

A Generic Algorithmic Framework for Aggregation of Spatio-Temporal Data

Seung-Hyun Jeong, Alvaro A. A. Fernandes, Norman W. Paton, Tony Griffiths
Department of Computer Science, University of Manchester
Manchester M13 9PL, UK

email: {jeongs|alvaro|norm|griffitt}@cs.man.ac.uk

Abstract

Spatio-temporal databases are often associated with analyses that summarize stored data over spatial, temporal or spatio-temporal dimensions. For example, a study of traffic patterns might explore average traffic densities on a road network at different times, over different areas in space, and over different areas in space at different times. The importance of temporal, spatial and spatio-temporal aggregation has been reflected in a significant number of proposals for algorithms for efficient computation of specific kinds of aggregation. However, although such proposals may be effective in particular cases, as yet there is no generic framework that provides efficient support for the wide range of partitioning and aggregation operations that a spatio-temporal database management system might be expected to support over both stored and derived data. This paper proposes an algorithmic framework that can be applied to many different forms of aggregation, and presents the results of performance studies on an implementation of the framework. These show that the framework provides a scalable solution for the many cases in which the aggregations required over stored and derived data may be widely variable and unpredictable.

1. Introduction

In query languages, **aggregation** is the process of computing a single value that summarizes a set of attribute values by means of an **aggregation function** (e.g., summation). For example, we may compute the sum total of employee salaries. The computed value (in the example, the sum total) is called an **aggregate**, and we refer to the attribute for which the aggregate is computed as the **aggregation attribute** (in the example, salary), and to the set of attribute values the aggregate is computed over as its **aggregation extent** (in the example, the extent of the salary attribute).

The aggregation extent need not span the entire extent of

the aggregation attribute. Query languages often offer the means to partition the values of the aggregation attribute into a collection of aggregation extents. For example, we may compute the sum total of employee salaries *per location*. If so, the aggregation process will be carried out over as many aggregation extents as there are partitions (in the example, location values actually occurring). The membership of an aggregation attribute value in a partition is decided by a **partitioning predicate** (in the example, simple equality). In the most general case, a partitioning predicate may also be a join predicate, i.e., the partitioning predicate may refer to both attributes of the entity whose property the aggregation attribute is and to attributes of other entities. In that case, one refers to the former as **membership attributes**, and to the latter as **partition attributes**. In the example, the partitioning predicate is not a join predicate and hence only one membership attribute (i.e., location) is referred to. Depending on the partitioning predicate, the resulting partitions may overlap. (It is well known disjoint partitions over multidimensional data such as spatial, temporal and spatio-temporal data are rare.)

Classical aggregation functions in conventional database management systems (DBMSs) include **count**, **sum**, **avg**, **min** and **max**. Conventional DBMSs also classically support equality-based partitioning. For example, consider the query Q1.1: What is the average rent of shops classified by type? and its query language expression as follows:

```
select      avg(s.rent)
from        shops s
group by    s.type;
```

In Q1.1, the aggregation attribute is `s.rent` and the partition attribute is `s.type`, where `s` is in the extent of `shops`. Thus, there are as many aggregation extents as there are different values for `type` (e.g., `rented`, `owned`, etc.) in the extent of `shops`. The aggregation function **avg** is applied to each such aggregation extent (i.e., each partition) in turn (within which all of the values for `rent` are for shops of a particular `type`) to yield the (bag of) aggregates.

The type extensions that lead to temporal, spatial and

spatio-temporal databases also lead to aggregation possibilities being greatly expanded. Moreover, the aggregation processes are more complicated and, hence, more prone to inefficiencies. Specifically, there are more aggregation functions (e.g., geometric union of spatial values, moving average over a time interval), there are more partition predicates than equality (e.g., spatial containment, temporal overlapping) and there are more partition strategies (e.g., group by a grid reference such as per square mile).

This means that supporting aggregation in spatio-temporal DBMSs is a far more complex problem than in conventional DBMSs. Researchers have tended to respond to this increase in complexity with tailor-made mechanisms. However, the solutions that have emerged from such tailor-made mechanisms are somewhat narrow. By narrow, we mean that the solution exploits, in a fundamental way, very specific properties of the problem, and, as a consequence of that, it is often not applicable (or not as efficient) if the parameters of the query are changed.

Thus, many solutions rely on specialist indexing structures specifically designed for evaluating aggregation queries. We observe that, while these techniques may guarantee desirable response times, reliance on indices causes these guarantees to be often dependent on a fixed partitioning strategy and on the aggregation process being applied to scanned collections (as opposed to intermediate results, upward from the leaves of a query execution plan¹).

This paper contributes a generic framework that, as shown by the accompanying experimental evaluation, provides efficient support for the wide range of partitioning and aggregation operations that a spatio-temporal DBMS is expected to support over both stored and derived data.

The remainder of the paper is structured as follows. Section 2 briefly describes related work. Section 3 introduces some background, viz., the spatio-temporal model that motivated the contributions of the paper, as well as the kinds of aggregation queries and of partition strategies that arise in that context. Section 4 contains the main contributions of the paper, in the form of a generic algorithmic strategy for aggregations based on multidimensional hashing. Section 5 describes experimental evidence that our proposal scales appropriately while being less dependent on the parameters of the aggregation process as well as on the nature (i.e., stored or derived) of the aggregation domains. Finally, Section 6 draws some conclusions.

¹Recall that a *query execution plan* is a tree whose nodes denote query algebraic operations and whose directed edges denote upwards flow of data from the child to the parent node. If a child node is a store, the parent is said to operate on *stored data*, else the parent operates on *derived data*, also referred to as intermediate results. Figure 8 is an example.

2. Related Work

Previous proposals for efficient aggregation over spatial, temporal and spatio-temporal data, can be categorized into three groups. The first is characterized by the use of plain search trees (e.g., R-trees [8]) augmented with additional information. An example in the spatial domain is the MRA-tree [13]. Each internal node of an MRA-tree maintains a search key and a set of aggregates (e.g., `sum`, `min`, `max`, etc.) of all the aggregation attribute values stored in the leaf nodes of its subtree and characterized by associated point data (e.g., representing locations) in space. The second group is characterized by the use of specialized search trees on scalar aggregates whose search keys are spatial or temporal attributes. Examples include the MR-tree [22] in the spatial domain, and the SB-tree [20] and the Multiversion SB-trees [21] in the temporal domain. These trees materialize aggregates in each node incrementally. Each node entry stores a spatial (e.g., a rectangle) or a temporal (e.g., a time interval) search key associated with the aggregates of the aggregation attribute values covered by that spatial search key and valid for the period covered by that temporal search key. The third group is characterized by the use of specialized search trees on precomputed partitions (e.g., aR-trees [15] in the spatial domain; aggregation trees [11] and PA-trees [10] in the temporal domain; and aRB-trees [16] in the spatio-temporal domain). aR-trees are similar to MRA-trees, but aR-tree leaf node entries are aggregation results over spatial partitions rather than raw aggregation attribute values as in MRA-trees. Aggregation trees and PA-trees store the start or the end point of time intervals representing a temporal boundary for an associated temporal partition at leaf nodes, and such time points are hierarchically categorized using binary search tree mechanisms. While PA-trees can support diverse scalar aggregation functions, aggregation trees can support only one chosen aggregation function. aRB-trees index aggregates that are precomputed over spatio-temporal partitions using an R-tree for spatial domain aggregates and a B-tree for each node entry of the R-tree for temporal domain aggregates.

It seems clear, therefore, that all the proposal above have focused on efficient retrieval of, or partitioning for, scalar aggregates by spatial, temporal and spatio-temporal attributes, respectively. None of them is robust across the diversity of aggregation functions and partitioning predicates that is to be expected by the richness of the underlying type extensions. This is probably because few complete prototype spatio-temporal DBMSs have been built, resulting in an emphasis on algorithms for specialized tasks rather than more general implementation strategies. As a result, such techniques, in general, have difficulties in supporting diversity in aggregation queries. Firstly, as more

aggregation functions turn out to be of interest to users, the amount of extra information to be stored in augmented search trees, or the number of specialized search trees, increases. It may then be problematic to maintain such large trees on all possible attributes for general query evaluation. Secondly, augmented search trees in the spatial domain can only support approximate aggregates, since the search key is usually an approximation (e.g., a minimum bounding rectangle (MBR) [8]) of the actual spatial value. Although, this may be useful for systems where prompt responses to user queries are more important than precision, exact answers are expected more often than not. Thirdly, disk-based search trees (e.g., MRA-trees, SB-trees, Multi-version SB-trees and aRB-trees) incur significant overheads if they aim to support aggregation on derived data (i.e., intermediate results in query processing) because very different policies/mechanisms are required for the incremental maintenance of different trees and the indices will need to be built on-the-fly. Finally, only a few index structures have proved to be sufficiently efficient and stable in practice to be incorporated into mainstream DBMSs. Often, the complexity and cost of developing new indexes and incorporating them into a DBMS are severe [12]. Therefore, it may not be an attractive long-term solution to set out to develop a different search tree technique for each of the many narrow groups into which different aggregation queries might fall.

3. Background

Spatio-Temporal Data

Although the results reported in this paper have broader applicability, they have been worked out in detail and implemented in the context of Tripod [5, 6], a prototype spatio-temporal object-oriented DBMSs. Tripod comes equipped with set-based spatial and timestamp data types over which a specialized mechanism, called a *history*, is defined. Histories are then used to track changes to spatial and aspatial data. The Tripod spatial data types are the **Points**, **Lines** and **Regions** of the ROSE algebra [7] plus their singleton counterparts, viz., **Point**, **Line** and **Region**. The Tripod timestamp types specialize the ROSE spatial types. Thus, **Instants** and **TimeIntervals** values are seen as one-dimensional versions of **Points** and **Lines** values, respectively, and the same is true for their singleton counterparts. A *history* is a collection of *states*, i.e., pairs of the form (τ, σ) , where τ and σ denote a *timestamp* and a (spatial or aspatial) *snapshot* value, respectively. The Tripod type system extends the ODMG standard type system [2] for object databases as shown in Figure 1. Each of the type extensions to the ODMG object model in Tripod is associated with a rich collection of operations and predicates [5, 6] extending those in the ROSE algebra [7]. These can be invoked

either from the Tripod query language or from application programs through the Tripod language bindings (extending, respectively the ODMG OQL and C++ language bindings).

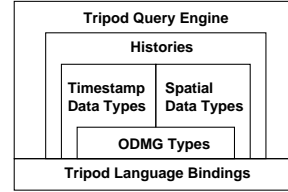


Figure 1. The Layered Tripod Architecture

Tripod was designed and implemented to deliver *orthogonality* (i.e., applications can make use of aspatial, spatial and temporal types in isolation or in any combination, without being penalized for doing, or not doing, so) and *synergy* (i.e., whenever distinct aspects *are* used in combination, then Tripod aims to respond with appropriate benefits to applications). One of the manifestations of these design principles is the choice of the same ROSE-algebraic foundations for the treatment of time as is used for space [5]. This allows Tripod to give semantics to spatio-temporal queries that relate precisely, and in principled ways, to both aspatial and spatial queries (as discussed in [4]). One step further leads Tripod to embrace approaches to query evaluation that are generic, but robust, across aspatial, spatial and spatio-temporal. Thus, this goal, which lies at the heart of this paper’s contributions, has also underpinned the generic evaluation strategies for spatio-temporal join queries proposed and studied in [9].

Aggregation Queries

We now briefly motivate the contributions of this paper by describing how rich type extensions give rise to a very diverse range of aggregation queries, both in terms of aggregation functions and in terms of partitioning predicates.

The aggregation process can be modelled using higher-order functions, familiar in functional programming [1]. For non-partitioned aggregation, **fold** suffices; for partitioned aggregation, **filter** is also needed. Given these higher-order functions, the function definitions in Figure 2 are one approach to defining aggregation semantics. Given those, **sum** (and its **group by** version) are definable as in Figure 3.

Now, the enrichment of the type system that underlies a DBMS causes there to be many more base aggregation functions and partitioning predicates than those on integer (exemplified by **add**, and by **odd** and **even**, respectively) in Figure 3. As the computational cost of either (or both) of these kinds of functions grows (as it does for both spatial, temporal and spatio-temporal DBMSs), one can respond (as the proposals cited in Section 2 testify) with tailor-made strategies to retain efficiency and scalability in evaluating

```

# aggregation can be defined in terms of fold
def agg(Function, Extent):
    return fold(Function, Extent)

# defining partitioning in terms of filter allows
# partitioned aggregation to be defined in turn
def partition(Extent, PredicateList):
    Partitions := []
    for Predicate in PredicateList:
        Partitions.insert(filter(Predicate, Extent))
    return Partitions

def agg_by_partition(Extent, Function, PredicateList):
    Bag := []
    Partitions := partition(Extent, PredicateList)
    for Partition in Partitions:
        Bag.append(agg(Function, Partition))
    return Bag

```

Figure 2. Aggregation Semantics

```

# given add as a base aggregation function
# and given even and odd predicates on integers,
# sum (with and without group by) is
# definable in terms of Figure 2
def add(X, Y): return X+Y

def even(Value):
    if Value mod 2 = 0: return True
    else: return False
def odd(Value):
    if not even(Value): return True
    else: return False

def sum(Extent): return agg(add, Extent)

def sum_with_group_by(Extent, PredicateList):
    return agg_by_partition(Extent, add, PredicateList)

# one can apply sum to some Extent to group by parity
Bag = sum_with_group_by(Extent, [odd, even])

```

Figure 3. Defining sum

aggregation queries. But, as pointed out, the diversity is such as to call into question whether tailor-made responses are adequate on grounds other than pure efficiency. Thus, this paper proposes a generic algorithmic framework that trades off some efficiency for greater versatility while retaining scalability (as Section 5 indicates).

Partitioning Strategies

There are many possible partitioning strategies of general interest to users in the case of temporal, spatial, and spatio-temporal data. For the purposes of providing an empirical setting to the contributions of this paper, we now describe some general strategies (inspired by [16, 17]) and propose example queries that presuppose them.

We first observe that both timestamp and spatial types draw values from an underlying abstract interpretation structure. For example, a point in two-dimensional space is defined by a pair of values over a coordinate system, and a line by a pair of such points. Likewise, an instant is defined by a value in a timeline, and a time interval by a pair of such instants. In the case of both Tripod timestamp and spatial types, the underlying abstract structure is a *realm* [7]. A realm is a relation on a finite set of integers satisfying some constraints [7]. In the case of Tripod spatial values, which are two-dimensional, the relation is binary and gives rise to a *grid* structure. Tripod timestamp types and operations are specializations to one dimension of their corresponding two-dimensional types (i.e., **Points** and **Lines** are specialized into **Instants** and **TimeIntervals**, respectively). In this case, the relation is unary and gives rise to a non-branching *timeline*.

Now, we consider three general kinds of partitioning strategy: one relying on predicates defined over the underlying abstract interpretation structure for data values, another on predicates defined on the data values themselves, and a third, in which the predicates are defined on the occurrence of events that cause snapshot values to change over time. We refer to these strategies as **(interpretation-)structure-based (IB)**, **value-based (VB)**, and **snapshot-change (SC)**. Example predicates defined over a grid and over a timeline might be containment within one-mile squares (in the spatial case of grids), or within year-long periods (in the temporal case of timelines). Example predicates on the data values themselves might be containment in an electoral district, or whether some rent rise took place during the time that a shop was being refurbished. Combinations of spatial and temporal, IB- or VB-, predicates give rise to spatio-temporal partitioning strategies, e.g., the average area occupied by shops per electoral district per decade. Example aggregation queries involving different partitioning strategies, or combinations thereof, are now given.

spatial-IB: What is the total space occupied by shops in each one-mile-square grid reference in Paris? (Q3.1)

spatial-VB: What is the total space occupied by shops per electoral district in Paris? (Q3.2)

temporal-IB: What was the total space occupied by shops per month in 2003? (Q3.3)

SC: What were the different values for time-evolving total space occupied by shops during 2003? (Q3.4)

spatial-IB, temporal-IB: What was the total space occupied by shops in each one-mile-square grid reference per month in 2003? (Q3.5)

spatial-VB, temporal-IB: What was the total space occupied by shops per electoral district in Paris between 2001 and 2003? (Q3.6)

spatial-IB, SC: What were the different values for time-evolving total space occupied by shops in each one-mile-square grid reference in 2003? (Q3.7)

The semantics of partitioning strategies often need to deal with boundary cases and that specific combinations require specific treatments. This argues for the likelihood of index-based approaches being forced to be narrow. Note also how the analysis above shows that there is little reason to expect certain partitioning strategies to be dominant: spatio-temporal DBMSs ought to be able to support a wide variety without undue performance penalties. Note, finally, that the aggregation functions and the partitioning predicates are likely to be more diverse across all user queries than in the examples above: again, spatio-temporal DBMSs ought to be able to support all of them without undue performance penalties. Thus, as pointed out in Section 2, while approaches based on index-based schemes and access methods can be efficient, they suffer from two shortcomings: firstly, the indices tend to be specific to specific partitioning strategies or predicates, therefore index-based approaches tend to fall short of being effective across sufficiently diverse queries; secondly, the indices tend to be prohibitively expensive to build on-the-fly, therefore such responses tend not to be applicable to derived data (i.e., to aggregation over the output internal nodes of a plan) and could be said to fall short of being effective across sufficiently diverse query plans.

4. A Generic Algorithmic Framework

The contributions of this paper are based on the observation that hash-based joins routinely deliver efficiency in conventional query evaluation on both stored and derived inputs and tend to be robust across a diversity of join predicates. We propose that using hash-based techniques for constructing the partitions leads to performance that is sufficiently efficient, in a scalable manner, over a greater diversity of queries and query plans than index-based ones.

Hash-based techniques have been suggested for aggregation. For example, [3] proposes to place incrementally computed aggregates in a hash table. However, our use of hash tables has more in common with their use in hash-based joins, where two collections are matched only after one of them is initially consumed to populate a hash table. For this purpose, we have extended the filtering stage of the spatial hash join technique proposed in [14] (and taken into account such contributions as [18]) to use, uniformly, a multidimensional envelope (consistently with Tripod’s uniform use of realms as the underlying abstract interpretation structure for its multidimensional data types). Thus, we speak of *minimum bounding envelopes* (MBEs), rather than *rectangles*, their two-dimensional specializations. Multidimensional hash techniques have to deal with the impossibility, in practice, of producing disjoint bucket extents. They must, therefore, devote specific attention to overlapping buckets and the consequent issue of multiple assignments, as we do.

Our overall strategy is to construct hash tables for the extents of both the partition attribute and the aggregation attribute. However, we build the former first and copy its carving of the multidimensional space into bucket-wide MBEs across to the construction of the latter’s buckets. This means that we ensure that no matter which partitioning strategy is used, the values of both the aggregate attribute and the partition attribute are placed into the same number of buckets with identical capacity and covering identical regions of the multidimensional space underlying the data. This is crucial in ensuring that the technique is robust across many partitioning strategies and predicates. Once the two hash tables are populated we can read them (as a join operation would) to determine which aggregation attribute value belongs to which partition according to the partitioning predicate used. The result is the determination of all the aggregation extents. Given these, we apply the aggregation function to each one and obtain the desired aggregates.

Hash-Based Multidimensional Partitioning

We now describe in more detail the determination of the aggregation extents using multidimensional hashing. The pseudo-code is given in Figure 4 and is cross-referenced to the informal description of its five stages now given.

Stage 1: Construct a hash table for partition attribute values with empty buckets whose MBEs spatially partition the space without overlaps.

Stage 2: Populate the hash table by assigning each partition attribute value to that bucket with whose MBE its own MBE overlaps the most, and adjust that bucket’s MBE to the extent necessary to fully contain the partition attribute value’s MBE.

Stage 3: Construct a hash table for aggregation attribute values with empty buckets whose MBEs are copied

from the corresponding bucket in the hash table for partition attribute values.

Stage 4: Populate the hash table by assigning each aggregation attribute value to every bucket with whose MBE its associated membership attribute MBE overlaps.

Stage 5: For each bucket in the aggregation-value hash table, traverse the bucket and, for each aggregation attribute value in it, check (for each partitioning attribute value in the corresponding bucket in the partition-value hash table) whether the partitioning predicate holds: if so, then map the aggregation attribute value to the partition defined by the matching partition attribute value.

Some assumptions made in Figure 4 are now made explicit. By `md.partition` we mean our specific proposal for the partitioning step in the aggregation semantics given in Figure 2 (which, therefore, instantiates the more general definition given there). Note that, in Figure 5, by [Predicate] we mean that the use of `md.partition` constrains the predicate list in Figure 2 to have cardinality equal to 1. Insofar as the experimental evaluation reported in Section 5 has only ranged over spatio-temporal data, by *multidimensional* we mean 2-dimensional spatial data with a third, temporal, dimension. Thus, our experiments have not dealt with aggregation on scalars (e.g., `count`, `sum`, `avg`, `min` and `max` on integers). By *overlap* we mean a polymorphic function on MBEs taking two multidimensional instances of compatible type and returning a Boolean. By *get_m_attr_val* we mean a function that returns the membership attribute values associated with an aggregation attribute value. By *key* we mean a function that computes an index into a sequence from a multidimensional value, e.g., it assigns a partition number to a spatial value. Finally, our notation uses tuple assignment as follows. Consider two expressions x and y . The expression $x := (y, z)$ binds (by packing, as it were) x to the pair (y, z) . If so, then the expression $a, b := x$ binds (by unpacking, as it were) a to y and b to z .

Generic Aggregation over Multidimensions

Given the partitioning approach described in Figure 4, the generic aggregation framework in Figure 5 can be defined by simply extending Figure 2 as indicated.

The framework is based upon the fact that, given the diversity of partitioning strategies and predicates, the dominant computational cost is the generation of the aggregation extents. This is dealt with using multidimensional hashing, which has the advantage of being generic across many strategies and predicates and of being applicable to both stored and derived data. The other cost is that of applying the aggregation function to each aggregation extent. The framework clearly separates these costs out, and this translates into query plan topologies that are more open to scrutiny by query optimizers and by adaptive query processors.

```

def md.partition(AggregationAttributeExtent,
                PartitionAttributeExtent,
                partitioning.predicate,
                bucket_size, number_of_buckets, max_MBE):
    # Stage 1
    P_ValueHashTable := []
    BucketMBEs := []
    i := 0
    while i < number_of_buckets:
        bucket := Bucket(bucket_size, [])
        # E := a multidimensional region
        # with size up to (max_MBE/number_of_buckets)
        # that does not overlap with any bucket.MBE
        # in P_ValueHashTable
        bucket.MBE := E
        BucketMBEs[i] := bucket.MBE
        P_ValueHashTable[i] := bucket
        i := i+1

    # Stage 2
    while not end_of(PartitionAttributeExtent):
        read p_value in PartitionAttributeExtent
        best := ((number_of_buckets-1), max_MBE, max_MBE)
        i := 0
        while i < number_of_buckets:
            if overlap(MBE(p_value), P_ValueHashTable[i].MBE):
                # how_much := the extension needed in the bucket MBE
                # for it to contain the value
                # extended_MBE := the MBE generated
                # by so extending the bucket MBE
                best_bucket, its_MBE, the_smallest_extension := best
                if how_much < the_smallest_extension:
                    best := (i, extended_MBE, how_much)
                i := i+1
        assigned_bucket, extended_MBE, how_much := best
        P_ValueHashTable[assigned_bucket].MBE := extended_MBE
        P_ValueHashTable[assigned_bucket].insert(p_value)

    # Stage 3
    A_ValueHashTable := []
    i := 0
    while i < number_of_buckets:
        bucket := Bucket(bucket_size, [])
        bucket.MBE := P_ValueHashTable[i].MBE
        A_ValueHashTable[i] := bucket
        i := i+1

    # Stage 4
    while not end_of(AggregationAttributeExtent):
        read agg_val in AggregationAttributeExtent
        MatchingBuckets := []
        i := 0
        while i < number_of_buckets:
            if overlap(MBE(get_m_attr_val(agg_val)),
                    A_ValueHashTable[i].MBE):
                MatchingBuckets.insert(i)
                i := i+1
        j := 0
        while j < length(MatchingBuckets):
            A_ValueHashTable[MatchingBuckets[j]].insert(agg_val)
            j := j+1

    # Stage 5
    Partitions = []
    i := 0
    while i < number_of_buckets:
        while not end_of(A_ValueHashTable[i]):
            read x in A_ValueHashTable[i]
            while not end_of(P_ValueHashTable[i]):
                read y in P_ValueHashTable[i]
                if partitioning_predicate(get_m_attr_val(x), y):
                    Partitions[key(y)].insert(x)
            i := i+1

    return Partitions

```

Figure 4. Multidimensional Partitioning

```

def agg_by_partition((AggAttrExtent,PartAttrExtent),
                    Function,
                    [Predicate]):

# assume default bucket_size, number_of_buckets, max_MBE

P := md_partition(AggAttrExtent,PartAttrExtent,Predicate)

Container := []
for p in P:
    Container.append(agg(Function,p))
return Container

```

Figure 5. Generic Aggregation Framework

Note that the computational cost of the aggregate function and of the partition predicate are likely to be characterized at the level of the DBMS kernel if, as in Tripod, the spatial, temporal and spatio-temporal algebras are implemented as primitive data types. Thus, the robustness and scalability of the framework needs to be demonstrated across different queries that make use of diverse partitioning strategies. In other words, the empirical question of interest is whether the multidimensional hashing approach in Figure 4 is robust and scalable.

5. Experimental Evaluation

The experimental evaluation described in this section was aimed at gathering evidence as to whether aggregation queries that make use of diverse partition strategies can be evaluated reliably efficiently using the framework contributed in Section 4. We aim to show that, across diverse partitioning needs, this hash-based approach is sufficiently economical to allow for it to be applied to both leaf and internal nodes of query plans. By sufficiently economical we mean that costs should not be extraordinary in comparison to other demanding operations, such as joins.

Schema

Figure 6 is the Tripod ODL schema used in the evaluation. Here, and elsewhere in the paper, reserved words are in **bold font**. Three classes are used, City, ElectoralDistrict and Shop. All are historical and have a historical spatial attribute, *boundary* for the first two and *land_parcel*, for the third, with the latter denoting the plot of land occupied by the shop.

Queries

The queries used were (Q3.1) to (Q3.5) from Section 3. Recall that they are designed to exhibit a variety of partitioning strategies, as follows:

- (Q3.1): spatial-structure-based
(Q3.2): spatial-value-based
(Q3.3): temporal-structure-based

```

historical (TimeIntervals, YEAR)
class City
  (extent Cities ){
  attribute string name;
  relationship
  set<ElectoralDistrict> has_districts
  inverse ElectoralDistrict::belongs_to;
  historical (TimeIntervals, YEAR)
  attribute Regions boundary;};
historical (TimeIntervals, YEAR)
class ElectoralDistrict
  (extent EDistricts ){
  relationship
  City belongs_to
  inverse City::has_districts;
  historical (TimeIntervals, YEAR)
  attribute Regions boundary;};
historical (TimeIntervals, YEAR)
class Shop
  (extent Shops ){
  historical (TimeIntervals, YEAR)
  attribute Regions land_parcel;};

```

Figure 6. Tripod Schema for Evaluation

- (Q3.4): snapshot-change
(Q3.5): spatial-, temporal-structure-based

Due to space constraints we only give the OQL and execution plan for query (Q3.2), where aggregation and membership attribute values are snapshots of time-evolving shop land parcels, partition attribute values are snapshots of time-evolving electoral district boundaries in Paris and the partitioning predicate is spatial containment. Figure 7 is the Tripod OQL expression corresponding to query (Q3.2). Note the **snapshot** reserved word: it can be conceived of as denoting an accessor function that returns the snapshot in a state in a history. Note also the spatial-value-based partitioning strategy in the **group by** clause. Note, finally, the aggregation function used is **spatial_union** [7] (i.e., the equivalent on spatial values to **sum** on integers). It is applied to each partition generated by the evaluation of the **group by** clause.

```

select      spatial_union(
            select lp.snapshot from partition)
from        Shops s, s.land_parcel lp
group by    lp.snapshot spatially_contained
            (select b.snapshot
             from  EDistricts e, e.boundary b
             where e.belongs_to.name = "Paris");

```

Figure 7. Query Expression for (Q3.2)

Figure 8 shows a query execution plan that might be generated by a query optimizer for the expression in Figure 7. It corresponds to the pseudo-code given in Figures 4 and 5 as follows. The first and second arguments to the **md_partition** operation in Figure 4 flow, respectively, from the left and right children of the **md_partition** node in Figure 8. The pseudo-code in Figure 5 covers the topmost edge in Figure 8 (i.e., edges are just flows). The call to **md_partition**

corresponds to the child node of the root. The root itself corresponds to the application of the aggregation function to the partitions flowing from the child node.

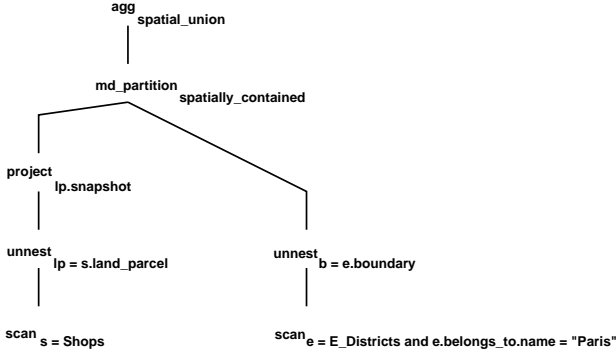


Figure 8. Query Plan for (Q3.2)

Datasets

The datasets used were generated by a slightly modified version of GSTD [19]. Our version of GSTD allows datasets to be parameterized by cardinality, number of states per object, spatial density (i.e., the percentage $100((\sum object_area)/workspace_area)$), the variation in the extent of spatial objects, and the variation in duration of consecutive snapshots. To obtain collections that vary in terms of size and density, we have varied the initial cardinality of spatial objects and the number of snapshots. As shown in [9], if we vary the initial cardinality (resp., the number of snapshots) while keeping the initial spatial density and the number of snapshots (resp., the initial cardinality) constant, we obtain collections of increasing overall temporal (resp., spatial) density as the cardinality of the collection increases. Since Tripod only models discrete change (i.e., in the spatial case, objects are not conceived of as having changed position between recorded timestamps) we keep locations fixed. Between states, the spatial extent of an object varies randomly by $\pm 10\%$ in both dimensions. The timestamps associated with states are consecutive in time (i.e., there are no gaps between timestamps in a history). The temporal density Δ_t , defined to be the length of the timeline divided by the number of states per object, is used as a parameter to compute the endpoint e_i of the timestamp of state i from the the endpoint e_{i-1} of the timestamp of state $i - 1$. Thus, $e_i = e_{i-1} + k\Delta_t$ where k is a randomly chosen value between 0.5 and 1. The distribution of spatial objects, of the changes in their extent, and of the duration of consecutive snapshots is uniform in all cases. These characteristics of the two data collections used, viz., DC1 and DC2, are summarized in Figure 9.

For generating partition attribute extents in the case of structure-based strategies, given a number of temporal spans (e.g., years) and spatial cells (e.g., one-mile squares)

Parameters	Sets of Data Collections	
	DC1	DC2
cardinality	5000 to 15000 in steps of 2500	200
number of states per object	20	500 to 1500 in steps of 250
spatial density	30 %	
variation in extent	[-10, 10] % of object size randomly	
variation in snapshot duration	[50, 100] % of temporal density randomly	
distribution	uniform	

Figure 9. Data Collections for Evaluation

we have divided the overall temporal and spatial spans of the datasets (defined with reference to the underlying temporal and spatial realms) proportionally.

Environment

The experiments were run on a 700MHz Pentium III PC with 256Mb main memory running RedHat Linux version 7.2. Each experiment was run three times with the operating system buffer cache being flushed after each run: we plot the average of the three runs.

Results

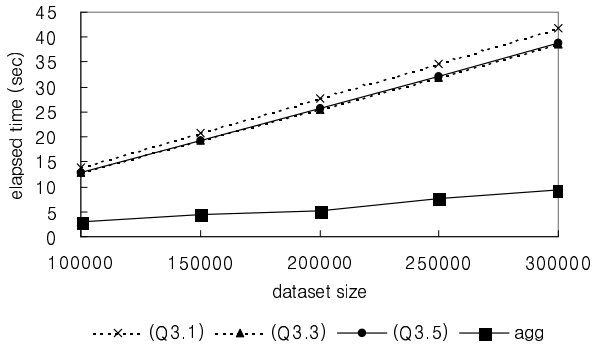
Figure 10 refers to data collection DC1. For each *dataset size* (measured as the number of spatial objects times the number of snapshots), Figure 10 plots, in (a), the *elapsed time*, and, in (b), the *partition size* (measured in number of matching pairs). Then, the *the number of partitions* is plotted against, in (c), the *elapsed time*, and, in (d), the *partition size*.

In Figure 10 (a) and (c), one extra curve, referred to as *agg*, plots, per dataset size and per number or partitions, respectively, the elapsed time of the aggregation process alone, i.e., of the topmost edge in Figure 8. The other three of the curves, in all the plots, correspond to queries (Q3.1), (Q3.3), and (Q3.5).

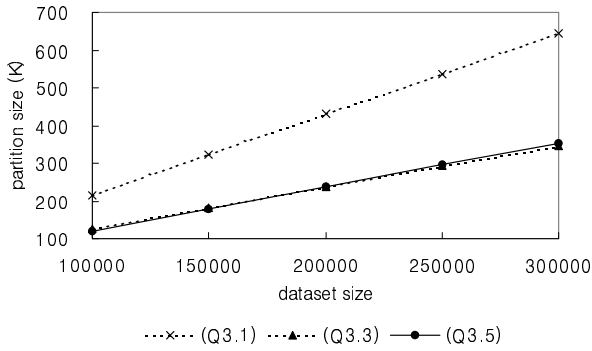
Results for queries illustrating value-based and snapshot-change partition strategies (viz., (Q3.2) and (Q3.4)) are not presented in Figure 10 because their measurements are very similar to those of (Q3.1) and (Q3.3), respectively. This is, we note, evidence for the robustness of our approach. Likewise, results for the data sets in data collection DC2 are not presented because the curves grew similarly to the growths observed in Figure 10 for corresponding partitioning strategies over data collection DC1, except that spatial-IB partitioning performs slightly poorer than temporal-IB and (spatial-IB, temporal-IB) partitioning. Again, this is evidence for the robustness of our approach.

Analysis

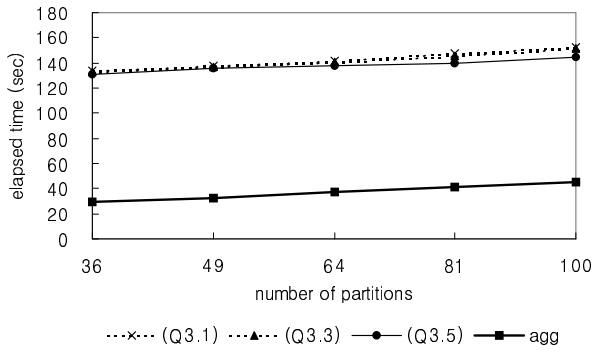
To show that the introduction of our approach in query execution plans is unlikely to lead to escalating costs, we note that:



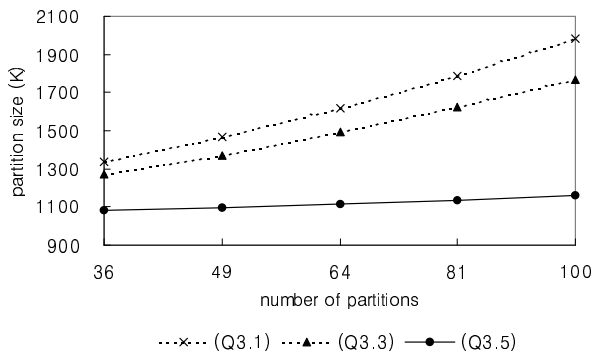
(a) Dataset Size v. Elapsed Time



(b) Dataset Size v. Partition Size



(c) Number of Partitions v. Elapsed Time



(d) Number of Partitions v. Partition Size

Figure 10. Experimental Results (on DC1)

1. all curves show linear (or near linear) growth for all the queries (and hence, for all partitioning strategies) used;
2. all slopes are gentle.

We interpret this to mean that our approach leads to elapsed times and partition sizes that scale across a range of partitioning strategies both on dataset size and on the number of partitions. In short, our approach seems to perform in a scalable manner across diverse aggregation queries, whereas tailor-made proposals, such as those discussed in Section 2, are, by definition, unlikely to do so.

To show that the cost of introducing our approach in query execution plans is not extraordinary, we note that:

1. as the *agg* curve shows, the cost of the *md_partition* \rightarrow *agg* edge in the query executions plotted in Figure 10 is modest, at around 25%, with respect to the time cost of the remainder of the query plan, and proportionate, as the edge comprises around 30% of the plan, in number of nodes;
2. the cost of the *md_partition* \rightarrow *agg* edge is likely to incur costs that are similar in magnitude to a multidimensional hash-join, given that in both cases the hashing process is the dominant one.

We interpret this to mean that our approach does not incur a performance penalty, if by penalty is meant that our approach will contribute (to the overall cost of the query) a cost that is significantly greater than that of an operation of similar complexity, such as a multidimensional hash-join. In short, our approach is likely to perform sufficiently well to be used in both internal and leaf nodes of a query execution plan, whereas tailor-made proposals, such as those discussed in Section 2, are, due to their reliance on indices, unlikely to do so.

6. Conclusions

Aggregation over temporal, spatial and spatio-temporal data is an important and challenging problem in query processing. This has been reflected in the significant number of proposals for algorithms for the efficient computation of specific kinds of aggregation. However, most such proposals tend to be narrowly applicable whereas there is reason to believe that temporal, spatial and spatio-temporal data suggest to users a great diversity of aggregation queries, especially with regards to partitioning strategies.

This paper has contributed:

1. An algorithmic framework that can be applied to many different forms of aggregation over spatial, temporal and spatio-temporal data.
2. Experimental evidence that the algorithmic framework provides a scalable solution across queries that are diverse on the partitioning strategy used and across query plans that are diverse on whether aggregation is carried out over stored or derived data.

The contributions of the paper succeed, therefore, in striking a balance with respect to a classical trade-off, viz., efficiency v. generality. A generic approach such as ours cannot expect to deliver comparable raw performance to that of a proposal which is tailor-made to a narrow class of aggregation queries. However, this tailoring process can sometimes hinder the applicability of the solution by overly narrowing the class of queries it evaluates efficiently. In contrast, the approach contributed in this paper seems capable of delivering reliably scalable performance over a significantly larger class of aggregation queries than previous proposals. Moreover, the cost of adopting it is not extraordinary in comparison to comparably costly operators (e.g., spatio-temporal hash-joins). This, we propose, makes it more suitable for deployment in implemented spatio-temporal DBMSs, and our experience with Tripod is evidence for that.

References

- [1] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.
- [2] R. G. G. Cattell et al. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
- [3] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [4] T. Griffiths, A. A. A. Fernandes, N. Djafri, and N. W. Paton. A Query Calculus for Spatio-Temporal Object Databases. In *Proc. TIME*, pages 101–110, 2001.
- [5] T. Griffiths, A. A. A. Fernandes, N. W. Paton, and R. Barr. The Tripod Spatio-Historical Data Model. *Data & Knowledge Engineering*, 49:23–65, 2004.
- [6] T. Griffiths, A. A. A. Fernandes, N. W. Paton, K. T. Mason, B. Huang, M. F. Worboys, C. Johnson, and J. G. Stell. Tripod: A Comprehensive System for the Management of Spatial and Aspatial Historical Objects. In *Proc. ACM GIS*, pages 118–123, 2001.
- [7] R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):243–286, 1995.
- [8] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, June 1984.
- [9] S. H. Jeong, N. W. Paton, A. A. A. Fernandes, and T. Griffiths. An Experimental Performance Evaluation of Spatio-Temporal Join Strategies. *Transactions in GIS*, 2004. Accepted for publication.
- [10] J. S. Kim, S. T. Kang, and M. Kim. Effective Temporal Aggregation Using Point-Based Trees. In *Proc. DEXA*, pages 1018–1030, 1999.
- [11] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proc. ICDE*, pages 222–231, 1995.
- [12] M. Kornacker. High-Performance Extensible Indexing. In *Proc. VLDB*, pages 699–708, 1999.
- [13] I. Lazaridis and S. Mehrotra. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *Proc. SIGMOD*, 2001.
- [14] M. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *Proc. ACM SIGMOD*, pages 247–258, 1996.
- [15] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *Proc. SSD*, pages 443–459, 2001.
- [16] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing Spatio-Temporal Data Warehouses. In *Proc. ICDE*, 2002.
- [17] R. T. Snodgrass, S. Gomez, and L. E. McKenzie. Aggregates in the Temporal Query Language TQuel. *IEEE TKDE*, 5(1):826–842, 1993.
- [18] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proc. ICDE*, pages 282–292, 1994.
- [19] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. SSD*, pages 147–164, 1999.
- [20] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *Proc. ICDE*, pages 51–60, 2001.
- [21] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proc. ACM PODS*, 2001.
- [22] D. Zhang and V. J. Tsotras. Improving Min/Max Aggregation over Spatial Objects. In *Proc. ACM GIS*, pages 88–93, 2001.