4-2012

Scalable activity-travel pattern monitoring framework for large-scale city environment

Youngki LEE
Singapore Management University, YOUNGKILEE@smu.edu.sg

Sang Jeong LEE

Byoungjip KIM

Jungwoo KIM

Yunseok Rhee

See next page for additional authors

DOI: https://doi.org/10.1109/TMC.2011.113

Citation

LEE, Youngki; LEE, Sang Jeong; KIM, Byoungjip; KIM, Jungwoo; Rhee, Yunseok; and SONG, Junehwa. Scalable activity-travel pattern monitoring framework for large-scale city environment. (2012). *IEEE Transactions on Mobile Computing*. 11, (4), 644-662. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/2080

Author
Youngki LEE, Sang Jeong LEE, Byoungjip KIM, Jungwoo KIM, Yunseok Rhee, and Junehwa SONG

Published in IEEE Transactions on Mobile Computing, 2012 April, Volume 11, Issue 4, Pages 644-662
https://doi.org/10.1109/TMC.2011.113

Scalable Activity-Travel Pattern Monitoring Framework for Large-Scale City Environment

Youngki Lee, SangJeong Lee, Byoungjip Kim, Jungwoo Kim, Yunseok Rhee, and Junehwa Song, *Member*, *IEEE*

Abstract—In this paper, we introduce *Activity Travel Pattern* (ATP) monitoring in a large-scale city environment. ATP represents where city residents and vehicles stay and how they travel around in a complex megacity. Monitoring ATP will incubate new types of value-added services such as predictive mobile advertisement, demand forecasting for urban stores, and adaptive transportation scheduling. To enable ATP monitoring, we develop ActraMon, a high-performanceATP monitoring framework. As a first step, ActraMon provides a simple but effective computational model of ATP and a declarative query language facilitating effective specification of various ATP monitoring queries. More important, ActraMon employs the shared staging architecture and highly efficient processing techniques, which address the scalability challenges caused by massive location updates, a number of ATP monitoring queries and processing complexity of ATP monitoring. Finally, we demonstrate the extensive performance study of ActraMon using realistic city-wide ATP workloads.

Index Terms—Activity-travel pattern (ATP), monitoring, location data processing, scalable architecture, large-scale, city.

1 Introduction

Understanding diverse aspects of complicated modern cities has long been a pivotal issue. *Activity-Travel Pattern* (ATP) has been used as one of the most useful tools to define and understand city residents' everyday lives in domains such as urban planning, geography, and transportation [1], [2], [4], [32]. ATP represents where people and vehicles are located, how they move in urban areas, and which activities they do at the places in certain patterns. As a city encompasses more people, vehicles, spaces and services, understanding ATP becomes more useful yet challenging. Because of its practical importance, governmental organizations have periodically observed and studied ATPs in terms of space-time use of citizens over major cities [22], [24], [25]. These observations have been performed through manual questionnaires and annotations, requiring huge amounts of time, labor, and expense.

Advances in mobile and ubiquitous computing technologies open up the opportunity for real-time *monitoring* of ATP. Real-time monitoring will significantly enhance the scale and timeliness of the ATP observation compared to conventional methods. More importantly, it will incubate a number of new advanced ATP-based urban services. For instance, mobile advertisement services will become far more intelligent and effective in targeting and appealing to potential customers by observing their ATPs. Also, there are many other applications such as adaptive and fine-granule scheduling of public transportations, demand forecasting of urban retail stores, and dynamic planning and management of urban districts. To effectively support diverse ATP-based services, an efficient ATP monitoring framework is compelling.

In this paper, we propose *ActraMon*, a high-performance ATP monitoring framework for large-scale city environments. ActraMon aims at providing an infrastructural support for a number of ATP-based services. (See Fig. 1) It helps service providers easily launch new services without building their own monitoring facilities. ActraMon monitors location data continuously updated from city-wide moving objects and evaluates ATP monitoring queries submitted by the service providers. Then, it detects ATP instances matching to the queries in a scalable manner. The services use the matching results to provide personalized situation-aware services (e.g., ATP-based mobile advertisement) or further aggregate them for collective observation of city dynamics (e.g., for dynamic scheduling of public transportations).

Developing an ATP monitoring framework is an important but challenging problem. The major challenge lies in developing efficient and scalable processing technologies to address the complexity in ATP monitoring semantics and the massive scale of location updates and ATP monitoring queries. First, ATP monitoring demands highly complicated processing, since it observes long-term behavior of the whole moving objects collectively spanning multiple geographic regions. Its time duration often becomes several hours or days, and the regions of interest are possibly distributed over the whole city domain. Second, an ATP monitoring framework is confronted with large-scale workloads. The framework should handle high-rate location updates from millions of people and vehicles. It should also concurrently evaluate more than thousands of city-wide ATP monitoring queries. Furthermore, the queries are mostly required to respond in real-time; for

- *Y. Lee, S. Lee, B. Kim, J. Kim, and J. Song are with the Department of Computer Science, KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Korea.*
  E-mail: {youngki, peterlee, bjkim, jwkim, junesong}@nclab.kaist.ac.kr.
- *Y. Rhee is with the School of Electronics and Information Engineering, HUFS, 89 Wangsan-ri, Mohyeon Yongin-si, Gyeonggi-do 449-791, Korea.*
  E-mail: rheeys@hufs.ac.kr.

Manuscript received 5 Nov. 2009; revised 14 Feb. 2011; accepted 4 Mar. 2011; published online 23 June 2011.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-2009-11-0484.
Digital Object Identifier no. 10.1109/TMC.2011.113.

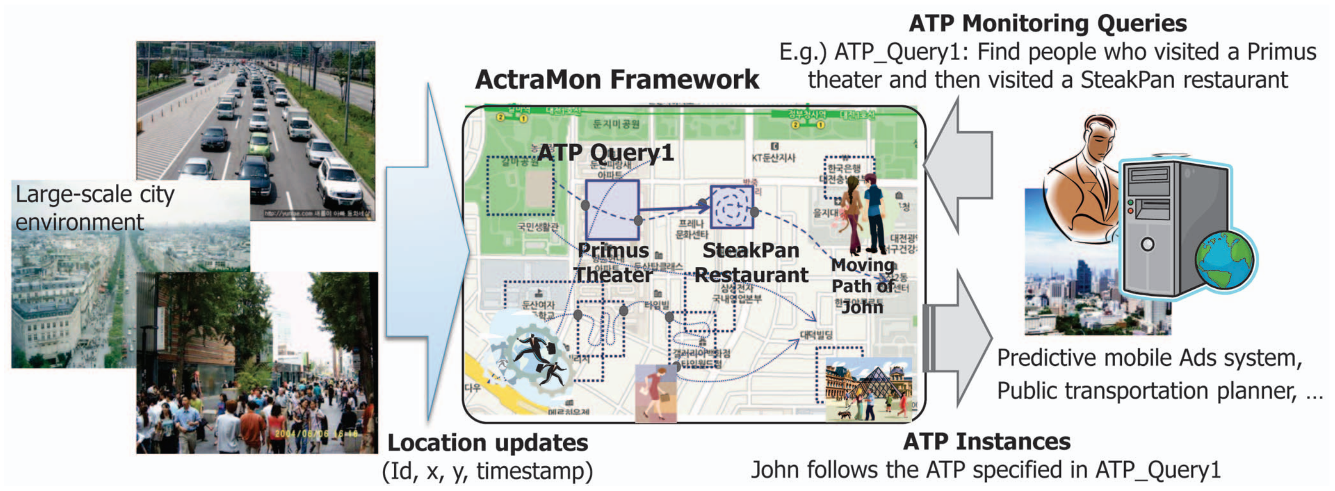

ATP Monitoring Queries
E.g.) ATP_Query1: Find people who visited a Primus theater and then visited a SteakPan restaurant
ActraMon Framework
ATP Query1
Primus Theater
SteakPan Restaurant
Moving Path of John
Large-scale city environment
Predictive mobile Ads system, Public transportation planner, ...
Location updates
(Id, x, y, timestamp)
ATP Instances
John follows the ATP specified in ATP_Query1

Fig. 1. ActraMon framework and environment.

instance, late-delivered coupons are usually worthless in ATP-based mobile advertisements.

To the best of our knowledge, our work is the first attempt to introduce the ATP monitoring in a large-scale city environment. Previous research on ATP in geography and transportation have mainly focused on developing logical models of ATP [1], [2], [4] and analyzing survey data statistically [24]. Our work tries to computationally model and automatically monitor city-wide ATPs, addressing an interdisciplinary research problem.

In terms of processing technologies, ATP monitoring is broadly related to location monitoring systems, trajectory-based systems, data stream management systems, and event processing systems. Recently, several systems such as SINA [7], MobiEyes [8], and MQM [9] have been proposed to enable real-time monitoring of moving objects. Also, data stream management systems [11], [12], [13] are proposed for general purpose data monitoring. However, their query semantics are mostly limited to range queries, k-Nearest Neighbor queries, and other relational queries. Thus, ATPs such as sequencing of multiple primitive activities can hardly be specified and processed efficiently by such monitoring systems. Event processing systems [18], [20] support various compositions; however, they do not efficiently handle massive location updates and a large number of monitoring queries. Several trajectory-based systems [5], [6] support queries over long-term behavior of an individual object. However, they aim at processing snapshot queries over stored trajectories but do not support real-time monitoring efficiently.

To support a city-wide ATP monitoring, we abstract ATP in two levels, i.e., *primitive* and *composite activities* (details in Section 3) and provide ATP monitoring query language based on the abstraction. This layered abstraction facilitates to capture diverse ATPs of interest and attain global understanding of the dynamics of a large-scale city, involving a huge number of residents, urban spaces, and facilities.

We design and develop a shared staging architecture to process ATP monitoring queries in a highly efficient and scalable manner. The architecture achieves the high performance in two aspects. (See Fig. 2)

First, it separates ATP monitoring into the detection of primitive activities and the composition of the detected activities. The separation of two processing stages allows ActraMon to effectively deal with the compound technical challenges of ATP monitoring, i.e., massive data processing and complex pattern composition. More important, the architecture performs efficient filtering of massive location updates at the early stage of the processing. The early filtering of insignificant updates eliminates a number of complex, unnecessary activity compositions. For this, ATP monitoring queries are fragmented into activity detection and composition operators and dispatched to proper stages, respectively.

Second, we develop novel shared processing techniques for each processing stage. Especially, we develop a shared activity detector to handle a number of activity detection operators and a shared activity composer to handle all types of activity composition operators. Each shared processor is equipped with shared data structures (*Activity Border Index* (ABI) and *Activity-centric Composable Queue* (ACQ)) and a localized processing method to efficiently process all registered operators with a single operation onto a small and critical part of the data structures. In addition, the processing is performed in an incremental manner for efficiency; the shared processors store the previous state of the computation and start the computation from the stored state. In this way, the shared processors significantly reduce processing overhead as well as memory consumption. The performance gain is studied in detail through the extensive experiments in Section 5.

The contribution of this paper is as follows: We first introduce ATP monitoring in a large-scale city environment. Then, we propose ActraMon, a high-performance ATP monitoring framework, which includes the computational

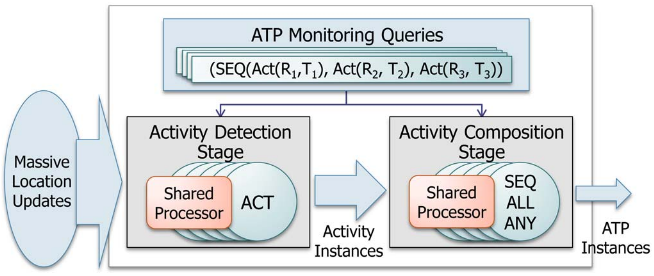


Fig. 2. Shared staging architecture of ActraMon.

model of ATPs and the novel shared staging architecture. Most important, we develop highly efficient and scalable processing techniques for city-scale ATP monitoring as a major contribution. They effectively address the scalability challenges caused by massive workloads and processing complexity. Finally, we conduct extensive performance studies to show the scalability and robustness of the techniques. The experimental results show that the techniques are highly scalable in terms of the number of ATP queries and moving objects, compared to existing state-of-the-art processing techniques [18], [27].

The rest of this paper is organized as follows: We first present related work in Section 2. In Section 3, we describe the computational model and language for ATP monitoring. Section 4 presents the shared staging architecture, and activity detection and composition techniques. We report the performance study in Section 5 and conclude the paper in Section 6.

2 Related Work

Understanding ATP with the intention of planning urban services and infrastructures has long been a pivotal issue in domains like geography, urban planning, and transportation [1], [2], [3], [4], [32]. It has been applied to many practical applications such as travel demand estimation and transportation planning [24]. This section reviews various computational technologies related to the proposed ATP monitoring framework. More comprehensive review of the related work can be found in [29, Section 2].

Several location-based systems have been proposed for real-time processing of continuous location updates of moving objects [7], [10], [14]. They provide continuous query semantics for effective monitoring and efficient processing techniques to deal with massive location updates and a number of queries. SINA [7] proposes a spatial join-based approach, and MAI [14] adopts motion-sensitive bounding box approach. Although they provide useful techniques to optimize location data processing, it is difficult to directly apply them to ATP monitoring; they do not deal with semantics and techniques for long-term transition behavior.

Recently in mobile computing domain, MobiEyes [8] and MQM [9] propose techniques for scalable computation of continuous range queries over location updates from mobile clients. They divide the whole monitoring domain into multiple cells [8] or regions [9] and offload partial processing to mobile clients. Although the techniques support simple range query semantics only, the idea may be applied to ATP monitoring. However, the detailed techniques should be redesigned for ATP monitoring since an ATP monitoring query is associated with multiple regions at the same time and requires to maintain long term states to generate final results.

Trajectory processing technologies [5], [6] can be considered for ATP monitoring, in the sense that both deal with the successive behaviors of moving objects. For efficient access of trajectories, they proposed trajectory indices, which are designed to effectively preserve the spatial proximity of trajectory data. Trajectory processing and ATP monitoring have several major differences. First, trajectory queries and indices are mainly designed to find the offline trajectory of a single object. Inherently, they can hardly support continuous real-time monitoring. For continuous monitoring, the indices built on the huge trajectory data need to be updated frequently by each location update, resulting in serious performance overhead.

Regarding the monitoring of continuous location updates, it would be possible to consider using data stream management systems such as Aurora [11], TelegraphCQ [12], and STREAM [13] and other related techniques [42]. Designed for generic data streams, these systems provide relational query models and a set of query processing techniques such as query planning, operator scheduling, and load shedding. The unique semantics of ATP such as activity detection and composition, however, is difficult to be efficiently specified and processed with the relational model and the proposed techniques.

Various indexing techniques have been proposed to handle various types of spatial and spatio-temporal queries [5], [6], [15], [23], [27]. Such techniques could be employed for the activity detection of ActraMon. However, the direct application of such indices is highly likely to cause inefficiency due to unique processing requirements of the activity detection; it requires continuous monitoring of spatio-temporal behavior of numerous moving objects with respect to a large number of activity regions and time conditions. More specific, diverse spatial query indexing techniques [15], [23], [27] have been proposed for efficient processing of multiple continuous range queries. Based on R-tree and grid structures, for instance, they quickly evaluate a number of range conditions in a shared manner. However, such spatial indices entail heavy postprocessing to trace long-term temporal behaviors over a set of regions. Also, several spatio-temporal indexing techniques [5], [6] have been proposed to facilitate trajectory query processing which retrieves offline trajectories for individual moving objects. However, as discussed above, they can hardly be applied to continuous activity detection processing over massive location updates from numerous moving objects. Hardly supported by existing indices, ActraMon develops a novel index structure, Activity Border Index, which addresses the unique requirements of the activity detection processing.

There has been some recent work on spatial alarms and location reminders in database, spatial data mining, distributed systems, and HCI domains [34], [35], [36]. They propose techniques to alarm users if they go into regions of interests by monitoring their location data. The techniques might be related to the activity detection of ActraMon, which also evaluate spatio-temporal conditions over location updates. Liu and colleagues recently propose a technique to reduce location updates utilizing safe region concept [35]; it helps servers to take less amount of data as input thereby achieving processing efficiency. Also, it distributes the intensive safe-region calculation to clients and imposes a little overhead to the server. Additionally, there has been work to further optimize location sensing cost in a mobile client using safe-distance concept [36]. The client-side techniques can be a complement to the location processing of ActraMon and help it achieve even higher level of scalability. Also, a server-side technique has been proposed to improve processing efficiency to handle multiple spatial alarms [36]. The technique utilizes Voronoi regions overlaid on top of grid cells to efficiently handle sparse and nonoverlapping spatial regions. However, activity regions are likely to be dense and overlapped with each other especially in downtown areas. Also, the technique requires heavy postprocessing as like conventional query-indexing techniques to evaluate temporal conditions for activity detection.

For the detection of complex patterns from primitive events, several event-based systems and techniques [18], [19], [20], [33] have been proposed in diverse application domains such as logistics and surveillance. SASE [18] and Cayuga [19] develop an efficient event composition technique extending Nondeterministic Finite Automata (NFA). Such event systems have rarely considered massive location data continuously updated from a number of moving objects. Also, it is highly challenging to detect composite ATP instances directly from raw location data, not from primitive events. To address the challenge, ActraMon employs the staging architecture; it performs activity detection and composition separately.

Regarding activity instances as primitive events, it may be possible to employ existing event processing systems for the activity composition. However, they also have rarely considered a large number of composition queries, which cause major performance challenges in large-scale ATP monitoring [16]. The processing time would increase significantly with the number of registered queries. Moreover, they consume considerable storage spaces to hold intermediary states separately for every single query, which makes in-memory processing for real-time ATP monitoring difficult. ActraMon, on the other hand, takes a shared processing approach based on *Activity-centric Composable Queue* that enables simultaneous evaluation of multiple ATP queries in a shared manner.

3 Activity-Travel Pattern Monitoring

ATP monitoring is useful to collectively observe population's city-wide behavior for diverse purposes. Consider a mobile advertising service. Issuing advertisements or coupons based on the current location of target customers is intended to be timely but in reality, not that useful. For instance, not every pedestrian walking close by a restaurant wants to have a meal there, nor cares about its lunch menu. Such conventional location-based advertisements would be ineffective and may rather give a bad impression, even recognized as annoying and unpleasant spam.

Observing the activity and travel patterns of city residents enables us to better identify and target the potential customers who would indeed benefit from timely and proactive advertisements [41]. For example, a franchise wine bar, leveraging the observation of activity-travel patterns, can target people who have stayed in nearby theaters for about two hours and visited Italian or French restaurants for about an hour. The wine bar then assumes that they are highly likely having a date at the moment and would be interested in a good wine bar.

ATP monitoring also enables developing more futuristic applications such as an adaptive public transportation planning. It captures occasional events such as festivals, sales, and demonstrations which would change population flows, and helps bus companies rearrange routes and schedules on-the-fly. For those purposes, we can install ATP queries on movements of interest over a city, and observe the occurrences in real time. We expect that a city planner or administrator will usually advise on the queries based on her interests and domain knowledge. The queries can be also crafted with the help of offline data mining systems running over a city. Meanwhile, we also expect that real-time ATP monitoring can facilitate complementarily the offline mining process on a tremendous amount of macroscopic city data.

3.1 Computational Model

Most work on ATP in the areas of geography, urban planning, and transportation has modeled activities at diverse urban spaces in terms of the space-time use of residents [24], [25], [32]. Based on the space-time-based models, they attempt to understand which activities the residents conduct, where, when, for how long, with whom, and which transportation mode they use between activities [1], [2] in a city. The idea behind the abstraction is that an urban space is usually designated by its unique service or an activity associated with it, e.g., a theatre for watching movie, a restaurant for dining-out, a hospital for healthcare, a school for education. People mostly stay in a space for a certain amount of time to perform a certain activity or to attain a certain service. Thus, an urban space and the duration of a stay have usually been taken as the basic units of observing urban activities, while concealing the details of the activities occurring within the space.

The computational model of ATPs is developed with a two-level approach; it first specifies *primitive activities* over location data streams from moving objects, and ATPs as the *compositions* of the primitive activities. The model provides necessary semantics for ATP monitoring. In addition, it provides an opportunity to develop a scalable and efficient processing technique by dividing complex ATP monitoring into two separate levels.

Primitive activities are modeled with the space-time abstraction. From massive location updates, we abstract the stay of a moving object within a specific urban space,R, for a certain time duration, t, where $T_{min} < t < T_{max}$, as a primitive activity. For the abstraction model, we assume that a location is uniquely defined in a given n-dimensional coordinate system, e.g., (x,y,z) in 3D, and a space is also specified as a range for each dimension. Thus, the model distinguishes multiple spaces in different levels of a building as well as movements between them.

Recent research demonstrates that location data are one of the most useful classifier features in macroscopic activity inference [22], [31]. The authors of [22] investigate the inference accuracy with respect to the activity taxonomy provided by American Time-Use Survey. When a location is combined with time-of-day, the accuracy rises up to 70 percent for coarse-granule (Tier 1) activities.

The composition of primitive activities enables the expression and monitoring of diverse, complex ATPs effectively. For the effective composition, the model provides several operations such as sequencing, grouping, selection, and negation of primitive activities. For example, a dating pattern can be presumed as the sequence of two primitive activities, i.e., "staying at a theater for two hours" and "visiting a restaurant for an hour." The activities may be grouped when the sequential order does not matter, e.g., visiting nearby stores in a shopping mall. Also, negation can be used to exclude people who have already visited another wine bar in the previous wine bar scenario.

There are many possible ways to model ATPs depending on their own purposes. The main purpose of the two-level approach is to provide a simple, practical means to attain

TABLE 1
Language Constructs

	Operators	Description	Parameters	Formal Definition
Primitive Activity	**ACT**(R, T_{min}, T_{max})	- Detecting activity instances based on **the space-time abstraction**	- R is an urban space where an activity occurs - T_{min} is a minimal stay time for an activity - T_{max} is a maximal stay time for an activity	For a location stream, *loc_strm* = <*loc*$_1$, …, *loc*$_j$, …, *loc*$_k$, …>, the operator generates activity instance stream, *act_strm*, a series of activity instances, act(o_{id} = $loc_k.o_{id}$, a_{id}, t_s, t_e), where $t_s \leftarrow loc_j.t$ such that $loc_{j-1}.coord \notin R$ and $loc_j.coord \in R$ $t_e \leftarrow loc_k.t$ such that $loc_{k-1}.coord \in R$ and $loc_k.coord \notin R$ for smallest k > j, and $T_{min} < t_e - t_s < T_{max}$
Composite Activity (ATP)	**SEQ**(ACT_{P1}, …, ACT_{Pn}, T_{tr}, WITHOUT({ACT_{Fk}, …}))	- Detecting **a time-ordered sequence** of multiple activities of an object within a given transition time	- ACT_{P1}, …, ACT_{Pn}, and ACT_{Fk}'s are a primitive activity - T_{tr} is a maximal transition time to complete ACT_{P1}, …, and ACT_{Pn} in the order - WITHOUT({ACT_{Fk}, …}) is an optional clause, specifying that ACT_{Fk}'s should not occur between ACT_{P1} and ACT_{Pn}	For an activity instance stream, *act_strm*, the operator generates ATP instances, atp(oid = $act_{Mn}.o_{id}$, c_{id}, t_s, t_e, (act_{M1}, ..., act_{Mn})), where $act_{Mk}.a_{id} = ACT_{Pk}.a_{id}$ for all $1 \le k \le n$, $act_{Mk-1}.t_e \le act_{Mk}.t_s$ for all $1 < k \le n$, $t_s \leftarrow act_{M1}.t_s$, $t_e \leftarrow act_{Mn}.t_e$, $t_e - t_s < T_{tr}$, and $act_i.a_{id} \notin \{ACT_F.a_{id}, …\}$ for all act_i such that $t_s < act_i.t_s \le act_i.t_e < t_e$
	ALL(ACT_{P1}, …, ACT_{Pn}, T_{wd}, WITHOUT({ACT_{Fk}, …}))	- Detecting **an unordered occurrence** of multiple activities of an object within a given time window	- ACT_{P1}, …, ACT_{Pn}, and ACT_{Fk}'s are a primitive activity - T_{wd} is a maximal time window to complete ACT_{P1}, …, and ACT_{Pn} - WITHOUT({ACT_{Fk}, …}) is an optional clause, specifying that ACT_{Fk}'s should not occur while ACT_{P1}, ..., and ACT_{Pn} occur	For an activity instance stream, *act_strm*, the operator generates ATP instances, atp(oid = $act_{Mn}.o_{id}$, c_{id}, t_s, t_e, (act_{M1}, ..., act_{Mn})), where $act_{Mk}.a_{id} = ACT_{Pk}.a_{id}$ for all $1 \le k \le n$, $t_s \leftarrow \min(\{act_{Mk}.t_s\})$, $t_e \leftarrow \max(\{act_{Mk}.t_e\})$, $t_e - t_s < T_{wd}$ and $act_i.a_{id} \notin \{ACT_F.a_{id}, …\}$ for all act_i such that $t_s < act_i.t_s \le act_i.t_e < t_e$
	ANY(ACT_{P1}, …, ACT_{Pn})	- Detecting **an occurrence** of an activity among multiple activities of an object	- ACT_{P1} , ..., and ACT_{Pn} are a primitive activity	For an activity instance stream, *act_strm*, of a moving object, the operator generates atp(oid = $act_{Mn}.o_{id}$, c_{id}, t_s, t_e, (act_M)), where $act_M.a_{id} \in \{ACT_{Pk}.a_{id}\}$ for all $1 \le k \le n$

macroscopic understanding of the collective behaviors of residents in a large-scale city.

3.2 ATP Query Language

ActraMon provides ATP query language based on the computational model. Table 1 shows the language constructs to specify ATP monitoring queries.

First, a primitive activity with space and time can be specified using $\mathbf{ACT}(R, T_{min}, T_{max})$ operator. It collectively observes all moving objects and detects the ones satisfying its spatio-temporal conditions. *Activity instances* are generated as its output when an object ends the activity by going out of the region. The following queries, i.e., A1 and A2, are examples queries of the ACT operators.

A1. $\mathbf{ACT}(R_{theater\_A}, 1.5h, 2.5h)$
A2. $\mathbf{ACT}(R_{GUCCI}, 10min, 60min)$

A1 specifies a query to detect people who have been "staying at a theater_A for about two hours." Similarly, A2 expresses a query to detect customers who "have shopped at a GUCCI shop." In practice, R is specified by a query issuer, e.g., using a map interface which enables her to designate a space of interest on a city map.

The composition operators are used to specify diverse ATP monitoring queries. First, the SEQ operator specifies a time-ordered sequence of multiple activities. For example, a query C1 as below, specifies a probable dating patterns, i.e., visiting a theater_A, a GUCCI shop and a restaurant_B in a sequence. Also, an example query on population flows for public transportation planning can be specified as C2.

C1. $\mathbf{SEQ}(\mathrm{ACT}(R_{theater\_A}, 1.5h, 2.5h),$
$\mathrm{ACT}(R_{GUCCI}, 10min, 60min),$
$\mathrm{ACT}(R_{restaurant\_B}, 0.5h, 1.5h), 6h)$

C2. $\mathbf{SEQ}(\mathrm{ACT}(R_{office\_area\_A}, 4h, 8h),$
$\mathrm{ACT}(R_{shopping\_district\_B}, 1h, 3h), 12h)$

The ALL operator expresses an unordered occurrence of multiple activities within a given time duration. For example, the marketing manager of a luxury shop may seek to identify brand shoppers who have visited all the neighbor GUCCI, PRADA, and CHANEL shops as below.

C3. $\mathbf{ALL}(\mathrm{ACT}(R_{GUCCI}, 10min, 60min),$
$\mathrm{ACT}(R_{PRADA}, 10min, 60min),$
$\mathrm{ACT}(R_{CHANEL}, 10min, 60min), 2h)$

Finally, the ANY operator provides a way to express the occurrence of an activity among multiple candidate activities. Note that the WITHOUT clause is optionally used to exclude certain activities for SEQ and ALL operators.

To help understand clear semantics of language constructs, we present the formal definition of each operator in Table 1. For the definition, we first define input and output of ActraMon and the operator definition is presented based on the input and output.

First, ActraMon takes a set of *location streams* from moving objects as input. A location stream of a single object, a series of *location data* generated from the object, is defined as follows:

Definition 1. Location data and location stream.

- loc *is an location data, formally defined as a tuple of (*o_{id}*, coord, t), where*
  - o_{id} *is an identifier of a moving object,*
  - coord *is a location coordinate in d-dimensional space (coord* $\in \mathbf{R}^{\mathrm{d}}$*), and*
  - t *is a timestamp.*

- loc_strm *is a location stream generated from a moving object, formally defined as an infinite series of location data* $<loc_1, loc_2, loc_3, \ldots>$, *where* $(loc_{k-1}.o_{id} = loc_k.o_{id})$ *and* $(loc_{k-1}.t < loc_k.t)$, *for all* $k > 1$.

Over the input location streams, ActraMon then detects activity instances of moving objects and generates activity instance streams, which are used as internal input for ATP composition.

Definition 2. Activity instance and Activity instance stream.

- act *is an activity instance of a moving object detected by space-time abstraction, formally defined as a tuple of* $(o_{id}, a_{id}, t_s, t_e)$, *where*
    - o_{id} *is an identifier of a moving object,*
    - a_{id} *is an identifier of a primitive activity, internally given by ActraMon, and*
    - t_s *and* t_e *are the timestamps of starting and ending the activity.*
- act_strm *is an activity instance stream of a moving object, formally defined as an infinite series of activity instances* $<act_1, act_2, act_3, \ldots>$, *where* $(act_{k-1}.o_{id} = act_k.o_{id})$ *and* $(act_{k-1}.t_e < act_k.t_e)$, *for all* $k > 1$.

ActraMon detects ATP instances of moving objects over activity instance streams, and generates resulting ATP instances as final output.

Definition 3. ATP instance.

- atp *is an ATP instance of a moving object, defined as a tuple of* $(o_{id}, c_{id}, t_s, t_e, (act_{M1}, \ldots, act_{Mn}))$, *where*
    - o_{id} *is an identifier of a moving object,*
    - c_{id} *is an identifier of a composite ATP, internally given by ActraMon,*
    - t_s *and* t_e *are the timestamps of starting and ending the atp instance, and*
- act_{Mi}*'s for* $1 \leq \mathrm{i} \leq \mathrm{n}$ *are the matching activity instances that compose the atp instance.*

ATP monitoring queries specified with the constructs have several common features as follows:

- ActraMon is designed mainly for collective monitoring of citizens and vehicles across a whole city. For the purpose, ATP monitoring queries are supposed to observe all moving objects.
- ATP monitoring starts to be effective with subsequent input data after the corresponding query has been registered into the system. ActraMon does not keep past location data and activity instances that might be used for immediate response to upcoming queries.
- Upon an input data of an object, each ATP query generates at most one ATP instance, which represents the most recent ATP behavior of the object. This can be considered as a *recent selection mode* in active database among several different composition modes to compose events [37].

The ActraMon language constructs can be used in combination with SQL-like continuous query languages [13]. The postprocessing such as aggregation, provided by the languages, can be used to further group and summarize ATP instances. It is useful for city-scale monitoring applications such as public transportation planning.

3.3 City-Wide ATP Monitoring

The primary purpose of city-wide ATP monitoring is to support and improve the planning, operation, and maintenance of various urban services in a complex modern city. Thus, the focus is on attaining global understanding of the dynamics of a large-scale city, which involves a huge number of residents, urban spaces, and facilities. Resultantly, it is important to capture and observe collective behaviors of city residents beyond personal behaviors of individuals. Also, the ATP monitoring focuses more on the macroscopic observation of residents' behaviors with regard to the usage of urban services and city facilities. Thus, it abstracts the urban activities in a high level with regard to the spatio-temporal occupancy of urban resources by the residents and its pattern. Then, it provides a basis for rich composition of diverse ATPs of interest to support various urban services. This high-level abstraction is advantageous for the observation and understanding of the dynamics of a modern city, especially a large-scale complex one.

Providing a framework to attain the integrated, complete understanding of a city is a challenging problem. Understanding the detailed personal behaviors of individual residents is not the objective of this city-wide ATP monitoring nor the scope of this paper. There are a number of researches for such microscopic understanding of personal behaviors, which mainly utilizes various sensor technologies such as accelerometers, gyroscopes, and physiological sensors. The design of the proposed model can be extended to attain a more complete understanding of city lives by incorporating such detailed observation.

4 Scalable Processing Architecture

4.1 Shared Staging Architecture

For a large-scale ATP monitoring framework, we devise a novel *shared staging* processing architecture. (Fig. 2) It is designed to be highly efficient and scalable in dealing with many diverse, complicated ATP queries as well as massive location updates from a large number of moving objects. Conventional systems [11], [18], [20] that support continuous processing of complicated monitoring queries mostly take a query plan-based approach; each query is planned as a chain of operators and processed separately from other queries. Although processing each query plan can be optimized, the separated query processing inherently limits scalability. Frequent location updates are repeatedly dispatched to all query plans, and they could cause unnecessary and redundant processing overhead.

The proposed architecture is advantageous as follows:

- The challenges are compound for efficient ATP processing, i.e., complex long-running ATP queries and huge-scale location updates. It is difficult to resolve both concerns at the same time. Regarding the ATP model, the architecture divides the processing into two stages: activity detection and composition. The staging of processing steps enables separation of the concerns and opens chances to develop efficient techniques properly optimized for each stage. The activity detection stage focuses on efficient space-time abstraction from massive and continuous location updates, while the activity composition stage supports flexible and complex composition.

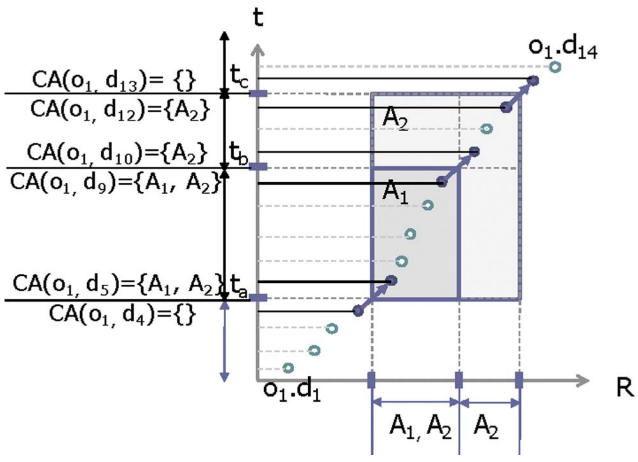
- For each stage, we design and implement novel shared processors which collectively handle all operators of the stage: a shared activity detector for ACT operators and a shared activity composer for composition operators, i.e., SEQ, ALL, and ANY. To efficiently build the shared processors, we develop shared data structures and localized processing methods adequate to each stage. It enables to process all the registered operators with a single operation. The data structures are specialized to maximize the effect of shared processing by collectively organizing individual operators into a single data structure. The localized processing methods are designed to process input data over all the operators by accessing only a small part of the data structures. In this way, the shared processors significantly reduce processing overhead as well as memory consumption.
- Staging also facilitates efficient filtering of massive location updates at the early stage of processing. The early filtering is important for the overall processing performance; it passes only a small number of meaningful activity instances as it filters out insignificant and unnecessary location updates. As a result, it avoids a number of complex, unnecessary compositions which would otherwise frequently be triggered by continuous location updates.

To enable the shared staging, ActraMon parses and decomposes ATP monitoring queries into individual operators. Then, it groups all ACT operators and dispatches them to the shared activity detector to be processed collectively. Similarly, all composition operators are grouped and processed within the shared activity composer.

4.2 Activity Detection Stage

The challenges in the activity detection stage lie in efficiently processing a large number of ACT operators over massive location updates. Each ACT operator should evaluate if a sequence of location updates within a time window satisfies its spatial and temporal conditions. A large time window often demands huge memory consumption. It also incurs huge processing overhead to evaluate the conditions repeatedly upon each location update from plenty of moving objects. Moreover, a number of ACT operators extremely aggravate such processing and memory overhead.

To address these challenges, we develop an efficient shared and incremental processing mechanism for the activity detector. We observe the moments of topological changes between moving objects and activity regions as the significant reference to effectively detect activity instances. To enable the efficient detection in a massive scale based on the observation, we build a novel index, *Activity Border Index*, over a large number of the operators and objects. ABI captures and encodes the topological changes. It localizes detection processing effectively to a small part of the whole index, and further quickly identifies all potentially relevant operators at a single step, achieving a high level of the scalability. ABI is based on our precedent work [39], [29], but it is further elaborated to efficiently support both spatial and temporal indices.


t
$o_1.d_{14}$
$CA(o_1, d_{13}) = \{\}$
t_c
$CA(o_1, d_{12}) = \{A_2\}$
A_2
$CA(o_1, d_{10}) = \{A_2\}$
t_b
$CA(o_1, d_9) = \{A_1, A_2\}$
A_1
$CA(o_1, d_5) = \{A_1, A_2\}$
t_a
$CA(o_1, d_4) = \{\}$
$o_1.d_1$
R
A_1, A_2
A_2
Fig. 3. Temporal borders for activity detection: for simple and clear explanation, we project 2D spatial region in 1D axis (R-axis).

4.2.1 *Activity Detection*

The activity detector takes a set of ACT operators and a set of location streams as input. As stated prior, the location and space are assumed to be defined in any n-dimensional space. For the simplicity of illustration, however, those are hereafter confined in 2-dimension. Thus, an ACT operator is defined formally as follows:

Definition 4. ACT operator.

- ACT *is formally defined as a tuple of* $(a_{id}, R, t_{min}, t_{max})$, *where*
  - a_{id} *represents the identifier of an ACT operator,*
  - R *is an activity region represented as* (x_l, y_l, x_u, y_u), *and*
  - t_{min}, *and* t_{max} *are the temporal conditions.*

The input, a set of location streams, is $\{loc\_strm\}$ defined in Definition 1. As output, the activity detector generates activity instances, *act* defined in Definition 2.

4.2.2 *Activity Border Index*

The ABI is a shared data structure for all moving objects which incorporates the specification of all ACT operators. The idea behind ABI are to concentrate on the changes in object-activity containment relations; an object has a containment relation with an activity when the location of the object is *contained* in the region of the activity. Location updates from a moving object may cause changes in its containment relations with some activities, i.e., "moving-in" to or "moving-out" of the regions of the activities, *ACT*s. However, most of the updates hardly cause any changes in the relations. For efficiency, we capture only the changes in the containment relationships and conceptualize the moments of the changes as the *temporal borders* of activities. ABI keeps the temporal borders dynamically occurring with regard to plenty of ACT operators and moving objects. Based on such temporal borders, the evaluation can be done progressively without repetitively tracing back the past location updates.

Temporal borders are illustrated in Fig. 3. The figure demonstrates an example with two ACT operators, A_1 and A_2, for a moving object o_1. o_1 generates 14 location updates from loc_1 at the lower left corner to loc_{14} at the upper right.

The spatial dimension, i.e., R-axis, designates the activity regions R_1 and R_2, of A_1 and A_2, respectively. The temporal dimension, i.e., t-axis, shows the time points of the location updates. Note that from 14 location updates, only three temporal borders, t_a, t_b, and t_c, have been created between the pairs of time points $(loc_4.t, loc_5.t)$, $(loc_9.t, loc_{10}.t)$, and $(loc_{12}.t, loc_{13}.t)$, when some changes occur in the containment relationship.

A core part in the development of the temporal borders is the computation of containment relations. It can be facilitated by using a shared region index such as R-tree or Containment Encoded Square (CES) [27]. Given a location data, the index enables the efficient computation of a set of containing regions. However, activity detection process should eventually obtain the changes in the containment relations, not the relations themselves. Utilizing such existing indices, therefore, the change detection inherently involves expensive set-difference operations, causing severe processing and memory overhead over successive and frequent location updates. For instance, SINA [7] incrementally reports the changes occurring to a region similar to our case. It performs the costly set-difference operation after retrieving all contained objects through a grid-based index. Yet, ABI employs a novel data structure and processing mechanism carefully designed to obtain the changes directly without computing containment relations.

Given a location stream $<loc_1, \ldots, loc_{k-1}, loc_k, \ldots>$ and a set of ACT operators $\{ACT_i\}$, the containing ACT set and the changes in the containing ACT sets (CAs) are defined in Definitions 1 and 2. $S(o_j, loc_k)$ denotes the set of the ACT operators whose activities are started by the location update loc_k. $E(o_j, loc_k)$ similarly specifies the set of the ending ACT operators. As described in Fig. 3, nonempty $S(o_j, loc_k)$ or $E(o_j, loc_k)$ indicate the occurrence of temporal borders at the time of the location update loc_k.

Definition 5. *Containing ACT set (CA) for an update loc_k of object o_j*

- $CA(o_j, \mathrm{loc_k}) = \{ACT_i | loc_k.(x, y) \in ACT_i.R,\ \ where\ loc_k.o_{id} = o_j\}$

Definition 6. *$S(o_j, loc_k)$ and $E(o_j, loc_k)$ over a set of activity $\{ACT_i\}$*

- $S(o_j, loc_k) = CA(o_j, loc_k) - CA(o_j, loc_{k-1})$
- $E(o_j, loc_k) = CA(o_j, loc_{k-1}) - CA(o_j, loc_k)$

For direct computation of $S(o_j, loc_k)$ and $E(o_j, loc_k)$, ABI works on the borders of activity regions rather than the activity regions themselves since it is the crossing over of a border that changes the containing ACT set. It divides the domain of location data into a number of segments with the borders. (See Fig. 4.) The differences in the containing ACT sets between adjacent segments are evaluated in advance and encoded in corresponding borders.

Designed as a stateful index, ABI holds the states of the most recent evaluations, such as the segments where moving objects are located. It further supports efficient incremental processing; $S(o_j, loc_k)$ and $E(o_j, loc_k)$ are computed incrementally starting from the previous states of moving objects toward new states. Effectively utilizing the stored states, the processing avoids the repetitive computation to start from scratch and reduces the processing cost significantly.

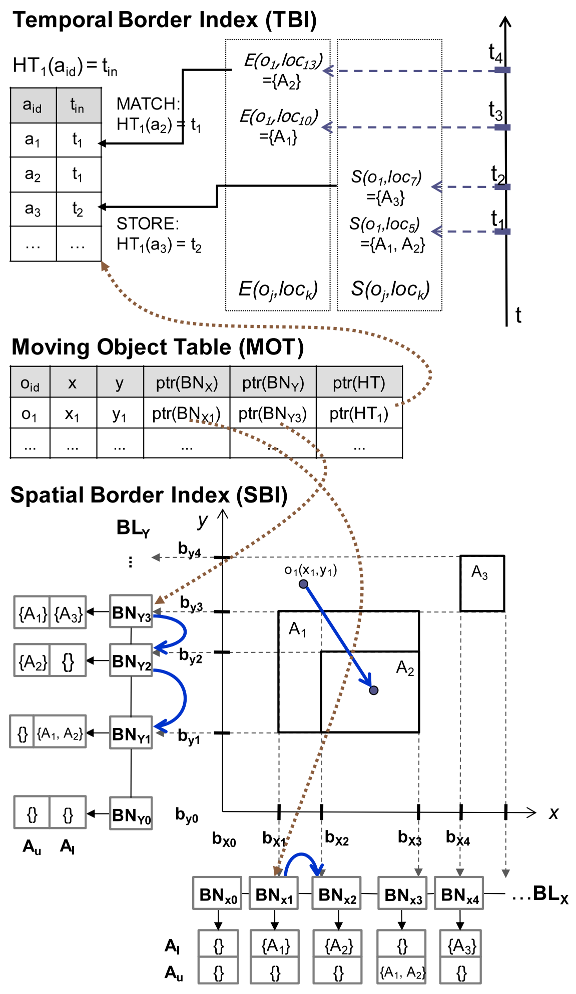


Fig. 4. Activity border index.

With the incremental processing, ABI best utilizes the locality of location streams. Location data mostly show gradual changes. This helps narrow the scope of the data structure to be accessed upon a location update. Thus, $S(o_j, loc_k)$ and $E(o_j, loc_k)$ are generated by accessing a small or no part of the index.

4.2.3 *Data Structure*

Fig. 4 depicts the data structures of Activity Border Index, which consists of three data structures: *Temporal Border Index* (TBI), *Spatial Border Index* (SBI), and *Moving Object Table* (MOT). TBI efficiently indexes dynamic temporal borders which are created by spatial border crossings. SBI indexes a large number of the spatial borders of ACT operators $\{ACT_i\}$ to enable efficient detection of temporal borders. MOT holds the state reflecting the last location update of each moving object and supports the incremental evaluations of TBI and SBI.

Temporal Border Index is a set of hash tables $\mathrm{HT_j}$s, one for each object o_j, with entries $<a_{id}, t_{in}>$, such that $\mathrm{HT_j}(a_{id}) = t_{in}$. $\mathrm{HT_j}$ keeps the temporal borders which a moving object o_j has created. An entry $<a_{id}, t_{in}>$ in $\mathrm{HT_j}$ represents that the object o_j has moved into the region of activity a_{id} at time t_{in}.

Spatial Border Index consists of two linked lists, $\mathrm{BL_X}$ and $\mathrm{BL_Y}$ defined as below. The two lists arrange the spatial borders of ACT operators in X- and Y-dimension, respectively. To effectively define the lists, we establish a set of the spatial borders in each dimension.

Border sets, $\mathbf{B_X}$ and $\mathbf{B_Y}$, contain the spatial borders on X- and Y-axes, respectively:

- $\mathrm{B_X} = \{b_X | b_X = ACT_i.R.x_l$ or $b_X = ACT_i.R.x_u$ for some $i\} \cup \{x_{min}, x_{max}\}$,
- $\mathrm{B_Y} = \{b_Y | b_Y = ACT_i.R.y_l$ or $b_Y = ACT_i.R.y_u$ for some $i\} \cup \{y_{min}, y_{max}\}$,

where $\mathrm{x_{min}}$ and $\mathrm{x_{max}}$ are the minimum and maximum values of X-dimension, and $\mathrm{y_{min}}$ and $\mathrm{y_{max}}$ are likewise for Y-dimension. For each border in $\mathrm{B_X}$ or $\mathrm{B_Y}$, we define a border node $\mathrm{BN_X}$ or $\mathrm{BN_Y}$, respectively.

Border Node $\mathbf{BN_X}$ is represented as $<b_X, A_l, A_u>$, where b_X is a spatial border of certain ACT operators on X-axis, i.e., a member of $\mathrm{B_X}$. A_l is the set of ACT operators whose regions are lower bounded by b_X in X-dimension, i.e., $A_l = \{ACT_i | ACT_i = (a_{id}, R, t_{min}, t_{max})$, such that $R.x_l = b_X\}$. Similarly, A_u is the set of ACT operators whose regions are upper bounded by b_X in X-domain, i.e., $A_u = \{ACT_i | ACT_i = (a_{id}, R, t_{min}, t_{max})$, such that $R.x_u = b_X\}$. A_l and A_u indicate the activities which would be started or ended by crossing the border of the node $\mathrm{BN_X}$. Border Node on Y-axis, $\mathrm{BN_Y}$, is defined similarly.

Border List of X-axis, $\mathbf{BL_X}$, is a sorted list of $\mathrm{BN_X}$s, i.e., $<\mathrm{BN_{X1}}, \mathrm{BN_{X2}}, \ldots, \mathrm{BN_{Xn}}, \ldots>$, where $\mathrm{BN_{Xi}}.b_X < \mathrm{BN_{X(i+1)}}.b_X$ for all i. Border List of Y-axis, $\mathrm{BL_Y}$ is similarly defined as $<\mathrm{BN_{Y1}}, \ldots, \mathrm{BN_{Yn}}, \ldots>$ over $\mathrm{BN_{Yi}}s$.

Moving Object Table is a table that maintains the last states of moving objects. The table is a list of tuples $<o_{id}, x, y, \mathrm{ptr(BN_X)}, \mathrm{ptr(BN_Y)}, \mathrm{ptr(HT}_{oid})>$, where o_{id} is the identifier of moving objects, and (x, y) is the coordinate of the previous location update. $\mathrm{ptr(BN_X)}$ points to a border node, $\mathrm{BN_{Xk}}$, where the previous location update is between $\mathrm{BN_{Xk}}.b_X$ and $\mathrm{BN_{Xk+1}}.b_X$. $\mathrm{ptr(BN_Y)}$ does likewise for Y-dimension. $\mathrm{ptr(HT}_{oid})$ points to the hash table, HT_{oid}, in TBI which stores the temporal borders of the object o_{id}.

4.2.4 Processing Algorithm

The activity detection algorithm consists of three steps.

Step 1. Detecting spatial border crossings

The algorithm computes $S_X(o_j, loc_k)$ and $E_X(o_j, loc_k)$ that denote the sets of ACT operators whose X-dimensional borders are crossed by a location update loc_k. Upon arrival of loc_k, SBI looks up the previous state of the object o_j, i.e., $(x_{prev}, y_{prev}, \mathrm{ptr(BN_{Xprev})}, \mathrm{ptr(BN_{Yprev})})$ in the Moving Object Table, and takes following procedures.

Border-crossing-test $(BN_{Xprev}, loc_k.x)$ tests if the update does not cross any borders in X-dimension, i.e., $loc_k.x$ is between $\mathrm{BN_{Xprev}}.b_X$ and $\mathrm{BN_{Xprev+1}}.b_X$. If not, the resulting $S_X(o_j, loc_k)$ and $E_X(o_j, loc_k)$ get empty. If $loc_k.x \geq \mathrm{BN_{X(prev+1)}}.b_X$, the procedure *Forward-traverse* is conducted. If $loc_k.x < \mathrm{BN_{Xprev}}.b_X$, the procedure *Backward-traverse* is conducted. Note that $S_X(o_j, loc_k)$ and $E_X(o_j, loc_k)$ would be empty for many of the location updates due to the locality of location data which mostly moves within a region.

Forward-traverse $(\boldsymbol{BN_{Xprev}, loc_k.x})$ traverses border nodes in $\mathrm{BL_X}$ from the $\mathrm{BN_{Xprev}}$ in the increasing order of X-coordinate. The traversal stops at the border node corresponding to the current location $loc_k.x$, i.e., $\mathrm{BN_{Xnew}}$ such that $\mathrm{BN_{Xnew}}.b_X \leq loc_k.x < \mathrm{BN_{X(new+1)}}.b_X$. By then, the traversal simply aggregates the activities in A_l and A_u of each traversed border node in order to compute $S_X(o_j, loc_k)$ and $E_X(o_j, loc_k)$. Precisely,

- $E_X(o_j, loc_k) = \cup_{p=prev+1}^{new} BN_{Xp}A_u$, and
- $S_X(o_j, loc_k) = \cup_{p=prev+1}^{new} BN_{Xp}A_l$

Backward-traverse $(\boldsymbol{BN_{Xprev}, loc_k.x})$ is similar except;

- $E_X(o_j, loc_k) = \cup_{p=new+1}^{prev} BN_{Xp}A_l$, and
- $S_X(o_j, loc_k) = \cup_{p=new+1}^{prev} BN_{Xp}A_u$

Note that A_l and A_u are interchanged in the $S_X(o_j, loc_k)$ and $E_X(o_j, loc_k)$ unlike *Forward-traverse*. The directions of the traversals are reverse in the two cases.

Lastly, the new state of the moving object o_j, i.e., $loc_k.x, loc_k.y, \mathrm{ptr(BN_{Xnew})}$, and $\mathrm{ptr(BN_{Ynew})}$, is stored into the MOT. The procedures for Y-dimension are similarly defined to compute $S_Y(o_j, loc_k)$and $E_Y(o_j, loc_k)$.

Step 2. Detecting temporal borders

The algorithm computes the temporal borders, $S(o_j, loc_k)$ and $E(o_j, loc_k)$. This is done by validating the border crossings in each dimension, i.e., $S_X(o_j, loc_k)$, $S_Y(o_j, loc_k)$, $E_X(o_j, loc_k)$, and $E_Y(o_j, loc_k)$, detected, respectively, in step 1. This validation checks if a crossing in either dimension causes an actual crossing of a two-dimensional border. Details on the validation can be found in [29], [39]. As a result, the activities validated in both dimensions are collected as temporal borders, i.e., $S(o_j, loc_k)$ and $E(o_j, loc_k)$.

Step 3. Temporal condition evaluation

To generate activity instances from $S(o_j, loc_k)$ and $E(o_j, loc_k)$, the temporal conditions are tested using TBI. The start times of the activities in $S(o_j, loc_k)$ are stored into TBI for later matching. The activities in $E(o_j, loc_k)$ are matched to previously started activities stored in TBI. For each operator ACT_i in $E(o_j, loc_k)$, $\mathrm{HT_j}$ is looked up with $ACT_i.a_{id}$ to get the start time of ACT_i, i.e., $t_{in} = \mathrm{HT_j}(ACT_i.a_{id})$. If the time difference between t_{in} and $loc_k.t$ satisfies the time condition of ACT_i, i.e., $ACT_i.t_{min} \leq (loc_k.t - t_{in}) \leq ACT_i.t_{max}$, an activity instance $(loc_k.o_{id}, ACT_i.a_{id}, t_{in}, loc_k.t)$ is created. The matched entries are removed from TBI.

4.2.5 ABI Update

ABI supports dynamic registration and deregistration of ACT operators. Upon registration and deregistration of an ACT operator, updating SBI with the spatial borders of the operator is essential. For the update, the X-dimension and Y-dimension borders are handled separately. Consider an ACT operator, ACT_i, whose activity region R is (x_l, y_l, x_u, y_u). When registering ACT_i to the ABI, an X-dimension range, (x_l, x_u), is registered to the border list of x-axis, $\mathrm{BL_x}$, and an Y-dimension range, (y_l, y_u), is registered to the $\mathrm{BL_y}$. Without loss of generality, we illustrate the procedure with respect to X-dimension.

In detail, for the update of $\mathrm{BL_x}$, activity detector locates the border node, BN_{Xm} which contains x_l, i.e., $b_m \leq x_l < b_{m+1}$. If x_l is equal to b_m, i.e., $BN_{Xm}.b_x$, then BN_{Xm} becomes responsible for x_l so that ACT_i is inserted into the A_l of BN_{Xm}. Otherwise, a new border node, BN_{Xnew} is inserted between BN_{Xm} and BN_{Xm+1} in BL_X, where

BN_{Xnew} is set to $<x_l, \{ACT_i\}, \{\}>$. The border node for x_u is also symmetrically updated.

When an ACT operator, ACT_i, is deregistered, the activity detector first locates the border node, BN_{Xm} whose lower bound is equal to x_l or x_u, and removes the ACT_i from A_l or A_u of the BN_{Xm}, respectively. If both A_l and A_u are empty, BN_{Xm} is merged with BN_{Xm-1}. The rest of the ABI structure, i.e., MOT and TBI, is updated accordingly with respect to the change of SBI.

4.3 Activity Composition Stage

The activity composer evaluates a set of composition operators over a number of activity instance streams, $\{act\_strm\}$, and generates a set of resulting ATP instances, $\{atp\}$. In this stage, we use a single definition of composition operators, which uniformly represents SEQ, ALL, and ANY operators defined in Section 3.2, as follows:

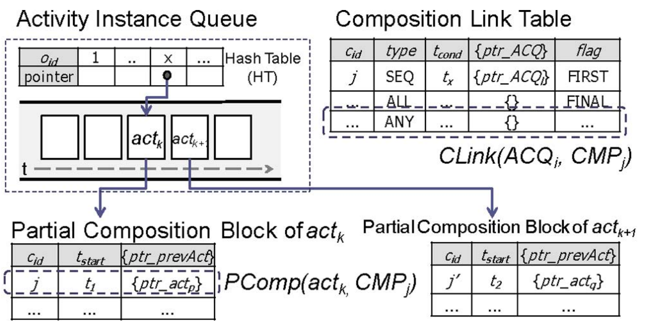
Definition 7. Composition operator.

- CMP *is a composition operator, formally defined as a tuple of* $(c_{id}, type, PASet, FASet, t_c)$*, where*
  - c_{id} *is an identifier of a composite operator, internally given by ActraMon,*
  - type *is either SEQ, ALL, or ANY,*
  - PASet *is a set of participating activities, formally defined as* $\{ACT_{P1}, \ldots, ACT_{Pn}\}$*,*
  - FASet *is a set of forbidden activities specified in the optional WITHOUT clause, formally defined as* $\{ACT_{F1}, \ldots, ACT_{Fm}\}$*, and*
  - t_c *indicates the length of the composition time window.*

The main challenge in this stage is that we should deal with a large number of diverse activity composition operators in large-scale ATP monitoring. As a key idea, we develop a shared and incremental processing method based on a novel data structure, called *Activity-centric Composable Queue*. ACQ is a shared queue that stores activity instances of a specific primitive activity. It serves as the basic building block of the activity composer. The composer constructs a single shared network of multiple ACQs, as many as the primitive activities specified in {*CMP*}, and evaluates all the composition operators in conjunction with the constructed network.

To effectively support the shared and incremental processing, ACQ has the following unique features:

1. For each primitive activity, only a single ACQ is allocated in the network and shared by multiple composition operators. Due to the sharing, processing an activity instance in a single ACQ evaluates all the composition operators which are interested in the instance.
2. Each ACQ is also shared by all moving objects. This design substantially reduces processing and storage overhead necessary for separately manipulating intermediate composition results of each moving object.
3. On top of the common ACQ structures, we design a unified processing method which is applicable to evaluating not only SEQ operators but also ALL and ANY.



Activity Instance Queue
Composition Link Table
Hash Table (HT)
o_{id} | 1 | .. | x | ...
pointer
act_k
act_{k+1}
t
c_{id} | type | t_{cond} | {ptr_ACQ} | flag
j | SEQ | t_x | {ptr_ACQ} | FIRST
... | ALL | ... | {} | FINAL
... | ANY | ... | {} | ...
$CLink(ACQ_i, CMP_j)$
Partial Composition Block of act_k
Partial Composition Block of act_{k+1}
c_{id} | t_{start} | {ptr_prevAct}
j | t_1 | {ptr_act_p}
j' | t_2 | {ptr_act_q}
$PComp(act_k, CMP_j)$

Fig. 5. Activity-centric Composable Queue.

4. ACQ supports incremental composition for individual composition operators. Upon arrival of an activity instance, it incrementally extends partial compositions of composition operators and progressively completes the compositions to the end.

The proposed ACQ-based processing is further advantageous especially in city-wide ATP processing. First, cities have many hot spots such as shopping complexes and stations. The activity composer efficiently handles multiple composition operators by sharing the common primitive activities over those popular spaces. Second, the activity compositions in city-wide ATP monitoring often involve long-term processing, e.g., for several hours or even days. Such long-term processing tends to rapidly increase intermediate composition results. The ACQ sharing, however, substantially reduces the amount of the intermediate results, and makes the long-term processing more effective in terms of storage consumption.

4.3.1 Activity-Centric Composable Queue

ACQ maintains three data structures of *Composition Link Table*, *Activity Instance Queue*, and *Partial Composition Block*. (See Fig. 5.) A single ACQ, ACQ_i, is designated to store the information to process all the activity instances delivered from ACT_i.

Composition Link Tables enable the construction of the ACQ network that supports the shared processing of composition operators. For an ACQ_i, its Composition Link Table contains a set of *composition links*, one for each composition operator that the ACQ_i participates in. Each link, denoted as $CLink(ACQ_i, CMP_j)$, represents the association of ACQ_i with the other ACQs specified in CMP_j.

Definition 8. *Composition link (*CLink*) in* ACQ_i *w.r.t.* CMP_j

- $CLink(ACQ_i, CMP_j) =$ $<c_{id}, type, t_{cond}, \{ptr\_ACQ\}, flag>$,
  - c_{id}*, type, and* t_{cond} *are the identifier, type and time condition of* CMP_j*, respectively,*
  - *{ptr_ACQ} presents the pointers to other ACQs which should be connected for* CMP_j*, and*
  - *flag marks the position of* ACQ_i *in* CMP_j*; FIRST indicates the first ACQ starting the composition, FINAL indicates the last ACQ ending the composition, and WITHOUT indicates that it is forbidden.*

{ptr_ACQ} and *flag* play a critical role for the network construction. Fig. 6 shows the examples of ACQ compositions using *CLink*'s with two sample CMP's. Here, CMP_1

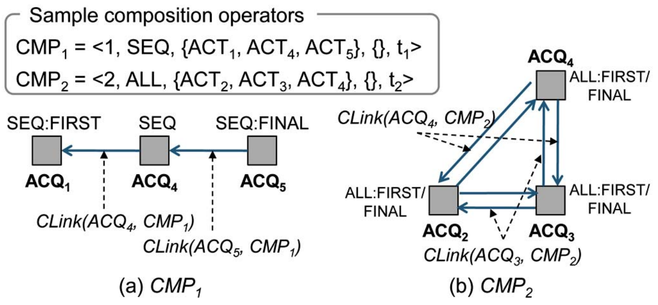


Fig. 6. ACQ composition by CLinks: (a) for SEQ, the participating ACQs, drawn as plain boxes, are linked sequentially and flagged accordingly as FIRST or FINAL, (b) for ALL, the participating ACQs are linked to each other and flagged as "FIRST&FINAL."

represents the query C1 given in Section 3.2, which monitors a probable dating pattern. ACT$_1$, ACT$_4$, and ACT$_5$, respectively, indicate the activities of visiting a theater_A, a GUCCI shop and a restaurant_B. CMP$_2$ represents the query C3 that identifies brand shoppers. ACT$_2$, ACT$_3$, and ACT$_4$ indicate the activities of shopping at PRADA, CHANEL, and GUCCI, respectively. Note that ACT$_4$, shopping at GUCCI, is shared by the two CMP's.

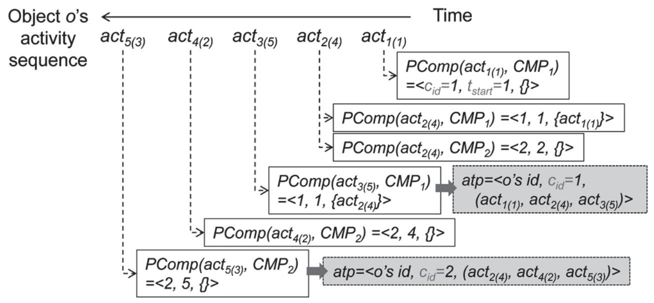
Activity Instance Queue manages the recent activity instances of all moving objects with regard to ACT_i. It stores the activity instances, act_k's, from ACT_i in the order of their arrivals. The single queue in ACQ_i is shared by a number of moving objects. It also uses a hash table that facilitates the access to the recent activity instance act_k and its partial composition states. The hash table uses o_{id} as its key, i.e., $\mathrm{HT}(o_{id}) = act_k$ for $o_{id} = act_k.o_{id}$.

Partial Composition Block is the main data structure for the incremental evaluation. A block is allocated to each activity instance act_k as shown in Fig. 5. It stores the current states of the partial compositions in which the instance act_k participates. The block has a set of partial composition entries, *PComp*'s, one for each composition operator. $PComp(act_k, CMP_j)$, if it exists, represents that a partial composition of CMP_j has been successfully extended by the instance act_k at ACQ_i. The block allows the incremental extension of the partial compositions until the compositions become completed.

Definition 9. *Partial composition (*PComp*) of* act_k *w.r.t.* CMP_j

- $PComp(act_k, CMP_j) =< c_{id}, t_{start}, \{ptr\_prevAct\}>$,
  - $\mathrm{c_{id}}$ *is the identifier of* CMP_j,
  - $\mathrm{t_{start}}$ *is the start time of the partial composition, and*
  - $\{ptr\_prevAct\}$ *is a set of pointers to the precedent activity instances, stored in other ACQs, leading to the current partial composition of* CMP_j. *When the composition is complete, the pointers are followed iteratively to obtain all the participating activity instances.*

Fig. 7 illustrates the example sequence of input activity instances and the correspondingly generated *PComp* and *atp* instances with CMP_1 and CMP_2. For simplicity, the activity instances of a single moving object, i.e., object o, are considered here. When the first instance, $act_{1(1)}$, comes in, it



Object o's activity sequence
Time
act$_{5(3)}$
act$_{4(2)}$
act$_{3(5)}$
act$_{2(4)}$
act$_{1(1)}$
PComp(act$_{1(1)}$, CMP$_1$) =<c$_{id}$=1, t$_{start}$=1, {}>
PComp(act$_{2(4)}$, CMP$_1$) =<1, 1, {act$_{1(1)}$}>
PComp(act$_{2(4)}$, CMP$_2$) =<2, 2, {}>
PComp(act$_{3(5)}$, CMP$_1$) =<1, 1, {act$_{2(4)}$}>
atp=<o's id, c$_{id}$=1, (act$_{1(1)}$, act$_{2(4)}$, act$_{3(5)}$)>
PComp(act$_{4(2)}$, CMP$_2$) =<2, 4, {}>
PComp(act$_{5(3)}$, CMP$_2$) =<2, 5, {}>
atp=<o's id, c$_{id}$=2, (act$_{2(4)}$, act$_{4(2)}$, act$_{5(3)}$)>

Fig. 7. Example sequence of activity instances and generated composition results: $act_{k(i)}$ denotes the kth activity instance of object o generated by ACT_i operator.

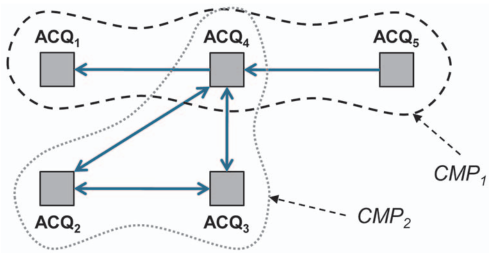


Fig. 8. N-ACQ with two CMP operators: each dashed line indicates the equivalent query plan, respectively, for CMP_1 and CMP_2.

is delivered to ACQ_1 and creates the initial partial composition of CMP_1, $PComp(act_{1(1)}, CMP_1)$. Upon $act_{2(4)}$'s arrival, it extends the partial composition of CMP_1 and initiates the new composition of CMP_2, resulting in $PComp(act_{2(4)}, CMP_1)$ and $PComp(act_{2(4)}, CMP_2)$, respectively. For $act_{3(5)}$, ACQ_5 finally completes the partial composition of CMP_1 and generates a resulting *atp* instance since $CLink(ACQ_5, CMP_1)$ is flagged as FINAL. Later, $act_{4(2)}$ generates its own *PComp* entry for CMP_2 within ACQ_2, and $act_{5(3)}$ finally completes CMP_2 within ACQ_3.

4.3.2 Construction of ACQ Network

Given $\{CMP_j\}$, the activity composer constructs a single ACQ network, called N-ACQ. In order to represent $\{CMP_j\}$ effectively, N-ACQ has all the comprising ACQs of each composition operator as its *nodes* and connects them using the composition links, *CLink*'s. With the links, an ACQ is effectively shared by multiple composition operators in the network. Fig. 8 shows the example N-ACQ with the two sample CMP's (CMP_1 and CMP_2 given in Fig. 6) using five primitive activities (from ACT_1 to ACT$_5$). In the figure, ACQ_4 is shared by both operators.

In brief, N-ACQ is constructed as follows: For each composition operator $CMP_j = <c_{id}, type, \{ACT_{P1}, \ldots, ACT_{Pn}\}, \{ACT_{N1}, \ldots, ACT_{Nm}\}, t_c>$, a new ACQ is instantiated for each activity, i.e., ACT_{Pk} or ACT_{Nk}. Some ACQs need not be created if they have already been involved in other composition operators. For the CMP_j, the ACQs are associated with each other by adding a *CLink* entry in their Composition Link Tables. In the case of SEQ operator, ACQ_{Pk}'s are linked sequentially; each $CLink(ACQ_{Pk}, CMP_j)$ points to the ACQ of the precedent activity, and the first and final ACQs are marked accordingly in the flag field. Detailed algorithms for the N-ACQ construction and maintenance can be found in [29], [40].

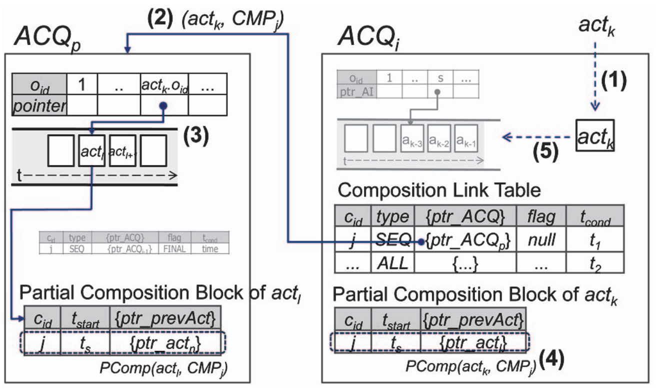

(2) (act_k, CMP_j)
ACQ_p
ACQ_i
act_k
(1)
o_id | 1 | .. | act_k.o_id | ...
pointer
(3)
act_l
act_l+1
t
o_id | 1 | .. | s | ...
ptr_AI
a_k-3
a_k-2
a_k-1
(5)
t
Composition Link Table
c_id | type | {ptr_ACQ} | flag | t_cond
j | SEQ | {ptr_ACQ_p} | null | t_1
... | ALL | {...} | ... | t_2
c_id | type | {ptr_ACQ} | flag | t_cond
j | SEQ | {ptr_ACQ_i-1} | FINAL | time
Partial Composition Block of act_l
c_id | t_start | {ptr_prevAct}
j | t_s | {ptr_act_n}
PComp(act_l, CMP_j)
Partial Composition Block of act_k
c_id | t_start | {ptr_prevAct}
j | t_s | {ptr_act_l}
PComp(act_k, CMP_j)
(4)

Fig. 9. Activity composition processing $(probe(), insert())$.

4.3.3 Activity Composition Algorithm

The composition process mainly consists of two major functions: ***probe()*** and ***insert()***. Upon arrival of an activity instance act_k, the function *probe()* evaluates whether act_k could lead to a partial or complete composition for some composition operators, and only at the instant that a composition happens, the function *insert()* updates the data structures of ACQ_i reflecting the arrival of act_k.

The process is illustrated in Fig. 9. First, act_k is dispatched to its corresponding ACQ, i.e., ACQ_i (step (1) in the figure). It quickly identifies a set of the CMP_j's associated with ACQ_i by looking up *CLink* entries in the Composition Link Table. Then, it invokes the *probe()* (steps (2), (3), and (4)) for each CMP_j. The figure illustrates that ACQ_p is probed by ACQ_i with respect to CMP_j. If the *probe()* extends partial compositions for any CMP, the *insert()* is performed to store act_k into the Activity Instance Queue of ACQ_i (step (5)). The *probe()* differently handles each type of the composition operators. We below sketch the *probe()* and *insert()* for the case of SEQ operator. Details of the functions for other types of operators are in [29, Section 4.3.4], [40].

Probe $(\boldsymbol{ACQ_i}, \boldsymbol{act_k}, \boldsymbol{CMP_j})$ performs a key role of the incremental processing for compositions. The pseudocode of the *probe()* is presented in Fig. 10.

Lines 1-3 (in Fig. 10). The function deals with the case that act_k initiates new partial compositions for some associated CMP_j's. For example, an activity instance from ACT_1, i.e., $act_{1(1)}$ in, is incoming to ACQ_1 for CMP_1. If ACQ_i (ACQ_1) is marked as *FIRST* with respect to CMP_j (CMP_1), the function starts a new partial composition of the CMP_j with act_k at ACQ_i; it creates $PComp(act_k, CMP_j)$ and sets the start time of the partial composition to the arrival time of act_k, i.e., $act_k.t_{out}$.

Lines 4-11. Otherwise, the function deals with the case that act_k extends existing partial compositions as like $act_{2(4)}$ in Fig. 7. It first visits its precedent ACQ referred by the pointer in $\{ptr\_ACQ\}$ (step (2) in Fig. 9). Let the ACQ be $ACQ_p(ACQ_1$ in Fig. 8). The function then identifies the existing partial composition of $CMP_j(CMP_1)$ for the moving object with $act_k.o_{id}$. For the purpose, it looks up the recent activity instance of the same moving object, i.e., act_l, such that $act_l.o_{id} = act_k.o_{id}$, in the Activity Instance Queue of ACQ_p (using $act_k.o_{id}$ as the hash key). It then finds from the Partial Composition Blocks of act_l in ACQ_p the existing partial composition entry, $PComp(act_l, CMP_j)$. With

```
Algorithm Probe(ACQ_i, act_k, CMP_j)
// If act_k initiates a new composition of CMP_j at ACQ_i
1.  if CLink_{i,j}.flag == FIRST then
2.    Create PComp(act_k, CMP_j)
3.    PComp(act_k, CMP_j).t_start = act_k.t_out

// Check the precedent ACQ to extend existing partial compositions
4.  else
5.    ACQ_p = getPointerInCLink(CLink(ACQ_i, CMP_j).{ptr_ACQ})
6.    act_l = HT(act_k.o_id) of ACQ_p  // look up the instance using HT
7.    if act_l != null and PComp(act_l, CMP_j) != null
          and (act_k.t_out - PComp(act_l,CMP_j).t_s) < CMP_j.t_cond then  // time cond
8.      Create PComp(act_k, CMP_j) unless it exists
9.      Add the pointer to act_l into PComp(act_k, CMP_j).{ptr_prevAct}
10.     PComp(act_k, CMP_j).t_start = PComp(act_l, CMP_j).t_s
11.   else unmatched = true  // fail to extend the partial composition

// If the composition is completed, generate resulting ATP instances
12. if unmatched != true and CLink(ACQ_i, CMP_j).flag == FINAL then
13.   Follow pointers in PComp(act_k, CMP_j).{ptr_prevAct} iteratively
          and retrieve participating instances
14.   Construct and output the ATP instance, (CMP_j.c_id, act_k.o_id, {..., act_l, act_k})
```

Fig. 10. *Probe()* algorithm for SEQ operator.

$PComp(act_l, CMP_j)$, the function tests if act_k can successfully extend the existing partial composition of CMP_j. If so, it designates the extension by creating and adding a new entry $PComp(act_k, CMP_j)$ to the Partial Composition Block of act_k in ACQ_i. (step (4) in Fig. 9). If either act_l or $PComp(act_l, CMP_j)$ does not exist, the function fails to extend the composition.

Lines 12-14. The function handles the case that the composition becomes complete, e.g., $act_{3(5)}$ into ACQ_5 for CMP_1. If ACQ_i is marked as *FINAL* (in Composition Link Table) and $PComp(act_k, CMP_j)$ has been already created, the function completes the composition of CMP_j with act_k and generates the corresponding ATP instance by following the pointer in $\{ptr\_prevAct\}$ of *PComp* entries.

Insert $(\boldsymbol{ACQ_i}, \boldsymbol{act_k})$ updates ACQ_i reflecting the dynamic composition status affected by the activity instance act_k. That is, act_k as well as the *PComp* entries newly created in the *probe()* should be stored into ACQ_i. First, the entries are added into the Partial Composition Block of act_k. Then, the function inserts act_k into the Activity Instance Queue (step (5) in Fig. 9). Finally, the function updates the hash entry HT($act_k.o_{id}$) so that it points to the newly added act_k in the queue. Note that, for an instance act_k, the function takes place at most once even for the case that act_k extends multiple partial compositions.

4.4 Performance Analysis

The processing and storage cost of ActraMon can be understood as a sum of activity detection and composition costs. The details of the analyses are in [29, Section 4.4].

According to our analysis, the total processing cost upon a location update is bounded by $O(N_a L_M)$, where N_a denotes the number of the ACT operators embedded in the ATP queries and L_M denotes the mobility level of the objects, i.e., the distance to the current update from the last location normalized with the domain size. The cost becomes very small in practice since the L_M value remains small with real-world moving objects, which means ActraMon achieves a high level of efficiency.

TABLE 2
Parameters (Activity Detector Experiments)

Parameter	Meaning	Range	Default Value
N_o	Number of objects	10K ~ 70K	10K
N_a	Number of ACT operators	10K ~ 70K	10K
R_w	Max. width of activity regions	0.5 ~ 4km	4km
R_{min_w}	Min. width of activity regions	N/A	50m
Dist	Distribution of activity regions	N/A	Uniform
T_{min}, T_{max}	Time conditions	N/A	10 min, 120 min

Also, the total storage cost of ActraMon is $O(N_q+N_o \times D_s)$ where N_o and N_q are the numbers of moving objects and composition queries, respectively, and D_s is the degree of ACQ sharing. The analysis shows that ActraMon is highly scalable in terms of memory consumption as well.

5 Experiment

In this section, we present the performance evaluation of the ActraMon framework. We intend to extensively study the performance behavior of separate modules under wide parameter ranges while reporting the overall performance over realistic workloads. First, we separately conduct performance studies on the activity detector and the activity composer using workloads of a wide range of input data and queries, and report the detailed results in Sections 5.1 and 5.2, respectively. Then, in Section 5.3, we report the result for overall ATP processing in combination with the activity detector and composer, using potential realistic city workloads generated on top of a carefully crafted city model. For the performance metric, we use the processing time and memory consumption which are commonly adopted for the evaluation of in-memory processing techniques [23], [27].

We implement ActraMon and alternative techniques with C++ language on a Linux 2.6.18-6-amd64 kernel. Experiments are conducted on a machine equipped with Intel Core 2 Quad Q9550 CPU (2.83 GHz) and 8 GB RAM. Due to the space limit, we are not able to fully report the results of our performance study in this manuscript. For more comprehensive report on the experiments and results, we refer the readers to [29, Section 5].

5.1 Shared Activity Detector

5.1.1 Experimental Setup

The activity detector takes location updates and a set of ACT operators as its input. Accordingly, we create location and ACT operator workloads for the experiment.

Location data workload. For the data generation, we assume a $20\ \mathrm{km} \times 20\ \mathrm{km}$ city where objects are continuously moving based on a random walk model [28]. Each object is randomly set with a next position in the map and move toward the position at a designated speed while reporting location updates every 30 seconds. Once it arrives at the position, a new position is set to a next destination. A location update consists of object identifier, longitude, latitude, and timestamp.

We consider two types of moving objects, i.e., vehicles and pedestrians. Vehicles represent fast moving objects whose locality in location data is low. They move at speeds between 20 and 60 km/h. Pedestrians, slow moving objects, move at 4 to 8 km/h. The number of moving objects, N_o, varies between 10 and 70K. We simulate 4 hours of location data, i.e., 480 updates per object, for the experiments.

ACT operator workload. We synthesize ACT operators with various parameter based on the scenarios described in Section 3. (See the ranges and default values in Table 2) In the ATP-based advertisement scenario, we expect that the number of activity regions such as restaurants and stores could be up to tens of thousands in a metropolitan city [26]. Based on the fact, we vary the number of primitive activities, N_a, between 10 and 70K. We also perform experiments by changing the size of activity regions in terms of the region width, R_w.

Alternative technique (CES-based approach). As the performance baseline for the server-side algorithm of activity detection, we implement a region index-based approach described in Section 4.2, for which we choose an efficient region index, *Containment Encoded Square* [27]. CES is a grid-based index that provides fast, i.e., constant, search performance, $O(c+k)$, where c and k denotes the depth of hierarchy and the result size, respectively. Adopting hierarchical grids, it consumes less memory than other grid-based indices exploiting huge memory space. Fig. 11 shows a brief and exact description of CES-based approach to evaluate ACT operators. The notations can be referred to in the definitions of Section 4.2.

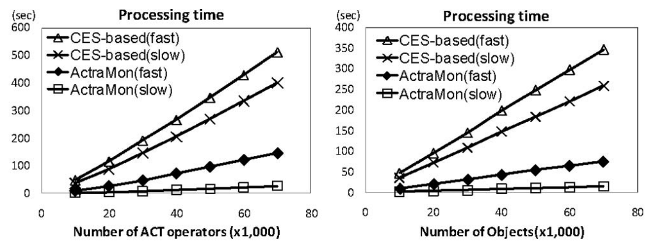
```
Algorithm: Evaluate ($o_j$, $loc_k$)
//$loc_k$ is a $k^{th}$ location data from an object $o_j$
1.  Retrieve the containing ACT set, $CA(o_j, loc_k)$, for $loc_k$ using CES
2.  Retrieve the containing ACT set, $CA(o_j, loc_{k-1})$, for $loc_{k-1}$ using CES
3.  Calculate $S(o_j, loc_k) = CA(o_j, loc_k) - CA(o_j, loc_{k-1})$
4.  Calculate $E(o_j, loc_k) = CA(o_j, loc_{k-1}) - CA(o_j, loc_k)$
5.  Store $S(o_j, loc_k)$ for later matching
6.  Match $E(o_j, loc_k)$ to the stored $S$'s of the object $o_j$
7.  For each $ACT_i$ in $E(o_j, loc_k)$,
8.      Find $S(o_j, loc_{k'})$ that contains $ACT_i$
9.      If $ACT_i.t_{min} < (loc_k.t - loc_{k'}.t) < ACT_i.t_{max}$, then generate a resulting act
```

Fig. 11. CES-based ACT operator evaluation.

5.1.2 Processing Performance

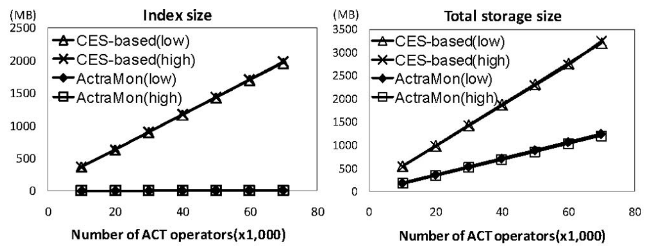
In this experiment, we measure the processing time as the elapsed time taken for processing all the location workload. The index update time for query registration is not included for both ABI and CES from the total processing time. In ATP processing, location updates occurs a lot more frequently than query updates since ATP monitoring queries are long-running, continuous queries. Accordingly, the query updates rarely affect the overall processing time; in a case of our experiment, it only takes 0.26 second to update 70K queries, which is 1 percent of total processing time with 4-hour amount of location updates from 10K slow moving objects and 0.17 percent for the fast-moving objects.

Scalability with operators and moving objects. We evaluate the scalability as increasing N_a and N_o from 10 to 70K, respectively, as shown in Figs. 12a and 12b. In the graphs, the activity detector appears highly scalable in terms of the number of ACT operators and moving objects. It processes ACT operators three to ten times faster than the CES-based method. The scalability is mainly achieved by taking advantage of the shared and incremental processing on ABI. Location updates are highly likely to stay within previous regions and do not cross any activity borders. Even if they do not stay, ABI efficiently starts an index search



(a) Number of operators (N_a) (b) Number of objects (N_o)

Fig. 12. Processing time of activity detector.



(a) Query-index only (b) Total memory consumption

Fig. 13. Memory consumption of activity detector (overN_a).

from the last border node that an object stays on, and the number of traversed border nodes is substantially reduced.

The processing time of the CES-based method, however, drastically increases as N_a and N_o increase. Even location updates staying within an activity region require the repetitive and unnecessary computation of the containing ACT sets and the difference between two consecutive CAs. Moreover, a large N_a increases the size of CAs which also increases the processing time to search the index and to calculate the set difference. As N_o increases, location updates become more frequent, and accordingly the amount of unnecessary computation increases.

The result also shows that the ABI performance is robust against the degree of the locality of location updates. The processing time increases moderately not only with the slow moving objects, but also with the fast moving objects. This shows that the locality-aware search of ABI is still efficient even with the fast moving objects in our experimental model. Interestingly, even for the CES-based method, the processing times of the fast moving objects are higher than that of the slow moving objects. The fast moving object creates temporal borders more frequently due to its fast speed. Thus, the size of $S(o_j, loc_k)$ and $E(o_j, loc_k)$ gets larger, which causes more processing.

As for the experiment with the varying sizes of the activity regions, the result shows that ActraMon is highly robust with the changes in R_w, showing a very slow increase, while the CES-based method shows a steep increase. Also, we experimented by varying the time conditions of activities, T_{min} and T_{max}, and observed that they rarely affect the processing time. This observation confirms the performance analysis in Section 4.4; T_{min} and T_{max} are not included as variables affecting the time complexity.

5.1.3 Storage Cost

In this experiment, we log the memory consumption whenever memory spaces are newly allocated and freed. We take a representative value for each run, after the memory consumption is stabilized. It is measured as varying N_a and R_w, which are the two influencing parameters according to our analysis. Overall, our activity detector consumes 2.7 to 21 times less memory space than the CES-based method.

Fig. 13 shows the memory consumption with respect to N_a. For detailed analysis, we separately measure the shared region index size only, i.e., the Spatial Border Index for ActraMon and CES itself for the CES-based method. As presented in Fig. 13a, the memory consumption of CES drastically increases with N_a while that of ActraMon increases barely. It is mainly because an activity region is repeatedly stored in multiple grids for fast searching, although CES employs an optimized memory management scheme based on hierarchical grids. Whereas a tree-base index [15] can reduce the storage cost, it slows down the processing by traversing the tree from the root. For ActraMon, on the other hand, the Spatial Border Index consumes much smaller storage, which amounts to less than 0.5 percent of the index size in CES for 70K operators. The concise index storing four borders only for each activity region reduces storage consumption significantly.

Fig. 13b shows total storage sizes including the temporal border information stored in the Temporal Border Index. The information amounts up to 1,249 MB for both cases equally. It covers 32 to 38 percent of the total memory consumption. However, the size of the information is expected to be saturated since an object would conduct a limited number of simultaneous activities for a certain time. While not reported, the results of experiments with varying R_w also show the same patterns to those in Fig. 13.

5.2 Shared Activity Composer

5.2.1 Experimental Setup

The activity composer takes activity instances (generated by the activity detector) and a set of composition operators as its input. Accordingly, we create activity instance and composition operator workloads for the experiment.

Composition operator workload. To evaluate the performance of the activity composer, we consider the SEQ operators, the main composition operator of ATP, as the basis for performance comparison. We generate SEQ operator workloads with the parameter ranges and default values as presented in Table 3. Given a set of primitive activities, we first control the number of SEQ operators, N_s, and the length of sequencing steps, n_s, to evaluate the effect of ACQ sharing, while the time window, T_t, is determined by n_s.

Activity instance workload. We simulate moving objects to generate activity instances following the sequences of the synthesized SEQ operators. Each object selects a SEQ operator randomly. The object generates a series of the activity instances which completely satisfy the activity sequence of the operator. If the generation is completed for the SEQ operator, the same process is repeated with another operator. In this experiment, each object is supposed to generate 1K activity instances by default, while we control the number of moving objects, N_o, to evaluate the scalability.

Alternative technique (NFA-based method). For comparison, we implement an automata-based event composition method, which is commonly used for event sequencing

TABLE 3
Parameters (Activity Composer Experiment)

Parameter	Meaning	Range	Default Value
N_s	Number of SEQ operator	10K – 70K	40K
n_s	Sequencing steps	2 – 14	5
T_t	Sequencing time window	(n_s+1) × 120 min	N/A
N_o	Number of objects	10K – 70K	10K
N_c	Memory cleaning period	10K – 400K	100K

[16], [18], [19]. The method creates a Nondeterministic Finite Automata for each SEQ operator. In addition, *activity instance stacks* are managed for each object to store the activity instances and keep track of the state efficiently [18]. Upon arrival of an activity instance, the method first identifies corresponding NFAs. Then, it tests state transitions in each NFA to evaluate the sequencing. If the transition is made, the instance is inserted into the stacks. When the transition reaches the final states of the NFAs, the stacks are reversely searched to construct the resulting sequence information. To support a number of NFAs, we employ a hash index on them, which enables activity instances to be dispatched efficiently to the related NFAs only. We call this method as the NFA-based method.

Memory cleaning period. Cleaning obsolete activity instances is critical for in-memory processing of the composite queries. It has significant impact on the processing as well as storage cost. We observe there exists a trade-off between the cleaning time and memory usage. Too frequent cleaning causes unnecessary scanning with rarely effective deletions, resulting in the excessive cleaning time. Also, it is hardly beneficial to defer cleaning beyond some extent due to huge memory consumption. The details of the experiment results can be found in [29, Section 5.2.1]. From the results, we set the base cleaning period to 100K in following experiments where the cleaning time starts to be saturated while the memory consumption increases linearly.

5.2.2 Processing Performance

We measure the elapsed times to process the simulated activity instance workloads, while varying N_s, N_o, and n_s.

Scalability with operators and sharing effect. We first demonstrate the scalability of the activity composer by increasing the number of SEQ operators, N_s. It is likely for a primitive activity to become shared more as more composition operators are employed in an urban area with a set of designated activities. Hence, assuming 5,000 primitive activities in an area, we study the scalability as increasing N_s, which also increases D_s, the degree of ACQ sharing, accordingly. Fig. 14a shows the results. We measure the processing time taken for activity composition (denoted as *Comp*) and memory cleaning (denoted as *Clean*) separately. Fig. 14a shows that our activity composer is faster than the NFA-based method with large N_s's. The performance benefit is mainly achieved by ACQ sharing and cleaning efficiency. Even for large N_s's, the processing is successfully localized to a single ACQ and a limited number of its adjacent ACQs. In specific, the cost of the *insert()* invocations is almost constant for each run, since an activity instance is inserted once at most. The increasing cost factor is the number of the *probe()* invocations, as an ACQ

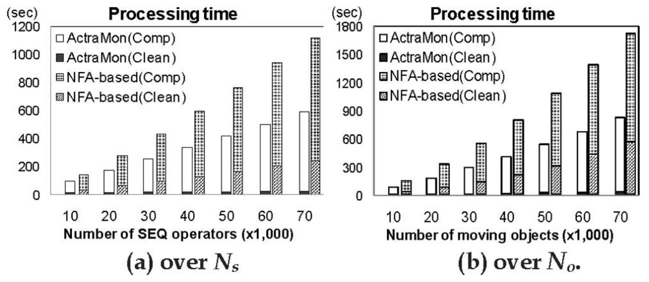


Fig. 14. Processing time of activity composer.

participates in more compositions for large N_s's. Meanwhile, the cleaning overhead of our composer appears almost negligible in the figure.

For the NFA-based method, on the other hand, the cleaning time increases substantially with N_s. It is an inherent cost to scan more NFAs accordingly to larger N_s. For composition, the NFA-based method also incurs processing overhead separately for each NFA without sharing. For large N_s's, much more NFAs are likely to be evaluated upon arrival of each activity instance.

We also evaluate the effect of the length of sequencing steps, n_s, on the processing time. With varying n_s, ActraMon consistently takes much less processing time than the NFA-based method (See [29, Section 5.2.2]).

Scalability with moving objects. We demonstrate the scalability with the number of moving objects, N_o. Note that the workload size increases proportionally to N_o, since each moving object is configured to generate 250 activity instances in this experiment. Fig. 14b shows that the processing time of ActraMon is linearly proportional to N_o. According to our analysis, the processing cost of the activity composer depends on the degree of ACQ sharing, D_s, only. Since D_s is fixed, i.e., 40 by default in this experiment, the unit processing time per incoming activity instance remains constant. For the NFA-based method, by contrast, the slope of the line is getting steeper, since the increase in the cleaning time becomes more significant for large N_o's. As N_o increases, more per-object stacks are created as well as scanned for cleaning. Consequently, this result shows that our activity composer is much more scalable with respect to the number of moving objects owing to the shared queue management.

5.2.3 Storage Cost

We evaluate the storage cost of the activity composer by measuring the average amount of memory consumption under the same experimental setting in Section 5.2.2. We periodically measure the memory consumption before and after memory cleaning. We take the minimum, maximum, and average values after they are stabilized.

Scalability with operators and sharing effect. We also investigate the scalability in terms of memory consumption with an increasing number of SEQ operators. Fig. 15a shows that our activity composer uses much less memory (from 60 to 80 percent reduction) than the NFA-based method does. The storage efficiency is mainly achieved by ACQ sharing. As far as the number of primitive activities is fixed, the same number of ACQs is created regardless of the number of SEQ operators, N_s. Even for large N_s's, the increase in storage consumption is restricted to a small part of ACQ,

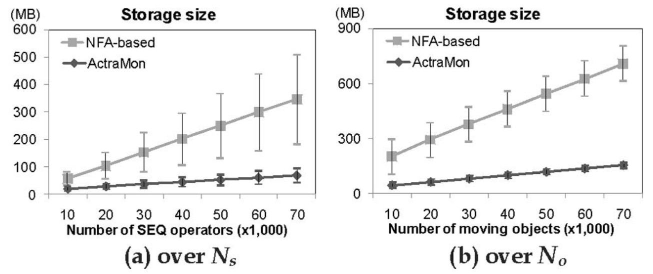


Fig. 15. Memory consumption of activity composer.

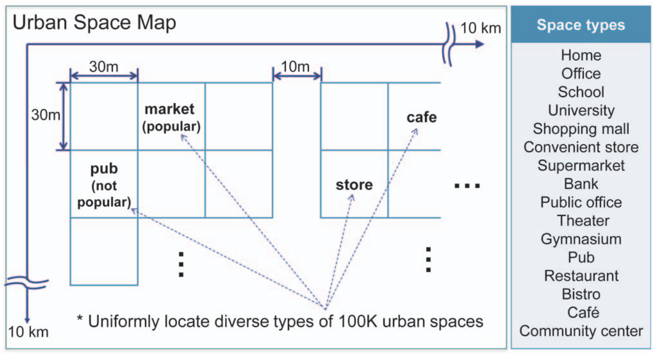


Fig. 16. Example of the urban space map.

i.e., the Composition Link Table and the Partial Composition Blocks. Thus, the increase becomes moderate. On the other hand, the NFA-based method creates an NFA and multiple activity instance stacks for each SEQ operator and does not share any data structures among them. Thus, the storage size increases substantially proportional to N_s.

From a separate experiment, we observed that, in ActraMon, the storage consumption hardly increases with the length of sequencing steps, n_s, while it does rapidly in the NFA-based method (See [29, Section 5.2.3]).

Scalability with moving objects. Fig. 15b compares the storage size as increasing N_o. For both ActraMon and the NFA-based method, the storage size is linearly proportional to N_o, since they are required to store more activity instances within the time windows specified in SEQ operators. The figure also shows that ActraMon reduces the memory consumption by up to 78 percent, compared to the NFA-based method. The NFA-based method maintains separate stacks for moving objects whereas ActraMon employs unified instance queues shared by the objects.

5.3 Overall ATP Query Processing

5.3.1 *Experiment Setup with Activity-Based City Model*

We report performance study on overall ATP query processing in combination with the activity detection and composition. For the experiment, we develop a well-crafted city model and produce realistic location and query workloads based on the model.

The city model consists of an *urban space map* and an *ATP model* of city residents. The urban space map represents a city that includes diverse types of spaces such as supermarkets and theaters. Fig. 16 shows an example of the map. We first divide the whole map into a number of square blocks and uniformly locate diverse types of 100K urban spaces into the blocks. The portion of space types is realistically determined by the statistics of Seoul city [26]. Also, we associate the degree of popularity with each urban space, which indicates people's level of interest on each space.

TABLE 4
Parameters for ATP Query Generation

Query template
SEQ(ACT$_1$(R$_1$, T$_{min\_1}$, T$_{max\_1}$), ..., ACT$_n$(R$_n$, T$_{min\_n}$, T$_{max\_n}$), T$_t$)

Parameter	Meaning	Range	Default Value
N_o	Number of objects	10K ~ 50K	10K
N_s	Number of SEQ queries	1K ~ 7K	2K
n_s	Sequencing step	N/A	2 ~ 3
D_s	Degree of Sharing	N/A	1 ~ 20
R_i	Region of ACT operator, ACT_i, in SEQ	N/A	Designated urban space for the activity
T_{min_i} T_{max_i}	Time conditions of ACT operator, ACT_i, in SEQ	N/A	Refer to the time duration value for the activity in Table 4

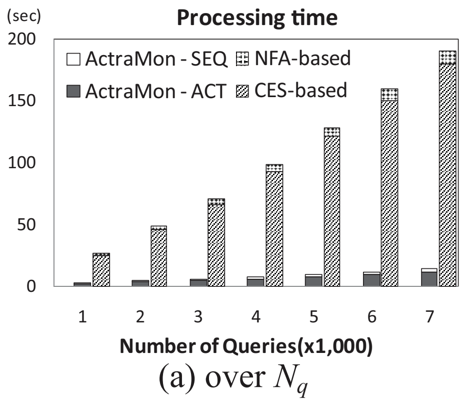
The ATP model describes which activities individuals perform, in which order, and which specific urban space they utilize for each activity. It also models how individuals travel from a space to a next space. We establish the ATP model based on the space-time use data studied in [24], [25]. The data describe statistics about what types of activities people do a day and how long and often they perform each type of the activities. Additionally, the model matches activity types to corresponding urban space types. The data that we use in this experiment have been summarized in [29, Table 5].

Specifically, the ATP model generates the activity travel sequence of an individual following the three steps: 1) *next activity selection*, 2) *space selection*, and 3) *moving method selection*. First, the model probabilistically determines the next activity of an individual based on the frequency of the activity types. Second, the model determines a specific urban space in the map for the selected activity. The space is selected more likely as its popularity value is higher and the distance to the previous space is smaller. Finally, the model determines how individuals move to the next space. If the distance between the two spaces is larger than 2 km, we assume that they use a car at the average speed of 20 km/h. If not, they walk at 4 km/h.

Location data generation. Based on the city model, we generate location traces of 50K moving objects. Each object follows one of the activity travel sequences generated by the ATP model; it stays and travels the urban spaces matching to the activity sequence. The location data are created every 30 seconds for each object, and 4-hour amount of location data are generated.

Query generation. We generate ATP queries that monitor the traces of the moving objects produced by the city model. We create two types of SEQ queries with the query template in Table 4 and use them in equal proportion. Table 4 also summarizes the parameters for the ATP query generation.

For the first type, SEQ queries are generated to detect frequent activity travel sequences of the moving objects. This type simulates the queries for mobile advertisement to people following the frequent patterns. For the generation, we first extract a number of frequent patterns in the activity travel sequences of individuals; we extract 3,500 sequential patterns with 2 or 3 activity steps, using an implementation of PrefixSpan algorithm [38]. Each activity



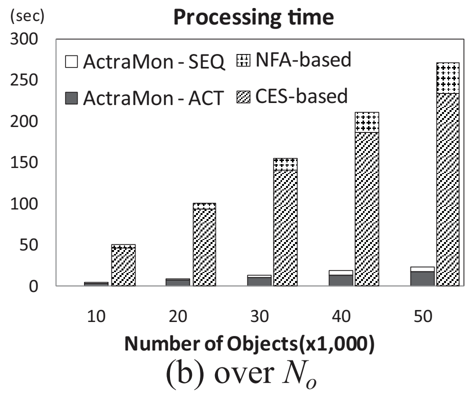
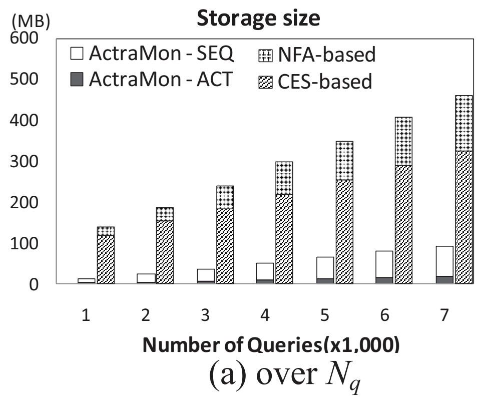


Fig. 17. Processing performance of ActraMon.



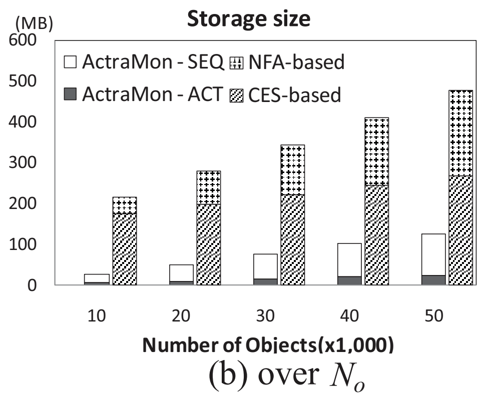


Fig. 18. Memory consumption of ActraMon.

in the patterns is instantiated as an ACT operator, ACT_i; the spatial condition, R_i, is designated as its corresponding urban space, and the temporal conditions, T_{min_i} and T_{max_i}, are determined based on its activity type. A SEQ operator is created for each extracted pattern, including its ACT operators; the time window, T_t, is set to include the duration of the activities and travels in the pattern.

For the second type, we target the queries that monitor larger regions for public transportation planning. For the query generation, we first randomly designate 500 interest regions with the sizes up to 4 km and create an ACT operator for each region. Then, we create SEQ queries combining 2 or 3 of the ACT operators in sequences. The temporal conditions of the operators are designated randomly less than 4 hours.

Alternative Technique. $\mathrm{SEQ}(\mathrm{ACT_1}, \ldots, \mathrm{ACT_n})$ queries are processed with the combination of CES-based activity detection and NFA-based activity composition. We call this method CES+NFA.

5.3.2 Processing Performance

First, we show the scalability of overall ATP query processing as increasing the number of ATP queries, N_q, from 1 to 7K. The number of objects, N_o, is set to 10K. Fig. 17a clearly shows that ActraMon is highly scalable with respect to N_q. The processing time of ActraMon slightly increases for a larger N_q, whereas that of CES+NFA increases significantly. The result conforms to our expectation derived from the previous experimental results presented in Sections 5.1 and 5.2. When N_q reaches 7K, CES+NFA takes about 14 times more time than ActraMon to process the same amount of location updates.

The figure also shows the activity detection takes the major portion of the processing time for both ActraMon and CES+NFA. This result demonstrates that the staging architecture facilitates filtering out a significant amount of location data at the early stage. Especially for ActraMon, the filtering is much more efficient by the incremental evaluation through the stateful index. Compared to CES+NFA, the high performance filtering technique greatly improves the overall processing performance.

Next, we increase the number of moving object, N_o, from 10 to 50K, where N_q is fixed to 2K. Fig. 17b shows that ActraMon is highly scalable with respect to N_o, whereas the processing time of CES+NFA increases drastically. The result also agrees with the previous experimental results of the activity detector and the composer.

Interestingly, the both figures show that the performance difference between ActraMon and CES+NFA is unexpectedly significant with the city model-based workload. The difference stems mainly from the fact that such realistic location streams feature relatively high locality; frequent activities have relatively long duration, and the next spaces to travel are often nearby in the city map. Such location streams are best suited for the high-performance filtering in the activity detection of ActraMon. As emphasized prior, the large portion with the detection stage doubles up the performance benefits of ActraMon.

5.3.3 Storage Cost

Fig. 18a shows the memory consumption as increasing the number of ATP queries, N_q, from 1 to 7K. ActraMon achieves a high level of memory efficiency since the activity detector only maintains the temporal and spatial borders in ABI and the activity composer shares ACQs for multiple composition queries and moving objects. Instead, CES+NFA consume more memory space since CES stores the query information in the grids repeatedly and NFAs store activity instances multiple times. Interestingly for ActraMon, the activity composer consumes more memory space than the activity detector, which is opposite to the result of processing time. The activity detector stores the border crossing information only, whereas the activity composer needs to maintain the activity instances and the partial composition information for a large time window.

We also increase the number of moving objects, N_o, from 10 to 50K. Fig. 18b shows that ActraMon consumes less than one third memory space that CES+NFA does. The increase of the NFA-based method is noticeable compared to that for a large N_q. The result indicates that the per-object stack management of the NFA-based method [18] often results in severe overhead in terms of storage cost with a large number of objects. Note that ActraMon only uses about 120 MB to deal with 50K objects, thanks to the shared data structures, whereas CES+NFA consumes up to 500 MB.

5.3.4 Discussion on Scalability

From the experiments, we confirm that ActraMon shows quite good scalability in terms of processing time and storage cost. Meanwhile, for a large scale city, the numbers of objects, activities, and composition operators could reach up to nearly millions. Even our system could not efficiently handle such a case with a single server. Fortunately, however, the proposed architecture is inherently well suited to a distributed processing since the processing can be individually divided by each object. That is, location stream data for each object are handled by one of processors which identically evaluate a duplicate set of same queries. In particular, we note that the number of objects strongly affects the storage cost as well as the processing time. For instance, although the size of ABI is quite small in the detection stage, the total memory consumption significantly increases with the number of objects. Thus, if we distribute objects over multiple servers, object-related storages as well as computation are naturally distributed to a server responsible for each object. We can also adapt a computational load to the capability of each server.

We expect that a single server with giga-byte memory could efficiently deal with a quite high number of objects and operators up to hundreds of thousands. If the memory cannot cope with total processing storage, some of that should be spilled over into a disk. Then, the limited disk bandwidth would impede the real-time processing significantly. Thus, we need to redesign the in-memory data structures well suited to such a disk-based system, and further develop aggressive caching schemes which exploit in-memory processing with disk-based indices.

6 Conclusion

This paper introduces ATP monitoring in large-scale city environments. To enable ATP monitoring, we develop ActraMon, a high-performance ATP monitoring framework. ActraMon is an initial attempt to support real-time city-scale services leveraging collective intellegence from mobile devices of city residents. Based on the computational model of ATPs, ActraMon provides a declarative query language that facilitates effective specification of various ATPs. Most important, it provides the shared staging architecture and efficient processing techniques for activity detection and composition. ActraMon effectively addresses the scalability challenges caused by massive input workloads and processing complexity of ATP monitoring. Our experimental results show that ActraMon is scalable to the number of ATP queries as well as moving objects, compared to the state-of-the art processing techniques.

Acknowledgments

The authors' special thanks go to professor K. Selcuk Candan for his valuable suggestions and discussion to improve the quality of this paper. This work was supported by the NRF of Korea under grants No. 2011-0018120 and No. 2011-0020519, ITRC supported by the Ministry of Knowledge Economy of Korea (NIPA-2011-C1090-1111-0004), and Hankuk University of Foreign Studies Research Fund of 2011.

References

[1] R. Kitamura, "An Evaluation of Activity-Based Travel Analysis," *Transportation,* vol. 15, pp. 9-34, 1988.

[2] C.H. Wen and F.S. Koppelman, "A Conceptual and Methodological Framework for the Generation of Activity-Travel Patterns," *Transportation,* vol. 27, pp. 5-23, 2000.

[3] T. Arentze and H.J.P. Timmermans, "A Learning-Based Transportation Oriented Simulation System," *Transportation Research Part B,* vol. 38, pp. 613-633, 2004.

[4] R.N. Buliung, P.S. Kanaroglou, and H.F. Maoh, "GIS, Objects and Integrated Urban Models," *Integrated Land-Use and Transportation Models: Behavioural Foundations,* pp. 207-230, Elsevier, 2005.

[5] R. Lange, F. Durr, and K. Rothermel, "Scalable Processing of Trajectory-Based Queries in Space-Partitioned Moving Objects Databases," *Proc. 16th ACM SIGSPATIAL Int'l Conf. Advances in Geographic Information Systems (GIS),* 2000.

[6] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approaches to the Indexing of Moving Object Trajectories," *Proc. Very Large Databases (VLDB),* 2008.

[7] M.F. Mokbel, X. Xiong, and W.G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries Spatio-Temporal Database," *Proc. SIGMOD Int'l Conf. Management of Data,* 2004.

[8] B. Gedik and L. Liu, "MobiEyes: A Distributed Location Monitoring Service Using Moving Location Queries," *IEEE Trans. Mobile Computing,* vol. 5, no. 10, pp. 1384-1402, Oct. 2006.

[9] Y. Cai, K.A. Hua, G. Cao, and Y. Xu, "Real-Time Processing of Range-Monitoring Queries in Heterogeneous Mobile Databases," *IEEE Trans. Mobile Computing,* vol. 5, no. 7, pp. 931-942, July 2006.

[10] G.S. Iwerks, H. Samet, and K. Smith, "Continuous K-nearest Neighbor Queries for Continuously Moving Points with Updates," *Proc. Very Large Databases (VLDB),* 2003.

[11] D.J. Adabi et al., "Aurora: A New Model and Architecture for Data Stream Management," *The Int'l J. Very Large Databases,* vol. 12, no. 2, pp. 120-139, Aug. 2003.

[12] S. Chandrasekaran and M.J. Franklin, "Streaming Queries over Streaming Data," *Proc. Very Large Databases,* 2002.

[13] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Queries Language: Semantic Foundations and Query Execution," *Very Large Databases J.,* vol. 15, no. 2, pp. 121-132, June 2006.

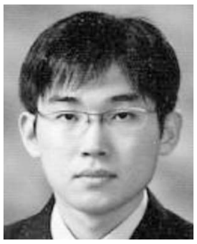
[14] B. Gedik, K. Wu, P.S. Yu, and L. Liu, "Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes," *IEEE Trans. Knowledge and Data Eng.,* vol. 18, no. 4, pp. 651-688, May 2006.

[15] H. Hu, J. Xu, and D. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," *Proc. SIGMOD Int'l Conf. Management of Data,* 2005.

[16] S. Chakravarthy and R. Adaikkalavan, "Ubiquitous Nature of Event-Driven Approaches: A Retrospective View (Position Paper)," *Dagstuhl Seminar 07191,* 2007.

[17] L. Bao and S.S. Intille, "Activity Recognition from User-Annotated Acceleration Data," *Proc. Pervasive,* 2004.

[18] E. Wu, Y. Diao, and S. Rizvi, “High-Performance Complex Event Processing over Streams,” *Proc. SIGMOD Int'l Conf. Management of Data,* 2006.
[19] A. Demers, J. Gehrke, and B. Panda, “Cayuga: A General Purpose Event Monitoring System,” *Proc. Third Biennial Conf. Innovative Data Systems Research (CIDR),* 2007.
[20] J. Agrawal, D. Gyllstrom, Y. Diao, and N. Immerman, “Efficient Pattern Matching over Event Streams,” *Proc. SIGMOD Int'l Conf. Management of Data,* 2008.
[21] M.F. Mokbel, X. Xiong, M.A. Hammad, and W.G. Aref, “Continuous Query Processing of Spatio-Temporal Data Streams in PLACE,” *Proc. Int'l Workshop Spatio-Temporal Database Management (STDBM),* 2004.
[22] K. Partridge and P. Golle, “On Using Existing Time-Use Study Data for Ubiquitous Computing Applications,” *Proc. 10th Int'l Conf. Ubiquitous Computing (UbiComp),* 2008.
[23] J. Lee, S. Kang, Y. Lee, S. Lee, and J. Song, “BMQ-Processor: A High-Performance Border-Crossing Event Detection Framework for Large-Scale Monitoring Applications,” *IEEE Trans. Knowledge and Data Eng.,* vol. 21, no. 2, pp. 234-252, Feb. 2009.
[24] A. Kulkarni and M.G. McNally, “An Activity-Based Travel Pattern Generation Model,” Center for Activity Systems Analysis, Paper UCI-ITS-AS-WP-00-6, 2000.
[25] R. Kitamura, C. Chen, R.M. Pendyala, and R. Narayanan, “Micro-Simulation of Daily Activity-Travel Patterns for Travel Demand Forecasting,” *Transportation,* vol. 27, pp. 25-51, 2000.
[26] Statistics of Seoul, http://www.kosis.kr/eng, 2011.
[27] K.L. Wu, S.K. Chen, and P.S. Yu, “On Incremental Processing of Continual Range Queries for Location-Aware Services and Applications,” *Proc. Second Ann. Int'l Conf. Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous),* 2005.
[28] Random Walk Model, http://en.wikipedia.org/wiki/Random_walk, 2011.
[29] Y. Lee, B. Kim, S. Lee, J. Kim, Y. Rhee, and J. Song, “ActraMon: Scalable Activity-Travel Pattern Monitoring Framework in Metropolitan City Environments,” KAIST technical report, CS-TR, pp. 2009-309, 2009.
[30] S. Kang et al., “SeeMon: Scalable and Energy-Efficient Context Monitoring Framework for Sensor-Rich Mobile Environments,” *Proc. Mobile Systems, Applications, and Services (MobiSys),* 2008.
[31] L. Liao, D. Fox, and H. Kautz, “Location-Based Activity Recognition Using Relational Markov Networks,” *Proc. 19th Int'l Joint Conf. Artificial Intelligence (IJCAI),* 2005.
[32] T. Hägerstrand, “What about People in Regional Science?” *Papers in Regional Science,* vol. 24, no. 1, pp. 7-24, 1970.
[33] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, “Composite Events for Active Databases: Semantics, Contexts and Detectiion,” *Proc. Int'l Conf. Very Large Databases,* 1994.
[34] A.K. Dey and G.D. Abowd, “CybreMinder: A Context-Aware System for Supporting Reminders,” *Proc. Handheld and Ubiquitous Computing (HUC),* 2000.
[35] B. Bamba, L. Liu, A. Iyengar, and P.S. Yu, “Distributed Processign of Spatial Alarms: A Safe Region-Based Approach,” *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS),* 2009.
[36] A. Murugappan and L. Liu, “An Energy Efficient Middleware Architecture for Processing Spatial Alarms on Mobile Clients,” *J. Mobile Networks and Applications,* vol. 15, no. 4, pp. 543-561, 2010.
[37] S. Urban, I. Biswas, and S.W. Dietrich, “Filtering Features for a Composite Event Definition Language,” *Proc. Int'l Symp. Applications and the Internet (SAINT),* 2006.
[38] P. Han et al., “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,” *Proc. Int'l Conf. Data Eng.,* 2001.
[39] J. Lee, Y. Lee, S. Kang, S. Lee, H. Jin, B. Kim, and J. Song, “BMQ-Index: Shared and Incremental Processing of Border Monitoring Queries over Data Streams,” *Proc. Seventh Int'l Conf. Mobile Data Management (MDM),* 2006.
[40] S. Lee, Y. Lee, B. Kim, K.S. Candan, Y. Rhee, and J. Song, “High-Performance Composite Event Monitoring System Supporting Large Numbers of Queries and Sources,” *Proc. Fifth ACM Int'l Conf. Distributed Event-Based Systems (DEBS),* 2011.
[41] B. Kim, J. Ha, S. Lee, S. Kang, Y. Lee, Y. Rhee, L. Nachman, and J. Song, “AdNext: A Visit-Pattern-Aware Mobile Advertising System for Urban Commercial Complexes,” *Proc. ACM HotMobile,* 2011.
[42] M. Al-Kateb and B. Lee, “Load Shedding for Temporal Queries over Data Streams,” *J. Computing Science and Eng.,* vol. 5, no. 4, pp. 294-304, Dec. 2011.


Youngki Lee received the BS degree in computer science from KAIST, Daejeon, South Korea, in 2004, where he is currently working toward the PhD degree. His research interests include mobile and ubiquitous computing systems, system support for context awareness, high-performance systems for city-scale ubiquitous services, and large-scale distributed systems and networking.

SangJeong Lee received the BS degree in physics and the MS degree in electrical engineering from KAIST, Daejeon, South Korea, in 1999 and 2001, respectively. He is working toward the PhD degree at KAIST. His research interests include mobile and pervasive applications and systems, and system supports for context-awareness and future games.

Byoungjip Kim received the BS and MS degrees in electrical engineering and computer science from KAIST in 2002 and 2004, respectively. He is currently working toward the PhD degree in computer science at KAIST. His research interests include mobile and ubiquitous computing, Internet computing, and data stream processing.

Jungwoo Kim received the BS and MS degrees in computer science from KAIST, Daejeon, South Korea, in 2010. His research interests include Internet service and technologies, user experience, social media, and ubiquitous computing.

Yunseok Rhee received the PhD degree in computer science from KAIST in 1999. He is a professor in the School of Electronics and Information Engineering, Hankuk University of Foreign Studies, South Korea. Before joining the faculty, he worked at Systems Engineering Research Institute, Korea, as a research staff member from 1988 to 1994. His research interests include distributed computing, embedded systems, and Internet technologies.

Junehwa Song received the PhD degree in computer science from the University of Maryland at College Park in 1997. He is a professor in the Department of Computer Science at KAIST, Daejeon, South Korea. Before joining KAIST, he worked at the IBM TJ Watson Research Center, Yorktown Heights, New York, as a research staff member from 1997 to 2000. His research interests include mobile and ubiquitous computing systems, Internet technologies such as intermediary devices, high-performance web serving, electronic commerce, and distributed multimedia systems. He is a member of the IEEE.