting relation i_j contains all the new tuples to be added to the join result, and d_j contains the tuples to be deleted. Input relations l and r are the state of the joined relations before the transaction. Input relations i_l and i_r are the tuples inserted into the joined relations as a result of transaction Φ, and relations d_l and d_r are the deleted tuples. The superscripts shown on the join operators in the following algorithm distinguish them to aid in further discussion.

```
function Incremental_View_Update(l, i_l, d_l, r, i_r, d_r)
    : return two relations
  begin
    i_j ← (i_l ⋈^0 r)
    i_j ← i_j ⊎ (l ⋈^1 i_r)
    i_j ← i_j ⊎ (i_l ⋈^2 i_r)
    i_j ← i_j − (i_l ⋈^3 d_r)
    i_j ← i_j − (i_r ⋈^4 d_l)
    d_j ← (r ⋈^5 d_l)
    d_j ← d_j ⊎ (l ⋈^6 d_r)
    d_j ← d_j − (d_r ⋈^7 d_l)
    return i_j and d_j
  end
```

The question is how to implement the join operations used in Incremental_View_Update(). Here we describe different algorithms for join operations. Each of the join operation algorithms will be compared experimentally for performance within function Incremental_View_Update() in Section 1.5.

The nested loop join [14] is a classic algorithm to implement spatial and non-spatial joins. The following function implements a nested loop join. Parameters s and t are the relations $s(S)$ and $t(T)$ to be joined. Parameter P is the join predicate defined on a subset of $S \cup T$. For two given tuples $\tau_s \in s$ and $\tau_t \in t$, if $\mathsf{P}(\tau_s, \tau_t)$ is true, then the tuples are included in the join result; otherwise they are omitted. Relation j is returned as the function value. The schema of j is the concatenation of schemas S and T where attribute names in the intersection of S and T are appropriately renamed to avoid conflict.

```
function Nested_Loop_Join(s, t, P) : return relation
  begin
    j ← ∅
    foreach tuple τ_s ∈ s do
      foreach tuple τ_t ∈ t do
        if P(τ_s, τ_t) = TRUE then
          j ← j ⊎ τ_s τ_t
        endif
      endfor
    endfor
```

```
    return j
  end
```

When predicate P is a check for some minimum distance between spatial attributes, then the join is called a spatial join. For example, $\mathsf{P} \equiv \mathsf{distance}(\tau_s[\alpha], \tau_t[\beta]) \leq d$ where $\alpha \in S$ and $\beta \in T$ and d is a nonnegative scalar value. The Nested_Loop_Join() requires $|s|$ scans of relation t and one scan of relation s.

Using an indexed spatial join algorithm, the number of disk accesses can be greatly reduced. The indexed spatial join algorithm that we use in our experiments is called the incremental spatial join [9]. We denote this algorithm as Incremental_Spatial_Join(). The algorithm works using spatial indices on the join attributes of each relation. In our case, the type of spatial index used is the PMR quadtree [13]. The indices form a tree structure where nodes cover a spatial extent defined by a polytope whose faces are orthogonal to the coordinate axes (e.g., a 2D rectangle, a 3D parallelepiped, etc.). Each bucket contains either a collection of spatial features (leaf nodes), or smaller buckets (gray nodes). The algorithm works by descending the index trees and maintaining a priority queue of object pairs (buckets or features), one object from each index, sorted by distance. When a pair of objects (λ_1, λ_2) is pulled off the priority queue and one of the objects, say λ_1, is a bucket, then for each object ω found in bucket λ_1, a pair (ω, λ_2) is reinserted into the priority queue. When the next pair of objects pulled off the queue are both features, then they are reported to the caller. The algorithm is used by initializing a search, then repeatedly calling a function that reports one pair of features for each call. Each pair of features reported is incrementally farther apart than the previous.

Function One_Index_Spatial_Join() shows a new spatial join operation algorithm. It is designed for use when the size of one of the join relations is known to be very small relative to the other.

```
function One_Index_Spatial_Join(s, α, t, β, d)
    : return relation
  begin
1.  j ← ∅
2.  foreach tuple τs ∈ s do
3.     Initialize_Incremental_Nearest_Neighbor(τs[α], t, β, d)
4.     while Incremental_Nearest_Neighbor_Not_Finished() do
5.        τt ← Next_Incremental_Nearest_Neighbor()
6.        j ← j ⊎ τsτt
7.     endwhile
8.  end
9.  return j
  end
```

Parameters s and t are relations such that $|s| << |t|$. Parameter $\alpha \in S$ and $\beta \in T$ and are both spatial attributes. Parameter d is a nonnegative scalar value. It is assumed that an index on the spatial join attribute β of relation t is maintained. The algorithm iterates through each tuple of relation s. For each tuple $\tau_s \in s$ it invokes the incremental nearest neighbor algorithm as described in [10].

The incremental nearest neighbor algorithm incrementally reports each successive tuple $\tau_t \in t$ where $\tau_t[\beta]$ is the next closest spatial feature to $\tau_s[\alpha]$. The incremental nearest neighbor algorithm is

initialized in line 1. The **while** loop in lines 4 to 7 iterates until all tuples τ_t in relation t have been reported such that $\mathsf{distance}(\tau_s[\alpha], \tau_t[\beta]) \leq d$.

1.5 Analysis

The join operations used in Incremental_View_Update() are the focus of the analysis. We compare the performance between three different versions on some test data sets and queries. We also compare these three incremental methods to that of recomputing queries from scratch after updates.

Each join operation in algorithm Incremental_View_Update() is identified with a superscripted number to distinguish it in the following discussion. For example, the join operation shown in the sixth line of Incremental_View_Update() , $d_j \leftarrow (r \bowtie^5 d_l)$, is identified as join operation $\bowtie^5$. A range of join operations is indicated using a hyphen. For example, the notation $\bowtie^{1-3}$ refers to join operations $\bowtie^1$, $\bowtie^2$, and $\bowtie^3$. A list of join operations is indicated using a comma. For example, the notation $\bowtie^{2,4}$ refers to join operations $\bowtie^2$, and $\bowtie^4$.

Version 1 of the incremental update algorithm uses the nested loop join to implement all the join operations. In other words, for all the join operations in Incremental_View_Update(), $\bowtie^{0-7}\equiv$ Nested_Loop_Join(). By the heuristic of inertia, differential tables are much smaller than base relations. The outer loop iterates over base relations when joining a base relation with a smaller differential table. We refer to this as *version 1.*

Version 2 of the incremental update algorithm uses the indexed spatial join algorithm to implement all the join operations. That is, for each spatial join operation in Incremental_View_Update() the Incremental_Spatial_Join() algorithm is used. We refer to this as *version 2.*

Version 3 is a little more complicated. Joins between a differential table and a base relation (e.g. relation l or r) are performed by One_Index_Spatial_Join(). In other words, $\bowtie^{0,1,5,6}\equiv$ One_Index_Spatial_Join() For the other join operations $\bowtie^{2,3,4,7}\equiv$ Nested_Loop_Join(). We refer to this new algorithm as *version 3* or the Spatial_Join_Incremental_Update() algorithm.

1.6 The Spatial Spreadsheet

The Spatial Spreadsheet [11] serves as a testbed for the algorithms presented in this paper (see Figure 1). It is a front end to a spatial relational database. In the classic spreadsheet paradigm, cell values are non-spatial data types whereas in the Spatial Spreadsheet, cell values are database relations.

The purpose of the Spatial Spreadsheet is to combine the power of a spatial database with that of the spreadsheet. The advantages of a spreadsheet are the ability to organize data, to formulate operations on that data quickly through the use of row and column operations, and to propagate changes in the data throughout the system. The Spatial Spreadsheet is made up of a 2D array of cells. Each cell in the Spatial Spreadsheet can be referenced by the cell's location (row, column). A cell can contain two types of relations: a base relation or a query result. A query result is a materialized view [7] defined on the base relations or other materialized views. The user can pose a simple query in an empty cell. For instance, a cell might contain the result of a spatial join between base relations. Simple queries are operations like selection, projection, join, spatial join [9], window [3], nearest neighbor [10], etc. Simple queries can be composed to create complex queries by using the result of one simple query as the input to another. If a base relation is updated, the effects of those changes are propagated to other cells. This is known as view maintenance. Due to the nature of the Spatial Spreadsheet, view maintenance is performed immediately. The displays of the spreadsheet must be updated immediately to reflect changes for the user.

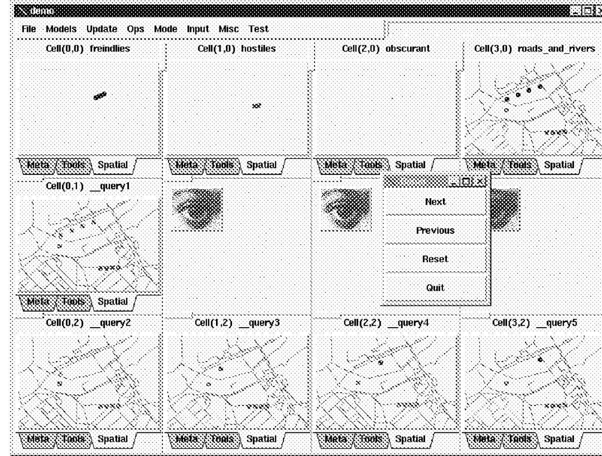

Figure 1: The Spatial Spreadsheet

1.7 Experiment Results

In the experiment, a test spatial join query Q is performed between two base relations a and b. Each base relation initially contains 1000 tuples. Each tuple has a 2D point feature attribute. All point features are distributed uniformly over a square area. The range of possible x and y coordinate values for each point is $[0, 1)$. Each relation is updated 30 times from transactions during each experimental run. A fixed number of N tuples are changed per relation during each of the 30 transaction updates within a run. The number N varies from run to run but not from transaction to transaction within a run. A tuple is changed by deleting it and replacing it with a new tuple. A deleted tuple is replaced with a new tuple such that the coordinate values of the new point feature within the tuple are within 5% of the coordinate values of the old point feature. For example, a point feature at $(0.5, 0.5)$ may be moved to $(0.54, 0.54)$ but not to $(0.6, 0.6)$ since $(0.54 - 0.5) \leq 0.05$ (less than 5% change), but $(0.6 - 0.5) > 0.05$ (more than 5% change). The test query Q joins the point features found in relation a to point features in relation b that are within a distance of 0.05 coordinate units of each other. Distance is measured using the Euclidean distance metric. The number of tuples changed per relation per update varies over 5 sets of runs where N =2,4,8,16, and 32.

Figure 2 presents a graphical summary of the experiments. The x-axis is the number of tuples changed per update. The y-axis is the time needed to recalculate the spatial join result in microseconds. Each data point is the average update time calculated from the 30 updates in each run. The *version 1* algorithm is never faster than any of the other algorithms. Only two data points are shown for the *version 1* algorithm because the runtime is very large for values of N greater than 4. The non-incremental algorithm, *version 2*, and the *version 3* algorithm converge at around 32 changed tuples per transaction update per relation. When the base relations change by 8 tuples per transaction update per relation, the *version 2* algorithm is 6.8 times faster than the non-incremental method, the *version 3* algorithm (our new Spatial_Join_Inc_Update() algorithm) is 3.4 times faster than the *version 2* algorithm, and is 23.1 times faster than the non-incremental algorithm.

2 Conclusion

The Spatial_Join_Inc_Update() algorithm, known as *version 3*, yields significant performance improvements under experimental conditions. The next best algorithm is *version 2*, which uses the

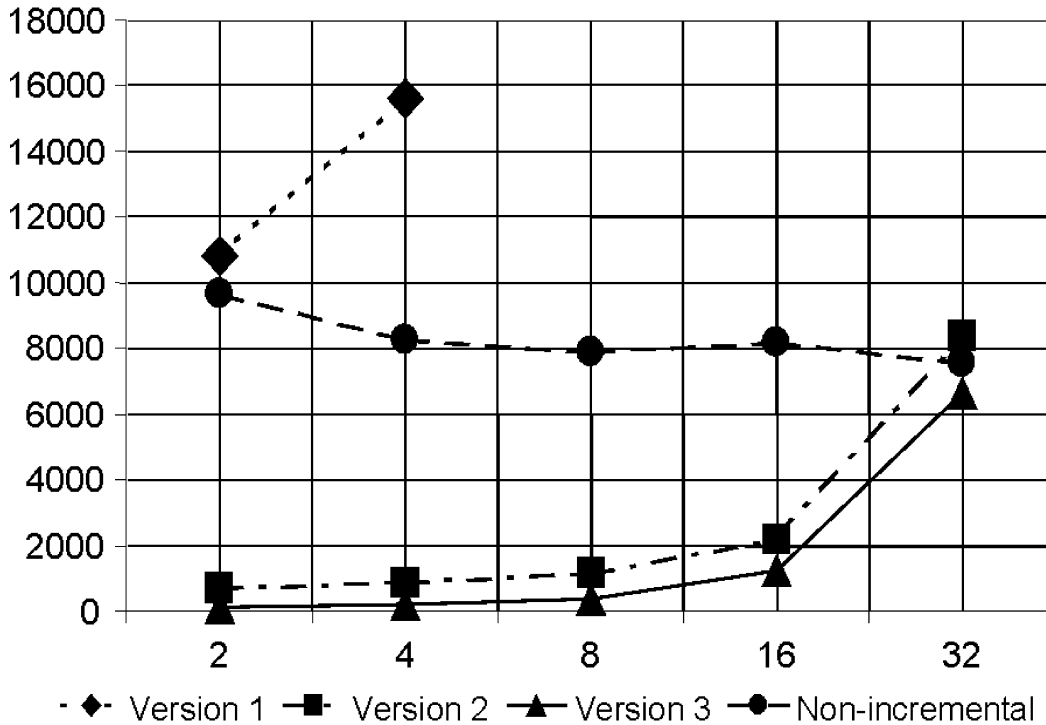

Figure 2: Experiment results

Incremental_Spatial_Join() for all join operations.

The operation $j' = (j \uplus (i_l \bowtie r)) - (d_l \bowtie r)$ requires three indices (2 new ones) for an index-based spatial join algorithm such as the incremental spatial join [9], one for each relation r, i_l, and d_l. The size of the tables i_l and d_l are small, $|i_l| << |r|$ and $|d_l| << |r|$, so the time needed to create the indices is relatively small; however, they will still require some disk accesses during processing. Differential tables can be loaded into memory once and used over and over again if no indices need to be created on them. This is one reason why the new Spatial_Join_Inc_Update() algorithm (*version 3*) performs better for small values of N. To more fully characterize the conditions under which the *version 3* algorithm outperforms the others, more testing is needed. For future work we plan to vary the size of the base relations, vary the join distance, and join other types of spatial features in our experiments.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 245–256, San Jose, CA, May 1995.

[2] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Sixth International Conference on Very Large Data Bases*, pages 86–91, Montreal, Quebec, Canada, October 1980.

[3] W. G. Aref and H. Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, April 1997.

[4] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., June 1986.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 469–480, Montreal, Quebec, Cananda, June 1996.

[6] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 405–416, Tucson, AZ, May 1997.

[7] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C., May 1993.

[8] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 440–453, San Francisco, May 1987.

[9] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. (Also University of Maryland Computer Science TR–3919).

[11] G. Iwerks and H. Samet. The spatial spreadsheet. In *Visual Information and information Systems: Third International Conference, VISUAL'99*, pages 317–324, Amsterdam, The Netherlands, June 1999. Springer-Verlag.

[12] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms. A snapshot differential refresh algorithm. In C. Zaniolo, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 53–60, Washington, D.C., May 1986.

[13] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 270–277, San Francisco, May 1987.

[14] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, New York, third edition, 1996.

[15] M. Stonebraker. Implementation of integrity constraints and views by query modification. In W. Frank King, editor, *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 65–78, San Jose, California, May 1975.

[16] J. Woodfill and M. Stonebraker. An implementation of hypothetical relations. In M. Schkolnick and C. Thanos, editors, *9th International Conference on Very Large Data Bases*, pages 157–166, Florence, Italy, Proceedings, October 1983.

3 Appendix: Rewrite Rule Proof

In this appendix, a sample proof of one of the rewrite rules is presented. The following definitions are used in the proof.

- Definition 3.1 *Equality of tuples*: Let tuple $\tau_s \in s(S)$ and tuple $\tau_t \in t(S)$. If $\forall \alpha \in S : \tau_s[\alpha] = \tau_t[\alpha]$ then $\tau_s = \tau_t$.

- Definition 3.2 *Subtraction of tuples*: Let tuple $\tau_s \in s(S)$ and tuple $\tau_t \in t(S)$. If $\tau_s = \tau_t$ then $\tau_s - \tau_t = \emptyset$ else $\tau_s - \tau_t = \tau_s$.

- Definition 3.3 *Join of tuples*: Let tuple $\tau_s \in s(S)$ and tuple $\tau_t \in t(T)$. Consider a join operation $s \bowtie_{\mathsf{P}} t$. P is a predicate defined on a subset of $S \cup T$. If $\mathsf{P}(\tau_s, \tau_t)$ is true then $\tau_s\tau_t \in (s \bowtie_{\mathsf{P}} t)$ else $\tau_s\tau_t \notin (s \bowtie_{\mathsf{P}} t)$.

In the following proof we consider a distinct set of tuples $\tau_s \in s(S), \tau_t \in t(S)$, and $\tau_v \in v(V)$, and a join operation $\bowtie_{\mathsf{P}}$ where P is a predicate defined on $S \cup V$.

Rule 1: $(s - t) \bowtie v \longleftrightarrow (s \bowtie v) - (t \bowtie v)$

Proof:

The proof is presented in the form of truth table in Figure 3. The first three columns of the truth table represent all the possible cases that we need to consider. The cases where predicate $\tau_s = \tau_t$ is true and $\mathsf{P}(\tau_s, \tau_v) \neq \mathsf{P}(\tau_t, \tau_v)$ are not shown. These states are not possible. If $\tau_s = \tau_t$ is true, then $\mathsf{P}(\tau_s, \tau_v) = \mathsf{P}(\tau_t, \tau_v)$ must be true. The value of the predicate in column 4, $\tau_s \in (s - t)$, is the inverse of predicate $\tau_s = \tau_t$. This follows from the definition of *equality of tuples* (Def. 3.1) and the definition of the *subtraction of tuples* (Def. 3.2). The value of the predicate in column 5, $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$, is false if predicate $\mathsf{P}(\tau_s, \tau_v)$ in column 2 is false by the definition of the *join of tuples* (Def. 3.3). The value of the predicate in column 5 is also false if $\tau_s \in (s - t)$ is false in column 4 since a tuple can not be joined if it doesn't exist. If $\mathsf{P}(\tau_s, \tau_v)$ is true and $\tau_s \in (s - t)$ is true, then $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$ is true. The predicate $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ in column 6 is true if predicate $\mathsf{P}(\tau_s, \tau_v)$, column 2, is true, else it is false. This follows from the definition of the *join of tuples* (Def. 3.3). The predicate $\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$ in column 7 is true if predicate $\mathsf{P}(\tau_t, \tau_v)$, column 3, is true, else it is false. This also follows from the definition of the *join of tuples* (Def. 3.3). Finally, in column 8, predicate $\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is true if predicate $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ in column 6 is true and $\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$ in column 7 is false. It is also true if $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ in column 6 is true, and $\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$ is true, and $\tau_s = \tau_t$ in column 1 is false. In all other cases, predicate $\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is false. This follows because if $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ is false, then $\tau_s\tau_v$ does not exist in the left operand of the subtraction, and will not be in the result of the subtraction. If $\tau_s\tau_v$ does exist in the left operand of the subtraction operation, then it can be removed if there exists an equivalent tuple in the right operand (Def. 3.2). Otherwise it will not be removed and $\tau_s\tau_v$ will be included in the resulting relation of the subtraction operation.

By examining the table in Figure 3 we see that either $\tau_s\tau_v \notin ((s - t) \bowtie_{\mathsf{P}} v) \wedge \tau_s\tau_v \notin ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is true or $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v) \wedge \tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is true. In other words, column 5, $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$, and column 8, $\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$, match in every row. Therefore rule 1 holds. $\square$

The proofs of the other rewrite rules are similar.

$\tau_s = \tau_t$	$\mathsf{P}(\tau_s, \tau_v)$	$\mathsf{P}(\tau_t, \tau_v)$	$\tau_s \in (s - t)$	$\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$	$\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$	$\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$	$\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$
F	F	F	T	F	F	F	F
F	F	T	T	F	F	T	F
F	T	F	T	T	T	F	T
F	T	T	T	T	T	T	T
T	F	F	F	F	F	F	F
T	T	T	F	F	T	T	F

Figure 3: Truth table where T is `TRUE` and F is `FALSE`. The conditions $(\tau_s = \tau_t) =$ T and $\mathsf{P}(\tau_s, \tau_v) \neq \mathsf{P}(\tau_t, \tau_v)$ are not shown since these states are not possible.

The proofs of the other rewrite rules are similar. Due to space limitaitons they are not presented here.