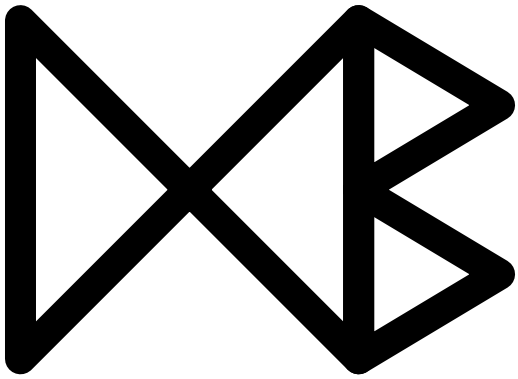


A Streams-Based Framework for Defining Location-Based Queries

Xuegang Huang, Christian S. Jensen

April 3, 2007

TR- 19

A DB Technical Report

Title	A Streams-Based Framework for Defining Location-Based Queries
Author(s)	Xuegang Huang, Christian S. Jensen
Publication History	April 2007. A DB Technical Report. Extended version of: X. Huang, C. S. Jensen, "Towards a Streams Based Framework for Defining Location-Based Queries," in *Proceedings of the Second Workshop on Spatial-Temporal Database Management*, Toronto, Canada, August 2004, pp. 78–85.

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

An infrastructure is emerging that supports the delivery of on-line, location-enabled services to mobile users. Such services involve novel database queries, and the database research community is quite active in proposing techniques for the efficient processing of such queries. In parallel to this, the management of data streams has become an active area of research.

While most research in mobile services concerns performance issues, this paper aims to establish a formal framework for defining the semantics of queries encountered in mobile services, most notably the so-called continuous queries that are particularly relevant in this context. Rather than inventing an entirely new framework, the paper proposes a framework that builds on concepts from data streams and temporal databases. Definitions of example queries demonstrates how the framework enables clear formulation of query semantics and the comparison of queries. The paper also proposes a categorization of location-based queries.

Keywords: Location-based service, data stream, continuous query, skyline query, range query, nearest-neighbor query.

1 Introduction

The emergence of mobile services, including mobile commerce, is characterized by convergence among new technologies, applications, and services. Notably, the ability to identify the exact geographical location of a mobile user at any time opens to range of new, innovative services, which are commonly referred to as location-based services (LBSs) or location-enabled services.

In an LBS scenario, the service users are capable of continuous movement, and changing user locations are sampled and streamed to a processing unit, e.g., a central server. The notion of a data stream thus occurs naturally. Service requests result in queries being issued against the data streams and other, typically relational, data.

Conventional queries are one-time queries, i.e., queries that are simply issued against the state of the database as of the time of issue, upon which they, at a single point in time, return a result. In our scenario, so-called continuous queries are also natural. Such queries are "active" (i.e., being re-evaluated) for a duration of time, and their results are kept up-to-date as the database changes during this time. As an example, an in-vehicle service may display the three nearest, reasonably priced hotels with rooms available along the route towards the vehicle's destination. The vehicle's location (a data stream) together with data about hotels (relational data) are continuously queried to provide the result (a data stream).

Significant results on the processing of location-based queries (LBQs) has already been reported. As LBQs are defined in different settings, no direct means are available for classifying and comparing these queries. As more and more work, considering more and more different kinds of queries, is reported, the need for comparison increases.

This paper, as an extended version of [24], presents a general framework within which the semantics of LBQs can be specified. This enables the definition of LBQs in a single framework, which in turn enables the comparison of queries. The framework is well defined—it is based on precise definitions of data structures and operations on these. The framework has the following characteristics.

- Streams as well as relations are accommodated.
- Because queries often involve ranked results, relations are defined to include order.
- Relational algebraic operators are extended to also apply to streams, by using mappings of streams to relations, and, optionally, mappings of relations to streams.

The result is an expressive yet semantically simple framework that may be extended with additional operators and mappings. To illustrate the extensibility, a new operator, the skyline operator, is introduced.

Rather than listing and defining all possible location-based queries, this paper represents several prominent location-based queries that have been under active discussions in the research community and then considers categorizations of LBQs.

The research area of stream data is quite active and has produced a number of interesting concepts in relation to the semantics of continuous queries. Specifically, significant research results have been reported on query processing for data streams (e.g., [5, 10, 42, 55]). Some works consider queries over data streams together with relations (e.g., [2, 35]), but only few works consider the formalization of queries over streams and relations.

Similarly, location-based query processing is an active area of research, and many interesting results have appeared. Much attention has been given to the indexing and query processing for moving objects. Numerous

index structures and algorithms have been proposed for a variety of location-based queries (e.g., [6, 17, 36, 40, 43, 45, 47, 48, 53]), such as nearest neighbor queries, reverse neighbor queries, spatial range queries, distance joins, and closest-pair queries. A new type of query, the skyline query, has recently received attention [7, 15, 31, 41]. However, only little attention has been paid to discuss formal frameworks for defining of location-based queries against relations and data streams.

Recently, Arasu et al. [2, 4] have offered an interpretation of continuous queries over streams, by formalizing streams, relations, and mapping operators among them. We build on their general approach. To accommodate ordering as well as duplicates, we use list-based relations and a variant of the list-based relational algebra proposed by Slivinskas et al. [46]. To be able to express query semantics precisely, our approach also accommodates the notions of activation and deactivation times and reevaluation granularity.

The paper is outlined as follows. Section 2 introduces related works. Section 3 defines the data structures underlying the framework and presents the application scenario. The next section completes the framework, by defining the operators that map between the different operators in the framework. Section 5 uses the framework to define different location-based queries and also discusses the categorization of location-based queries. The last section summarizes and offers directions for future research.

2 Related Work

Numerous works have been conducted on topics in the data stream management. In the Tapestry project [51], data streams are modelled as append-only databases and support continuous queries. Due to the fact that an effective data stream management system requires extensive modifications to traditional database systems. Many academic projects have been generated, e.g., the Aurora [10] system for workflow-processing, the COUGAR [9] system for sensor databases, the Gigascope [16] system that provides a distributed network monitoring architecture, the NiagaraCQ [12] and OpenCQ [33] systems that are designed for monitoring dynamic Web content, the StarStream [58] system that computes on-line statistics across many streams, the STREAM [2, 5] system which is a general-purpose stream processing system, and the TelegraphCQ [11] system that focuses on adaptive query processing. Also, the Tribecca [44] system is an early on-line Internet traffic monitoring tool.

Three types of paradigms have been proposed for querying data streams [21], i.e., relation-based system, object-based system, and procedural system. Among all the systems and languages, we are particular interested in the semantics used by the continuous query language (CQL) in the STREAM system. It includes definitions of streams, relations and mapping operators among them [2]. With the abstract semantics, any continuous query can be constructed from three building blocks, i.e., any relational query languages, a window specification language, and three relation-to-stream operators. Having the three building blocks, a continuous query can simply be assembled in a type-consistent way from streams, relations, and the mapping operators. Since the building blocks are open to new elements, extensibility is allowed in the semantics. The simple way of expressing continuous queries makes the CQL language easily applicable to any specific scenario. Thus, in this paper, we choose to extend the abstract semantics defined in CQL to the location-based query processing scenario. To follow the denotational semantics proposed [4], we use the list-based relations and a variant of the list-based relational algebra proposed by Slivinskas et al. [46]

Location-based queries can be seen as spatial and spatio-temporal queries in the context of location-based service. One of the most popular queries, is the $k-$nearest neighbor (kNN) searching [47], which retrieves the k nearest neighbors to a given source point in certain spatial scenario. Quite a few extensions have been made to the kNN problem, e.g., reverse k nearest neighbor [6], continuous k nearest neighbor [53, 29], constrained nearest neighbor [19], group nearest neighbor [40], all nearest neighbor [56]. Since R-tree [22] and R*-tree [8] have already been accepted as general ways for spatial data storage and access, the so-called window query and range query, have been frequently discussed and analyzed in the research literature [8] since the structures of R-tree and its variants are naturally designed for such range search. Spatial join [27, 30, 38] is a type of query that involves join operation on multi-sets of tuples. Discussions have been made on several extensions of spatial joins, such as distance join [27] operators where the join output is ordered by the distance between the spatial attributes of the joined tuples and the spatial closest pair query [17] that has been proposed for finding the k closest pairs between two spatial data sets. In addition, spatio-temporal aggregation [57, 59] is another topic that has been discussed in the literature.

With the popularity of stream processing, research works in the spatio-temporal area begin to consider the data

management and query processing in presence of data streams [23, 37, 39]. However, no formalized framework has been proposed to express location-based queries with respect to stream data. In this paper, we will present a streams-based framework and present definitions of several prominent location-based queries based on the proposed semantics. To show the extensibility of our framework, we present the definition of the skyline operator [7] within the framework and utilize the operator for defining a location-based skyline query. Another motivation is that the skyline operation has been frequently-discussed [15, 31, 41, 50] and is shown to be useful in LBS scenario [25].

3 Data Structures and Application Scenario

3.1 Data Model Definition

Building on the relation concept defined by Slivinskas et al. [46], we define relations as lists to capture duplicates and ordering. We define schemas, tuples, and relation instances, then define the same concepts for streams.

Definition 3.1. A *relation schema* (Ω, Δ, dom) is a three-tuple where Ω is a finite set of attributes, Δ is a finite set of domains, and $dom : \Omega \to \Delta$ is a function that associates a domain with each attribute.

usr_id	usr_name
1001	Adam
1002	Brain
1003	Clark
1004	Debby

Figure 1: Relation r_{usr}

Relation r_{usr} in Figure 1 has schema (Ω, Δ, dom), where $\Omega = \{usr\_id, usr\_name\}$, $\Delta = \{\text{number}, \text{string}\}$, and $dom = \{(usr\_id, \text{number}), (usr\_name, \text{string})\}$.

Definition 3.2. A *tuple* over *schema* $\mathcal{S} = (\Omega, \Delta, dom)$ is a function $t : \Omega \to \bigcup_{\delta \in \Delta} \delta$, such that for every attribute A of Ω, $t(A) \in dom(A)$. A *relation* over $\mathcal{S}$ is a finite sequence of tuples over $\mathcal{S}$.

The definition of a relation corresponds to the definition of a list or a sequence. A relation can thus contain duplicate tuples, and the ordering of tuples is significant. Relation r_{usr} from Figure 1 is the list $\langle t_1, t_2, t_3, t_4 \rangle$, where, e.g., $t_1 = \{(usr\_id, 1001), (usr\_name, \text{A})\}$.

Definition 3.3. A *stream schema* is a relation schema (Ω, Δ, dom), where Ω includes a special attribute T, Δ includes the time domain $\mathbb{T}$, and $dom(\text{T}) = \mathbb{T}$.

We assume that domain $\mathbb{T}$ is totally ordered. While, for simplicity, we use the non-negative numbers as the time domain in the sequel, other domains may be used. For example, the real or natural numbers, the TIMESTAMP domain of the SQL standard, or one of the domains proposed by the temporal database community may be used.

Stream s_{usr} in Figure 2(b) has schema (Ω, Δ, dom), where $\Omega = \{usr\_id, usr\_v, usr\_loc, \text{T}\}$, $\Delta = \{\text{number}, \text{velocity}, \text{location}, \mathbb{T}\}$, and $dom = \{ (usr\_id, \text{number}), (usr\_v, \text{velocity}), (usr\_loc, \text{location}), (\text{T}, \mathbb{T})\}$.

Definition 3.4. A *stream* is a possibly infinite multiset of tuples over *stream schema* $\mathcal{T}$.

For a stream tuple t_i, the time $\tau_i = t_i(\text{T})$ indicates when the tuple became available in the stream. While a relation is ordered, we have chosen to not introduce an inherent order on streams. Streams come with the natural (partial) order implied by their time attribute.

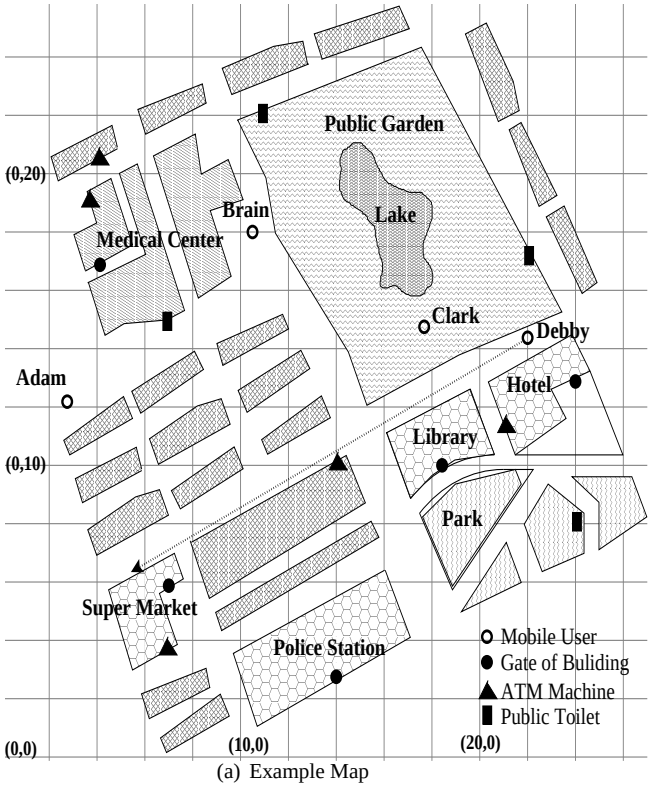
Stream s_{usr} in Figure 2(b) is the possibly infinite multiset $s_{usr} = \{\!\{ \ldots, (1004, (-1.5, -3), (21.5, 14.5), 8), \ldots, (1004, (-5, -8), (5, 6.5), 15), \ldots \}\!\}$.

While a query is issued against an entire relation state, intuitively, a query issued at some time τ_q will only see either what has appeared in the stream so far, i.e., all tuples with timestamp less than or equal to τ_q, or what has appeared in the stream between some past time and τ_q. The latter may be assumed if what has appeared in the stream so far does not fit in the available memory.

3.2 Discussion

As we pointed out earlier, we use streams for modeling the locations of moving objects such as pedestrians, cars, and buses. We use relations for modeling aspects of an application domain that change discretely.

As we aim for a generic framework, we make no assumptions about the representations of the geographical locations and extents of objects that limit the applicability of the framework. However, to be specific, we assume that positions are simply points (x, y) in the two-dimensional map in Figure 2(a); in accord with this, a velocity vector is given by (v_x, v_y). We note that in some application scenarios, positions of objects are given in terms



(a) Example Map

usr_id	usr_v	usr_loc	T
...	...	...	...
1002	$(1, -2)$	$(10.5, 18)$	8
1001	$(1, 2)$	$(2, 12)$	8
1003	$(2, 1)$	$(17.5, 14.5)$	8
1004	$(-2, -1)$	$(21.5, 14.2)$	8
1002	$(3, 8)$	$(12, 15)$	9
1001	$(0.5, 2)$	$(3, 13.5)$	9
1003	$(2.5, 1.5)$	$(21, 16)$	10
1004	$(-3, -1.5)$	$(17.5, 12.2)$	10
1002	$(1, 2.5)$	$(14, 12)$	10
1001	$(0, 1)$	$(3, 15)$	10
1004	$(-6, -3)$	$(14.5, 10.7)$	11
1003	$(-3, 5)$	$(21, 17)$	11
1001	$(1, 0.5)$	$(3, 16.5)$	11
1002	$(-2, 1)$	$(13, 11)$	12
1003	$(-3, 5.5)$	$(19, 21)$	12
1004	$(-1, -0.5)$	$(9.5, 8.2)$	12
1001	$(0, 0)$	$(3.5, 16.5)$	12
1002	$(-4, 5)$	$(10, 9.5)$	13
1004	$(-2, -1)$	$(8.5, 7.7)$	14
1002	$(-8, 8)$	$(7, 8)$	14
1003	$(-3, -5)$	$(15, 23)$	15
1004	$(0, 0)$	$(5.5, 6.2)$	15
1001	$(0, 0)$	$(3.5, 16.5)$	15
...	...	...	...

(b) Stream s_{usr}

Figure 2: Example Scenario

of road networks, using linear referencing [26]. The framework is also applicable in the context of this kind of positioning.

In the example map (Figure 2(a)) we use throughout this paper, stream s_{usr} in Figure 2(b) captures positions and velocities of moving users listed in Figure 1. Attribute usr_id records the ID of a user, and usr_v and usr_loc record the velocity and location of the user at the time instant recorded in attribute T.

obj_id	obj_loc	obj_type
301	$(4, 17)$	medical center
302	$(7, 6)$	supermarket
303	$(14, 2.5)$	police station
304	$(18.5, 10)$	library
305	$(24, 13)$	hotel
306	$(7, 3.5)$	ATM machine
307	$(4, 19)$	ATM machine
308	$(4, 20.6)$	ATM machine
309	$(14, 10)$	ATM machine
310	$(21, 11.5)$	ATM machine
311	$(7, 15)$	public toilet
312	$(11, 22)$	public toilet
313	$(22, 17)$	public toilet
314	$(24, 8)$	public toilet

Figure 3: Relation r_{obj}

In a real-world application, multiple streams may well be present. For example, users moving by bicycle and by car may be captured by separate streams. For simplicity, we only use one stream. The relation r_{usr} in Figure 1 captures discretely changing properties of the service users. The attribute usr_id records the ID of a user; and attribute usr_name capture the first name of a user. In real-world applications, additional attributes and relations may of course be used, beyond the ones introduced above.

Finally, relation r_{obj} in Figure 3 records the points of interest that service users may query. Attribute obj_id captures the ID of a point of interest, obj_loc records its location, and obj_type records its type. For the points of interest such as medical center, super market, police station, library and hotel, we simply use the coordinates of the gates to these buildings as their location. In real world applications, the minimum bounding rectangles (MBRs) of the builds' actual geographical shapes can be used instead.

In our scenario, the users of the services that issue the queries are moving, and the points of interests being queried are static. However, in other equally valid scenarios, a static user can query moving objects, e.g., a supermarket wants to know all the potential customers who are near the supermarket between 8:00 a.m. and 5:00 p.m. Also, a moving user may query other moving users—this may be typical of location-based games.

4 Mapping Operators

Queries are either one-time or continuous, they apply to relations and streams, and their results are either relations or streams.

Relations are well known, and the semantics of queries against relations are generally agreed upon. In contrast, what the appropriate semantics of queries against streams should be and how these should be defined are less obvious. Following Arasu et al. [2], we aim to maximally reuse the relational setting in defining the semantics of

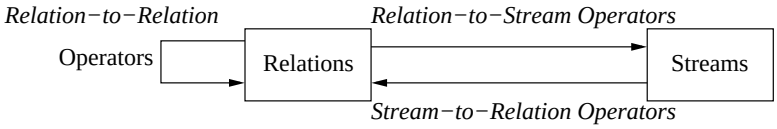


Figure 4: Mapping Operators

queries against streams. We do this by introducing mapping operations between streams and relations, so that a query against a stream can be defined by mapping the stream to a relation, then applying a relational query, and then, optionally, mapping the result to a stream. This results in the framework of representations and operators outlined in Figure 4. Note that direct *stream-to-stream* operators are absent.

4.1 Relation-to-Relation Operators

4.1.1 Basic Algebra Operators

A relation-to-relation operator takes one or more relations $r_1, \cdots, r_n$ as arguments and produces a relation r as a result. As our relations are ordered, we use operators introduced by Slivinskas [46] as our relation-to-relation operators (listed in Figure 5) which are defined with λ-calculus to contend with the order. There are also relation-to-relation operators given in the CQL language [3], i.e., binary-join, mjoin, intersect, antisemijoin. Since these operators are also expressed with λ-calculus elsewhere [4], they can also be included in the framework.

Operator	Denotation
Selection	σ
Projection	π
Union-all	$\sqcup$
Cartesian Product	$\times$
Difference	$\setminus$
Duplicate Elimination	$rdup$
Aggregation	ξ
Sorting	$sort$
Top	top

Figure 5: Basic Algebra Operators

These carry their standard meanings when applied to relations without order. As an example of how the operators are defined, consider selection σ. Based on the definitions in Section 3, we use $\mathcal{R}$ to be the set of all relations and let $r = \langle t_1, t_2, \ldots, t_n \rangle \in \mathcal{R}$. We let $p \in \mathcal{P}$, where (following standard practice) $\mathcal{P}$ is the set of all selection predicates (also termed propositional formulas, see, e.g., [1, pp. 13–14]) that take a tuple as argument and return True or False.

The selection operator $\sigma : [\mathcal{R} \times \mathcal{P} \rightarrow \mathcal{R}]$ is defined using λ-calculus rather than tuple relational calculus, to contend with the order. Being a parameter, argument p is expressed as a subscript, i.e., $\sigma_p(r)$.

$$\begin{aligned}\sigma \triangleq \lambda r, p.(r = \bot) \rightarrow r,\\ (tail(r) = \bot) \rightarrow (p(head(r)) \rightarrow head(r), \bot),\\ (p(head(r)) \rightarrow head(r), \bot) \mathbin{@} \sigma_p(tail(r))\end{aligned}$$

The arguments are given before the dot, and the definition is given after the dot. Thus, if r is empty (denoted as $\bot$), the operation returns it. Otherwise, if r contains only one tuple (the remaining part of the relation, $tail(r)$, is empty), we apply predicate p to the (first) tuple, $(head(r))$. If the predicate holds, the operation returns the tuple; otherwise, it returns an empty relation. If these conditions do not hold, the operation returns the first tuple or an empty relation (depending on the predicate), with the result of the operation applied to the remaining part of r appended ($@$). The common auxiliary functions $head$, $tail$, and $@$ are defined elsewhere (e.g., [46]). Since the objective is to obtain an expressive framework, the framework is kept open to the introduction of such auxiliary functions, although they may increase the conceptual complexity.

4.1.2 Skyline Operator

We proceed to demonstrate how a $skyline$ operator, which is of particular interest in location-based services, can be expressed in the framework.

To understand the operator, consider a set of points in l-dimensional space. One point p_1 dominates another point p_2 if p_1 is at least as good as p_2 in all dimensions and is better than p_2 in at least one dimension [7]. It is assumed that a total order exists on each dimension, and "better" in a dimension is defined as smaller than (alternatively, larger than) with respect to the dimension's total order. Next, we assume a relation r with attributes $\{a_1, \ldots, a_l, b_1, \ldots, b_m\}$ so that the sub-tuples corresponding to attributes $\{a_1, \ldots, a_l\}$ make up the l-dimensional points. The skyline operator then returns all tuples in r that are not dominated by any other tuple in r.

To be precise, we first define two auxiliary functions. Let $\mathcal{T}$ denote the set of all tuples of any schema. The first function is $Dmnt$: $[\mathcal{T} \times \mathcal{R} \times \Omega^l] \rightarrow \{\text{True}, \text{False}\}$, which returns True if there exists a tuple in the (second) relation argument that dominates the first argument tuple with respect to the argument attributes.

$$\begin{aligned} Dmnt \triangleq \lambda t, r, a_1, \ldots, a_l.(r = \perp) \rightarrow \text{False}, \\ Eql(t, head(r), a_1, \ldots, a_l) \rightarrow Dmnt(t, tail(r), a_1, \ldots, a_l), \\ Comp(t, head(r), a_1, \ldots, a_l) \rightarrow \text{True}, \\ Dmnt(t, tail(r), a_1, \ldots, a_l) \end{aligned}$$

Function Eql returns True if the two argument tuples are identical on all argument attributes $a_1, \ldots, a_l$. Function $Comp$ returns True if the second argument tuple is no worse than the first argument tuple on any of the argument attributes.

In the first line, if r is empty, the operation returns False. Otherwise, if the first argument tuple t is the same as the head of argument relation r on the argument attributes, the operation continues to consider the rest of r. Else, the third line checks if $head(r)$ is no worse than t. If so, t is dominated by $head(r)$, and the operation returns True. Otherwise, the operation proceeds with the rest of r.

Next, we define auxiliary function $Fltr$: $[\mathcal{R} \times \mathcal{R} \times \Omega^l] \rightarrow \mathcal{R}$. For two relations r_1 and r_2 having the same attributes $a_1, a_2, \ldots, a_l$, $Fltr$ collects all the tuples in r_1 that are not dominated by any tuple in r_2 with respect to attributes $a_1, a_2, \ldots, a_l$.

$$\begin{aligned} Fltr \triangleq \lambda r_1, r_2, a_1, \ldots, a_l.(r_1 = \perp) \rightarrow r_1, \\ Dmnt(head(r_1), r_2, a_1, \ldots, a_l) \rightarrow Fltr(tail(r_1), r_2, a_1, \ldots, a_l), \\ head(r_1) \mathbin{@} Fltr(tail(r_1), r_2, a_1, \ldots, a_l) \end{aligned}$$

Here, if r_1 is empty, the operation returns it. Otherwise, if the head of r_1 is dominated by any tuples in r_2 on the argument attributes, the operation continues with the rest of r_1. Else, it returns the $head(r_1)$ with the result of the operation applied to $tail(r_1)$ appended.

The skyline operator $skyline : [\mathcal{R} \times \Omega^l] \rightarrow \mathcal{R}$ is defined next. Arguments Ω^l are parameters and are expressed as subscripts, i.e., $skyline_{a_1,\ldots,a_l}(r)$.

$$\begin{aligned} skyline \triangleq \lambda r, a_1, \ldots, a_l.(r = \perp) \rightarrow r, \\ Fltr(r, r, a_1, \ldots, a_l) \end{aligned}$$

4.2 Stream-to-Relation Operators

A *stream-to-relation* operator takes a stream as input and produces a relation. As relations are finite while streams can be infinite, windowing is commonly used to extract a relation from a stream [5]. We describe three types of sliding windows [2]: time-based, tuple-based, and partitioned. Other types of windows can be easily incorporated into the framework, as this does not affect other parts of the framework. The stream-to-relation operators map multisets into lists. We assume that each operator described next orders its result according to the time attribute T (tuples with the same time value may be in any order).

4.2.1 Time-Based Windows

A time-based sliding window operator $\mathcal{W}^a$, with absolute or now-relative time parameter τ_s, on a stream s returns all tuples $t \in s$ for which $\tau_s \leq t(\mathrm{T}) \leq \tau_c$, where τ_c is the current time. We present its formal definition in the following.

$$\mathcal{W}^a_{\tau_s}(S) = \{t \mid t \in S \wedge \tau_s \leq t(\mathrm{T}) \leq \tau_c\}$$

usr_id	usr_v	usr_loc	T
1002	$(3, 8)$	$(12, 15)$	9
1001	$(0.5, 2)$	$(3, 13.5)$	9
1003	$(2.5, 1.5)$	$(21, 16)$	10
1004	$(-3, -1.5)$	$(17.5, 12.2)$	10
1002	$(1, 2.5)$	$(14, 12)$	10
1001	$(0, 1)$	$(3, 15)$	10

Figure 6: Result of $\mathcal{W}^a_9(s_{usr})$ at Time 10

Note that $\mathcal{W}^a_{\tau_c}(s)$ consists of tuples that made their appearance in s at time τ_c, while $\mathcal{W}^a_0(s)$ consists of all tuples that appeared in the stream so far.

To give an example for this operator, suppose we are at now at time 10 and want to find all the data in the stream since time 9. Then for the stream s_{usr} in Figure 2(b), we set $\tau_c = 10$ and $\tau_s = 9$. The result of $\mathcal{W}^a_9(s_{usr})$ can be seen in Figure 6. This operator is the same as the Range(*T*) operator defined in the semantics of the CQL language [4]. For the following two operators, i.e., tuple-based and partitioned windows, we choose to only informally describe them as the formal definitions can be found in the CQL language.

4.2.2 Tuple-Based Windows

usr_id	usr_v	usr_loc	T
1001	$(0, 1)$	$(3, 15)$	10
1004	$(-6, -3)$	$(14.5, 10.7)$	11
1003	$(-3, 5)$	$(21, 17)$	11
1001	$(1, 0.5)$	$(3, 16.5)$	11

Figure 7: Result of $\mathcal{W}^b_4(s_{usr})$ at Time 11

A tuple-based sliding window operator $\mathcal{W}^b$, with positive integer parameter N, on a stream s returns the N most recent tuples in s, i.e., the tuples $t \in s$ for which $t(\mathrm{T}) \leq \tau_c$ and such that no other tuples exist in S that have larger time values (that do not exceed τ_c). If ties exist, tuples are chosen at random among the ties. Note also that fewer than N qualifying tuples may exist.

A tuple-based window is specified as $\mathcal{W}^b_N(s)$. Note that $\mathcal{W}^b_\infty(s) = \mathcal{W}^a_0(s)$. As an example, to find 4 most recent (current time is 12) tuples in the stream s_{usr} in Figure 2(b), we set $\tau_c = 12$. Then $\mathcal{W}^b_4(s_{usr})$ is given in Figure 7.

4.2.3 Partitioned Windows

A partitioned sliding window over stream s takes a positive integer N and a subset of s's attributes, $\{A_1, \ldots, A_m\}$, as parameters. This operation first partitions S into substreams based on the argument attributes, then computes a tuple-based sliding window of size N independently on each substream, and then returns the union of these windows.

usr_id	usr_v	usr_loc	T
1002	$(1, 2.5)$	$(14, 12)$	10
1004	$(-6, -3)$	$(14.5, 10.7)$	11
1003	$(-3, 5)$	$(21, 17)$	11
1001	$(1, 0.5)$	$(3, 16.5)$	11

Figure 8: Result of $\mathcal{W}_{1,usr\_id}(s_{usr})$

Using $\mathcal{W}$ as the operator name, the partitioned window can be expressed as $\mathcal{W}_{N,A_1,\ldots,A_m}(s)$. To exemplify, we consider to find most recent tuples of each user until the current time $\tau_c = 11$ in stream s_{usr} in Figure 2(b). Thus, $N = 1$ and the set of attributes is $\{usr\_id\}$. Then the result of $\mathcal{W}_{1,usr\_id}(s_{usr})$ is given in Figure 8.

4.3 Relation-to-Stream Operators

A relation r may be subject to updates, so that its state varies across time. We use the notation $r(\tau)$ to refer to the state of r at time τ. With this definition, we can specify the three relation-to-stream operators Istream, Dstream, and Rstream (adapted from [2, 3, 4]). The operators $\sqcup$, $\times$, and $\setminus$ are the algebra operators defined in Section 4.1.1.

Istream ("Insert" stream) maps relation R into a stream S so that a tuple $t \in r(\tau) \setminus r(\tau - 1)$ is mapped to $(t, \tau) \in s$. Analogously, Dstream ("Delete" stream) forms a stream based on when tuples were deleted from the argument relation. Next, Rstream maps relation r into stream s by tagging each tuple in r with each time that it is present in r. Assuming that 0 is the earliest time instant, the operators are defined as follows.

$$\mathsf{Istream}(r) = \sqcup_{\tau>0}((r(\tau) \setminus r(\tau-1)) \times \{\tau\}) \sqcup (r(0) \times \{0\})$$
$$\mathsf{Dstream}(r) = \sqcup_{\tau>0}((r(\tau-1) \setminus r(\tau)) \times \{\tau\})$$
$$\mathsf{Rstream}(r) = \sqcup_{\tau\geq 0}(r(\tau) \times \{\tau\})$$

Assume that a moving tourist wants to continuously know the nearest hospitals. The result, which is subject to change as the tourist moves, may be returned as a stream produced using one of the windowing operators and a relation-to-stream operator. We discuss nearest-neighbor queries in the next section.

We have so far defined relational operators and mapping operators between streams and relations. As location-based queries involve operations on spatial data, spatial operators are intrinsic to such queries. We treat spatial operators as black boxes and simply assume a set of such operators. Specifically, we will use spatial operators proposed by the OpenGIS Consortium [13] (see Appendix 6).

4.4 One-Time and Continuous Queries

The recent discussions over location-based queries in the database research community have a particular interest over the so-called continuous queries [28, 34, 36, 37, 53]. In our framework, a one-time query is a combination of stream-to-relation and relation-to-relation operators, while a continuous query is a possibly infinite numbers of one-time queries that are run repeatedly within a specified time interval according to a specified time granularity. The result of a continuous query can either be relations or streams. To generate a stream result, relation-to-stream operators are naturally employed by the continuous query; see Figure 9.

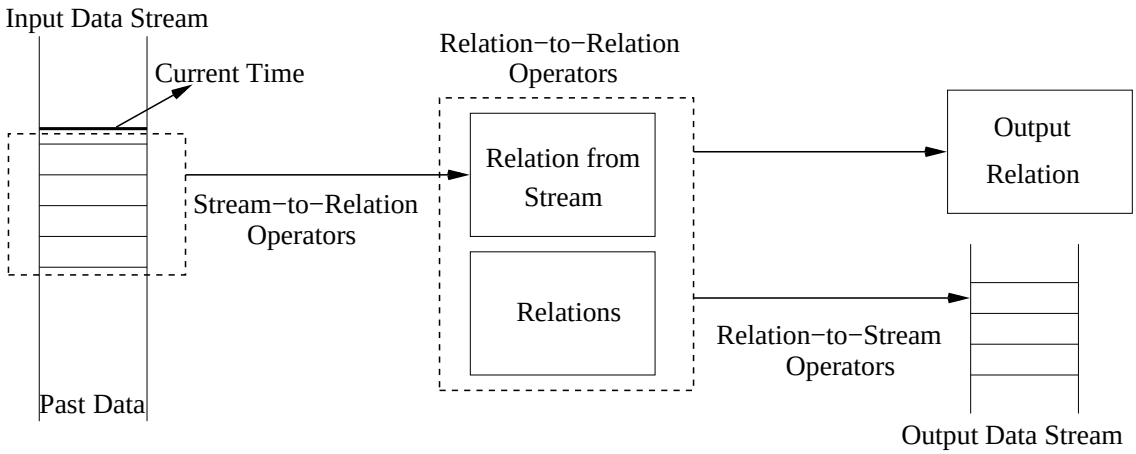


Figure 9: The Working of a Location-Based Query

Let a one-time query be expressed as $LBQ_p(\mathcal{S}, \mathcal{R})$, where $\mathcal{S}$ and $\mathcal{R}$ are argument sets of streams and relations and p is the parameters of the query. Then a continuous query can be expressed as $CLBQ_p(\mathcal{S}, \mathcal{R})$ $[T_s, T_e, \mathcal{G}]$, where T_s and T_e are the start and end time of the continuous query and $\mathcal{G}$ is the time granularity of the query. The result of a continuous query can be either a relation (e.g., a relation containing aggregate results) or a stream. If the result is a stream, relation-to-stream operators should also be included in the expressions of continuous queries to map the results into stream.

We choose to use λ-calculus in the definitions of relations and the associated algebraic operators for the purpose of accommodating duplicates and order. An additional outcome of this choice is that we are able to represent any queries that can be expressed using traditional relational algebra in the framework. Thus, the framework is open to new kinds of queries, as new algebraic and mapping operators can be added.

Another interesting question comes from the dissemination of results from continuous location-based queries. A general way is to provide the time-parameterized [52] or "distance-parameterized" [53] query results and employ a "trigger" mechanism either in the server or mobile clients to automatically generate outputs by looking up the time-parameterized query results based on the current location of mobile clients. As we will demonstrate in the following section, such ways of dissemination can also be captured in our framework.

5 Location-Based Queries

The literature covers the processing of a good amount of spatial and spatio-temporal queries which serve as basis and proto-types for LBQs, including range, nearest-neighbor, and reverse nearest-neighbor queries, as well as spatial join queries. Queries can concern past states of reality, or present and (anticipated) future states. Our focus will be on queries that concern the current state.

We proceed to demonstrate how the semantics of several frequently-discussed queries can be specified: spatial range search, nearest neighbor query, spatial join and a new, so-called location-based skyline query. We end by discussing the categorization of location-based queries.

5.1 Spatial Range Query

Various kinds of spatial range queries are used commonly. A range query may be used for finding all moving objects within a circular or rectangular region around a point of interest; and a continuous range query may be used for the monitoring of a region.

We assume a spatial range sr is given and define a range query (RQ) and continuous range query (CRQ) using a combination of stream-to-relation, relation-to-relation, and relation-to-stream operators.

To define the range query, we first obtain the most recent positions of all users. This is done by applying a partition window $\mathcal{W}_{1,usr\_id}(s_{usr})$ to stream s_{usr} and by applying a function CurLoc, using an extended projection. The function takes as argument a tuple $t \in s_{usr}$ that records the movement of an object, and it returns the location at time τ_c of the object.

$$r_s = \pi_{usr\_id,usr\_v,\mathrm{CurLoc}(t)\text{ AS loc}}(\mathcal{W}_{1,usr\_id}(s_{usr}))$$
$$\mathrm{CurLoc}(t) = t(usr\_loc) + t(usr\_v) \cdot (\tau_c - t(\mathrm{T}))$$

Then a selection retrieves all users that are inside the spatial region. Operator $\mathrm{within}(sr, \mathrm{loc})$ returns true if loc is within spatial range sr. The one-time range query is then given as follows.

$$\mathrm{RQ}_{sr}(\{s_{usr}\}, \emptyset) = \sigma_{\mathrm{within}(sr,\mathrm{loc})}(r_s)$$

Next, assume that the start and end times of the continuous range query are T_s and T_e, and that the time granularity is $\mathcal{G}$. By applying the Rstream operator, the definition of the continuous range query is given next.

$$\mathrm{CRQ}_{sr}(\{s_{usr}\}, \emptyset)[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}] = \mathsf{Rstream}(\sigma_{\mathrm{within}(sr,\mathrm{loc})}(r_s))[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}]$$

usr_id	usr_v	loc	T
...	...	...	...
1003	$(2, 1)$	$(17.5, 14.5)$	8
1004	$(-2, -1)$	$(21.5, 14.2)$	8
1003	$(2, 1)$	$(19.5, 15.5)$	9
1004	$(-2, -1)$	$(19.5, 13.2)$	9
1004	$(-3, -1.5)$	$(17.5, 12.2)$	10
1002	$(1, 2.5)$	$(14, 12)$	10
...	...	...	...

Figure 10: Result of $\mathrm{CRQ}_{sr}(\{s_{usr}\}, \emptyset)[0, 20, 1]$

To exemplify, let spatial range sr be a circle with radius 6 and center $(18.5, 10)$ (i.e., the gate of the library), and let $\mathrm{T}_s = 0$, $\mathrm{T}_e = 20$, and $\mathcal{G} = 1$. Suppose the current time is 10, the result of the continuous range query against stream s_{usr} is given in Figure 10. Note that the result is a stream.

According to the above definition, all the users' locations are calculated at each time instant. However, it may be that some users have not reported location data for several hours, rendering their location data useless. This suggests an alternative definition where only location data that has arrived within some time duration from the current time is used. To give the definition, we first obtain the stream of the users' most recent location data in stream S_{usr} by combining a time-based window operator $\mathcal{W}^a_{\tau_s}(S_{usr})$ and a Rstream operator. Then apply the spatial operator "Within" to retrieve users that are inside the spatial region. Using the same T_s, T_e, and $\mathcal{G}$ as above and assuming that we are only interested in location data that arrived since time τ_s, an alternative definition follows.

$$\mathrm{CRQ}_{sr,\tau_s}(\{s_{usr}\}, \emptyset)[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}] = \mathsf{Rstream}(\sigma_{\mathrm{within}(sr,\mathrm{loc})}(r'_s))[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}]$$

In this definition, r'_s is r_s where s_{usr} is replaced by $\mathsf{Rstream}(\mathcal{W}^a_{\tau_s}(s_{usr}))$, i.e.,

$$r'_s = \pi_{usr\_id,usr\_v,\mathrm{CurLoc}(t)\text{ AS loc}}(\mathcal{W}_{1,usr\_id}(\mathsf{Rstream}(\mathcal{W}^a_{\tau_s}(s_{usr})))) \ .$$

Intuitively, this query may miss some users who are actually inside the spatial range, but have not reported their location for some time.

Another observation from the result in Figure 10 is the redundant data. For instance, the tuples related to users with usr_id as $1003, 1004$ frequently appear in the output stream. If we only want to continuously know all the moving users that appeared inside the spatial range, we can obtain the one-time result, as shown in the following.

$rdup(\pi_{usr\_id}(\mathcal{W}^a_{\tau_c - \mathrm{T}_s}(S_{usr})))$

And then map the result into stream with the Istream operator.

$$\mathrm{CRQ}^{range}_{sr,\tau_s}(\{s_{usr}\}, \emptyset)[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}] = \mathsf{Istream}(rdup(\pi_{usr\_id}(\mathcal{W}^a_{\tau_c - \mathrm{T}_s}(S_{usr}))))[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}]$$

There are a variety of choices for representing one type of location-based query. For these range queries and subsequent location-based queries in our discussion, we choose, from several alternative representations, the one that we felt most expressive and understandable.

5.2 Nearest Neighbor Query

The k nearest neighbor query (kNNQ) is another basic type of location-based query. Example uses include locating the nearest hospitals or emergency vehicles. To formulate the query that finds the k nearest neighbors of a moving user with usr_id as m_id, we first define several auxiliary functions.

A partitioning window query $\mathcal{W}_{1,usr\_id}(s_{usr})$ first retrieves the most recent position data for each object from the stream. Then a selection with predicate $usr\_id = m\_id$ is applied to retrieve the position data for our object. Let r_l denote the relation resulting from this selection, i.e.,

$$r_l = \sigma_{usr\_id = m\_id}(\mathcal{W}_{1,usr\_id}(s_{usr}))$$

To compute the k objects nearest to m_id, we calculate, using a spatial operator "dist," the distance between m_id's current location and the locations of all other objects, which are stored in attribute obj_loc of r_{obj}. As the next step in computing the query, we apply a generalized projection to associate the distance to the user object with each other object:

$$r_s = \pi_{obj\_id, obj\_loc, obj\_type, \mathrm{dis}}(r_{obj} \times r_l)$$

Here, "dis" denotes $\mathrm{dist}(obj\_loc, \mathrm{CurLoc}(t))$, function $\mathrm{CurLoc}(t)$ was defined earlier, and t denotes a tuple from the argument relation.

Then we apply the $sort$ and top operators to r_s to express the query.

$$\mathrm{kNNQ}_{m\_id,k}(\{s_{usr}\}, \{r_{obj}\}) = top_k(sort_{\mathrm{dis}}(r_s))$$

Let $r_p = \sigma_{obj\_type = \text{'ATM machine'}}(r_{obj})$ contain all AMT machines and consider the query $\mathrm{kNNQ}_{1004,1}$ $(\{s_{usr}\}, \{r_p\})$ issued at time $\tau_c = 9$. The query finds the ATM machine nearest to user 1004. Using the definition, a window operator extracts all the most recent tuples for each user from stream s_{usr}. Then the tuple with $usr\_id = 1004$, $usr\_v = (-1.5, -3)$, $usr\_loc = (21.5, 14.2)$, and $\tau = 8$ is selected. The current location is approximated as $(21.5, 14.2) + (-2, -1) \cdot (9 - 8) = (19.5, 13.2)$. (We choose to use Manhattan distance throughout this paper. It calculates the distance between points (x_1, y_1) and (x_2, y_2) as $|x_2 - x_1| + |y_2 - y_1|$). Among all objects in relation r_p, the object with $obj\_id = 310$ is selected.

If the user is interested in keeping knowing the nearest ATM machine while moving, we need to add relation-to-stream operators and represent the query as a continuous nearest neighbor query. The expression is as follows:

$$\mathrm{CkNNQ}_{m\_id,k}(\{s_{usr}\}, \{r_{obj}\})[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}] = \mathsf{Rstream}(\mathrm{kNNQ}_{m\_id,k}(\{s_{usr}\}, \{r_{obj}\}))[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}]$$

The result of $\mathrm{CkNNQ}_{1004,1}(\{s_{usr}\}, \{r_p\})[0, 20, 1]$ is shown in Figure 11.

obj_id	obj_loc	obj_type	dis	τ
...	...	...	...	...
310	$(21, 11.5)$	ATM machine	3.2	8
310	$(21, 11.5)$	ATM machine	3.2	9
310	$(21, 11.5)$	ATM machine	4.2	10
309	$(14, 10)$	ATM machine	1	11
309	$(14, 10)$	ATM machine	3.5	12
...	...	...	...	...

Figure 11: CkNNQ over Relation r_p

The continuous query keep running the one time nearest neighbor query with the current location of the moving user at each time instance.

A popular strategy for processing such continuous queries is, by assuming that a trajectory or route of the moving user is known, to pre-compute the nearest neighbors to each location of the route so that the whole trajectory is cut into segments with each segment corresponding to one nearest neighbor. Then the continuous query just need to look through the pre-computed relations for real-time results. For instance, take the

moving user with $usr\_id = 1004$ as an example, suppose we are at time 8 and the user's future trajectory (also shown in Figure 2(a)) is a line segment containing the following sequence of two-dimensional locations, $\langle(21.5, 14.2), (17.5, 12.2), (14.5, 10.7), (9.5, 8.2), (8.5, 7.7), (5.5, 6.2)\rangle$. We pre-compute the nearest ATM machine to any locations on this trajectory and save the distance-parameterized result into relation r_{ATM}. As shown in Figure 12, following each nearest neighbor, the whole trajectory is partitioned into several segments, with each segment corresponds to the $dist\_range$ column in the r_{ATM} relation. The value of the $dist\_range$ column is computed as the distance (note that this distance is the 1-dimensional Manhattan distance on the linear trajectory) from a location on the trajectory to the start location $(21.5, 14.2)$. The efficiency of processing a continuous query will be improved with this pre-computed result. Using the same r_l, dist and $\mathrm{CurLoc}(t)$ described above, the continuous query can be expressed as in the following.

$$\mathrm{CkNNQ}_{(21.5,14.2),1}(\{s_{usr}\}, \{r_{ATM}\})[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}] = \mathsf{Rstream}(\sigma_{dist((21.5,14.2),\mathrm{CurLoc}(t))\in dist\_range}(r_{ATM} \times r_l))[\mathrm{T}_s, \mathrm{T}_e, \mathcal{G}]$$

NN_obj_id	$dist\_range$
310	$(0, 5.31)$
309	$(5.31, 13.752)$
306	$(13.752, 17.89)$

Figure 12: Relation r_{ATM}

Another interesting variant of the nearest neighbor query is the so-called **reverse nearest neighbor** search. Given a set of objects and one of them as a query object, the (monochromatic) reverse nearest neighbor query finds all the objects that have the query object as their closest object. Another interesting extension is the bichromatic reverse nearest neighbor search. Given two categories of objects, the bichromatic reverse nearest neighbor query returns the objects that belong to the opposite category than the query object and have this query object as the closest from all the objects in the same category as the query object. We consider the representation of the bi-chromatic reverse nearest neighbor query. For instance, for the point of interest with $obj\_id = 304$ (the library in Figure 2(a)), we want to search its reverse nearest neighbor users listed in relation r_{usr} (shown in Figure 1) with locations in stream s_{usr}. We can use the kNNQ query to find nearest neighbor point of interest to all the moving users. Then the RkNNQ query is just to find those moving users having nearest neighbor point of interest as $obj\_id = 304$.

$$\mathrm{RkNNQ}_{304,1}(\{s_{usr}\}, \{r_{obj}, r_{usr}\}) = (\sigma_{\mathrm{kNNQ}_{usr\_id,1}(\{s_{usr}\},\{r_{obj}\}).obj\_id=304}(r_{usr}))$$

It is also possible to re-use the distance join (denoted as D-Join) and distance semi-join (denoted as DS-Join) operators described in [27] for representing nearest neighbor and reverse nearest neighbor queries. We omit these redundant representations and proceed to discuss these two spatial join queries in the following.

5.3 Spatial Join Query

Although a variety of spatial join queries exist [38], we focus on the distance join query and its variants [17, 27] as it is of more interest to location based services. We omit the formal definition of spatial distance join and semi-join as they have been represented with a SQL-like language elsewhere [27] and include them directly into the framework.

Given two sets of objects with spatial locations, the distance join operation computes a subset of the Cartesian products of the two sets and, based on the distance, specifies an order on the result. The distance is computed in terms of location attributes of objects in two sets. We can also apply a predicate to limit the resulting data to a range. The distance semi-join is a special case of the distance join. It finds, for each object in the first set, its nearest object in the second set.

To give an example for the distance join (denoted as D-Join) operation, let us consider to find all pairs of moving users in stream s_{usr} and toilets in relation r_{obj} whose distance is less than a given value, d_{mt}. The query can be expressed as in the following.

$$\mathsf{D\text{-}Join}_{dis\le d_{mt}}(\pi_{usr\_id,obj\_id,\mathrm{dis}}(\mathcal{W}_{1,usr\_id}(s_{usr} \times \pi_{obj\_id,obj\_loc}(\sigma_{obj\_type=\text{'public toilet'}}(r_{obj})))))$$

Suppose current time is 12, then the locations of the four moving users can be calculated from the window operation over s_{usr}. If we set $d_{mt} = 6$, the result of the query, based on the expression above, can be found in Figure 13.

usr_id	obj_id	dis
1001	311	5
1002	312	3
1004	312	5.3

Figure 13: Result of Distance Join

The distance semi-join operator can be used for representing reverse nearest neighbor query. Here we consider to use this operator to express the so-called "closest pair" query [17]. Given two sets of objects with their spatial locations, this query retrieves k closest pairs. Following the above example, if we want to find k closest pair of moving users and toilets, the expression of the query, denoted as kCPair, is in the following.

$$\text{kCPair}_k(\{s_{usr}\},\{r_{obj}\}) = top_k(sort_{\text{dis}}(\text{DS-Join}(\pi_{usr\_id,obj\_id,\text{dis}}(\mathcal{W}_{1,usr\_id}(s_{usr} \times \pi_{obj\_id,obj\_loc}(\sigma_{obj\_type=\text{'public toilet'}}(r_{obj})))))))$$

Due to the natural similarity and inheritance from spatial and spatio-temporal queries, a lot of other location-based queries can be found by further exploring on the research literature over the spatial database area. We choose to turn our focus of discussion on accommodating the multiple preferences considered by the location-based queries. Such queries require definition of novel relation-to-relation operators based on the basic operators and combination of the new operators with other mapping operators. In the following, as an example, we discuss a location-based skyline query.

5.4 Location-Based Skyline Query

The query assumes the following scenario. A user drives along a pre-defined route towards a destination. The user wants to visit one or several points of interest enroute. The most attractive of the qualifying points of interest are those that are nearest to the user's current location and that result in the smallest detour. The detour is the extra distance traveled if the user visits the point of interest and then travels to the destination.

Let r_l, CurLoc and t be as defined earlier. We assume that spatial operator "dist" takes into account the user's route, and we denote the user's destination by $dest$. Then the detour fe can be expressed as follows.

$$fe(obj\_loc, t, dest) = \mathrm{dist}(obj\_loc, \mathrm{CurLoc}(t)) + \mathrm{dist}(\mathrm{obj\_loc}, dest) - \mathrm{dist}(\mathrm{CurLoc}(t), dest)$$

Next, a (generalized) projection is applied to the Cartesian product of r_{obj} and r_l to get all the objects' distances and detours to the user:

$$r_s = \pi_{obj\_id,obj\_type,\text{dis,det}}(r_{obj} \times r_l)$$

Here, "dis" denotes $\mathrm{dist}(obj\_loc, \mathrm{CurLoc}(t))$ and "det" denotes $fe(obj\_loc, t, dest)$.

Finally, the skyline operation generates the result.

$$\text{SQ}_{m\_id,dest}(s_{usr}, r_{obj}) = skyline_{\text{dis,det}}(r_s)$$

obj_id	obj_type	dis	det
306	ATM machine	19.2	6.4
307	ATM machine	20.3	16.6
308	ATM machine	21.9	20.8
309	ATM machine	5.7	0
310	ATM machine	4.2	8

Figure 14: Intermediate Result

Following the scenario described above, let the current time $\tau_c = 10$ and assume a user with $usr\_id = 1004$ wants to go to an ATM machine enroute to the destination, the location of which is $(5.5, 6.2)$. For simplicity, we use direct line segments as routes between two points. The current location of the user is $(17.5, 2.2)$. For all static objects with usr_type "$\mathrm{ATM\ machine}$," the distance and detour are listed in Figure 14. The skyline operator returns the last two tuples.

5.5 Towards a Categorization of Location-Based Queries

A complete list of all LBQs may be impossible as novel queries may appear due to the discover of new location-based services. We proceed to explore the space of existing LBQs by presenting several orthogonal categorizations of such queries. Our discussion also considers a recent paper [20] addressing the issue related to the taxonomy of location-based services.

First, as most location-based queries are issued from query objects over data objects, we can categorize these queries based on whether the query and data objects are moving or stationary. A stationary query object, e.g., a

supermarket, may ask for moving mobile users passing by in a recent hour. A moving mobile user, as a moving query object, may ask for nearest gas stations.

Second, queries can be categorized according to whether they are one-time or continuous queries. Continuous queries may be classified further, based on whether they are constant or time-parameterized. The latter occurs when a query refers to the (variable) current time. An example is a continuous query that retrieves all objects currently within a spatial range. A corresponding constant query might retrieve all objects that are (currently believed to be) within a spatial range at some fixed near future time. Constant continuous queries have been termed "persistent" in the literature.

Third, one can also classify the queries based on the spatial scenario where the query is concerned with, i.e., the non-constrained Euclidean space (e.g., movement of aircraft), the transportation networks, and the infrastructure-constrained space. As an example for the last scenario, let us consider user Clark in Figure 2(a), this user's movement is partially-constrained by the lake.

Fourth, queries can be categorized based on whether they refer to data concerning the past, present, or future states of reality. For instance, a mobile user driving on a route may be interested in finding restaurants open in the next 2 hours.

Fifth, based on the cardinality of query objects and data objects, queries may be classified as being either "one-to-many" or "many-to-many." The former queries apply one predicate to many objects, returning one set, multiset, or list of objects. The latter conceptually repeatedly applies many different predicates to many objects, potentially retrieving many objects for each predicate. A simple selection is thus an example of the former. The k nearest neighbor query in Section 5.2 retrieves (up to) k objects that are the nearest to some (i.e., "one") specified object; it is thus also a "one-to-many" query. In contrast, joins are "many-to-many" queries: The predicate involving one (left hand side) object is applied to many (right hand side) objects, and this repeated many times. The so-called "closest pair" query, which finds pairs of objects from two different groups that are closest, is also a "many-to-many."

Sixth, based on the time at which a query is registered to the system, it can be "pre-defined," meaning that it is present before the streams it uses start, or it can be "ad hoc," meaning that it is registered after at least one of its streams has started.

6 Summary and Future Work

Substantial research has been reported on query processing in relation to mobile services, in particular location-enabled mobile services. Different techniques are applicable to different kinds of queries. Based on results from stream and temporal databases, this paper proposes a framework for capturing the semantics of the diverse kinds of queries that are relevant in this context. By enabling the definition of queries in a single framework, the paper's proposal enables the comparison of queries.

The framework consists of data types, relations and streams, as well as algebraic operations on relations and operations that map between streams and relations. The specific representations of spatial data and the associated operations on these are treated as black boxes, in order to enable applicability across different such representations and operations. The extensibility of the framework was exemplified by adding a skyline operator.

The use of the framework was illustrated by the definition of several location-based queries and corresponding variants, spatial range query, nearest-neighbor query, spatial join query, and location-based skyline query. Toy examples were given for illustrating these queries. Focus has been on the capture of the semantics of LBQs, and how to use one-time or continuous queries in actual location-based services is beyond the scope of the paper.

This paper represents initial work, and future work may be pursued in several directions. First, the framework may be enriched in various ways. One is to introduce explicit representations of the space within which the spatial objects are located and move, e.g., road networks. Second, while this paper has given one definition of several location-based queries, it would be worthwhile to explore the different possible semantics that may be given to queries within the framework. Such a study may reveal whether or not desirable semantics can be specified in all cases. Third, although interesting initial steps have been taken in this direction (e.g., [20, 43]), more work on taxonomies for location-based queries is desirable.

References

[1] P. Atzeni and V. De Antonellis. *Relational Database Theory.* Benjamin Cummings, 1993.

[2] A. Arasu, S. Babu, and J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Technical Report. Department of Computer Science, Stanford University, 12 pages, 2002.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report. Department of Computer Science, Stanford University, 32 pages, 2003.

[4] A. Arasu and J. Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. Technical Report. Department of Computer Science, Stanford University, 8 pages, Mar. 2004

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. PODS*, pp. 1–16, 2002.

[6] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Šaltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *VLDB J.*, 15(3): 229–249, 2006.

[7] S. Borzsonyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proc. ICDE*, pp. 421–430, 2001.

[8] N. Beckmann, H. P. Kriegel, R. Schneider, B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. SIGMOD*, pp. 322–331, 1990.

[9] P. Bonnet, J. Gehrke, P. Seshadri. Towards Sensor Database Systems. In *Proc. MDM*, pp. 3–14, 2001.

[10] D. Carney, U. Centintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proc. VLDB*, pp. 215–226, 2002.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. 1st Biennal Conf. on Innovative Data Syst. Res*, pp. 269–280, 2003.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Sclalable Continuous Query System for Internet Databases. In *Proc. SIGMOD*, pp. 379–390, 2000.

[13] E. Clementini and P. D. Felice. Spatial Operators. In *SIGMOD Record*, 29(3): 31–38, 2000.

[14] S. Chandrasekaran and M. Franklin. Streaming Queries Over Streaming Data. In *Proc. VLDB*, pp. 203–214, 2002.

[15] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *Proc. ICDE*, pp. 717–816, 2003.

[16] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A Stream Database for Network Application. In *Proc. SIGMOD*, pp. 647–651, 2003.

[17] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *Proc. SIGMOD*, pp. 189–200, 2000.

[18] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. SIGMOD*, pp. 319–330, 2000.

[19] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained Nearest Neighbor Queries. In *Proc. SSTD*, pp. 257–276, 2001.

[20] K. Gratsias, E. Frentzos, V. Delis, Y. Theodoridis. Towards a Taxonomy of Location Based Services. In *Proc. W2GIS*, 2005.

[21] L. Golab and M. Tamer Özsu. Issues in Data Stream Management. In *SIGMOD Record*, 32(2): 5–14, 2003.

[22] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. SIGMOD*, pp. 47–57, 1984.

[23] Q. Hart, M. Gertz. Indexing Query Regions for Streaming Geospatial Data. In *Proc. STDBM*, pp. 49–56, 2004.

[24] X. Huang, C. S. Jensen. Towards A Streams-Based Framework for Defining Location-Based Queries. In *Proc. STDBM*, pp. 78–85, 2004.

[25] X. Huang, C. S. Jensen. In-Route Skyline Querying for Location-Based Services. In *Proc. W2GIS*, pp. 120–135,2004.

[26] C. Hage, C. S. Jensen, T. B. Pedersen, L. Speičys, and I. Timko. Integrated Data Management for Mobile Services in the Real World. In *Proc. VLDB*, pp. 1019–1030, 2003.

[27] G. R. Hjaltason and H. Samet. Incremental Distance Join Algorithms for Spatial Databases. In *Proc. SIGMOD*, pp. 237–248, 1998.

[28] H. Hu, J. Xu, D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *Proc. SIGMOD*, pp. 479–490, 2005.

[29] G. S. Iwerks, H. Samet, K. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *Proc. VLDB*,, pp. 512–523, 2003.

[30] G. S. Iwerks, H. Samet, K. Smith. Maintenance of Spatial Semijoin Queries on Moving Points. In *Proc. VLDB*, pp. 828–839, 2004.

[31] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proc. VLDB*, pp. 275–286, 2002.

[32] X. Liu, H. Ferhatosmanoglu. Efficient k-NN Search on Streaming Data Series. In *Proc. SSTD*, pp. 83–101, 2003.

[33] L. Liu, C. Pu, W. Tang. Continual Queries for Internet-Scale Event-Driven Information Delivery. In *IEEE Trans. Knowledge and Data Eng.*, 11(4): 610–628, 1999.

[34] K. Mouratidis, D. Papadias, M. Hadjieleftheriou. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *Proc. SIGMOD*, pp. 634–645, 2005.

[35] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries Over Streams. In *Proc. SIGMOD*, pp. 49–60, 2002.

[36] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *Proc. SIGMOD*, pp. 623–634, 2004.

[37] M. F. Mokbel, X. Xiong, M. A. Hammad, W. G. Aref. Continuous Query Processing of Spatio-temporal Data Streams in PLACE. In *Proc. STDBM*, pp. 57–64, 2004.

[38] O. Günther. Efficient Computation of Spatial Joins. In *Proc. ICDE*, pp. 50–59, 1993.

[39] K. Patroumpas, T. K. Sellis. Managing Trajectories of Moving Objects as Data Streams. In *Proc. STDBM*, pp. 41–48, 2004.

[40] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group Nearest Neighbor Queries. In *Proc. ICDE*, pp. 301–312, 2004.

[41] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proc. SIGMOD*, pp. 467–478, 2003.

[42] L. Qiao, D. Agrawal, and A. E. Abbadi. Supporting Sliding Window Queries for Continuous Data Streams. In *Proc. SSDBM*, pp. 85–94, 2003.

[43] A. Y. Seydim, M. H. Dunham, and V. Kumar. Location Dependent Query Processing. In *Proc. MobiDE*, pp. 47–53, 2001.

[44] M. Sullivan, A. Heybey. Tribecca: A System for Managing Large Databases of Network Traffic. In *Proc. ICDE*, pp. 80–89, 1995.

[45] S. Šaltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. ICDE*, pp. 463–472, 2002.

[46] G. Slivinskas, C. S. Jensen, R. T. Snodgrass. Bringing Order to Query Optimization. In *SIGMOD Record*, 31(2): 5–14, 2002.

[47] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proc. SSTD*, pp. 79–96, 2001.

[48] S. Shekhar and J. S. Yoo. Processing In-Route Nearest Neighbor Queries: A Comparison of Alternative Approaches. In *Proc. GIS*, pp. 9–16, 2003.

[49] M. Sullivan. Tribeca: A Stream Database Manager for Network Traffice Analysis. In *Proc. VLDB*, pp. 594, 1996.

[50] K. L. Tan, P. K. Eng, B. C. Ooi. Efficient Progressive Skyline Computation. In *Proc. VLDB*, pp. 301–310, 2001.

[51] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries Over Append-Only Databases. In *Proc. SIGMOD*, pp. 321–330, 1992.

[52] Y. Tao, D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *Proc. SIGMOD*, pp. 334–345, 2002.

[53] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *Proc. VLDB*, pp. 287–298, 2002.

[54] J. S. Yoo, S. Shekhar. In-Route Nearest Neighbor Queries. In *GeoInformatica*, 9(2): 117–137, 2005.

[55] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal and Spatio-Temporal Aggregations Over Data Streams Using Multiple Time Granularities. In *Inf. Syst.* 28(1–2): 61–84, 2003.

[56] J. Zhang, N. Mamoulis, D. Papadias, Y. Tao. All-Nearest-Neighbors Queries in Spatial Databases. In *Proc. SSDBM*, pp. 297-306, 2004.

[57] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos and B. Seeger. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proc. PODS*, pp. 237–245, 2001.

[58] Y. Zhu, D. Shasha. StarStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proc. VLDB*, pp. 358–369, 2002.

[59] D. Zhang, V. J. Tsotras and D. Gunopulos. Efficient Aggregation over Objects with Extent. In *Proc. PODS*, pp. 121–132, 2002.

Appendix

Table 1: Operators defined on the class Geometry by OpenGIS [13]

Basic operators	Spatial Reference	Returns the Reference systems of the geometry
	Envelope	The minimum bounding rectangle of the geometry
	Export	Convert the geometry into a different representation
	IsEmpty	Tests if the geometry is the empty set or not
	IsSimple	Returns True if the geometry is simple
	Boundary	Returns the boundary of the geometry
Topological operators	Equal	Tests if the geometries are spatially equal
	Disjoint	Tests if the geometries are disjoint
	Intersect	Tests if the geometries intersect
	Touch	Tests if the geometries touch each other
	Cross	Tests if the geometries cross each other
	Within	Tests if the given geometry is within another given geometry
	Contains	Tests if the given geometry contains another given geometry
	Overlap	Tests if the given geometry overlaps another given geometry
	Relate	Returns true is the spatial relationship specified by the 9-Intersection matrix holds
Spatial analysis operators	Distance	Returns the shortest distance between any two points of two given geometries
	Buffer	Returns a geometry that represents all points whose distance from the given geometry is less than or equal to the specified distance
	ConvexHull	Returns the convex hull of the given geometry
	Intersection	Returns the intersection of two given geometries
	Union	Returns the union of two given geometries
	Difference	Returns the difference of two given geometries
	SymDifference	Returns the symmetric difference of two given geometries