GPAC: Generic and Progressive Processing of Mobile Queries over Mobile Data

Mohamed F. Mokbel Walid G. Aref*

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{mokbel, aref}@cs.purdue.edu

ABSTRACT

This paper introduces a new family of *Generic* and *Progressive* algorithms (GPAC, for short) for continuous mobile queries over mobile objects. GPAC provides a general skeleton that can be tuned through a set of methods to behave as various continuous queries (e.g., continuous range queries and continuous k-nearest-neighbor queries). GPAC algorithms aim to provide three goals: (1) Online evaluation through an in-memory processing of the incoming mobile data. (2) Progressive evaluation through employing an *incremental* evaluation paradigm. (3) Fast query response through employing an *anticipation* paradigm. Query answer is anticipated and is *cached* in memory to allow for fast evaluation. GPAC algorithms are encapsulated in physical pipelined query operators. GPAC pipelined operators can be combined with traditional query operators in a query execution plan to support a wide variety of continuous queries. Experimental results based on a real implementation inside a prototype streaming database engine show the efficiency of GPAC operators in providing incremental and fast response for continuous queries.

1. INTRODUCTION

The rapid increase of spatio-temporal applications calls for new query processing techniques to deal with both the spatial and temporal domains. Examples of these applications include location-aware services [18], traffic monitoring [23], and enhanced 911 service. Such applications receive streaming data from mobile objects continuously (e.g., from moving vehicles in road networks). Recent research efforts for continuous spatio-temporal query processing (e.g., see [12, 14, 15, 16, 19, 25, 28, 29, 31, 33, 34]) rely on the ability to store and index spatio-temporal data. Although working well in theory, such indexing schemes fail in practical to cope with high arrival rates of spatio-temporal data streams. With the notion of data streams, only in-memory algorithms can be realized.

In this paper, we propose the *Generic Progressive Algorithm* (GPAC, for short) for continuously evaluating mobile queries over spatio-temporal data streams. GPAC provides a ***generic*** skeleton that can be tuned through a set of methods to behave as different continuous mobile queries (e.g., continuous range queries and k-nearest-neighbor queries). The GPAC family of algorithms is mainly designed to achieve three goals: (1) Online evaluation. Incoming data is processed in-memory without the need for secondary storage. (2) Progressive evaluation. Only the updates of the previously reported result are computed progressively as new tuples arrive. (3) Fast query response. Once a change in the previously reported result is recognized, GPAC immediately sends the update to the user.

Unlike most of the existing algorithms for continuous spatio-temporal queries (e.g., see [3, 19, 30, 33, 37]) that are implemented as high-level functions at the application level, GPAC algorithms are encapsulated in physical pipelined query operators that can be part of a query execution plan. By having GPAC as pipelined query operators, we achieve three goals: (1) GPAC operators can be combined with other operators (e.g., distinct, aggregate, and join) to support online and progressive evaluation for a wide variety of continuous spatio-temporal queries. (2) Pushing GPAC operators deep in the query execution plan reduces the number of tuples in the query pipeline where GPAC operators act as filters to other operators. (3) Flexibility in the query optimizer where multiple candidate execution plans can be produced. In general, the contributions of this paper are summarized as follows:

1. We propose GPAC; a *generic* and *progressive* family of algorithms that achieves online, progressive, and fast response for continuous spatio-temporal queries over spatio-temporal streams.

2. We introduce two instances of GPAC for continuously evaluating spatio-temporal range queries and spatio-temporal k-nearest-neighbor queries.

3. We encapsulate GPAC algorithms into physical pipelined query operators that can be combined with other operators as part of a query execution plan.

4. We give experimental evidence, based on a real system implementation [20], that GPAC provides the query optimizer with a variety of pipelined query plans.

*This work was supported in part by the National Science Foundation under Grants IIS-0093116, IIS-0209120, and 0010044-CCR.

.

The rest of the paper is organized as follows: Section 2 highlights related work. In Section 3, we propose GPAC. Section 4 provides two instances of GPAC that behave as continuous range queries and k-nearest-neighbor queries. Encapsulation of GPAC into physical query operators is presented in Section 5. Section 6 provides an experimental study of GPAC. Finally, Section 7 concludes the paper.

2. RELATED WORK

Most of the existing continuous spatio-temporal query processing techniques assume that incoming data from mobile objects are materialized in secondary storage. Various external memory index structures are used, e.g., simple grid [8, 19, 26, 35], B-tree-like [13], R-tree-like [14, 16], and TPR-tree-like [17, 28, 29, 34] structures. However, issues of high arrival rates, infinite nature of data, and spatio-temporal streams are overlooked by these approaches. On the other side, numerous research efforts are devoted to stream query processing (e.g., see [1, 5, 22]). However, the spatial and temporal properties of data streams are not exploited.

Three approaches are investigated for continuously evaluating continuous spatio-temporal queries: (1) Reevaluation. A query has an additional temporal parameter (e.g., valid time [38]) or a spatial parameter (e.g., a valid region [37] or a safe period [8]) that indicates the region where the query answer is valid. Once a query is out of the temporal or spatial region, it needs to be reevaluated. (2) Caching the results. Previous results are cached either in the client side [30] or in the server side [15] and are used to prune the search for the new results. (3) Incremental evaluation. The query answer is limited to only the *positive* or *negative* updates of the previously reported answer [19, 35]. Positive (Negative) updates indicate that a certain object has to be added to (removed from) the previously reported answer. In GPAC, we utilize the third approach with incremental updates being handled through the query pipelines. In addition, GPAC *progressively* computes the incremental answer and sends it immediately to the user rather than bulk the updates and send them once as in [19, 35].

Our proposed *Generic Progressive Algorithm* (GPAC) distinguishes itself from the above approaches where it has the following unique properties: (1) GPAC goes beyond the idea of materializing incoming data into secondary storage. Instead, GPAC evaluates incoming spatio-temporal streams online. (2) GPAC is encapsulated in physical pipelined query operators and can be *extended* to a variety of continuous spatio-temporal queries. In addition, GPAC combines the advantages of other techniques, where (1) GPAC is an incremental approach, (2) GPAC outputs immediately any change to the query result once a change takes place, and (3) GPAC deals with stationary as well as mobile queries.

3. PROGRESSIVE EVALUATION OF CONTINUOUS SPATIO-TEMPORAL QUERIES

In this section, we introduce the *Generic Progressive Algorithm* (GPAC) for continuous spatio-temporal queries over spatio-temporal streams. GPAC is similar in spirit to generalized search tree indexes (e.g., GiST [11] and SP-GiST [2]), but GPAC is in the context of spatio-temporal query processing algorithms. GPAC is introduced as a general skeleton that can be adjusted through a set of methods to behave as various continuous spatio-temporal queries (e.g., continuous range queries and nearest-neighbor queries). In GPAC, each mobile query is bounded to one *focal* object. For example, if a moving object M submits a query Q that asks about its nearest police car, then M is considered the *focal* object of Q. Mobile objects and queries are required to send updates of their locations every T seconds. Failure to do so results in considering the mobile object or query as disconnected. As GPAC can be implemented either at the application level or as a physical query operator, the output of GPAC is sent either directly to the user or to the next query operator in the pipeline. Thus, throughout the rest of the paper, we use the terms *"user"* and *"next query operator"* as synonyms.

3.1 Main idea of GPAC

To overcome the massive size of incoming data streams, traditional data stream query processors (e.g., see [1, 5, 22]) employ the so-called *sliding-window queries*, where the query is limited to only the recently received tuples. Two types of sliding windows are distinguished, *time-based* window (e.g., a query is interested only in received tuples within the last hour) and *tuple-based* windows (e.g., a query is interested only in the last 100 tuples). In sliding window queries, once a tuple becomes old enough, it is expired (i.e., deleted) from the memory leaving its space to a more recent tuple. Thus, incoming streaming data follow a first-in-first-expire model.

Traditional sliding-window queries can support only (recent) historical queries. However, in mobile environments, most of the queries are concerned with the current state of data. Thus, in GPAC, we go beyond the ideas of time-based and tuple-based queries. Instead, GPAC employs a new kind of sliding window queries, in which we call, *predicate-based* window queries. In *predicate-based* window queries, an incoming data tuple is stored in memory only if it satisfies a query predicate. Once an object becomes out of the predicate, we expire (i.e., delete) that object from the memory. Thus, data tuples are expired out-of-order.

To support *predicate-based* window queries in GPAC, we store the tuples that satisfy the query predicate in a data structure termed $Q.Answer$. Then, for each newly incoming tuple P, GPAC performs two tests: Test I: Is $P \in Q.Answer$? Test II: Does P satisfy the query predicate?. Based on the results of the two tests, GPAC distinguishes among four cases:

- **Case I:** $P \in Q.Answer$ and P still satisfies the query predicate. As GPAC processes only the updates of the previously reported result, P will neither be processed nor will P be sent to the user.
- **Case II:** $P \in Q.Answer$, however, P does not qualify to be part of the answer anymore (i.e., P does not satisfy the query predicate anymore) . In this case, GPAC reports a *negative* update P^- to the user. The *negative* update indicates that P needs to be removed from the query answer and hence is discarded from the system.
- **Case III:** $P \notin Q.Answer$, however, P qualifies to be part of the current answer (i.e., P satisfies the query predicate currently). In this case, GPAC reports a *positive* update to the user. The *positive* update indicates that P needs to be added to the query answer.

Procedure Q.ReceiveTupleI(Tuple P)
Begin

1. If query Q is moving and P is the focal point
   (a) $Q.UpdateCriteriaI(P)$ (Figure 2)
   (b) **return**
2. if **Q.satisfy**(P) AND $P \notin$ **Q.Answer**
   (a) Add P to **Q.Answer**
   (b) Send the *Positive*update P to the user
   (c) If **Q.IsDynamic()**
   - $Q.UpdateCriteriaI(P)$ (Figure 2)
   (d) **return**
3. If (!**Q.satisfy**(P)) AND ($P \in$ **Q.Answer**)
   (a) Delete P from $Q.Answer$
   (b) Send the *Negative* tuple P^- to the user.

End

Figure 1: Pseudo code of skeleton of GPAC.

Procedure Q.UpdateCriteriaI(Tuple P)
Begin

1. **Q.Update**(P)
2. For all moving objects $M \in Q.Answer$
   - If NOT **Q.satisfy**(M)
     (a) Send the *Negative* output M^- to user.
     (b) Delete M from $Q.Answer$.

End

Figure 2: Updating query information in GPAC

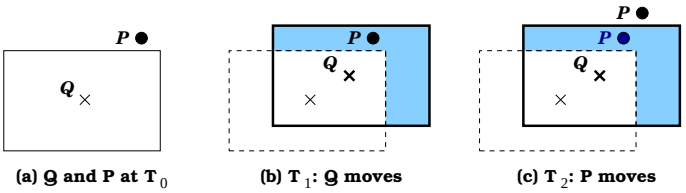


Figure 3: Uncertainty in moving queries.

- **Case IV:** $P \notin Q.Answer$ and P still does not qualify to be part of the current answer. In this case, P has no effect on Q. Thus, P will neither be processed nor will P be sent to the user.

Figures 1 and 2 give the pseudo code of the main idea of GPAC upon receiving a tuple P. Functions and variables written in **bold** font need to be implemented separately for each query type as will be addressed in Section 4. Initially, GPAC checks if P is the *focal* object of the moving query Q (Step 1 in Figure 1). If this is the case, we update the spatial region covered by Q. Based on the update, some tuples from $Q.Answer$ may be out of the new query spatial region. These tuples are deleted (expired) from $Q.Answer$ and corresponding *negative* updates are sent to the user or the next query operator (Step 2 in Figure 2).

If the newly incoming tuple P is not the query *focal* object (Step 2 in Figure 1), we check if P qualifies to be in the query answer (Test II). If this is the case, we check if P is part of the recently reported answer (Test I). In this case, we do not process or send P since P is still in the reported answer (Case I). However, if P is not part of the recently reported answer (Case III), we add P to $Q.Answer$ (Step 2a in Figure 1) and send P as a *positive* update to the user (Step 2b in Figure 1). Then, we update the query information (if needed) based on P's effect on the query spatial region (Step 2c in Figure 1). The predicate `Q.IsDynamic()` returns *"true"* if the query spatial area is changed as a result of P.

If the incoming tuple P does not qualify to be part of the answer, then we check if P is part of the recently reported answer (Step 3 in Figure 1). In this case (Case II), we delete P from the current answer (Step 3a in Figure 1) and report P as a *negative* update to the user (Step 3b in Figure 1). Notice that if P was not in the previously reported answer, we do not have to process or send P to the user (Case IV).

3.2 Uncertainty in GPAC

One of the goals of GPAC is to provide a fast and up-to-the-moment answer to continuous queries over spatio-temporal streams. However, this goal is hindered by the fact that spatio-temporal streams are not materialized on secondary storage. The basic GPAC algorithm stores only the tuples that satisfy the query predicate Q. Such implementation may result in having *uncertainty* areas in Q. We define the *uncertainty* area of a query Q as the spatial area that may contain potential moving objects that satisfy Q, with Q not being aware of contents of this area. Uncertainty areas in GPAC may result in erroneous query result. We distinguish among three cases for producing uncertainty areas within the basic GPAC framework:

1. New query. Initially, there are no outstanding queries in the system. Thus, continuously arrived data streams are neither processed nor stored. Once a query Q is submitted to the system, we cannot provide a fast answer to Q, simply because there is nothing currently being stored in the database. In this case, all the area covered by Q is considered an *uncertainty* area. Later on, moving objects update their locations and the answer of Q is progressively built.

2. Moving queries. Figure 3 gives example *uncertainty* areas that result from moving range queries. Figure 3a represents a snapshot at time T_0 where point P is outside the area of query Q. Thus, P is not physically stored in the database. At time T_1 (Figure 3b), Q is moved to cover a new spatial area. The shaded area in Q represents its *uncertainty* area. Although P is inside the new query region, P is not reported in the query answer, simply, because P is not stored in the database. At T_2 (Figure 3c), object P moves out of the query region. Thus, P is never reported at the query result, although it was physically inside the query region in the interval $[T_1, T_2]$.

3. Stationary queries. Figure 4 gives an example *uncertainty* area in stationary k-nearest-neighbor queries ($k = 2$). At time T_0 (Figure 4a), the query Q has P_1 and P_3 as its answer. P_2 is outside the query spatial region, thus P_2 is not stored in the database. At T_1 (Figure 4b), P_1 moves far from Q. Since Q is aware of P_1 and P_3 only, we extend the spatial region of Q to include the new location of P_1. Thus, an *uncertainty* area is produced. Notice that Q is unaware of P_2 since P_2 is not stored in the database. At T_2 (Fig-

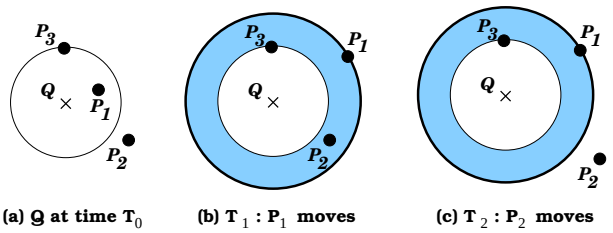


Figure 4: Uncertainty in static NN queries.

ure 4c), P_2 moves out of the new query region. Thus, P_2 never appears as an answer of Q, although P_2 should have been part of the answer in the time interval $[T_1, T_2]$.

Handling uncertainty in GPAC. GPAC does not handle the *uncertainty* area that results from newly submitted queries. Continuous queries are issued to run for hours and days. Thus, having a *warm-up* period for a few seconds does not affect neither the accuracy nor the efficiency of the query result. However, *uncertainty* areas that result from stationary or moving queries are crucial and are treated by GPAC. The following section gives the complete GPAC algorithm for handling uncertainty in stationery and moving queries.

3.3 Anticipating the Query Answer in GPAC

In this section, we modify the basic GPAC algorithm given in Section 3.1 to avoid having *uncertainty* areas in both stationary and moving queries. The main idea is to *anticipate* the change in the query spatial region and *cache* all moving objects that lie inside the anticipated area in an in-memory structure called $Q.Cache$. A *conservative* approach for determining the *anticipated* area is to expand the query region in all directions with the maximum possible distance that a moving object can cover between any two consecutive updates. Such *conservative* approach completely avoids uncertainty areas. Once a query changes its spatial region, we probe $Q.Cache$ for all objects that lie inside the new spatial region. Thus a fast answer of Q is retrieved. Notice that with the *conservative* approach, the change of the query spatial region is guaranteed to be completely inside the *anticipated* area. To realize GPAC with *caching*, we equip each query Q with the following: (1) A variable $Q.CacheArea$ that contains the boundary of the *anticipated* area. (2) The data structure $Q.Cache$ that keeps track of all moving objects within $Q.CacheArea$. (3) The function `Q.InCacheArea()` that takes an input tuple P and outputs *true* if P lies inside $Q.CacheArea$.

The *conservative* caching approach requires only the knowledge of the maximum object speed, which is typically available in moving object applications (e.g., moving cars in road network have limited speeds). This is in contrast to all validity region approaches (e.g., the safe region [27], the valid region [37], and the No-Action region [36]) that require the knowledge of the locations of other objects. This information is not available in our case since GPAC is aware only of objects that satisfy the query predicate. Thus, validity region approaches are not applicable in the case of spatio-temporal streams.

Figures 5 and 6 give the pseudo code of GPAC when *caching* is employed as a means to avoid *uncertainty* areas. The changes in the basic GPAC algorithm are limited to the following: (1) When the *focal* point of a moving query

Procedure Q.ReceiveTupleII(Tuple P)
Begin

1. If query Q is moving and P is the focal point
   (a) $Q.UpdateCriteriaII(P)$ (Figure 6)
   (b) **return**
2. if **Q.satisfy**(P) AND $P \notin$ **Q.Answer**
   (a) Add P to **Q.Answer**
   (b) Send the *Positive* update P to the user
   (c) If $P \in$ **Q.Cache**
   - delete P from **Q.Cache**

   (d) If **Q.IsDynamic()**
   - $Q.UpdateCriteriaII(P)$ (Figure 6)

   (e) **return**
3. If (!**Q.satisfy**(P)) AND ($P \in$ **Q.Answer**)
   (a) Delete P from **Q.Answer**
   (b) Send the *Negative* tuple P^- to the user
   (c) If **Q.InCacheArea**(P)
   - Insert P in **Q.Cache**

   (d) If **Q.IsDynamic()**
   - $Q.UpdateCriteriaII(P)$ (Figure 6)

   (e) **return**
4. If **Q.InCacheArea**(P)
   (a) If $P \notin$ **Q.Cache**
   - Insert P in **Q.Cache**

   (b) **return**
5. If $P \in$ **Q.Cache**
   - delete P from **Q.Cache**.

End

Figure 5: Pseudo code of GPAC with caching.

moves, we update the new $Q.CacheArea$. Then, we go over all the objects in $Q.Cache$ to determine whether any of them become part of the query answer (Step 2 in Figure 6). Also, for moving objects that are out of the new query region, we check whether they need to be moved into $Q.Cache$ or not (Step 3b in Figure 6). (2) When the input P is inside the query area but was not in the previously reported answer, we check if P is stored in $Q.Cache$. In this case, we delete P from $Q.Cache$ (Step 2c in Figure 5). (3) When the input P is not inside the query region but was in the old answer, we check if the new value of P lies in the query region. In this case, we add P to $Q.Cache$ (Step 3c in Figure 5). (4) If P is neither inside the query region nor in the previous query answer, we maintain the status of P with respect to the query region (Steps 4 and 5 of Figure 5).

3.4 Scalability of GPAC

In this section, we discuss the effect of large number of objects/queries and the size of the cache area on the scalability of GPAC.

Large number of objects. The infinite nature of data is a common problem to all traditional data streams where the incoming data is truly massive and is beyond the systems capabilities to store. GPAC avoids this problem by employing

Procedure Q.UpdateMovingQueryII(Tuple P)
Begin

1. Update **Q.Criteria** and **Q.CacheArea** based on P
2. For all moving objects M in **Q.Cache**
   - If **Q.satisfy**(M)
     (a) Move M from **Q.Cache** to **Q.Answer**
     (b) Send the *Positive* update M to the user
   - If NOT **Q.InCacheArea**(M)
     – Delete M from **Q.Cache**
3. For all moving objects $M \in$ in **Q.Answer**
   - If NOT **Q.satisfy**(M)
     (a) Send the *Negative* tuple M^- to the user
     (b) if **Q.InCacheArea**(M), move M from **Q.Answer** to **Q.Cache**, else, delete M from **Q.Answer**.

End

Figure 6: Updating query information in GPAC with caching

predicate-based windows where incoming tuples that do not satisfy any query predicate are discarded. Similarly, once a stored tuple does not satisfy the query predicate any more, that tuple is immediately expired from the system. Thus, stored tuples are limited only to those tuples that satisfy query predicates.

Large number of queries. GPAC algorithms and operators are concerned with evaluating one outstanding continuous query with an incoming spatio-temporal stream. Being encapsulated into physical operators provide the flexibility that GPAC can be plugged into any scalable framework that employs a *shared execution* paradigm as a means to achieve scalability, e.g., NiagaraCQ [7], PSoup [6], or SINA [19]. The main idea behind shared execution is to abstract the operation of evaluating a set of concurrent continuous queries into a join between objects and outstanding queries. GPAC can be plugged in these frameworks as the basic operator to join each query with the incoming objects.

Cache area. The cache area enlarges the query size and hence more input tuples need to be stored. However, this increase in size is limited and can be neglected in many cases. For example, consider a square range query with side length x. A *conservative* cache area would increase the side length to be $x + d$ where d is the maximum distance an object can travel between any two consecutive updates. The ratio of area increase would be $\frac{(x+d)^2-x^2}{x^2}$. A typical query region would be orders of magnitude of d, i.e., $x = md$. Thus, the ratio of increase is $\frac{2md^2+d^2}{m^2d^2} = \frac{2m+1}{m^2}$, which can be approximated to $\frac{2}{m}$. In a typical scenario, m can be in the order of tens, which results in a slight overhead in the query size. For example, consider a square range query with side length 2 miles that monitors the traffic in a downtown area. If objects are moving with speed 25 miles/hour while updating their locations every 30 seconds, then the maximum traveled distance for each object is $d = 1/8$. This will result in increasing the query area by only 12.5%. Similarly, for the same setting, a query about objects within 3 miles suffers only an 8.5% increase in size. Notice that the overhead in having a cache area is reduced by the increase in the area of the original query.

4. INSTANCES OF GPAC

In this section, we develop two instances of GPAC, namely, for continuous spatio-temporal range queries and continuous k-nearest-neighbor queries. Other instances of GPAC (e.g., reverse nearest-neighbor [3], group nearest-neighbor queries [24], and time-parameterized queries [32]) can be developed in a similar way.

4.1 Spatio-temporal Range Queries

$Q.Answer$ is represented by a hash table. $Q.Cache$ is represented as a linked list that is sorted on the distance from the moving object to the boundary of the query region. The functions $Q.satisfy()$ and $Q.InCacheArea()$ represent a test of object P inside the rectangular region of Q and the cache area, respectively. The function $Q.IsDynamic()$ always returns *false* for stationary queries and *true* for moving queries. This is because static range queries never change their spatial regions.

4.2 Spatio-temporal k-nearest-neighbor

$Q.Answer$ and $Q.Cache$ are represented by a linked list that is sorted on the distance from the moving object to the query focal point. The functions $Q.satisfy()$ and $Q.InCacheArea()$ represent a test of object P inside the circular region of Q and the cache area, respectively. The circular region has the focal point as its center and the distance to the furthest k point as its radius. The function $Q.IsDynamic()$ always returns *true* for both stationary and moving queries.

5. PIPELINED SPATIO-TEMPORAL QUERY OPERATORS

We encapsulate GPAC algorithms for continuous range queries and continuous k-nearest-neighbor queries into the pipelined query operators `GPAC-IN` and `GPAC-kNN`, respectively. The pipelined operators are implemented inside the PLACE (Pervasive Location-Aware Computing Environments) server [20, 21]. A typical SQL query submitted to the PLACE server may have the following form:

```
SELECT select_clause
FROM from_clause
WHERE where_clause
GPAC-IN in_clause
GPAC-kNN knn_clause
```

The *in_clause* may have one of the following two forms:

- Static range query (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) represent the top left and bottom right corners of the rectangular range query.
- Moving rectangular range query $('M', ID, xdist, ydist)$, where $'M'$ is a flag indicates that the query is moving, ID is the identifier of the query *focal* point, $xdist$ is the length of the query rectangle, and $ydist$ is the width of the query rectangle.

Similarly, the *knn_clause* may have one of the following two forms:

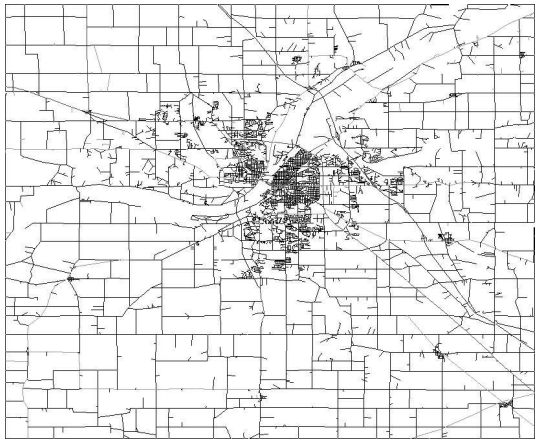


Figure 7: Greater Lafayette, Indiana, USA.

- Static kNN query (k, x, y), where k is the number of the neighbors to be maintained, and (x, y) is the center of the query point.

- Moving kNN query $('M', k, ID)$, where $'M'$ is a flag indicates that the query is moving, k is the number of neighbors to be maintained, and ID is the identifier of the query *focal* point.

As will be discussed in Section 6, pushing the operators `GPAC-IN` and `GPAC-kNN` to the bottom of the execution query plan always achieves the best performance. However, having the spatio-temporal operators at the bottom or at the middle of the query evaluation pipeline requires that all the above operators be equipped with special handling of *negative* tuples. The NILE query processor [10] handles *negative* tuples in pipelined operators as follows: *Selection* and *Join* operators handle *negative* tuples in the same way as *positive* tuples. The only difference is that the output will be in the form of a *negative* tuple. *Aggregates* update their aggregate functions by considering the received *negative* tuple. The *Distinct* operator reports a *negative* tuple at the output only if the corresponding *positive* tuple is in the recently reported result. For more details about handling the *negative* tuples in various query operators, the reader is referred to [9].

6. EXPERIMENTAL RESULTS

In this section, we give experimental evidence that encapsulating GPAC algorithms with appropriate cache size into physical pipelined query operators outperforms high level implementations. Mainly, the experiments in this section are divided into two categories:

- **Pipelined operators.** This set of experiments compare the high level implementation of GPAC with the encapsulation of GPAC algorithms in pipelined query operators.

- **Properties of GPAC.** In this set of experiments, we study some properties of GPAC, namely dealing with high data rates and various spatio-temporal selectivities.

All the results in this section are based on a real implementation of GPAC algorithms and operators inside our prototype database engine for spatio-temporal streams, PLACE [20, 21]. PLACE extends the Nile [10] streaming database management system to handle spatio-temporal streams. We run PLACE on Intel Pentium IV CPU 2.4GHz with 512MB RAM running Windows XP. Without loss of generality, all the presented experiments are conducted on stationary and moving continuous spatio-temporal queries. Similar results are achieved when employing continuous k-nearest-neighbor queries.

We use the *Network-based Generator of Moving Objects* [4] to generate a set of moving objects and moving queries in the form of spatio-temporal streams. The input to the generator is the road map of Greater Lafayatte (a city in the state of Indiana, USA) given in Figure 7. The output of the generator is a set of moving points that move on the road network of the given city. Moving objects can be cars, cyclists, pedestrians, etc. Any moving object can be a *focal* of a moving query. Unless mentioned otherwise, we generate 110K moving objects as follows: Initially, we generate 10K moving objects from the generator, then we run the generator for 1000 time units. At each time unit, we generate new 100 moving objects. Moving objects are required to report their locations every time unit T. Failure to do so results in disconnecting the moving object from the server.

Although, it is appealing to have a *conservative* cache, a large cache size may encounter high overhead while maintaining objects inside the cache area. Thus, for the rest of the experiments, we use the cache size 75% of the *conservative* cache area. In most cases, a cache size of 75% would have similar performance as that of the *conservative* cache. Notice that a *conservative* cache is designed for the most speedy moving object. Most likely, the query *focal* object is not that object of maximum speed.

6.1 GPAC Operators in a Pipelined Query Plan

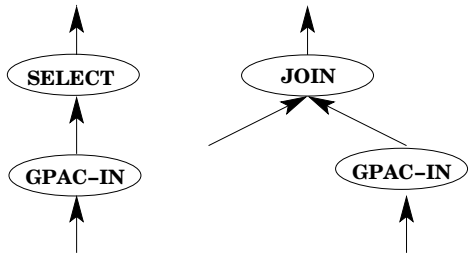
In this section, we compare the implementation of GPAC at the application level with the encapsulation of GPAC inside query operators.

6.1.1 Pipeline with a Selection Operator

Consider the query Q: *"Continuously report all trucks that are within MyArea"*. *MyArea* can be either a stationary or moving range query. A high level implementation of this query is to have only a selection operator that selects only the *"trucks"*. Then, a high level algorithm implementation would take the selection output and incrementally produce the query result. However, an encapsulation of GPAC into the `GPAC-IN` operator allows for more flexible plans. Figure 8a gives a query evaluation plan when pushing the `GPAC-IN` operator before the *selection* operator. The following is the SQL presentation of the query.

```
SELECT M.ObjectID
FROM MovingObjects M
WHERE M.type = "truck"
GPAC-IN MyArea
```

Figure 9 compares the high level implementation of the above query with pipelined `GPAC-IN` operators for both stationary and moving queries. The selectivity of the queries varies from 2% to 64%. The selectivity of the selection op-



(a) SELECTION **(b) JOIN**

Figure 8: Pipelined GPAC operators.

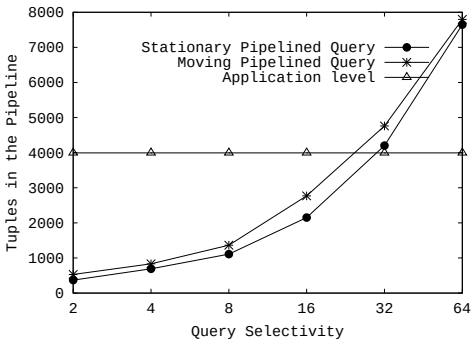


Figure 9: Pipelined operators with SELECT.

erator is 5%. Our measure of comparison is the number of tuples that go through the query evaluation pipeline. When GPAC is implemented at the application level, its performance is not affected by the query selectivity. However, when `GPAC-IN` is pushed before the *selection*, it acts as a filter for the query evaluation pipeline, thus, limiting the tuples through the pipeline to only the progressive updates. With `GPAC-IN` selectivity less than 32%, pushing `GPAC-IN` before the selection greatly affects the performance. However, with selectivity more than 32%, it would be better to have the `GPAC-IN` operator above the *selection* operator.

6.1.2 Pipeline with a Join Operator

In this section, we consider a more complex query plan that contains a *join* operator. Consider the query Q: *"Continuously report moving objects that belong to my favorite set of objects and that lie within MyArea"*. A high level implementation of GPAC would probe a streaming database engine to join all moving objects with my favorite set of objects. Then, the output of the join is sent to the GPAC algorithm for further processing. However, with the `GPAC-IN` operator, we can have a query evaluation plan as that of Figure 8b where the `GPAC-IN` operator is pushed below the *Join* operator. The SQL representation of the above query is as follows:

```
SELECT M.ObjectID
FROM MovingObjects M, MyFavoriteCars F
WHERE M.ObjectID = F.ObjectID
GPAC-IN MyArea
```

Figure 10 compares the high level implementation of the

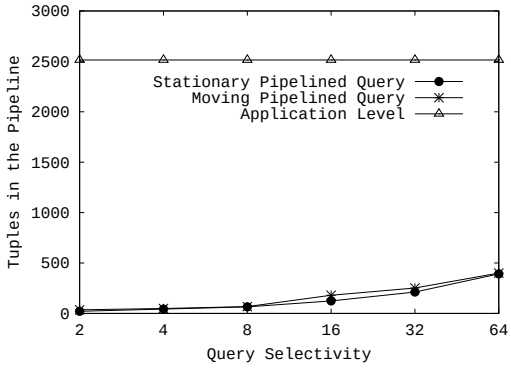


Figure 10: Pipelined operators with Join.

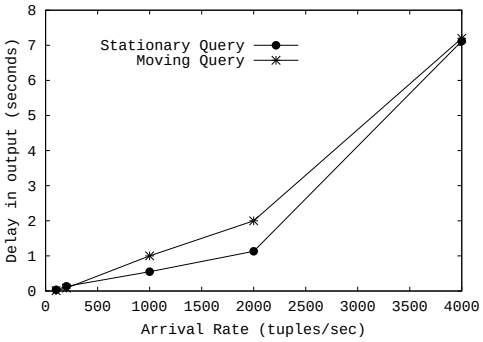


Figure 11: High arrival rates.

above query with the pipelined `GPAC-IN` operator for both stationary and moving queries. The selectivity of the queries varies from 2% to 64%. As in Figure 9, the selectivity of GPAC does not affect the performance if it is implemented in the application level. Unlike the case of *selection* operators, GPAC provides a dramatic increase in the performance (around an order of magnitude) when implemented as a pipelined operator. The main reason in this dramatic gain in performance is the high overhead incurred when evaluating the *join* operation. Thus, the `GPAC-IN` operator filters out the input tuples and limit the input to the join operator to only the incremental *positive* and *negative* updates.

6.2 Properties of GPAC

In this section, we study some properties of GPAC algorithms, namely, dealing with high rates of data arrival and the spatio-temporal query selectivity.

6.2.1 High Arrival Rates

Figure 11 gives the result of an experiment that deals with high arrival rates in GPAC for stationary and moving queries. Spatio-temporal data arrives exponentially with an arrival rate that varies from 100 tuples per second to 4000 tuples per second. Our measure is the average output delay. The output delay of a tuple P is the difference from the time that P enters the system to the time that P has an effect on the output result. As shown in Figure 11, GPAC algorithms can afford up to 2000 tuples per second with only one second in output delay.

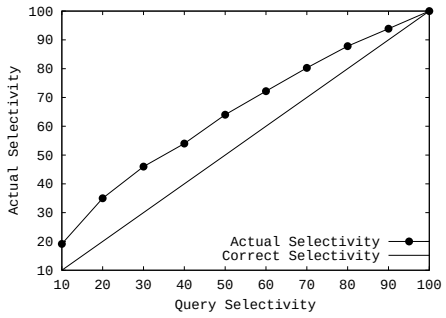


Figure 12: Query selectivity.

6.2.2 Query Selectivity due to Incremental Evaluation

Unlike the selectivity of traditional queries, the selectivity of spatio-temporal queries is more sophisticated. Figure 12 gives the result of an experiment that shows the selectivity of spatio-temporal queries. We run a continuous spatio-temporal query Q that should have a selectivity that varies from 10% to 100%. We call this selectivity as the *correct* selectivity where it is induced from the spatial area covered by Q. However, the *actual* selectivity of the spatio-temporal query is higher than its *correct* selectivity. The main reason is that in spatio-temporal queries, moving objects can go back and forth and report themselves in the query answer as multiple *positive* and *negative* tuples. Thus, it may happen that a query with a smaller area produces more output results than a query with a larger area. For example, consider a query that covers all the spatial area (i.e., selectivity 100%). Such a query would never output *negative* tuples. In addition, once all objects are inside the query area, no output will be produced due to the progressive property. Consider another query that has a slightly less area. Due to the area not covered by this query, it may happen that some tuples go out of the query region and produce *negative* tuples. Then, these tuples can move again inside the query area to produce a set of *positive* tuples. As a result, a query with smaller area may produce more output tuples.

7. CONCLUSION

In this paper, we introduced a new family of *Generic and Progressive Algorithms* (GPAC, for short) for continuous query evaluation over spatio-temporal data streams. GPAC is a general skeleton that can be tuned through a set of methods to behave as various continuous spatio-temporal queries. GPAC provides online, progressive, and fast response to continuous spatio-temporal queries. We described two versions of GPAC. The first version (with no *caching*) is simple to maintain, however, produces inaccurate answers. In the second version (with *caching*), we introduce the concept of *anticipation*, where the query answer can be anticipated beforehand and is cached in a *cache* structure. We show how to realize two types of continuous spatio-temporal queries form GPAC, namely, continuous range queries and continuous k-nearest-neighbor queries. Moreover, we encapsulate GPAC algorithms into physical pipelined query operators. Pipelined operators are combined with traditional operators (e.g., selection and join) to provide online, progressive, and fast response of a wide variety of continuous spatio-temporal queries. Experimental results determine the appropriate size of *caching* in GPAC. In addition, we show that encapsulating GPAC into pipelined query operators is an order of magnitude better than implementing GPAC at the application level. Also, GPAC is stable with high data arrival rates. For arrival rate of 2000 tuples per second, GPAC results in only one second delay in the query answer.

8. REFERENCES

[1] Daniel J. Abadi. et. al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[2] Walid G. Aref and Ihab F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Info. Systems, JIIS*, 17(2–3):215–240, 2001.

[3] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, 2002.

[4] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2), 2002.

[5] Sirish Chandrasekaran. et. al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[6] Sirish Chandrasekaran and Michael J. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[7] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2000.

[8] Bugra Gedik and Ling Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2004.

[9] Moustafa A. Hammad, Thanaa M. Ghanem, Walid G. Aref, Ahmed K. Elmagarmid, and Mohamed F. Mokbel. Efficient execution of sliding-window queries over data streams. Technical Report CSD-03-035, Department of Computer Science, Purdue University, 2004.

[10] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann C. Catlin, Ahmed K. Elmagarmid, Mohamed Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette Marzouk, and Xiaopeng Xiong. Nile: A Query Processing Engine for Data Streams (Demo). In *Proceedings of the International Conference on Data Engineering, ICDE*, 2004.

[11] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1995.

[12] Glenn S. Iwerks, Hanan Samet, and Ken Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *Proceedings of the International Conference on Very*

Large Data Bases, VLDB, 2003.

[13] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2004.

[14] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, 2002.

[15] Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic Queries over Mobile Objects. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2002.

[16] Mong-Li Lee, Wynne Hsu, Christian S. Jensen, and Keng Lik Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

[17] Bin Lin and Jianwen Su. On Bulk Loading TPR-Tree. In *Mobile Data Management, MDM*, 2004.

[18] Mohamed F. Mokbel, Walid G. Aref, Susanne E. Hambrusch, and Sunil Prabhakar. Towards Scalable Location-aware Services: Requirements and Research Issues. In *Proceedings of the ACM workshop on Advances in Geographic Information Systems, ACM GIS*, 2003.

[19] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2004.

[20] Mohamed F. Mokbel, Xiaopeng Xiong, Walid G. Aref, Susanne Hambrusch, Sunil Prabhakar, and Moustafa Hammad. PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams (Demo). In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2004.

[21] Mohamed F. Mokbel, Xiaopeng Xiong, Moustafa A. Hammad, and Walid G. Aref. Continuous Query Processing of Spatio-temporal Data Streams in PLACE. In *STDBM*, 2004.

[22] Rajeev Motwani. et. al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proceedings of the International Conference of Innovative Innovative Data Systems Research, CIDR*, 2003.

[23] Tamer Nadeem, Sasan Dashtinezhad, Chunyuan Liao, and Liviu Iftode. TrafficView: A Scalable Traffic Monitoring System. In *Mobile Data Management, MDM*, 2004.

[24] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group Nearest Neighbor Queries. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2004.

[25] Hyun Kyoo Park, Jin Hyun Son, and Myoung-Ho Kim. An Efficient Spatiotemporal Indexing Method for Moving Objects in Mobile Communication Environments. In *Mobile Data Management, MDM*, 2003.

[26] Jignesh M. Patel, Yun Chen, and V. Prasad Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2004.

[27] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Trans. on Computers*, 51(10), 2002.

[28] Simonas Saltenis and Christian S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2002.

[29] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2000.

[30] Zhexuan Song and Nick Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, 2001.

[31] Zhexuan Song and Nick Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. In *Mobile Data Management, MDM*, 2003.

[32] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2002.

[33] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous Nearest Neighbor Search. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2002.

[34] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2003.

[35] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2005. To Appear.

[36] Xiaopeng Xiong, Mohamed F. Mokbel, Walid G. Aref, Susanne Hambrusch, and Sunil Prabhakar. Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 2004.

[37] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based Spatial Queries. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2003.

[38] Baihua Zheng and Dik Lun Lee. Semantic Caching in Location-Dependent Query Processing. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, 2001.