An Experimental Performance Evaluation of Spatio-Temporal Join Strategies

Seung-Hyun Jeong, Norman W. Paton, Alvaro A. A. Fernandes, Tony Griffiths
Department of Computer Science, University of Manchester,
Manchester M13 9PL, UK
email: (jeongs,norm,alvaro,griffitt)@cs.man.ac.uk

May 8, 2004

Abstract

Many applications capture, or make use of, spatial data that changes over time. This requirement for effective and efficient spatio-temporal data management has given rise to a range of research activities relating to spatio-temporal data management. Such work has sought to understand, for example, the requirements of different categories of application, and the modelling facilities that are most effective for these applications. However, at present, there are few systems with fully integrated support for spatio-temporal data, and thus developers must often construct custom solutions for their applications. Developers of both bespoke solutions and of generic spatio-temporal platforms will often need to support the inter-relating of large spatio-temporal data sets. Supporting such requests in a database setting involves the use of join operations with both spatial and temporal conditions – spatio-temporal joins. However, there has been little work to date on spatio-temporal join algorithms or their evaluation. This paper presents an evaluation of several approaches to the implementation of spatio-temporal joins that build upon widely available indexing techniques. The evaluation explores how several algorithms perform for databases with different spatial and temporal characteristics, with a view to helping developers of generic infrastructures or custom solutions in the selection and development of appropriate spatio-temporal join strategies.

1 Introduction

Real world objects are naturally associated with space and time: *spatial information* represents positions and/or extents of the objects in space, and *temporal information* represents the existence of the objects in time. When the spatial properties of an object evolve over time, the object has a *history* of spatial data.

A significant number of applications in fields such as the Earth Sciences, Cartography and Land Information Systems involve the storage and analysis of large amounts of historical spatial data (e.g. [4, 5, 26]). Such data can be challenging to represent and manage, giving rise to much work on, for example, ST (Spatio-Temporal) data models [31, 3, 8, 6] and indexing techniques [17, 29, 19]. We note, however, that the development of systems that provide comprehensive support for spatio-temporal data is challenging, and that many application developers have resorted to custom solutions in the absence of widely available generic solutions. For example, a survey of experience managing spatio-temporal data describing administrative boundaries is given by [5], in which many custom-built systems are described.

Many applications that associate collections of spatio-temporal data on the basis of their aspatial and temporal properties are likely to end up developing or using spatio-temporal join algorithms. In ST databases, ST join queries, that combine two sets of ST objects according to join predicates embracing both spatial and temporal attributes, are likely to be important and expensive. For example, a spatio-temporal join query can identify which bus routes passed through a town in 2001. Surprisingly, however, there has been much less attention given to ST join algorithms in the literature than to, for example, spatio-temporal data models or indexes. In this context, this paper contributes to the understanding of ST joins by introducing and evaluating strategies for processing ST join queries. It is hoped that these results will be of interest not only to the developers of spatio-temporal data management systems, but also to the developers of custom applications.

Although there has been little work on ST joins, some adjacent areas have been extensively explored. For example, there has been intensive research on spatial join algorithms (e.g., [2, 16, 21]) and temporal join algorithms (e.g., [25, 32]) employing diverse index structures, hash tables, and sort-merge techniques. In this paper, the strategies are implemented using techniques based on index structures, following *Index-based Nested-Loop* [7, 18] and *Synchronized Tree Traversal* [2, 32] approaches, as discussed further in Section 2. As such, the paper shows how approaches proposed for supporting spatial joins can be extended for use with spatio-temporal data, and describes a performance evaluation of the resulting implementations. This activity can be seen to complement research on other aspects of spatio-temporal querying, which, for example, has investigated algorithms for moving objects [15] or for identifying nearest neighbours in space and time [24].

As the ST join algorithms employed in this paper might be considered to be "obvious" extensions of their spatial counterparts, the principal contribution is the experimental evaluation of the approaches rather than the algorithms themselves. The experiments compare alternative strategies over various database sizes, numbers of snapshots and input selectivities, using ST data collections generated by *GSTD* [28], a ST data generator. Although there have been several performance studies of spatial and of temporal join algorithms before, we know of no comparable study for spatio-temporal joins. In the absence of such a study, developers lack hard evidence on which to base design decisions that may be important to their development activities. Synthetic data is used because of the control it provides in support of experiment design, in which it is often necessary to vary a single factor at a time in order to understand the effect that factor has on performance. Clearly the risk exists in studying algorithms in use with synthetic data that this data is not representative of that found in real applications. However, although some benchmarks use real data (e.g. [27]), this approach may also involve unrepresentative data, and current practice in spatial data management seems to favour synthetic data for performance studies (e.g., [30]).

This paper is structured as follows : Section 2 presents an overview of existing spatial and temporal join algorithms; Section 3 describes the processing of ST joins; Section 4 explains the design of the experiments including QEPs (Query Execution Plans); in Section 5, an analysis of the experimental results is given; and lastly in Section 6, some conclusions are drawn from the experiments.

2 Related Work

Join algorithms are typically classified as *hash-based, sort-merge-based,* or *index-based*, depending on the data structure or data property exploited by the algorithm. Spatial and temporal join algorithms can also be classified into these groups. This paper, however, focuses on index-based algorithms, as (i) indexes on spatial, temporal and ST attributes of objects are likely to be widely available, as many different query operations can benefit from their presence; and (ii) previous evaluations of spatial (e.g., [1]) and temporal (e.g., [32]) joins testify to the effectiveness of index-based approaches.

Well known index-based spatial and temporal join algorithms fall into two groups: *Index-based Nested-Loop* (*INL*) [7, 18] and *Synchronized Tree Traversal* (*STT*) [2, 32]. INL is a variant of the simple nested-loop algorithm, where each object of the outer collection is scanned, and the join attribute of the object is used as the search key to an index structure for the inner collection. The pseudo-code for an INL algorithm is given in Figure 1 using an intersection predicate. In the INL algorithm, each index lookup for the inner collection is independent of all other lookups to that collection. Thus the index is accessed in an uncoordinated manner.

The main idea of STT is to reduce the number of visits to internal nodes of two index trees by traversing the indexes of the operands synchronously, as illustrated in Figure 2 for spatial joins with intersection predicates.

INL is specifically useful when only one operand is indexed (as may be the case, for example, for joins involving intermediate results), while STT requires indexes on both operands. Note that the intersection predicates in line 2 of *SpatialLookup*() and in line 3 of *SpatialSTT*() can be replaced with other predicates, e.g. a containment predicate.

Index-based join algorithms are, of course, affected by the index structures used, and various structures for spatial and temporal indexes have been proposed. This section focuses on R-trees [9] as R-trees can be applied to spatial [9], temporal [12] and ST data [29]. R-trees have also been shown to perform well in both spatial [1] and temporal [32] join algorithms, and are widely available (e.g. in the PostgreSQL DBMS (*www.postgresql.org*)). This paper can be seen as providing insights into the use of widely available indexes

```
$SpatialINLJoin(R,\ root_{T_S},\ O)$
input:
        $R$, a collection of spatial objects;
        $root_{T_S}$, the root node of $T_S$, the R-tree indexing the join attribute of $S$, the other input collection;
output:
        The result $O$ is a collection of pairs of objects from $R$ and object identifiers from $S$;

/* $e_s$ is an entry in leaf nodes of $T_S$ which is a pair of MBR, $mbr$, and an object identifier, $oid$;
$r$ is an object of $R$, having $val$ as its spatial attribute;
$Mbr()$ is a function that computes an MBR enclosing its argument;
$S'$ is a lookup result which is a collection of $e_s$. */

begin
1: $O := \emptyset$;
2: for each object $r \in R$ do
3:   $S' := \emptyset$;
4:   $SpatialLookup(root_{T_S},\ Mbr(r.val),\ S')$;
5:   for each $e_s \in S'$ do
6:         $O := O \cup \{(r, e_{s.oid})\}$;
end.


$SpatialLookup(N_T,\ window,\ O)$
input:
        $N_T$, which is a node of an R-tree; $window$ is a searching area;
output:
        The result $O$ is a collection of entries in leaf nodes;

/* $readNode()$ returns the R-tree node referred to by its argument. */

begin
1: for each $e \in N_T$ do
2:   if $N_T$ is a leaf node where $(e.mbr$ intersects $window) \neq \emptyset$
3:   then $O := O \cup \{(e.oid)\}$;
4:   else if $N_T$ is an internal node where $(e.mbr$ intersects $window) \neq \emptyset$
5:         then $SpatialLookup(readNode(e),\ window,\ O)$;
end.
```

Figure 1: An INL Algorithm

such as R-trees, including their 3D extensions [29], for supporting spatio-temporal requests.

Note that the algorithms in Figures 1 and 2 correspond to the *filtering* step [2], in which approximations (such as Minimum Bounding Rectangles) of the join attribute values are used to reduce the cost of pairwise comparisons of spatial values; the subsequent *refinement* step [2] compares the actual values. In what follows, all spatial and ST joins require a refinement step, whereas purely temporal joins do not – intervals can be represented precisely within R-trees.

3 Approaches to Spatio-Temporal Join Processing

In a query language, there is often no specialised syntax for an ST join – it is simply a join with spatial and temporal conditions, as in:

```
select *
from   $R$, $S$
where  $C_s$ and $C_t$
```

In the above query, R and S are input data collections for a join, and C_s and C_t are spatial and temporal conditions respectively.

```
$SpatialSTTJoin(root_{T_R}, root_{T_S}, O)$
input:
        $root_{T_R}$ and $root_{T_S}$, each of which is the root node of R-trees, $T_R$ and $T_S$,
        over input collection $R$ and $S$, respectively;
output:
        $O$, a set of pairs of object identifiers;

begin
1: $O := \emptyset$;
2: $SpatialSTT(root_{T_R}, root_{T_S}, O)$;
end.


$SpatialSTT(N_r, N_s, O)$
input:
        $N_r$ and $N_s$, each of which is a node of an R-tree;
output:
        $O$, a set of pairs of object identifiers;

/* an entry of an R-tree node, $e_r$ and $e_s$, is a pair of MBR, $mbr$, and an object identifier, $oid$;
$readNode()$ returns the R-tree node referred to by its argument. */

begin
1: for each $e_r \in N_r$ do
2:   for each $e_s \in N_s$ do
3:       if $N_r$ and $N_s$ are leaf nodes where ($e_r.mbr$ intersects $e_s.mbr$) $\neq \emptyset$
4:       then $O := O \cup \{(e_r.oid, e_s.oid)\}$;
5:       else if $N_r$ and $N_s$ are internal nodes where ($e_r.mbr$ intersects $e_s.mbr$) $\neq \emptyset$
6:            then $SpatialSTT(readNode(e_r), readNode(e_s), O)$;
end.
```

Figure 2: A STT Algorithm

This paper compares three alternative ST join strategies depicted in Figure 3 in relational algebra: (a) performs a spatial join followed by a temporal selection; (b) performs a temporal join followed by a spatial selection; and (c) performs a specialised ST join. In all cases a spatial selection is required to support the refinement step. The conditions representing the *filtering* and *refinement* steps are denoted by ${C_s}'$ and C_s respectively. The prime is also used in Figures 4 and 5 to label the *filtering* conditions.

These alternatives are explored so that experiments can compare specialised ST algorithms with combinations of spatial and temporal algorithms for answering ST queries. These comparisons not only assess the efficiency of the different approaches, but can also inform the decision as to how many indexes are required. For example, an ST index enables indexed access to both spatial and temporal data, but may be expected to be less effective in these cases than the more specialised spatial-only or temporal-only indexes.

The experiments reported in Sections 4 and 5 compare each of the strategies depicted in Figure 3, using both INL and STT algorithms for spatial, temporal and spatio-temporal joins.

To implement the strategies, R-trees are used for spatial, temporal and ST joins. The ST joins use 3D R-trees [29], a 3D variant of R-trees in which two dimensions are occupied by space, and time is treated as a third dimension.

In spatial, temporal and ST joins, the approximate values used in the 2D and 3D R-trees are: normal MBRs (*Minimum Bounding Rectangles*) represented by two pairs of the x- and y-coordinates of the lower-left and the upper-right corner points of a rectangle (i.e. (x_{ll}, y_{ll}), (x_{ur}, y_{ur})) for spatial indexes; MBRs with upper bound x-coordinates set to the maximum value for the *now* time point and y-coordinates set to zero for temporal indexes; and MBCs (*Minimum Bounding Cuboids*) with x- and y-coordinates representing the spatial dimension and z-coordinates representing the temporal dimension for spatio-temporal indexes.

The STT algorithms used in the implementation adopt the *local optimization* and *search space restriction* techniques of Brinkhoff *et al.*'s algorithm [2]. By way of local optimization, the *plane sweep* [20] technique is used to reduce the CPU time required for comparison of indexed entries from nodes of two

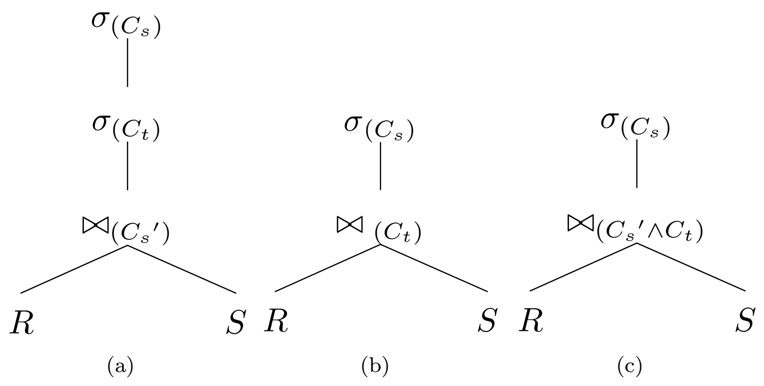


Figure 3: Logical Expressions for ST Joins

distinct R-trees. For the spatio-temporal case, the notion of *space sweep* [11] is used, which is analogous to the plane sweep, but for 3D. The first attempt to employ the space sweep for sort-merge-based spatial joins over 3-dimensional spatial data is found in [21]. The search space restriction technique reduces the number of indexed entries to be compared at each tree traversal stage by restricting the space of comparison using the intersecting area of the two upper level node entries from distinct trees: only the entries overlapping the intersecting area can be candidates of comparison at the tree traversal stage since only that area is of interest for comparison.

4 Experiments

This paper tests the following hypotheses:

(i) Specialized ST joins perform better than spatial joins followed by a temporal selection and temporal joins followed by a spatial selection.

(ii) Joins using STT algorithms perform better than INL algorithms in ST databases.

(iii) Spatial and temporal joins perform better using spatial and temporal indexes respectively than with ST indexes.

These hypotheses would normally be expected to be true, so the interesting cases are those in which the hypotheses do not hold. However, as well as testing these hypotheses, the experiments provide many details from a range of scenarios that should be useful to the developers of spatio-temporal systems.

4.1 Queries and Query Execution Plans (QEPs)

The tables *Cities*, *BusRoutes* and *Shops* are used as the context for the queries used in the experiments:

```
Cities(id: integer, description: string, boundary: Polygon, vt: Interval)
BusRoutes(id: integer, description: string, centreline: Polyline, vt: Interval)
Shops(id:integer, description: string, location: Point, vt: Interval)
```

A city can change in boundaries by growth or shrinkage and keep a boundary during a time interval; a bus route can change to reflect requirements of local areas, for financial reasons, etc; and a shop can often change location for expansion or contraction.

The *vt* attribute represents the valid time of a tuple in the table. The precise spatial model used is not crucial to the experiments, as these focus on the *filtering* step of join processing involving spatial values, using MBRs or MBCs as abstract representations of the *centreline*, *boundary* and *location* attributes. Thus the results presented in this paper are of relevance in the presence of different spatial models, as long as these can be abstracted for representation in indexes using MBRs or MBCs.

The following queries are used in the experiments:

Q1.1 Which bus routes intersected a city at some time?

```
select *
from   Cities c, BusRoutes b
where  c.boundary intersects b.centreline
and    c.vt intersects b.vt;
```

Q1.2 Which shops were contained by a city at some time?

```
select *
from   Cities c, Shops s
where  c.boundary contains s.location
and    c.vt intersects s.vt;
```

Q2 Which bus routes intersected a city within a given area (represented by a *spatial-literal*) during a given period (represented by *valid-time-literal*).

```
select *
from   Cities c, BusRoutes b
where  c.boundary intersects b.centreline
and    c.boundary intersects <spatial-literal>
and    b.centreline intersects <spatial-literal>
and    c.vt intersects b.vt
and    c.vt intersects <valid-time-literal>
and    b.vt intersects <valid-time-literal>;
```

Q3.1 Which bus routes have at some time intersected an area that at some time was a city?

```
select *
from   Cities c, BusRoutes b
where  c.boundary intersects b.centreline
```

Q3.2 Which shops have at some time been contained by an area that at some time was a city?

```
select *
from   Cities c, Shops s
where  c.boundary contains s.location
```

Q4 Which bus routes and cities co-existed at some time?

```
select *
from   Cities c, BusRoutes b
where  c.vt intersects b.vt
```

Queries *Q1.1*, *Q1.2* and *Q2* are all spatio-temporal joins, in that their join conditions have both spatial and temporal aspects; the principal distinguishing feature of *Q2* from *Q1.1* and *Q1.2* is that it allows changes to the selectivity of the inputs to the join operations using spatial and temporal selection conditions. These conditions are used to vary the selectivity of the inputs to the join, as described in Section 5.4. Queries *Q3.1* and *Q3.2* are purely spatial queries run over spatio-temporal data collections – there is no temporal condition in the join. Similarly, *Q4* is a purely temporal query – there is no spatial condition in the join.

Queries *Q1.1* and *Q3.1* use intersection relationships between spatial values with extent in their approximate representation (viz. polygon and polyline values) for join predicates, but queries *Q1.2* and *Q3.2* involve containment relationships between spatial values with extent and without extent (viz. point values).

To evaluate such queries, a query processor translates a query written in a declarative surface language, e.g. SQL, into a logical query expression as shown in Figure 3 comprising logical operators at internal nodes and input collections at leaf nodes; and that expression is in turn translated into a physical query execution plan (QEP) comprising physical operators in a similar structure to the logical expression. The QEP is then evaluated and the corresponding algorithms and access methods are invoked.

QEPs used to evaluate *Q1.1* are illustrated in Figure 4. The strategies illustrated are spatial joins followed by a temporal selection and spatio-temporal joins; temporal joins followed by a spatial selection involve QEPs that are analogous to those for spatial joins followed by temporal selections. QEPs for *Q1.2* are analogous to counterparts for *Q1.1* except for changes to the input collections.

In the QEPs, C and B represent the input collections $Cities$ and $BusRoutes$ respectively; P is a pointer to a node of 2D spatial or 3D ST R-trees, *sT* and *stT*, respectively; and *oid* is for an object identifier. The operators of the physical algebra used are: *SeqScan* – sequential scan of a collection; *IdxScan* – indexed scan of a collection; *INL* – indexed nested loop join; STT – synchronised tree traversal join; *ObjFetch* – read an object given its identifier; *Select* – keep the objects in a collection only if they satisfy a predicate; and *NodeFetch* – retrieve a page from an index given its address.

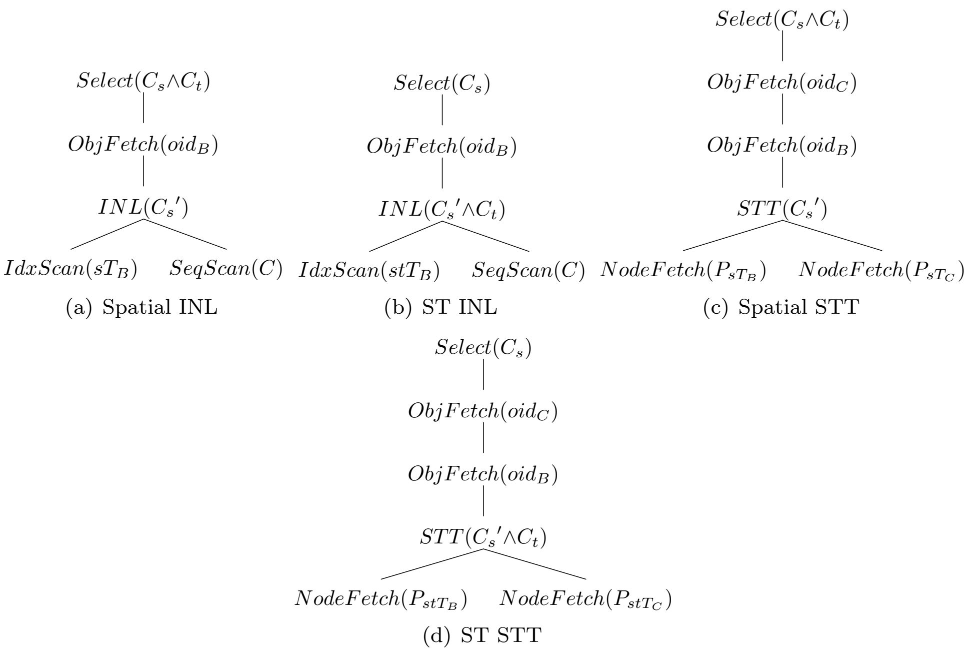


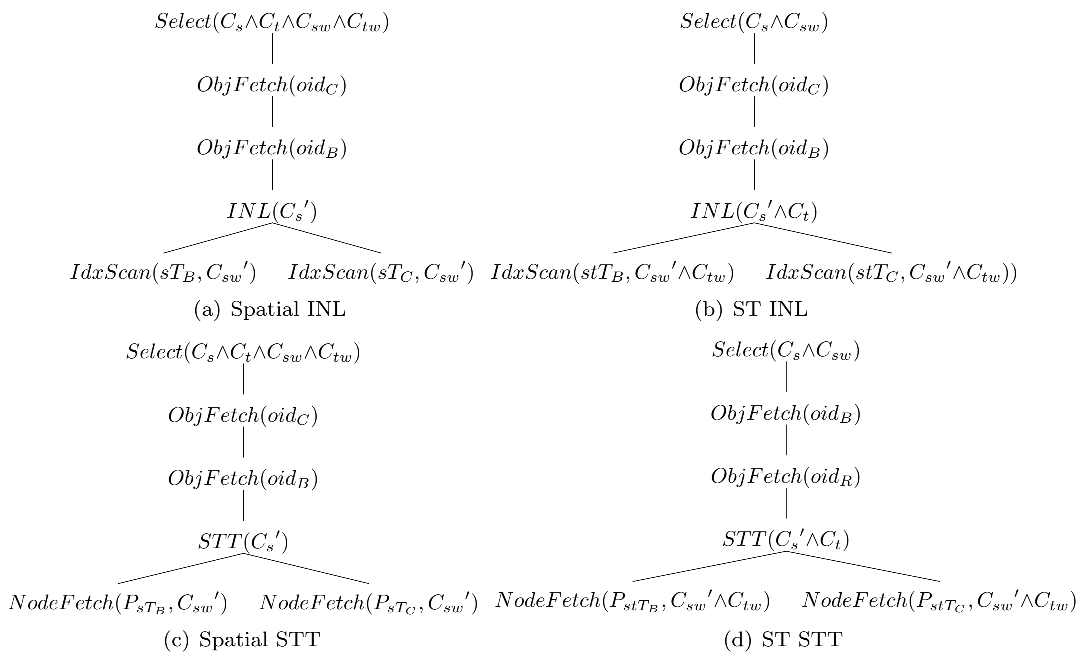
Figure 4: QEPs for Query *Q1.1*

In Figure 4 (a) and (b) *INL*() reads objects by calling *SeqScan*(), and uses the join attributes of the objects as the search keys for *IdxScan*(). *IdxScan*() therefore retrieves object identifiers satisfying the given join condition with the given search keys. *INL*() produces pairs consisting of objects scanned by *SeqScan*() and object identifiers retrieved by *IdxScan*(). *ObjFetch*() then reads objects from store using their identifiers. The objects that satisfy the refinement conditions are then selected by *Select*().

In Figure 4 (c) and (d), $STT()$ reads nodes of two R-trees referenced by the given pointers by calling *NodeFetch*(), and matches index entries across the two R-trees. The operator produces pairs of object identifiers satisfying the given join condition for the filtering step. The identifiers in a pair are subsequently consumed by *ObjFetch*(), which reads the corresponding objects from store. The objects read are eventually fed into *Select*() to check whether they indeed satisfy the join conditions in the refinement step.

The QEPs used to evaluate *Q2* are given in Figure 5. The notation is as in Figure 4, except for the addition of C_{sw} and C_{tw} that represent the spatial and temporal windows of the selection conditions from *Q2*. In Figure 5, *NodeFetch*() and *IdxScan*() are overloaded to have additional arguments for the given spatial and temporal selection conditions. In addition, *IdxScan*() is used where *SeqScan*() was used in Figure 4, so that the extra condition from *Q2* is applied at the earliest possible moment.

The implementations of the join algorithms used are all single-pass (i.e., they were designed for use with intermediate data sets that are smaller than the available memory, as is the case in all the experiments reported).



$Select(C_s \wedge C_t \wedge C_{sw} \wedge C_{tw})$
$ObjFetch(oid_C)$
$ObjFetch(oid_B)$
$INL(C_s')$
$IdxScan(sT_B, C_{sw}')$
$IdxScan(sT_C, C_{sw}')$
(a) Spatial INL
$Select(C_s \wedge C_{sw})$
$ObjFetch(oid_C)$
$ObjFetch(oid_B)$
$INL(C_s' \wedge C_t)$
$IdxScan(stT_B, C_{sw}' \wedge C_{tw})$
$IdxScan(stT_C, C_{sw}' \wedge C_{tw}))$
(b) ST INL
$Select(C_s \wedge C_t \wedge C_{sw} \wedge C_{tw})$
$ObjFetch(oid_C)$
$ObjFetch(oid_B)$
$STT(C_s')$
$NodeFetch(P_{sT_B}, C_{sw}')$
$NodeFetch(P_{sT_C}, C_{sw}')$
(c) Spatial STT
$Select(C_s \wedge C_{sw})$
$ObjFetch(oid_B)$
$ObjFetch(oid_R)$
$STT(C_s' \wedge C_t)$
$NodeFetch(P_{stT_B}, C_{sw}' \wedge C_{tw})$
$NodeFetch(P_{stT_C}, C_{sw}' \wedge C_{tw})$
(d) ST STT

Figure 5: QEPs for Query *Q2*

4.2 System Environment

The experiments have been run using the PostgreSQL 7.1.3 object-relational DBMS. PostgresSQL allows users to extend its functionality through *user-defined functions* and *user-defined types.* In addition, R-trees are implemented using the GiST (Generalized Search Tree) [10] that is part of PostgreSQL. The GiST is an Abstract Data Type for search trees, which enables users to implement additional index structures. All the implementations of physical operators and R-trees rely on the buffer management of PostgreSQL, in which 64 8K buffers are used.

The underlying experimental environment consists of a 700MHz Pentium III PC with 256Mb main memory running RedHat Linux version 7.2. In the experiments each QEP was run three times, and the average used. Each execution is performed after the PostgreSQL server has been shut down and restarted, and the operating system buffer cache has been flushed (i.e., queries are run "cold").

4.3 Data Generation

For generating data collections of *Cities*, *BusRoutes* and *Shops* a slightly modified version of the data generator GSTD [28] has been used. Data collections generated with GSTD can be tuned by various parameters:

1. The initial cardinality of a collection – this is a straightforward way of varying the sizes of the data collections being studied.
2. The initial density of a collection (i.e., the percentage $\frac{100 \times \sum area\ of\ a\ spatial\ object}{workspace\ area}$) – this can be used to reflect the fact that different geographical environments may have different numbers of features in a given area.
3. The number of snapshots – this can be used to capture the fact that applications differ in the period for which historical data is available and that data capture rates vary.
4. The duration between consecutive snapshots – this can be used to reflect the fact that different data sets may be updated at different rates;
5. The change in location of time-evolving spatial objects – this can be used to capture the nature of the changes taking place.

Parameters	**Data Collections**	
	DC1	DC2
number of initial spatial objects	500 to 1500 in steps of 250	200
number of snapshots	20	50 to 150 in steps of 25
initial spatial density(%)	0 for point collections; 30 for rectangle collections	
change in location(%)	[-10%, 10%] of the entire spatial data space at random for point collections; [-10%, 10%] of extents' size at random for rectangle collections	
duration between changes(%)	[50%, 100%] of time resolution at random	
size of objects(bytes)	460	
distribution	uniform	

Figure 6: Data Collections

The following parameters are selectively varied to yield databases of different sizes in the performance evaluation of queries *Q1.1*, *Q1.2*, *Q3.1*, *Q3.2* and *Q4*:

- the initial cardinality of spatial objects; and
- the number of snapshots.

In evaluations involving query *Q2*, the spatial and temporal windows are varied in size.

The data collections used in the experiments simulate only discrete change in the locations of spatial objects. The change in location of the object with extent in each direction is a random value up to 10% of the size of the object in that direction; and when the object does not have extent (i.e. a point) the change is a random value up to 10% of the entire spatial data space size. Changes happen consecutively after a period that is a random value from 50% to 100% of the temporal resolution. The temporal resolution is determined by the number of snapshots. The distribution of initial spatial objects, the changes in their location, and the duration between consecutive snapshots follow a uniform distribution.

The details of the data collections used are presented in Figure 6. Thus data collections $DC1$ and $DC2$ explore changes in database size by varying the number of initial spatial objects and snapshots respectively. Polygon, polyline and point values of $Cities$, $BusRoutes$ and $Shops$ are approximated by rectangles in the collections. The data collections are inserted into PostgreSQL tables in chronological order and R-trees are built on the tables. To give an indication of the database sizes used in the experiments, the largest databases in both *DC1* and *DC2* are around 55Mb. We note that the absolute size of the databases is not the principal issue – the principal issue is the relative performance of the approaches and the growth curves for single-pass algorithms; we do not explore cases in which disk space is required to store intermediate results.

Figure 7 visualises some snapshots of an example data collection, and Figure 8 shows the simplified distribution of time intervals of input data collections by transforming 1-dimensional intervals into 2-dimensional points. Intervals of the last snapshots of each spatial object, whose ending instant is the maximum value, *now*, in the given temporal dimension, constitute the upper distribution.

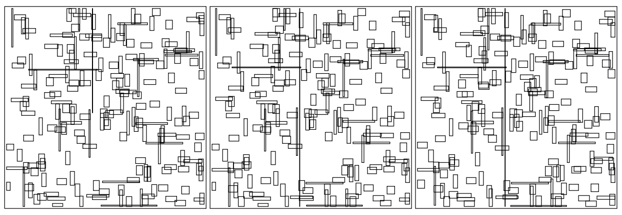
There is clearly a limit to the number of aspects of a data set that can be changed in an experimental study. This paper has not considered aspects such as data skew, or variable complexities of individual spatial objects. These factors could influence experimental results, and thus could form the basis for continuing studies on spatio-temporal join performance.

5 Results

This section presents the results of the experiments, for which the data collections, queries and query execution plans were presented in Section 4.

5.1 Varying Spatial Cardinality

This subsection describes how the different join strategies perform using spatio-temporal queries *Q1.1* and *Q1.2* over data collections in *DC1*, in which the size of the database is varied as a result of changing the number of initial spatial objects. So that the experiment is focused on the number of data items, and not influenced by changing density, the initial spatial density of rectangle collections is kept at 30% throughout


(a) 200 Rectangles (1st, 75th, 150th snapshots from left)

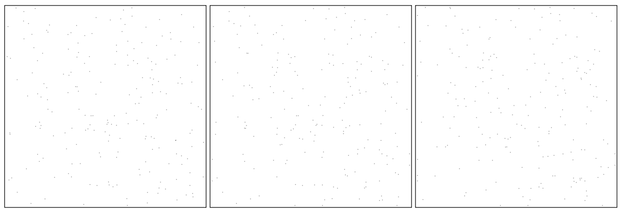

(b) 200 Points (1st, 75th, 150th snapshots from left)

Figure 7: Examples of Data Collections (with 200 spatial objects)

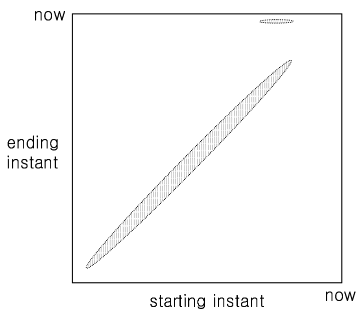


Figure 8: Distribution of Time Intervals

for *DC1*, which means that 30% of the area is occupied at each point in time. This is done by decreasing the average area occupied by each spatial object as the cardinality increases.

Figure 9 shows elapsed times for four of the join strategies for *Q1.1*. To clarify the graphs, the performance of QEPs using temporal INL and temporal STT joins are excluded from the graphs since they show significantly poorer performance than the others. The poor performance of the joins based on temporal indexes arises from the high temporal density of the data collections used. This high density gives rise to significant overlaps in interior MBRs of temporal R-trees (and thus inefficient lookup) and to large result sizes from purely temporal filtering for spatio-temporal joins, which in turn gives rise to high refinement costs. We note that high temporal densities are realistic overall, as it is to be expected that many data items will exist at the same time.

In Figure 9 and hereafter: the spatial, temporal and spatio-temporal INL strategies are referred to as *s_inl*, *t_inl* and *st_inl*, respectively; and the spatial, temporal and spatio-temporal STT strategies are referred to as *s_stt*, *t_stt* and *st_stt*, respectively.

The results in Figure 9 exhibit several features that are worthy of comment:

(i) The STT algorithms perform better than their INL counterparts. This result can be explained with reference to Figure 10 (a) and (b), where it can be seen that the STT algorithms perform many fewer visits to index nodes than the INL algorithms. This confirms earlier results [2] on the effectiveness of the STT approach over spatial data, and shows that the results, as expected, carry over to joins over 3D R-trees.

(ii) The *st_stt* (or the *st_inl*) performs better than the *s_stt* (or the *s_inl*). This is principally because the number of results returned by the ST join is considerably less than the number of results from the

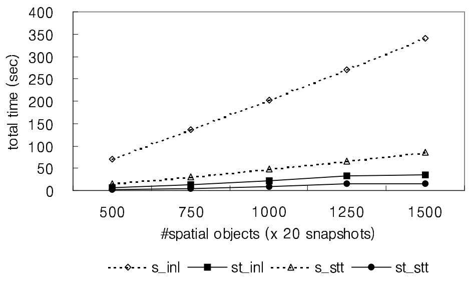
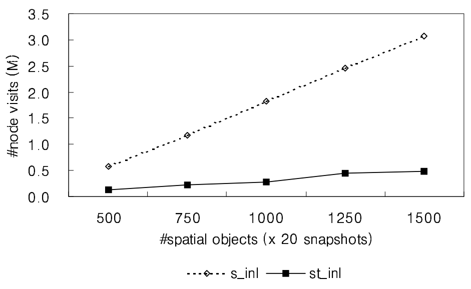
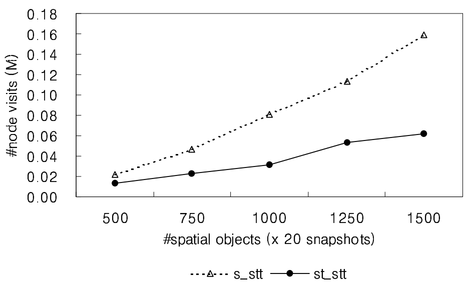


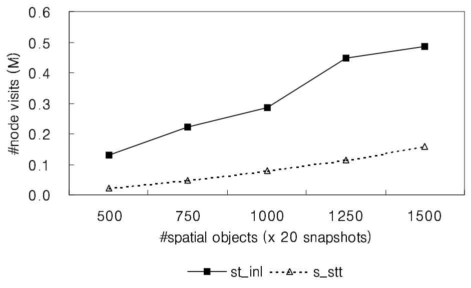
Figure 9: Elapsed time for query *Q1.1* with data collections *DC1*



(a)

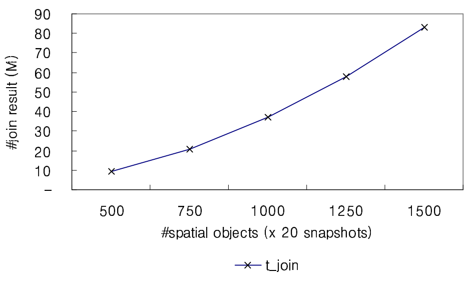


(b)

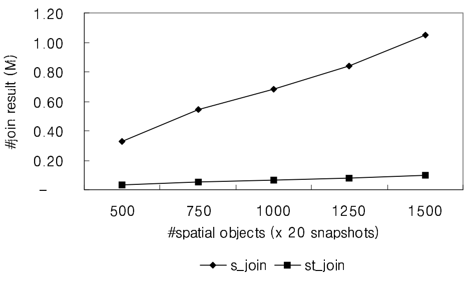


(c)

Figure 10: Number of node visits for the query *Q1.1* with data collections *DC1*



(a)



(b)

Figure 11: Number of join result for the query *Q1.1* with data collections *DC1*

spatial join, as shown in Figure 11. The join selectivity is lower in the ST case simply because the join condition is more precise, including both spatial and temporal aspects. The larger join result from the spatial query means that many more objects must be fetched from the store after the join,

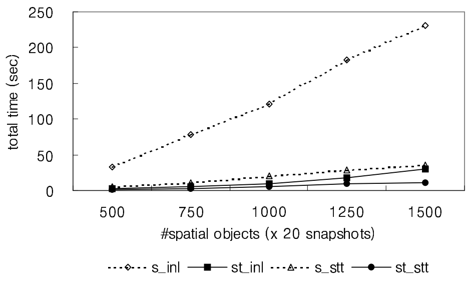


Figure 12: Elapsed time for query *Q1.2* with data collections *DC1*

and that many more objects are processed in the *refinement* step of the join[1]. We note also that the figures given here will generally underestimate the cost of the refinement step for real examples, as only approximations of actual spatial values such as MBRs and MBCs are stored and analysed.

(iii) The *st_inl* performs better than the *s_stt*. In the filtering step of spatio-temporal join processing, the *s_stt* visits many fewer nodes than the *st_inl* as shown in Figure 10 (c). This confirms again the effectiveness of the coordinated tree traversal technique over the uncoordinated approach as indicated in (i) above. However, the overall performance does not correspond to the performance of the filtering step. This can be explained by the fact that there is a significant difference in the number of the join results between the *st_inl* and the *s_stt*, which affects the refinement step as indicated in (ii) above, and the cost of the refinement step becomes dominant in the overall join processing.

Figure 12 shows elapsed times for four of the join strategies for *Q1.2*. The pattern of performance curves is similar to Figure 9 for *Q1.1*. It stems from the fact that INL and STT algorithms with containment predicates are only different in processing at leaf nodes from those with intersection predicates: viz. in line 2 of *SpatialLookup*() in Figure 1 and in line 3 of *SpatialSTT*() in Figure 2.

What lessons are learned from this for the developers of ST applications and systems? Perhaps the clearest lesson is that specialised ST indexes were crucial for obtaining scalable performance with ST joins – in the experiment, evaluation strategies that used only spatial or temporal indexes were not competitive. This benefit derived principally from the reduced cost of the refinement step in this case. The performance difference associated with index selection was more significant than that associated with algorithm selection, although the STT algorithms significantly out-performed INL, as anticipated.

5.2 Varying Number of Snapshots

Figure 13 displays the experimental result of processing the query *Q1.1* over data collections *DC2*. These experiments assess the effect of different numbers of data values from the same geographical area, reflecting different frequencies or periods of data capture. The results in Figure 13 exhibit the following feature that is worthy of comment:

- As the number of snapshots increases, the benefits from using the ST indexes increase. This is reflected in the fact that *st_inl* and *st_stt* significantly out-perform their spatial counterparts where there are large numbers of snapshots. This is to be expected, as the purely-spatial condition testing provided by the indexes in the *s_inl* and *s_stt* approaches becomes less discriminating as the number of snapshots increases. This is reflected in Figure 15, where the size of the spatial join result is seen to grow more rapidly than that of the ST join.

Figure 16 shows elapsed times for four of the join strategies for *Q1.2* over data collections *DC2*. The pattern of performance curves is also similar to Figure 13 for *Q1.1* as observed between Figures 9 and 12.

What lessons are learned from this for the developers of ST applications and systems? Perhaps the clearest lesson is that specialised ST joins become important as the number of snapshots increases, as

[1]The large join result from the temporal join illustrated in Figure 11 (a) helps to explain the poor overall performance (not shown but an indication of the poor performance can be derived from Figure 21 for pure temporal joins) of the temporal algorithms for Q1.1 over *DC1*.

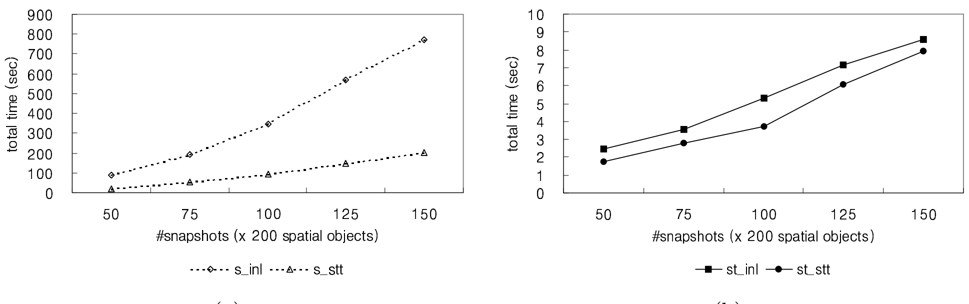


Figure 13: Elapsed time for query *Q1.1* with data collections *DC2*

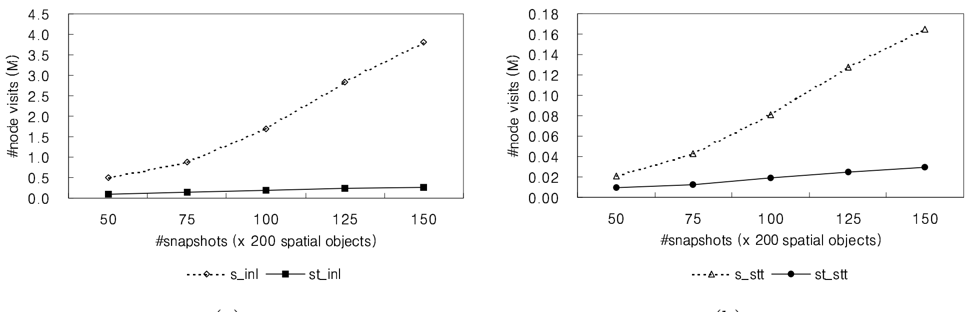


Figure 14: Number of node visits for query *Q1.1* with data collections *DC2*

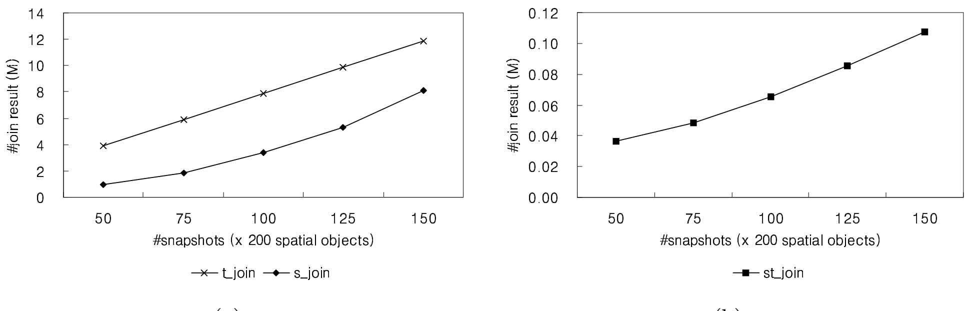
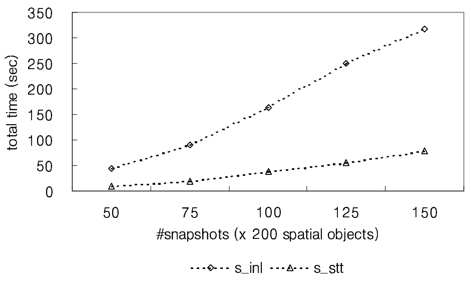


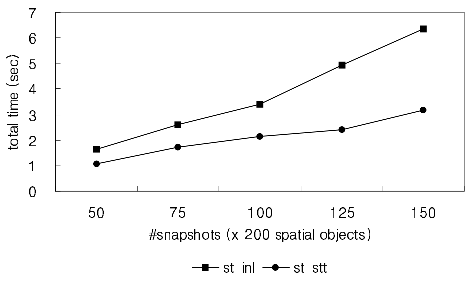
Figure 15: Number of join result for query *Q1.1* with data collections *DC2*

would be expected. This means that developers with historical data collections with shallow histories may well be able to get by without specialist ST indexing and joins, but that this will become increasingly less viable as more historical information is accumulated.

Comparing the results in this section and those in Section 5.1, there are differences in the behaviours of the algorithms for the two data collections, with *DC1* varying only cardinality (and keeping density constant) and *DC2* allowing the spatial density to increase as the number of snapshots grows. This gives rise to differences in the numbers of nodes visited for *Q1.1* (and *Q1.2*) with data collections *DC1* and *DC2*, as illustrated in Figures 10 and 14. This can be understood with reference to the spatial, temporal and ST densities in Figure 17. In particular, the spatial density becomes much higher in *DC2* than in *DC1*, which in turn gives rise to the much larger results for the spatial joins in *DC2*.

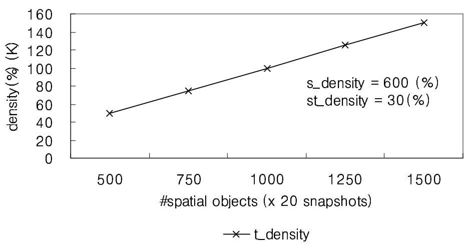


(a)

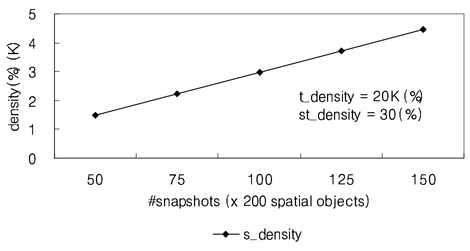


(b)

Figure 16: Elapsed time for query *Q1.2* with data collections *DC2*



(a) data collections DC1



(b) data collections DC2

Figure 17: Density for *DC1* and *DC2*

5.3 Pure Spatial and Pure Temporal Joins

In the above experiments, ST queries are evaluated using spatial, temporal and ST join strategies. In this section, joins with purely spatial and purely temporal conditions are run over the same range of strategies. In these examples the refinement step is identical in each case (i.e., the filtering steps all return the same number of results for the purely spatial or purely temporal joins, respectively), so all time differences must be attributed to the number of node visits in the filtering step and the quality of R-trees in preserving clusters of physical addresses of spatio-temporal data stored, which can affect the costs of object fetch operations in the refinement step. In the *st_inl* and *st_stt* algorithms only spatial (or temporal) information is used to to search and traverse the 3D R-trees.

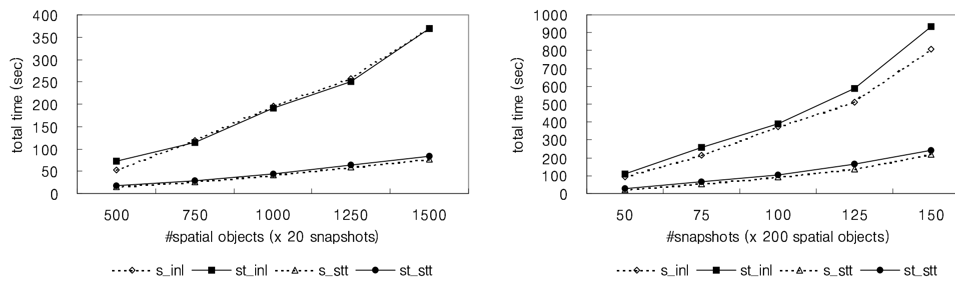
Firstly, the purely spatial join:

Q3.1 over DC1 and DC2 The results in Figure 18 show several features: (i) The STT algorithms outperform their INL counterparts (i.e., *st_stt* is faster than *st_inl*, and *s_stt* is faster than *s_inl*); and (ii) The specialist spatial joins perform somewhat better than their ST counterparts (i.e. *s_stt* is faster than *st_stt*, and *s_inl* is faster than *st_inl*). This latter result is principally explained by the fact that the number of node visits of the spatial specialists is usually less than that of STT counterparts as shown in Figure 19.

Q3.2 over DC1 and DC2 The results in Figure 20 show very similar performance for the different algorithms to that in Figure 18.

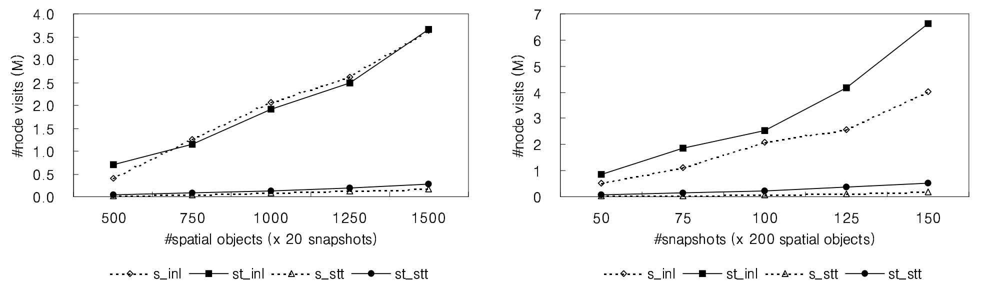
Secondly, the purely temporal join:

Q4 over DC1 As to the pure temporal join algorithms over *DC1* the results in Figures 21 (a) confirms the efficiency (i) of STT algorithms over their INL counterparts (i.e., *st_stt* is faster than *st_inl*, and *t_stt* is faster than *t_inl*); and (ii) of the specialist temporal joins over their ST counterparts (i.e. *t_stt* is faster than *st_stt*, and *t_inl* is faster than *st_inl*). In pure temporal joins, the benefit using the temporal indexes is increasing as the density is much higher in the temporal dimension than in



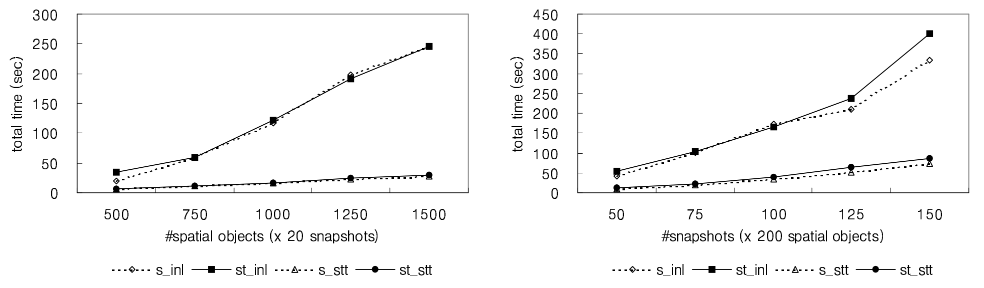
(a) data collections DC1 (b) data collections DC2

Figure 18: Elapsed time for the pure spatial query *Q3.1*



(a) data collections DC1 (b) data collections DC2

Figure 19: Number of node visits for pure spatial query *Q3.1*

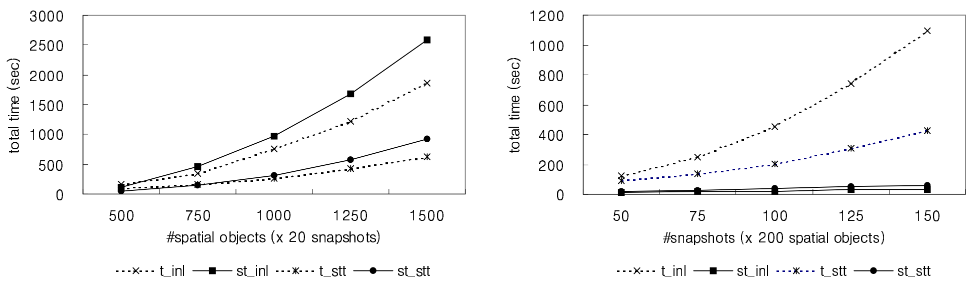


(a) data collections DC1 (b) data collections DC2

Figure 20: Elapsed time for the pure spatial query *Q3.2*

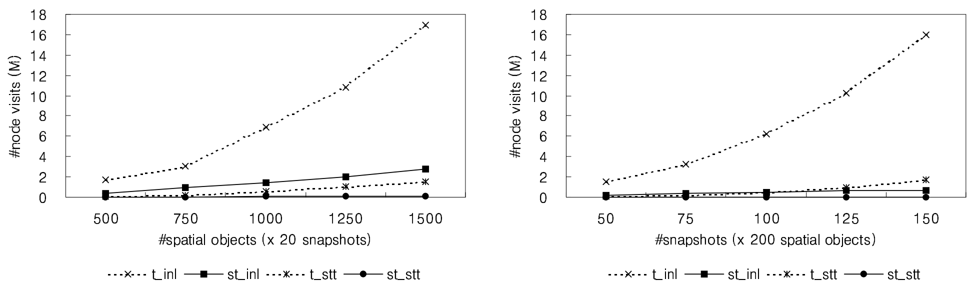
the spatial dimension of *DC1* as shown in Figure 17. The number of node visits shown in Figure 22 (a), however, is not a good guide as to the behaviours of the algorithms. Since the number of temporal join results is large, as indicated in Figure 11, the cost of object fetch operations becomes the dominant factor of the overall join processing in this case. Figures 21 (a) and 22 (a) imply that ST indexes can provide better temporal clustering over spatio-temporal data than temporal indexes.

Q4 over DC2 Figures 21 (b) over *DC2* shows interesting behaviours of algorithms for pure temporal joins compared with those over *DC1*: (i) the efficiency of STT algorithms over their INL counterparts is still sustained; and (ii) all the specialist temporal joins, however, perform poorer than any ST algorithm. In essence, the principal difference from the same query and algorithms for *DC1* is that the spatial density of *DC2* is considerably higher than that of *DC1*, but the temporal density is still considerably high compared with the spatial density, as indicated in Figure 17. Figures 21 (b) and 22 (b) imply that ST indexes provide not only better temporal clustering than temporal indexes,



(a) data collections DC1 (b) data collections DC2

Figure 21: Elapsed time for the pure temporal query *Q4*



(a) data collections DS1 (b) data collections DS2

Figure 22: Number of node visits for query *Q4*

but also good physical clustering over the given spatio-temporal data collections. Variations of R-trees used as temporal indexes are found in [12, 13, 14]. [22] provides analysis of such R-tree based temporal indexes. Although they show good performance characteristics on average for querying, they sometimes behave pathologically and performance becomes quite poor.

What lessons are learned from this for the developers of ST applications and systems? The experiments show that purely spatial joins perform quite well over ST indexes, which suggests that it will not always be necessary to maintain both spatial and ST indexes in ST applications with both spatial and ST queries. This is an encouraging result, in that reduced storage overheads and update times result from fewer indexes.

We observe, however, that both the temporal and the ST indexes seem to be necessary to obtain adequate temporal join performance depending on data collections.

5.4 Varying Selectivity

Some join strategies are sensitive to the selectivity of their inputs (e.g., this is important when comparing navigational and value-based joins in object databases [23]). Figure 23 shows the elapsed times for *Q2* over one of the databases from *DC1* (i.e. 1000 spatial objects $\times$ 20 snapshots), for varying input selectivities. We note that few radical changes can be seen from the results reported, but that from those shown and others obtained for lower selectivities (not shown), it seems that the differentials observed for large selectivities are somewhat diluted for lower selectivities. This topic could benefit from a more thorough investigation.

6 Conclusion

This paper has described several approaches to the evaluation of spatio-temporal joins and described a comparative evaluation of the approaches. We know of no other similar study.

In terms of the hypotheses described in Section 4, the following can be observed:

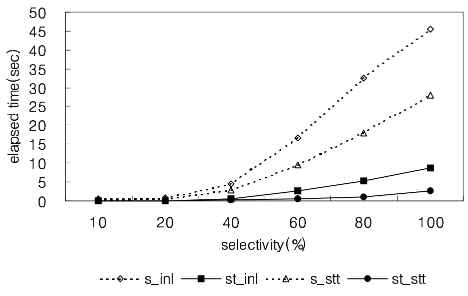


Figure 23: Elapsed time for *Q2* for varying selectivities

(i) *Specialised ST joins perform better than spatial joins followed by a temporal selection and temporal joins followed by a spatial selection.* This hypothesis is supported by the results in Figures 9, 12, 13 and 16. It is clear that in certain cases specialist ST joins provide significantly better performance than joins using purely spatial or temporal filtering steps, and thus that the extra overhead associated with maintaining ST indexes can be beneficial. The benefits from use of specialised ST joins are particularly great for data sets with large numbers of snapshots, where ST joins substantially reduce the number of candidates forwarded to the refinement step (see, for example, Figure 15). Conversely, where data sets are in use with modest numbers of snapshots, the benefits from using specialised ST joins are much reduced.

(ii) *Joins using STT algorithms perform better than INL algorithms in ST databases.* The good behaviour of STT joins that is familiar from spatial databases essentially carries over to ST databases. STT performs much better than INL where there is high density, as shown in figures for elapsed time, and in particular, this is likely to be common in the temporal dimension of ST databases. Thus there is some evidence that STT approaches will be even more effective in ST databases than they have been shown to be in earlier work on spatial databases. However, the experiments show that the difference in performance for ST joins is influenced more by the use of specialised ST indexes than by the specific join algorithm used (see, for example, Figures 12 and 13). For data sets with large numbers of snapshots, the use of ST indexes can provide orders of magnitude improvements in performance, as illustrated in Figure 16.

As STT requires indexes on both operands, and thus is not directly usable over intermediate join results, more work is required to understand the trade-offs involved in building indexes for intermediate collections.

(iii) *Spatial and temporal joins perform better using spatial and temporal indexes respectively than with ST indexes.* This hypothesis was broadly true in the experiments for spatial joins, but not to the extent that we anticipated. In the spatial experiments (Figures 18 and 20) the performance of both the STT and the INL approachs over the ST indexes were generally close to the times recorded for the spatial joins using spatial indexes. This suggests that ST indexes may be able to support queries with purely spatial conditions.

The hypothesis did not hold for all the cases of temporal joins depending on configurations of input data collections: when spatial density is fixed and temporal density increases, there is clear benefit from using specialised joins (Figure 21 (a)), but in the reversed case, specialised joins performed less well than those using ST indexes (Figure 21 (b)). This was something of a surprise, and may suggest either that ST indexes can be quite widely employed, or that R-trees are not particularly effective as temporal indexes. Further study is required to understand behaviours of R-trees in the mixture of spatial and temporal dimensions where the two dimensions show different characteristics, e.g. in density.

In reporting on an experimental study, this paper does not claim to present a definitive position on spatio-temporal join performance. However, it does provide empirical results in an important area, which it is hoped will be of value to developers of spatio-temporal database systems and algorithms, designers of cost models for spatio-temporal systems, and developers of data-intensive spatio-temporal applications.

Acknowledgements: This work is partly supported by a grant from the UK Engineering and Physical Sciences Research Council (EPSRC), whose support we are pleased to acknowledge.

References

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J.S. Vitter. A Unified Approach for Indexed and Non-Indexed Spatial Joins. In *7th International Conference on Extending Database Technology*, pages 413–429, Konstanz, Germany, March 2000.

[2] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-trees. In *ACM SIGMOD International Conference on Management of Data*, pages 237–246, Washington, D.C., May 1993.

[3] C.X. Chen and C. Zaniolo. SQL^{ST}: A Spatio-Temporal Data Model and Query Language. In *Proceedings of ER 2000, 19th International Conference on Conceptual Modeling*, pages 96–111, Salt Lake City, Utah, USA, October 2000.

[4] A. Frihida, D.J. Marceau, and M. Theriault. Spatio-Temporal Object-Oriented Data Model for Disaggregate Travel Behavior. *Transactions in GIS*, 6(3):277–294, 2002.

[5] I.N. Gregory. Time-variant GIS Databases of Changing Historical Administrative Boundaries:A European Comparison. *Transactions in GIS*, 6(2):161–178, 2002.

[6] T. Griffiths, A.A.A. Fernandes, N.W. Paton, K.T. Mason, B. Huang, and M.F. Worboys. Tripod: A Comprehensive Model for Spatial and Aspatial Historical Objects. In *Proceedings of ER 2001, 20th International Conference on Conceptual Modeling*, pages 84–102, Yokohama, Japan, November 2001.

[7] O. Günther. Efficient Computation of Spatial Joins. In *Proceedings of the 9th International Conference on Data Engineering*, pages 50–59, Vienna, Austria, April 1993.

[8] R.H. Güting, M.H. Bühlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representating and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, March 2000.

[9] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, pages 47–57, June 1984.

[10] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[11] S. Hertel, M. Mätylä, K. Mehlhorn, and J. Nievergelt. Space Sweep Solves Intersection of Convex Polyhedra. *Acta Informatica*, 21:501–519, 1984.

[12] C.P. Kolovson and M. Stonebraker. Indexing Techniques for Historical Databases. In *Proceedings of the 5th IEEE International Conference on Data Engineering*, pages 127–137, 1989.

[13] C.P. Kolovson and M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. In *Proceedings of ACM SIGMOD Conf. on the Management of Data*, pages 138–147, 1991.

[14] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. Technical report, AT&T Bell Laboratories, 1995.

[15] J. Antonio Cotelo Lema, L. Forlizzi, R.H. Guting, E. Nardelli, and M. Schneider. Algorithms for Moving Objects Databases. *Computer Journal*, 46(6):680–712, 2003.

[16] M. Lo and C.V. Ravishankar. Spatial Hash-Joins. In *ACM SIGMOD International Conference on Management of Data*, pages 247–258, Montreal, Canada, June 1996.

[17] M.A. Nascimento and J.R.O. Silva. Towards Historical R-trees. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, pages 235–240, February 1998.

[18] A. Papadopoulos, P. Rigaux, and M. Scholl. A Performance Evaluation of Spatial Join Processing Strategies. In *Proceedings of the 6th International Symposium on Large Spatial Databases*, pages 286–307, Hong Kong, China, July 1999.

[19] D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 395–406, Cairo, Egypt, September 2000.

[20] F.P. Preparata and M.I. Shamos. *Computational Geometry: AN INTRODUCTION*. Springer-Verlag, 1985.

[21] S. Ramaswamy and T. Suel. I/O-Efficient Join Algorithms for Temporal, Spatial, and Constraint Databases. Technical report, Bell Labs Technical Report, 1996.

[22] B. Salzberg and V.J. Tsotras. Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2):158–221, June 1999.

[23] S.F.M Sampaio, J. Smith, N.W. Paton, and P. Watson. An experimental performance evaluation of join algorithms for parallel object databases. In *Proceedings of Euro-Par*, pages 280–290, 2001.

[24] Z. Song and N. Roussopoulos. K-Nearest Neighbour Search for Moving Query Point. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 79–96, Redondo Beach, CA, USA, July 2001.

[25] M.D. Soo, R.T. Snodgrass, and C.S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proceedings of the 10th International Conference on Data Engineering*, pages 282–292, Houston, Texas, USA, February 1994.

[26] L. Spery, C. Claramunt, and T. Libourel. A Spatio-Temporal Model for the Manipulation of Lineage Metadata. *Geoinformatica*, 5(1):51–70, 2001.

[27] M. Stonebraker, K. Gardens J. Frew, and J. Merideth. The sequoia 2000 storage benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 2–11, Washington, D.C., USA, May 1993.

[28] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proceedings of the 6th International Symposium on Large Spatial Databases*, pages 147–164, Hong Kong, China, July 1999.

[29] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-Temporal Indexing for Large Multimedia Applications. In *Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems*, pages 441–448, Hiroshima, Japan, June 1996.

[30] T. Tzouramanis, M. Vassilakopoulis, and Y. Manolopoulos. On the Generation of Time-Evolving Regional Data. *Geoinformatica*, 6(3):207–231, 2002.

[31] M. Yuan. Use of a Three-Domain Representation to Enhance GIS Support for Complex Spatiotemporal Queries. *Transaction in GIS*, 3(2):137–159, 1999.

[32] D. Zhang, V. Tsotras, and B. Seeger. A Comparison of Indexed Temporal Joins. Technical report, Time Center, 2000.