BWHERE

Composite Spatiotemporal Query Operators in a Streaming and Mobile Context

Aalborg University

D631a

Master Thesis

AALBORG UNIVERSITY

Department of Computer Science

Title: BWHERE

Subtitle: Composite Spatiotemporal Query Operators in a Streaming and Mobile Context

Theme: Master Thesis

Time period: February 1st, 2005 - June 7th, 2005

Term: F10S/CIS4

Group: D631a

Members:

Kim R. Bille, F10S

Asbjørn Strøm Jensen, CIS4

Martin Maach, F10S

Supervisor:

Janne Skyt

Number printed: 7

Report pages: 63

Total page count: 99

Ended: June 7th, 2005

Abstract:

This project, BWHERE, extends our last project "AWHERE - Handling of Data Streams in a Mobile Context" with area subscription. The project creates an interesting interplay between three worlds of technologies, namely mobile environment, streams, and GISs. The project addresses the issues by providing a theoretical and practical solution. The theoretical solution examines the worlds, and provides a set of composite spatiotemporal query operators for the streaming and mobile environment. Subsequently, the result is applied to a experimental solution, with the convergent technologies from the three worlds, i.e., *Java 2 Platform, Micro Edition (J2ME)*, *TelegraphCQ*, and *PostgreSQL*. A practical implementation of the composite spatiotemporal query operators, enter and leave, binds the worlds of technologies together.

Both theory and experiments revealed that such a combination of mobility, streaming, and GIS, was a viable solution, although some of the technologies involved needs some maturity.

Preface

This project has been prepared by group D631a at Department of Computer Science at Aalborg University, consisting of students of the F10S and CIS4 semesters. The project extends from February 1st, 2005 to June 7th, 2005.

This report is the product of the second half of the master thesis period spanning a full year (2 semesters). The area of specialization of this master thesis is *database systems and programming technology*. The target audience of the project is students, researchers, and other interested with a technical insight corresponding to that of the group. The purpose and content of the project is specified in the F10S, and CIS4 study regulations, which are outlined below.

With an offset in a central problem within a chosen area of specialization, the student must be able to utilize the theories and methods in the field of research for an in-depth analysis and possibly a theoretical or practical solution.

Thanks goes to Stardas Pakalnis and Wladyslaw Andrezej Pietraszek (Department of Computer Science, Aalborg University) for providing GPS modules and external access for our database server, respectively.

A special thanks to our supervisor Janne Skyt (Department of Computer Science, Aalborg University) for informative guidance throughout the course of this project.

In the report all literature references have been composed with author and year in square brackets, e.g., [Silbershatz et al., 2002]. The position of the reference determines which specific part the reference relates to. If the reference is positioned before a period, it relates to the previous sentence, otherwise if the reference is positioned after a period the reference related to the previous section.

Further, we in the report use a number of abbreviations. All abbreviations are written in non-abbreviated form at first occurrence. Moreover, all abbreviations and their meaning are listed in the Glossary on page 93.

The focus of the report is analysis of the problem and design of a solution in preference to the implementation, according to the study regulations.

Aalborg, June 7th, 2005

Kim Rud Bille

Asbjørn Strøm Jensen

Martin Maach

Contents

Chapter 1

Introduction

In this chapter the initial thoughts and motivation of the project are described. As this project partially builds on a previous project, namely AWHERE*, this is briefly introduced. Further, based on ideas from* AWHERE *we present scenarios that* AWHERE *is not capable of handling and that we wish to address in this project. This leads to a description of how this project can be viewed as a combination of different worlds. This, further, leads to which points we need to analyze in these worlds. In addition, related applications in the field of Location-Aware Services are presented.*

In this project we proceed our examination and development of mobile services, from our last project "AWHERE - Handling of Data Streams in a Mobile Context" [Bille et al., 2005].

The AWHERE system is a *Continuous Location-Aware Service (CLAS)*[1], which in real-time provides users of mobile devices with location information on other users. The system was based on a developed *Data Stream Management System (DSMS)* and a client application. In order to minimize the transmission cost for the user, we introduced *Repeated Discrete Evaluation (RDE)* of queries. Clients of the system could subscribe to position, distance, direction, and speed information based on *Global Positioning System (GPS)* information.

The back-end of the AWHERE system is the AWHERE-DSMS, a DSMS we developed during the course of the last project. The AWHERE-DSMS is limited in different ways, because the primary objective of it was to support the functionality in AWHERE and only secondly be a full-fledged DSMS.

The two main foci of this project stems directly from the future work of AWHERE and are:

Real life test In AWHERE we tested our software exclusively on mobile phone emulators. In this project we wish to remedy this and create software that will run on a physical mobile phone.

Extended area subscription With AWHERE a user has a contact list, and when people on his/her contact list are online he/she is constantly updated with their speed, current direction, and the distance between the two users. In this project we will extend this with subscription to geographical areas, where the user receives a notification when the status of the users on his/her contact list change in the subscribed areas.

Achieving the goal of a *Real life test* is relatively uninteresting from a theoretical standpoint and will therefore not be a major part of the coming analysis. The system that we have developed, which is an extension of AWHERE and which include the *extended area subscription*, will be called: BWHERE.

Extending the features of the AWHERE system with the proposed area subscription creates an interesting interplay of technologies. In the following, possible scenarios

[1] Notice: All abbreviations used in this report are listed on page 93.

of area subscription will be presented and the different technologies needed will be introduced.

1.1 Area Subscription

In this section we give some descriptions of scenarios in which area subscription is useful for users of AWHERE, e.g., pedestrians.

The scenarios we supported in AWHERE were mostly scenarios where a user only needed information on the distance to another user, e.g., "I am at this bar, who of my friends are nearby?". In some cases you might also make use of the direction and speed information, e.g., "I get an alert that a friend is nearby only to discover that she is moving away from me at 50 km/h". So she must be in a moving vehicle and following her on foot would be pointless.

Below is a list of possible scenarios which are not explicitly possible in the current version of the AWHERE system.

- Locate friends in the vicinity (area) of the user.
- Locate friends in a specific area independent of the user's position.
- Alert when other users enters/leaves an area of interest.
- Warning system in case of tsunami, weather disaster, accidents, etc.

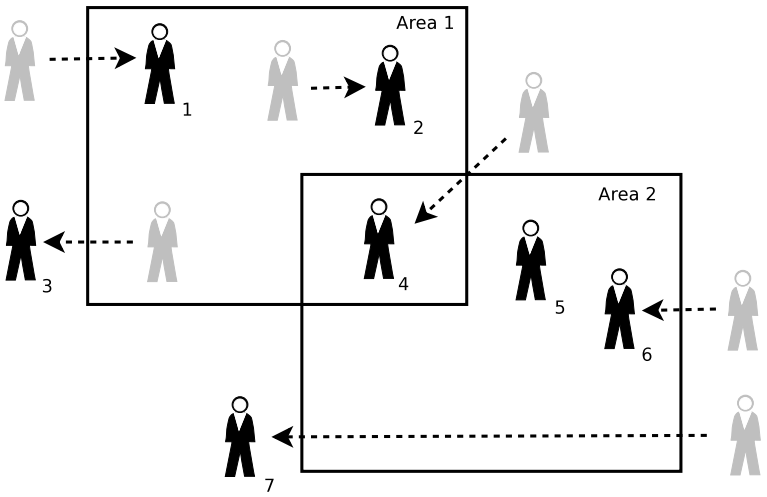


Figure 1.1: Illustrations of possible scenarios.

Figure 1.1 depicts seven users and their movement in relation to two areas. An unfaded person icon denotes the users current position, whereas the faded icons denotes the users previous location. The arrows denote the trajectory of a given user, from their previous location to their current location. In the following we will refer to the users and areas in the figure as illustration for various scenarios. References to Figure 1.1 are in the following marked in an emphasized font.

1.1.1 Scenario: In area

These types of scenarios are most similar to what functionality AWHERE has. In addition to subscribing to other users, you can limit your subscription to only users in a specified area, e.g., only contacts that are in Area 1.

Example

- You are out taking a walk and are wondering if any of your friends are in the area of your favorite "hang out" place, e.g., the local park. If anyone are, you want to drop by, if not you are going to walk in another direction. You consult your mobile device and subscribe to the park. The mobile device then informs you whether any of your friends are in the park, e.g., person 5 (your friend) is in Area 2.

1.1.2 Scenario: Leaving an area

In these scenarios the user subscribed to changes in who of the contacts are in a specific area, specifically in these scenarios when a contact leaves an area. As person 3 does from Area 1.

Examples

- You are throwing a party for your friends and in order to be well prepared you subscribe on the area of your friends' homes. When they leave that area you will get a notification thereby knowing that they are on their way.
- In a game of capture the flag you want to know when a member of the opposite team, e.g., person 3, leaves your base, Area 1, with your flag. You subscribe to the area of your base and on the members of the opposite team. Whenever an enemy leaves your base you can try to intercept and re-capture your flag.

1.1.3 Scenario: Entering an area

This is the reverse of the leaving scenarios, i.e., person 1 enters Area 1.

Examples

- You are playing capture the flag and you want to know when any of your team members enters the enemy base. As in the above example you subscribe to the area of the enemy base as well as your team members. When, e.g., three or more from your team has entered the enemy base you can coordinate a tactical strike against the enemy in order to capture their flag.
- You have just gotten home from work and realize that you have forgotten some important papers. Instead of calling all your colleagues in order to find out whether they are still at work you subscribe to the area of your workplace and receives a list of the colleagues who enters, and if combined with the **in area** scenario, also of as those already in the area. Then you just call a colleague and asks if he or she will drop by with the papers.

- You are home sick and need to get some penicillin from the pharmacy. You subscribe the area around the pharmacy and if anyone on your contact list, e.g., person 4 and person 6 (notice that person 4 enters both Area 1 and Area 2 as they overlap), enters that area you can call him or her and ask for a favor.

- A thunderstorm is raging in the neighbor county and you would like to know if the storm enters the outskirts of the county you live in. You subscribe on the county borders as well as the position of the thunderstorm (the thunderstorms position could be retrieved from, e.g., a weather service). If the storm enters the county you will be notified and can take action accordingly.

1.1.4 Scenario: In the vicinity of

This is in principle also close to the functionality of AWHERE as it is an **in area** scenario, where the subscribed area is a circle or a box surrounding yourself.

Examples

- You are out browsing windows and would like to know if any of your friends happen to be in the vicinity of, e.g., 500 meters of you. You subscribe to the area with you as its center and has a radius of 500 meters. If anyone on your contact list is inside, i.e., **in area** or **enters** the defined area, e.g., person 1 and person 2 in Area 1, you will be notified and can then meet up with him or her.

- You have a friend you have not seen for a very long time. You set a proximity alert on that friend, so your phone will tell you when that friend is nearby.

1.2 World View

In the above we have shown possible scenarios where mobile users use information on areas to determine whether their any of their contacts are near. In this project we look at the different aspects as an effort to combine different worlds of technologies. We have three different worlds, namely:

GIS Database systems that are characterized by their ability to handle geographical referenced information in an efficient way. They store data in special data structures and have special spatial operators. Both for use in spatial queries.

Streams In streaming database systems the data is characterized by being non-persistent and continuously changing. Further, often in data stream systems only the present state of data is relevant.

Mobile environment In a mobile context only a relative low network bandwidth is available and the data sending cost is high compared to wired networks. Further, mobile devices have limited system resources, and therefore mobile applications must have minimal CPU and RAM requirements.

In Figure 1.2 on the next page the three worlds are illustrated as points in space, spanning a triangle. This project, being an effort to combine these worlds, is at the center of this triangle.

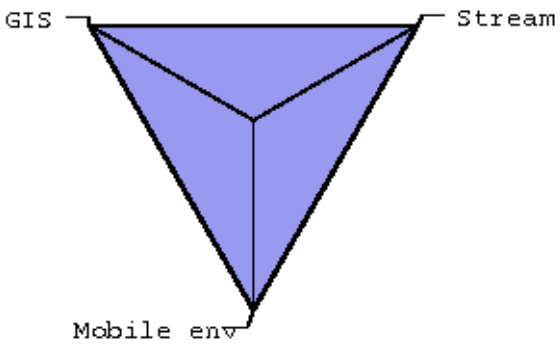


Figure 1.2: Triangle of worlds.

Another effort of this project is to implement an actual combination of these worlds and we have therefore chosen representative systems from each world and built our combination based on these. In Chapter 2 we argue for the choice of those systems. These three systems can also be illustrated as a triangle. The two triangles, one with worlds and one with systems at the corners, can be viewed as the theoretical and experimental planes of this project. Figure 1.3 shows this two-plane model. We will use this model as a reference point in the analysis of convergent points between the three worlds. In Figure 1.3, *J2ME* is not actually a system as such, but a collection of technologies that represents the mobile environment, i.e., *General Packet Radio Service (GPRS)*, *Mobile Information Device Profile (MIDP)*, *Connected, Limited Device Configuration (CLDC)*, and *Java 2 Platform, Micro Edition (J2ME)*.

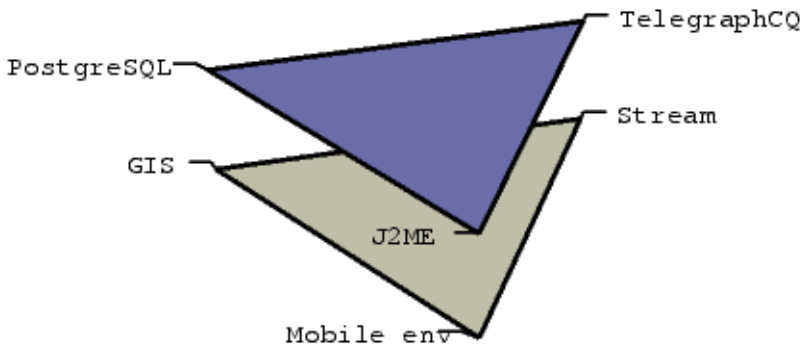


Figure 1.3: The two-plane model.

Some systems already exists and can be placed in the triangle or on one of its edges. AWHERE is such a system and would be placed on the edge between `mobile environment` and `stream`, as it works on a mobile device with an application that sends and receives streaming data to/from a DSMS. Also, the implementation that we use in the experiments, *TelegraphCQ*, is actually a convergence between `GIS` and `stream`, because it expands PostgreSQL to have stream handling capabilities. Nevertheless, we have chosen TelegraphCQ to represent the "stream" corner, for reasons that will become apparent later. More systems and their placement in the triangle will be presented in

Related Work, Section 1.4 on the next page.

1.3 Points of Analysis

Based on the triangle model presented above, this section will expand on what we know from the AWHERE project and what we need to examine further to successfully combine the three worlds of technologies.

1.3.1 Mobile Environment

From the AWHERE project we gained knowledge about the requirements and limitations of the mobile environment. The following is a list of those problems and how we dealt with them.

Which mobile devices? Because we wanted a broad user base, we chose to base our service on **mobile phones**, and consequently analyzed the requirements the area of mobile technology set for our service, and vise versa. This resulted in a requirement of MIDP 2.0 and CLDC 1.1 for phones to run the AWHERE client. For further information see [Bille et al., 2005].

Geographically locating users? At the moment the only viable solution to geographically position mobile devices, e.g., mobile phones, is **GPS**. Although only a few mobile phones today have a GPS module build-in, the tendency goes towards GPS enabled mobile phones.

Bandwidth cost? Almost all mobile phone on the market today support GPRS technology and therefore we chose to utilize this. Unfortunately, the data sending cost is relative high. This had an impact on the choice of client-server vs. broadcast and the choice of using a specialized form of continuous updates, the **RDE**. Also in this project the cost of data transfer has had impact on our choices.

Choice of networking topology? We chose a **central server** because, even though a server is a *single point of failure*, it is not practical to let the mobile phones broadcast their positions and let other phones pick this up and do the processing. This for two reasons, the limited bandwidth and the limited processing power of a mobile device.

Continuously Moving POIs? The fact that we in AWHERE had moving *Point Of Interest (POI)*, i.e., the other users, and that we wanted the position information up-to-data at all times, resulted in us choosing that users both send and receive **continuous updates**.

As this project builds on the knowledge and ideas from AWHERE all of the circumstances concerning the mobile environment that were true in AWHERE, are true in this project. As in AWHERE the mobile environment limits the feasible amount of data we can transmit between server and client and, as in AWHERE, we have continuously moving *Point Of Interest (POI)*s, and so on.

1.3.2 GIS

In order for queries to support the scenarios from Section 1.1 on page 2, they require more complex operators from the *Geographical Information System (GIS)* world. These operators rely on more complex data types, e.g., *enter(trajectory, box)*, which will be discussed in Section 4.4 on page 24. Because of this and the limitation of the mobile environment some of the operators will have to be modified to fit this new context, while still maintaining their intended functionality.

1.3.3 Streaming

In this world we also have some knowledge from the AWHERE project, here in the same format as above:

Keep all users up-to-date? To do this we chose to **push** data to the mobile phones, in contrast to *Location Based Service (LBS)* were users pull information. The data being pushed comes from other users that must send data in a continuous manner.

Continuously changing information? As data is continuously both streamed in and out, the queries, upon input data, produces output data, must also run continuously. This resulted in choosing a **DSMS** for the back-end.

The queries this time is more complex, since they also have a temporal aspect, therefore the data streamed from the mobile clients will have to be handled differently.

Because of the changes in this project, in relation to the AWHERE project, we will reevaluate the choice of DSMS. In AWHERE we created our own DSMS, but because we in this project will focus more on modifying complex GIS operators to fit the stream world, we will consider other DSMSs to find a better fit.

In summation, the main areas of the analysis are:

- DSMS systems.
- Mobile environment.
- GIS systems.
- Operators and data types in a streaming and mobile context.

Notice that we analyze two corners in the triangle, the `GIS` and the `Stream` corner, and only briefly touch the `Mobile environment` corner. The fourth item in the above list, can be viewed as the center of the theoretical triangle, i.e., an analysis of how the three worlds can be integrated.

1.4 Related Applications

According to the related work in the AWHERE project, there exists a number of LBSs. The two main categories are *people tracking* and *map based services*. With people tracking users are able to request positions of other users, without providing a map of the surroundings. In this category, the services do not provide the possibility of

requesting user appearance in specific areas. The map based services provides only a map of the surroundings and can, in general, not be used to locate moving objects, but only fixed POIs.

In BWHERE the *Location Based Service (LBS)* will utilize features from both *people tracking* and *map based services*, because people will be tracked according areas defined from a map. In the following we will look at two related services, in which it is possible to request information on users related to specific areas. Both services can be places near the center of the theoretical triangle as they both utilize streaming capability, uses mobile phones, and uses GIS to determine the positions.

1.4.1 MobiLife

MobiLife [MobiLife, 2005] is the name of a consortium developing mobile applications and services for users in their everyday life. They are currently developing a new application, which we were introduced to at the CeBIT 2005 trade fair [CeBIT, 2005] in Hannover. The application bases its positioning technology on GPS and *cell identity*, both technologies introduced in the AWHERE project. All users of the system are reporting their actual geographical position to a central server, based on an *update when moved* strategy. Furthermore, the textual description of the positions that the users are supposed to be at, according to their calendar, are reported to the server. The actual geographical positions are mapped to the textual positions at the server, an adaption process that develops over a period of time used. When a user is requesting information on other users, the position information returned will be the textual description, e.g., "office".

The system relies primarily on the user's calendar and requires that it is kept updated according to the actual geographical positions measured. Lack of updates can cause the system to adapt incorrect information, which further results in incorrect information provided for requesting users. In this system it is neither possible to request for users in a given area, nor to be informed on changes when users enter or leave a region.

1.4.2 Trygghetsmobilen

The Norwegian company ZEEKIT [ZEEKIT, 2005] has developed a product named "Trygghetsmobilen". The system is introduced to the market as a child-supervision system, where parents can monitor the motion of their children. The positioning technologies utilized are *Global System for Mobile communications (GSM)* and GPS, and the tracked positions are send to a central server. The tracked person has to allow the tracking according to the Act on Processing of Personal Data, which is performed via SMS. Likewise, SMS is used when discontinuing an already started tracking session of a person. To prevent abuse of the service, a reminder or confirmation is send to the tracked mobile phone. The system can be setup with user-defined areas, through a website, and the parents can get an alert via SMS when their child moves outside the defined agreed areas. In case the parents are just curious to see where their child moves around, they can also use the website to see the tracked phone as a spot on a map. Finally, the service can be used for emergency assistance, because you can press one button to send an alert to predefined contacts, who will get the alert with position information.

Chapter 2

Specification of Technologies

In this chapter we examine the possible technologies for the BWHERE *system and choose the best solutions for each of the areas in the world views presented in the Introduction.*

The arrows in Figure 2.1 illustrates the purpose of this chapter, namely to present the preliminary analysis of how we might combine the three worlds and further to specify the concrete technologies we use throughout this report.

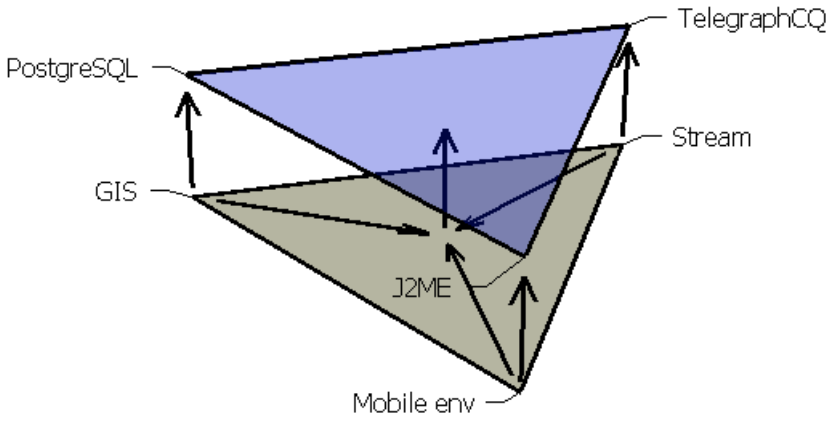


Figure 2.1: Purpose of this chapter.

As mentioned in the previous chapter we will not deal with the mobile environment in details in this report, an in this chapter not at all, because the knowledge needed to combine the three world is already listed in Section 1.3.1 in the previous chapter.

The two remaining corners are closely related, i.e., both DSMSs (in the figure denoted stream) and GISs are kinds of database systems, and will therefore not be treated separately in this chapter.

Finally, the middle vertical arrow will not be completely covered by this chapter. The full coverage on that arrow is in Chapters 5, 6, and 7.

2.1 Data Modeling and Operators

In order to support the area subscription functionality, an underlying data model for the moving objects is needed, i.e., users and regions, and a set of operators for spatial querying. There exists a number of data and trajectory models, and we have chosen the ones we have found most appropriate, in the articles in this field of research, namely the model for *spatiotemporal data types*, and *trajectory stream model*, respectively. In the following topics in the chosen data model will be examined with the purpose of determining which is needed for the system.

2.1.1 Data and Trajectory Model

When handling spatial entities that may change continuously with time, a model for these characteristic properties has to be utilized.

The model for *spatiotemporal data types* handles the data of this concept and was introduced in [Erwig et al., 1998]. The fundamental abstractions of spatial objects in spatial databases are *point*, *line*, and *region*. A *point* describes an object whose location, but not extent, is relevant. A *line* (curve in space, usually a polyline) describes facilities for moving through space or connections in space. Finally, a region is the abstraction for an object whose extent is relevant.

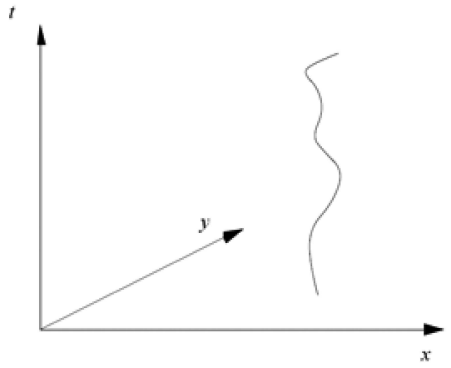


Figure 2.2: A moving point [Erwig et al., 1998].

The representation of a moving point can be illustrated as a curve in the 3D space (x,y,t), where the position (x,y) is given as a function of time (t). An example of a moving point is illustrated in Figure 2.2. It is assumed that space as well as time dimensions are continuous.

In case of a moving region, a value is set of volumes in the 3D space. A moving region at a given time t yields a region value, describing the moving region at that time.

The abstract models of continuous moving points and regions can not be implemented in computers, and therefore discrete models have to be used. The discrete models are considered as approximations or finite descriptions of infinite shapes. Examples of discrete representations of moving points and regions are illustrated in Figure 2.3 on the next page

Since we in BWHERE also only need two spatial and one temporal dimension, the presented model for spatiotemporal data types will suffice to define the moving clients and the specified regions.

Concerning trajectories, [Patroumpas and Sellis, 2004] model position data from moving obejcts with the underlying assumption that the *trajectories* of the objects are essentially continuous, time varying, and possibly unbounded data streams. The model is named *trajectory stream model*, and is restricted to point objects, and thereby handling of spatial objects are considered to have no extent. A trajectory is obtained by recording successive positions a point object takes across time. The continuous data flow from a number of sources to a central processing system is modeled as *trajectory streams*. A trajectory stream therefore only deals with historical data, and not with predictions

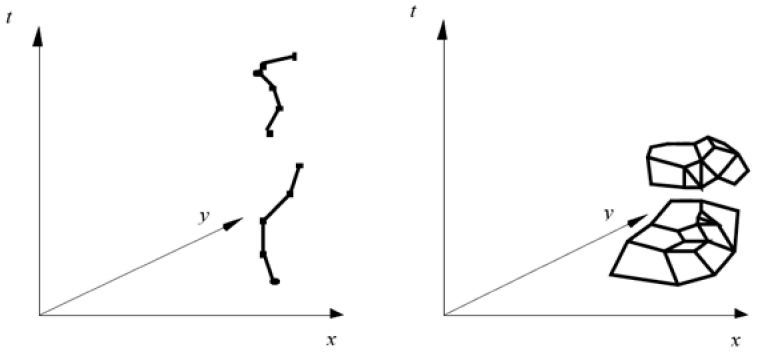


Figure 2.3: Discrete representations for moving points and regions [Erwig et al., 1998].

about future positions of moving objects.

The basic entities included in the presented trajectory stream model, can be drawn from the following main components [Patroumpas and Sellis, 2004]:

Relations for spatial relations that need to be supported for stationary entities.

Data streams which are modeled as ordered multisets of tuples. Used for data that changes with time and is based on valid time.

Trajectory streams is a specialized class of data streams, which models the continuous movement of spatial entities, thus the entity is a moving object.

Derived streams used for local views, summaries or synopses, and are derived from the base stream.

Query language for expressing continuous queries on the contents of streams and relations.

This trajectory stream model, just presented, nearly meets our requirements of the scenarios for the BWHERE system. The fact that the scope of the model is restricted to point objects must be compensated for in an implementation of the model.

2.1.2 Spatial Operators

From the need of detecting spatial relations between the clients and the specified regions, comes the need for spatial operators. Depending on the interaction types we need to detect, we can derive the specific operators.

Our system is dealing with the trajectories of the clients, based on the trajectory stream model just presented. In case of moving regions, we will have trajectories of them as well. Examples of interactions between regions and points in our system are illustrated in Figure 2.4 on the following page. In the figure, the arrows describe movement and the dotted objects are previous positions.

For detecting the types of interactions we need in this project, the novel queries, presented in [Pfoser et al., 2000], become important due to the spatiotemporal data. The novel queries are trajectory-based, and are classified as *topological* and *navigational*. The topological queries involve the whole information of the movement of an object,

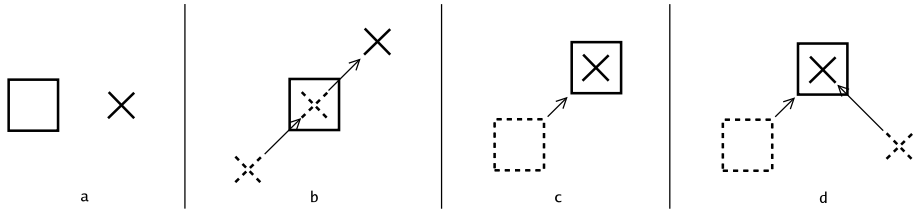


Figure 2.4: Examples of interactions between regions and points.

and the navigational queries involve derived information, such as speed and heading. Further, [Pfoser et al., 2000] distinguish between two types of spatiotemporal queries, namely:

Coordinate-based queries such as point, range, and nearest-neighbor queries in the resulting 3D space.

Trajectory-based queries involves the topology of trajectories (topological queries) and derived information, such as speed and heading of objects (navigational queries).

The trajectory-based query type is the most relevant, according to our chosen trajectory stream model for representing objects. In this category, the topological queries touches the very central part of the project, whereas the navigational queries are of little interest.

In the field of spatial operators, the 4- and 9-intersection models [Egenhofer and Herring, 1992, Egenhofer and Sharma, 1993] are well known models of how objects can interact with each other. The 4-intersection model describes binary topological relations between two objects, A and B, which are defined in terms of the four intersections of A's boundary and interior with the boundary and interior of B. The model is concisely represented by a 2x2-matrix, called 4-intersection. The 9-intersection model is an extension of the 4-intersection model, which considers the location of each interior and boundary with respect to the other object's exterior. Therefore, the binary topological relation between two objects, A and B, in R^2 is based on the intersection of A and B's interior, boundary, and exterior with B's interior. The nine intersections between the six object parts describe a topological relation and can be concisely represented by a 3x3-matrix, called the 9-intersection model. In Appendix A the 9-intersections and their geometric representations for a region and a line are shown.

According to [Pfoser et al., 2000] the definition of the 4- and 9-intersection model for spatial data is not yet available for spatiotemporal data. It has been discussed to extend SQL with the spatiotemporal versions of the basic spatial predicates *disjoint, meet, overlap, equal, covers, contains, coveredby,* and *inside*, defined by the 9-intersection model. Furthermore, the composite predicates based on the basic ones, namely enter (and its reverse, leave), cross, and bypass, have been considered too. At the current state of time, the operators have not been implemented in the SQL standard.

Since we only want to detect simple topological changes of points and regions, the composite predicates enter and leave will be sufficient for the basic system. The cross and bypass operators will not be as useful in the context of BWHERE, due to, among other factors, how humans move.

2.2 Stream and GIS

With the data and trajectory model in place, we continue by examining which parameters a DSMS for BWHERE must be capable of handling and why we cannot make the same choices in this project as in our last. This section concludes by choosing a DSMS for BWHERE.

2.2.1 AWHERE-DSMS

In our last project, AWHERE, we developed a DSMS, the AWHERE-DSMS, which utilized the RDE technique. In the AWHERE system, RDE was the best solution since we wanted to minimize the cost for the clients. This was done by modeling queries in the *Limited Continuous Query Language (LCQL)* developed to accompany RDE. When taking area subscription into consideration, LCQL and RDE may not be powerful enough to express and evaluate whether someone has entered or left an area.

To illustrate this consider the following scenario. A human travels, while walking, at about five to six kilometers per hour, or roughly 500 meters in five minutes. If a user of AWHERE has set the update frequency, i.e., the `REPEAT` clause in LCQL, to its maximum 5 minutes and has defined a relatively small area, e.g., a 100 by 100 meter area as shown in Figure 2.5, the user might not detect that someone had entered and then left the area, since the distance of the area's diagonal can be traversed in less time than the time between the user's updates.

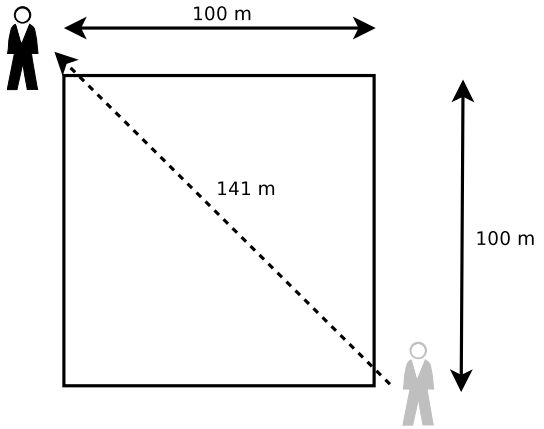


Figure 2.5: The small area is traversed within the update interval.

Even if the update interval is at its lowest, i.e., 30 seconds, areas can be defined too small since a person can traverse roughly 50 meters in 30 seconds.

A naive approach to overcome this problem will be to set the frequency to zero seconds, i.e., `REPEAT 0`, letting the query run all the time and thereby simulating a continuous query. As we chose not to implement a query optimizer in AWHERE-DSMS the resources would be exhausted fast, as shown by the experiments in [Bille et al., 2005].

Considering the limitations in RDE, LCQL, and AWHERE-DSMS we look at implementations of DSMSs which utilizes true continuous evaluation of queries hereby abandoning

System	Motivating app.	Data handling	Reference
Aurora	sensors	streams	[Aurora, 2005]
COUGAR	sensors	streams	[COUGAR, 2005]
Gigascope	network monitoring	streams	[Cranor et al., 2003]
NiagaraCQ	Internet XML databases	relations	[NiagaraCQ, 2005]
OpenCQ	tracking changes to web pages	Internet and relations	[OpenCQ, 2005]
StatStream	data analysis	streams	[StatStream, 2005]
STREAM	all-purpose	streams and relations	[STREAM, 2004]
TelegraphCQ	sensor data	streams and relations	[TelegraphCQ, 2005]
Tribeca	network traffic analysis	single input stream	[Tribeca, 1998]

Table 2.1: DSMS systems.

RDE, LCQL, and the AWHERE-DSMS as a solution for the BWHERE system.

2.2.2 Other DSMSs

In [Bille et al., 2005] we examined some of the implementations in the field of DSMSs, listed in Table 2.1. The way most of these DSMSs handle data are via streams only. As we base BWHERE on AWHERE we need support for relations as well as streams making these DSMSs unsuitable for BWHERE.

The remaining DSMSs which handle streams as well as relations are TelegraphCQ and *STREAM*. Both DSMSs are suitable for BWHERE.

In order to choose which DSMS to use, we take area subscription into consideration and how areas can be defined in these DSMSs. As mentioned in Section 2.1 on page 9 we need to be able to model moving objects as well as the subscribed areas. Modeling the moving objects, i.e., users, can be done by using trajectory streams, also mentioned in Section 2.1.

Using streams for the areas is only sensible if areas are non-static, i.e., Scenario 1.1.4: *Vicinity of*. When subscribing on static areas it is, however, not a sensible solution, because it would create more traffic, in that data on areas would be transmitted on every update. As all communication is done using GPRS, transmitting this data will only end up in a higher cost for the users.

Storing static areas in a GIS is more efficient. Most systems that handle geographic data is referred to as a GIS. Internally a GIS stores the geographic data in a *Database Management System (DBMS)* capable of handling the data. As the queries in BWHERE use spatial operators, we require that the DSMS have GIS capability or is able to connect to a GIS. With the above considerations in mind we examine TelegraphCQ and STREAM.

From [Patroumpas and Sellis, 2004] we can see that both TelegraphCQ and STREAM

can express spatial entities, however, only TelegraphCQ has built-in spatial data types, functions, and operators as it is built on top of the *PostgreSQL* DBMS.

In STREAM an area is defined by its range of coordinates and in order to detect whether a person is inside an area the query will need to model this. Consider a simplified stream, `person (x,y)`, which is transmitted by the person in Figure 2.6. The query to detect if the person is inside the area would be:

```
SELECT x,y
FROM person
WHERE x>=2 AND x<=8 AND y>=2 AND y<=8
```

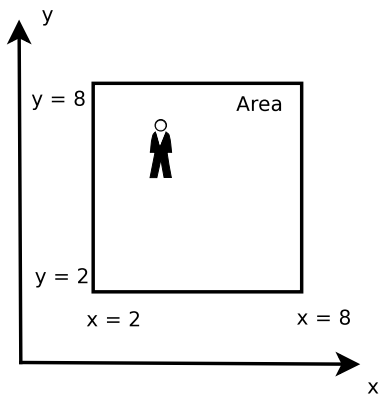


Figure 2.6: Detecting the person.

In TelegraphCQ we take advantage of the spatial data types in PostgreSQL. Consider the simplified stream, `person (position)`. The query is now:

```
SELECT position
FROM person
WHERE position @ box '((2,2),(8,8))'
```

The *box* is a PostgreSQL data type and the operator @ means `Contained or on`. Note that the @ operator is not a part of the SQL standard.

When comparing the STREAM query with the TelegraphCQ query, the query from TelegraphCQ expresses the same as the STREAM query in more powerful way, since it takes advantage of the spatial data types and operators from PostgreSQL. TelegraphCQ has another advantage, as it seemingly is not possible to connect STREAM to an external GIS. A disadvantage of both TelegraphCQ and STREAM is that currently they do not support sub-queries [TelegraphCQ, 2005, STREAM, 2004].

By the presiding arguments in this section, TelegraphCQ seems to be the best DSMS solution to use in BWHERE.

Chapter 3

Project Statement

In this chapter we summarize the technologies from Chapter 1 and Chapter 2 and state what this project is intended to examine and document.

In Chapter 1 we presented the triangle of worlds in which the corners represents the mobile environment, streams, and GIS. In Chapter 2 we proceeded to specify representative systems for the stream and GIS worlds. As we had covered the mobile environment in AWHERE we chose not to specify this world. Further, we specified the data and trajectory model to be used and the operators to accommodate BWHERE. The specified technologies that are situated on the corners of the triangle of worlds are:

Mobile environment: The mobile environment is the same as it was in AWHERE.

Stream: TelegraphCQ was selected as the best solution for handling the stream functionality.

GIS: PostgreSQL was chosen in combination with TelegraphCQ, due to the integration of the two systems.

The remaining technologies are data and trajectory modeling, and operators. The data model is the spatiotemporal model, and the trajectory model is the trajectory stream model, which combined handles the moving points and regions. The operators, enter and leave, accommodates the needs in BWHERE. In the triangle of worlds, the operators are situated in the center and can be thought of as, what binds the worlds together.

The purpose of this project is to examine the presented technologies in order to accommodate the demands of the intended Location-Aware System. The problem can thereby be split in these goals:

1. A theoretical solution to the problem of combining the three worlds of technologies. A solution that is based on the spatiotemporal model and the trajectory model. Further, a solution that includes both basic and composite spatial and spatiotemporal predicates.

2. An experimental system design and implementation, based on the theoretical solution. The experimental system will be implemented with the use of these technologies: J2ME, GPS, GPRS, TelegraphCQ, and PostgreSQL.

3. Proving that the experimental system can function with a real mobile phone in a satisfactory way.

4. Proving that the experimental system scales in a acceptable manner according to an estimation of a reasonable number of users.

In the following chapters the technologies of the theoretical solution will be analyzed and the technologies will be utilized to design and implement the execution solution.

Chapter 4

Theoretical Analysis

In this chapter we concentrate on the theoretical triangle presented in the Introduction, and present theory supporting that a system combining the three world of technologies is feasible.

The analysis is built in four steps; first a brief analysis of the stream world, that will summarize what we know from AWHERE*, second, we briefly recap on the knowledge we gained on mobile environment. Third we give a detailed analysis of GIS theory, and finally we analyze how to deal with the problems springing from combining the three worlds.*

The chapter concludes with a summation of how a system like BWHERE *could work and further shows a conceptual architecture of it.*

4.1 Streams

As mentioned in the Introduction on page 1, we need continuous queries in the BWHERE system, in order to provide the users with continuous position updates. DSMSs are designed to handle these type of queries, whereas the traditional DBMS is targeted at applications that require persistent data storage and complex querying [Golab and Özsu, 2003]. The BWHERE system does not fit the data model and querying paradigm of the DBMS, because information will occur in the form of sequences (streams) of data values reported between the clients and the server.

A data stream is defined in [Golab and Özsu, 2003] as "*a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items*". The order in which items arrive is impossible to control and it is not feasible to locally store a stream in its entirety. Moreover, as in AWHERE we are not interested in saving old data, because it is just that, old. Since most people are mobile and do not stay in the same place all the time, and since systems as AWHERE and BWHERE only are interesting if they can tell you where people are now, old data is irrelevant.

Since the data should be pushed continuous to the users in the BWHERE system, users will have to request the data through a continuously evaluating query. Such queries are described as "*long-running, continuous, standing, and persistent*" [Golab and Özsu, 2003]. As in AWHERE each user of BWHERE will, in most cases, only have one query running on the server, which we referred to as a subscription. A query can, however, involve multiple streams from other users, and also relations.

There are two different strategies when evaluating continuous queries; the time-based, and the event-based approach. In the time-based approach query evaluates at specified intervals. Whereas, in the event-based approach the queries evaluate when new data arrives. At first glance the latter of the two seems the most sensible, because data is evaluated and send on as fast as possible, but when in a mobile environment, speed and always up-to-date data is not the only things that matter. When the data sending cost is high, as it is in a mobile environment, it should be left up to the end user how he/she will weigh speed and accuracy against cost.

In AWHERE we chose a time-based evaluation strategy, namely RDE, for the reason mentioned above. In this project the continuous queries are a bit different. This time speed is a bigger factor, in that we would like to know as quickly as possible when one of our contacts enter one of our subscribed regions. Of course, data sending cost is still a factor, especially in queries that does not involve a enter or leave, and it is certainly possible to create at system that support both evaluation strategies, but since the focus of this project is to combine mobile environment and streams with operators from the GIS world, the best choice of evaluation strategy is the event-based strategy.

A final similarity with AWHERE, is that the ratio between incoming and outgoing streams is very close to one-to-one. All users on the system have one position stream to the server, and all users have one (possibly two) continuous queries resulting in an stream. There might be an in-stream to the system that does not have a query, e.g., if the disaster-warning functionality is implemented. Further, some users might have two queries if a system contains event- and time-based query evaluation, then a user might have one of each.

The one-to-one feature is quite unique for these systems in the stream world. Most systems have many input streams and a few queries. This fact must be considered when designing systems like AWHERE and BWHERE, because it will influence the scalability of the system.

4.2 Mobile Environment

In AWHERE the goal was to develop a service that could provide users with information on other users location via their mobile phone. This goal is still present in BWHERE and as mentioned in Section 1.3.1 on page 6 we utilize many of the same technologies which we used in AWHERE. In this section we briefly recap on these technologies. Then we give an estimate on the possible number of potential users who could be interested in a system such as BWHERE.

In the following we list the technologies used:

GPS As GPS is available globally, everybody can use it provided they have a GPS-receiver. This makes GPS suitable for LBSs, as network based solutions may have restrictions depending on the operator. It is also the most accurate positioning system available which in a mobile context is important.

J2ME J2ME is with its constituents, MIDP and CLDC, the most widespread programming environment used on mobile phones.

GPRS Reliable and fast communication is essential in a mobile environment. Using GPRS to communicate between the mobile phones and the server provides a stable and, in the mobile environment, a relative fast connection.

As a goal of BWHERE is to service users we look at statistics to determine the number of potential users.

We estimate that the age of our target group is between 15 years and 30 years. We take base in the city of Aalborg, Denmark where, according to Statistics Denmark, there live close to 40,000 people with the age between 15 and 30 years. Also, according to Statistics Denmark, 85% of the Danish population had in 2003 a mobile phone. [DST, 2004]

In order to give a real estimation on the number of potential users we looked at the system *CBB Mobil Messenger*, developed by the mobile phone service provider CBB Mobil. The system was an online chat service that enabled the customers of CBB to chat with each other. The service was launched in January 2005 and closed in March 2005 due to dispute between CBB and Microsoft. In the time the service was available the client application had been downloaded more than 10,000 times. [CBB, 2005]

This indicates that the population of Denmark has an interest in using such systems. If we presume that at least 10% of our target group is interested in BWHERE then in Aalborg alone, about 4,000 people can be considered as potential users.

4.3 GIS

As mentioned in Section 2.1 Egenhofer (in collaboration with various others) has done extensive research in the field of spatial relations, including topological relations between regions and the 9-intersection model [Egenhofer and Herring, 1992]. We need topological relations between points and regions and have based our analysis on Egenhofer and will therefore briefly present Egenhofer and Herring's categorization of topological relations between regions.

But first, to fully understand the Egenhofer and following analysis, we will introduce some basic algebraic topology.

4.3.1 Algebraic Topology

The spatial data model in algebraic topology is based on *cells*. Cells are primitive geometric objects defined for a specific spatial dimension: A 0-cell is a cell with zero extend, also known as a *node* or *point*. A 1-cell is a link between two 0-cells. A 2-cell is three 1-cells connected by non-intersecting lines, and so on. In this project we will, however, not consider more than two spatial dimensions. In GISs 0-, 1-, and 2-cells, are called: points, lines, and regions, respectively. Further, we define a *face* of an n-cell A to be any 0–n-cell that is contained in A. [Egenhofer and Herring, 1992]

We now define the closure of an n-cell, denoted by $\overline{A}$, and then the boundary δA, interior A^o, and exterior A^- of an n-cell ($\mathcal{U}$ denotes universe). [Egenhofer and Herring, 1992]

$$\overline{A} = \bigcup_{r=0}^{n} r\text{−face} \in A$$

$$\delta A = \bigcup_{r=0}^{n-1} r\text{−face} \in A$$

$$A^o = \overline{A} - \delta A$$

$$A^- = \mathcal{U} - \overline{A}$$

We will return to these formulae to discuss relations between points and regions, which we will use in BWHERE, but first we will examine how relations between regions are defined.

4.3.2 Relations Between Regions

The 9-intersection model, introduced in Section 2.1.2, is so named because there are nine ways the three sets, exterior, boundary, and interior, of two regions can relate. In R^2 this yields $2^9 = 512$ possible relations, however, not all of these can be realized, e.g., in some the region boundary will have to be disconnected[1]. In this project we will only look at regions with a connected boundary, for two reasons. First, it is hard to imagine a scenario where a BWHERE user would like to define a region with disconnected boundaries and second, none of the DSMSs and DBMSs we have examined, support regions with disconnected boundaries. Egenhofer and Herring finds that only eight of the 512 relations can be realized if the region must have connected boundaries. Those eight are graphically presented in Figure 4.1.

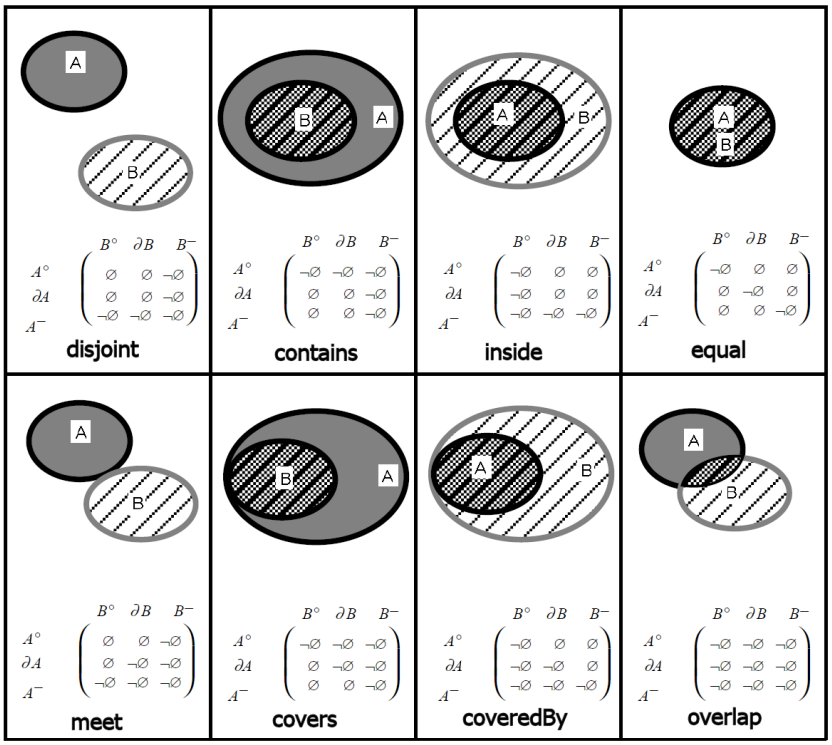


Figure 4.1: The eight relations between regions.

4.3.3 Relations Between Regions and Points

To determine how points relate to regions, we will take offset in the above. Then we must first determine the exterior, boundary, and interior of a point P, i.e., a 0-cell:

$$\overline{P} = \bigcup_{r=0}^{0} r\text{−face} \in P = P$$

[1] A disconnected boundary is a boundary where you cannot get from any point on the boundary to any other point by only tracing the boundary, e.g., a region with a hole in the middel, will have a disconnected boundary.

$$\delta P = \bigcup_{r=0}^{-1} r\text{–face} \in P = \varnothing$$

$$P^o = \overline{P} - \delta P = P - \varnothing = P$$

$$P^- = \mathcal{U} - \overline{P} = \mathcal{U}/P$$

From the equations above we see that a point has no boundary, it's interior consists of a single point, and the exterior is everything except that single interior point.

By examining Figure 4.1 and imagining the B region as a point, it can quickly be determined that *contains* and *inside* not can be distinguished from *covers* and *coveredBy*, respectively, because a point's boundary set is empty.

Further, by examining the case of *equal*, one discovers that $B^- \cap A^o$ and $B^- \cap \delta A$ can only be empty if $\overline{A} = \overline{B}$. Ergo, if B is a point and B *equal* A, then A must be a point too. Therefore, a point cannot be *equal* to a region.

Furthermore, the case of *overlap* is by equivalent logic. If all intersections are non-empty and one the "regions" only contains one entry, i.e., $|\overline{B}| = 1$, then the other region can only contain one entry too. Ergo, there can be no overlap between a point and a region.

Finally, a point is a 0-cell and therefore can be a face in a region (2-cell) and is therefore in either[2] A^-, δA, or A^o. Ergo a point P can have one of three relationships to a region A described with the following predicates:

1. P disjoint A or A disjoint P, if $P^o \in A^-$
2. P meet A or A meet P, if $P^o \in \delta A$
3. P inside A or A contains P, if $P^o \in A^o$

With the algebraric topology and relation in place we will move on to define spatiotemporal predicates.

4.3.4 Spatiotemporal Predicates

The spatial predicates from the above section, only evaluate at a single point in time, but if a spatial predicate is valid for each point in time in a certain time interval, we have a basic *spatiotemporal* predicate. These are written like the spatial predicates, but with initial capital letter, and can be combined to create more complex predicates describing *developments*. A development is a series of spatial or spatiotemporal predicates that evaluate to true, if the predicates evalutate to true in sequence. [Erwig and Schneider, 1999] defines these developments:

Enter := Disjoint meet Inside
Leave := Inside meet Disjoint
Cross := Disjoint meet Inside meet Disjoint
Touch := Disjoint meet Disjoint
Bypass := Disjoint Meet Disjoint
Graze := Disjoint meet Overlap meet Disjoint

First, *Graze* can be eliminated, since it requires a predicate that does not exist between points and regions, namely *Overlap*. We know *enter* and *leave* from the Introducion, the rest is visualized in Figure 4.2 on the next page.

[2] Only one of them, because their intersection is empty.

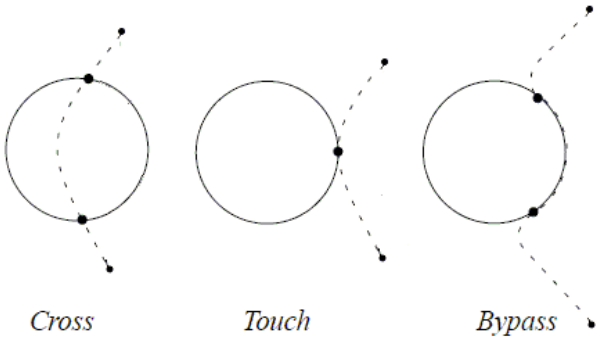


Figure 4.2: Visualization of *Cross*, *Touch*, and *Bypass*.

These development predicates evaluate on a sequence of (point, time)-pairs, a trajectory. Trajectories are, naturally, also a part of GIS systems, and are simply an abstraction over a sequence of (point, time) pairs.

In the project we will focus mainly on enter and leave and will therefore in the following discuss only these.

When we examine the definition of the development predicates enter and leave we discover that to implement these predicates, without modification, time will have to be continuous, we need to know the position of a given point at any given time, because, e.g., *meet* is true only at single point in time. When we consider this in context with the mobile enviroment, it is not feasible to stream positions at very low intervals, thereby making time as continuous as posible, because it will be too expensive for the user.

Because of the above, we will need to redesign the development predicates, while still preserving their intented function. This analysis and redesign will be presented in the next section.

4.4 Spatiotemporal Operators

In this section the operators selected for the BWHERE project, i.e., enter and leave, are further analyzed. The analysis provides a theoretical solution for the enter and leave operators, which will constitute the basis of the design and implementation of these.

4.4.1 Geometric Representation

The movement history of the objects are referred to as trajectories, and transferred via trajectory streams. This data constitutes the basis for the operators enter and leave.

An abstract model of trajectories, described in [Erwig et al., 1998], allows us to view a moving point as a continuous curve in the 3D space (x, y, t). In [Egenhofer and Herring, 1992] this would be a 0-cell with a timestamp, this because Erwig discuss spatiotemporal entities, whereas Egenhofer only discuss spatial entities.

The abstract model results in an arbitrary mapping from an infinite time domain into an also infinite space domain. This is, however, not an appropriate representation in a computer for storage and manipulation. The discrete representations are therefore

the only usable. "Data structures and algorithms have to work with the discrete models of the infinite point sets." [Erwig et al., 1998]

In order to compensate for the discrete representation, an approximation method like linear approximation can be utilized between points. A curve in space, e.g., a client of BWHERE, is thereby described by a polyline, and a region, e.g., a subscription area, is described in terms of polygons. The linear approximations are attractive because they are easy to handle mathematically.

4.4.2 Operators on Streams

The predicates of the enter and leave operators were introduced in Section 4.3.4 on page 23. Making the operators work on streams, sets requirements to the operators, due to available data and possible data loss.

Example
In [Patroumpas and Sellis, 2004] the enter operator is simulated with CQL in the DSMS STREAM by use of two points with an `antisemijoin`[3]. In the following CQL example from [Patroumpas and Sellis, 2004], the simulated enter is illustrated with the query *"Find all vehicles entering now into the area of interest"*. The utilized stream is `Vehicles (vID, vType, x, y, t, TS)` describing (Vehicle ID, Vehicle type, x position, y position, valid time, transaction time). The simulation is carried out by the `antisemijoin` between two temporary relational views. The *InsideAreaNow* view stores vehicles located now inside the area, and *InsideAreaRec* keeps those recorded within the area in the last two time instances, utilizing a partitioning window for each object. [Patroumpas and Sellis, 2004]

```
InsideAreaNow:  SELECT vID, t
                FROM Vehicles NOW
                WHERE x>=475000 AND x<=480000
                AND y>=4204000 AND y<=4208000
InsideAreaRec:  SELECT vID, t + 1 AS t
                FROM Vehicles PARTITION BY vID ROWS 2
                WHERE x>=475000 AND x<=480000
                AND y>=4204000 AND y<=4208000

SELECT InsideAreaNow.vID, InsideAreaNow.t
FROM InsideAreaNow ANTISEMIJOIN InsideAreaRec
```

In Figure 4.3 the CQL example with the simulated enter is illustrated, neglecting the actual x and y positions. In the left side of the figure, two time instances of the same area is illustrated. The numbers are referring to Vehicle IDs, and the arrow indicates the movement of Vehicle ID 2, which makes an enter at time t = 0. In the middle of the figure, the relevant streamed information from the three vehicles is provided in the table Vehicles. According to the CQL example there has to be performed an `antisemijoin` between the vehicles inside the area now ($t = 0$), and the vehicles recorded inside the area in the last two time instances ($t = 0$ and $t = -1$). The result of the two respectively

[3] `Antisemijoin` of R and S is defined as the multiset of tuples in R that do not agree with any tuple of S in the attributes common to both R and S.

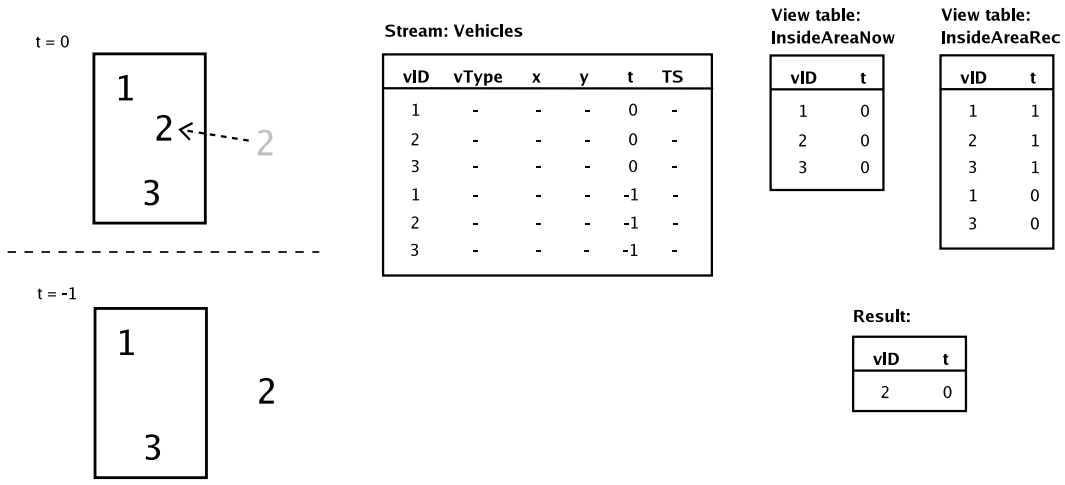


Figure 4.3: Example of simulated enter.

temporary relational views, InsideAreaNow and InsideAreaRec, is shown in the right side of the figure. The result of the `antisemijoin`, and thereby the multiset of tuples in InsideAreaNow that do not agree with any tuple of InsideAreaRec in the attributes common to both InsideAreaNow and InsideAreaRec, is $vID = 2$ and $t = 0$. The result is illustrated as the table Result in the figure. ■

According to [Patroumpas and Sellis, 2004] the enter operator could not be simulated similarly in TelegraphCQ, due to lack of support for nested subqueries and views. On the contrary, other queries involving geometric functions like *distance between* (<->), *Intersects or overlaps* (?#), and *Contained or on* (@) [PostgreSQL, 2005], are formulated rather simply.

The example takes advantage of the monotonicity of time, and is thereby assuming that position information is available for every time instant. In the streaming environment of BWHERE this will be rather inefficient and could not be guaranteed either.

4.4.3 Operator Algebra

Pursuing the idea of basing the enter and leave operators in BWHERE on two points without approximation, as mentioned in the above example, demands new definitions of the operators. The new operators thereby omit the meet predicate in the definitions presented in Section 4.3.4 on page 23, and leads to the following definitions:

Enter := Disjoint Inside
Leave := Inside Disjoint

As illustrated in the previous example with the simulated enter, meet can be presumed to have taken place, when the first point was "disjoint" and the second "inside". Likewise for the leave operation. The two reference points must be successive concerning time, although there might be several time instances in between. No approximation is necessary, due to the calculation method. The operators can thereby be defined by

use of two registered successive and timestamped points.

To define the needed formal algebra for the enter and leave operators, the basic entities point and region have to be defined first, as well as the trajectory providing the data.

The formal algebra of the point domain can be defined as follows.

Definition 1 (Point domain)
The underlying domain is a two-dimensional Euclidean space, where a point is defined as a pair of coordinates with a timestamp added $p = (x, y, t)$. *The coordinates* $x, y \in \mathsf{R}$ *and* $t \in \mathcal{T}$, *which is the time domain represented by the natural numbers. The set of all the points is denoted by* $\mathcal{P}$. *[Civilis et al., 2004]* ■

In [Erwig et al., 1998] they define a moving point, which is represented by a sequence of quadruples. The moving point thereby contains trajectory information. "Each quadruple (p_i, t_i, b_i, c_i) contains a position in space p_i and a time t_i. It also contains a flag b_i which tells whether the point is defined at times between t_i and t_{i+1} (b_i = true). This allows for the representation of partial functions (at the conceptual level). Finally, there is a flag c_i which states whether between t_i and t_{i+1} a stepwise constant interpretation is to be assumed, i.e., the point stayed in p_i, did not move (c_i = true), or linear interpolation, i.e., a straight line between p_i and p_{i+1}, is to be used (c_i = false). This representation has been chosen in order to support different classes of applications for moving point, e.g., unique events, stepwise constant locations, etc." [Erwig et al., 1998]. Since we omit the *meet* predicate, and do not user any interpolation or approximation between points, the above point domain definition will suffice for the BWHERE application.

Definition 2 (Region)
The region definition is limited to a square-shaped area. A region is defined by two points $A, B \in \mathcal{P}$, *where* $A.x \leq B.x$, $A.y \leq B.y$, *and* $A.t = B.t$. ■

In [Erwig et al., 1998] they also define a moving region, which represents a set of volumes in the 3D space (x, y, t). An intersection with the set of volumes at a given time t, yields a region value, describing the moving region at time t.

Based on the point definition just presented, the trajectory domain can be presented as follows.

Definition 3 (Trajectory domain)
A trajectory tr *is defined as a tuple of points* (p_1, p_2), *where* $p_1, p_2 \in \mathcal{P}$ *and* $p_1.t < p_2.t$. *The set of all the trajectories is denoted by* $\mathcal{TR}$. ■

The point and region definitions, combined with the trajectory definition, can be used to represent the moving points and regions needed in BWHERE.

Once the point, region, and trajectory definitions are stated, the enter and leave functions can be defined. The enter function can be defined as follows.

Definition 4 (Enter)
The enter function can be defined as follows.

$$\begin{aligned} Enter(tr, r) = \{ b \,|\, b \in \{true, false\} \wedge tr \in \mathcal{TR} \wedge r \in R \wedge \\ r.B.x < tr.p_1.x < r.A.x \ \wedge \ r.B.y < tr.p_1.y < r.A.y \wedge \\ r.A.x \leq tr.p_2.x \leq r.B.x \ \wedge \ r.A.y \leq tr.p_2.y \leq r.B.y \ \} \end{aligned}$$

■

Definition 5 (Leave)
The definition of leave is as follows.

$$\begin{aligned} Leave(tr, r) = \{b \,|\, b \in \{true, false\} \wedge tr \in \mathcal{TR} \wedge r \in R \wedge \\ r.A.x \leq tr.p_1.x \leq r.B.x \ \wedge \ r.A.y \leq tr.p_1.y \leq r.B.y \wedge \\ r.B.x < tr.p_2.x < r.A.x \ \wedge \ r.B.y < tr.p_2.y < r.A.y \ \} \end{aligned}$$

■

4.5 Summation

We here summarize in key points and requirements, that the different worlds of technologies must include/support to create a combination that will work and further what extra is needed to combine them.

- Streams
  Event-based evaluation To best fit the needs in area subscription, event-based evaluation of continuous queries is the best choice.
  One-to-one The system must be capable of handling as many queries as input streams.
  Old data irrelevant In this category of system only current data is relevant.
- Mobile environment
  GPS The mobile device must have a GPS receiver either build in or be capable of communicating with an external one to determine its position.
  GPRS The position data should be send through GPRS, so the device must, of course, support this data communication protocol.
- GIS
  Support of needed data types The GIS system must have data types supporting points, regions, and possible trajectories or the GIS must support user-defined data types.
  Point-region predicates Again, the GIS must either support user-defined functions/operators or must have the predicate *inside*.
- Operators
  Trajectory of 2 points In order to provide the enter and leave functionality, a trajectory of 2 points is needed.
  Enter and leave The operators must be defined to operate on the points and regions defined, and naturally simulate the composite predicates enter and leave.

This combination is illustrated in Figure 4.4 on the facing page which shows a conceptual overview of the architecture for a system binding the three worlds together.

In the figure the three world from the triangle; mobile, stream, and GIS, is represented by the clients, DSMS, and GIS, respectively. Operators represent what is needed to bind the the three worlds together, i.e., the center of the triangle.

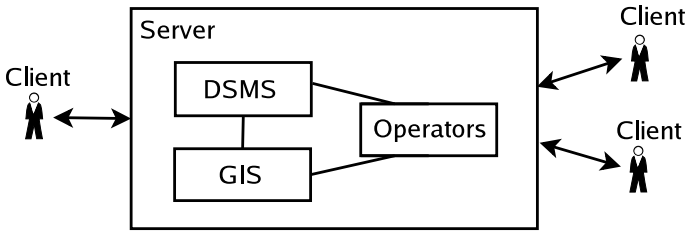


Figure 4.4: BWHERE architecture.

How exactly the architecture will look in systems like BWHERE may differ from system to system, but it must contain these four elements. The exact architecture for BWHERE will be described in the next chapter.

Chapter 5

Experimental System - Overview

In this chapter and the following we will, based on the theoretical analysis and design of systems in the center of the triangle, describe the experimental system, and thereby proving the theory. With an offset in the conceptual architecture, this chapter presents an overall system architecture.

As in Chapter 2, the arrows in Figure 5.1 illustrates the purpose of this and the following two chapters, namely to describe the developed system in the experimental plane.

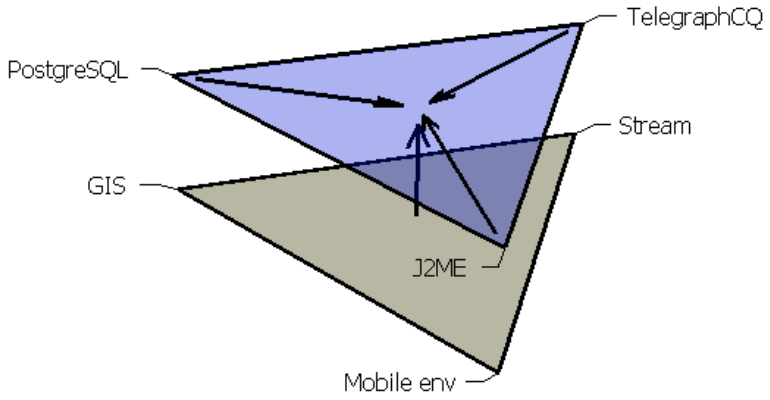


Figure 5.1: Purpose of this and the two following chapters.

The arrow from mobile environment is handled in the Client chapter, Chapter 6, and the two remaining horizontal arrows in the Server chapter, Chapter 7. This chapter describes the overall architecture for BWHERE and how we progress from the theoretical solution to the experimental solution, i.e., the vertical arrow that was started in Chapter 2.

5.1 System Architecture

The architecture presented in Figure 5.2 on the next page depicts an ideal architecture of BWHERE. The architecture is based on the conceptual architecture shown in Figure 4.4 on page 29, but with the specific DSMS and GIS systems from Chapter 2 on page 9, namely TelegraphCQ and PostgreSQL. Because TelegraphCQ builds on PostgreSQL and since the extensions we make, denoted operators, affect PostgreSQL, we here depict these as nested boxes.

In this architecture the clients stream data to the server, i.e., TelegraphCQ, initiates continuous queries on the data and receives the results accordingly. This architecture works well if the clients are capable of issuing SQL queries. Although an SQL API exists for J2ME, the *JDBC Optional Package for CDC/Foundation Profile* [JSR169,

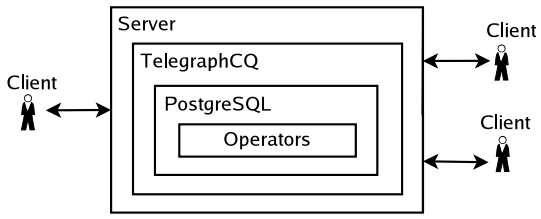


Figure 5.2: Ideal system architecture.

2004], we have chosen not to use it as not all mobile phones will be able to utilize it.

An architecture that takes this issue into consideration is therefore presented in Figure 5.3.

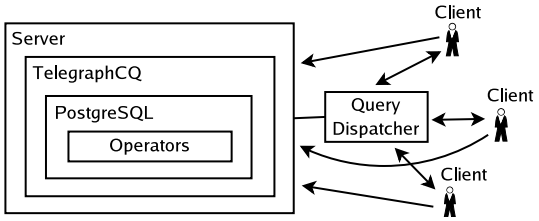


Figure 5.3: Modified system architecture.

The modified architecture is very similar to the ideal architecture, except a *Query Dispatcher* has been put in between the clients and the server. The mobile clients still stream data to TelegraphCQ but queries are routed through the Query Dispatcher. The Query Dispatcher receives queries from clients and initiates the queries on the server and returns the results to the right client.

In the following two chapters we will address issues on the client- and server-side separately. The Query Dispatcher will be handled in the Client chapter, since it is needed only because of limitations in J2ME and since both relate to the mobile environment corner of the triangles. Whereas the server side of BWHERE in concert covers the remaining two corners and the center of the triangle.

Chapter 6

Experimental System - Client

In this chapter we will discuss the considerations taken in developing the mobile client application. This discussion will be brief and will not cover all aspects of developing for the mobile environment, because we covered that in the AWHERE *project.*

Further, this chapter describes the Query Dispatcher, which is needed to bind the mobile client to the server, in that J2ME does not include an SQL API.

6.1 Mobile Client Application

In BWHERE the requirements to the client are very similar to the ones presented in AWHERE. The design of the client for BWHERE will therefore rely on the existing design from AWHERE, and only the new subjects will be handled here.

The first subject to be handled is the frequency the clients are reporting their positions and receive subscription updates. In the AWHERE prototype the subscription update frequency was equal to the position report-in frequency. The frequency was specified as a part of the queries, defined in the query language LCQL, due to the use of RDE. Since we base BWHERE on the continuous query language provided in TelegraphCQ, and the evaluation is continuous, we can not specify the either of the frequencies. The frequency of the evaluation, and thereby the subscription updates, will be based on the incoming data from other clients. What we can specify separately, is the position report-in frequency. Ideally, this frequency should be determined according to an update strategy, e.g., update when moved. Since this is not the focus of this project, we set the frequency to a predefined value.

The second subject to be handled is area specification and subscription. Concerning the definition of the areas, we have seen in Section 1.4 that the existing system "Trygghetsmobilen" utilized a web application for specifying the areas. The areas specified in this case were directly converted into queries running on the server. This feature requires a website providing a map system with an underlying conversion of selected areas to position information, i.e., GPS or GSM information. The area information could also be read out on the computer screen and typed in on the mobile client. We will, however, not implement these features as we deem them as out of the scope of this project. Instead, we will define a number of predefined regions on the mobile client, which will be used in the queries. The areas will be referred to by a name, and the actual position information will therefore be transparent for the user. Regarding subscriptions, this was handled through the website in the "Trygghetsmobilen". According to [ZEEKIT, 2005] the only subscription option was leave, which was targeted at detecting when child was getting outside the agreed area. In BWHERE both enter and leave functionality are provided, and therefore the user must specify what to subscribe on for each area. This will be provided on the mobile client, since we choose not to implement a website.

6.2 Query Dispatcher

In Section 5.1 on page 31 we introduced the Query Dispatcher which routes queries and results to the respective clients.

In Figure 6.1 the flow of data inside the Query Dispatcher is depicted. The Query Dispatcher consists of three threads, a *listen thread* which listens for new client connections, a *client thread* that handles communication between the Query Dispatcher and the client and finally a *query thread* which handles the query.

When a client initiates a connection, the listen thread accepts this connection and spawns a client thread. The listen thread then returns to listen state. When a client thread is spawned it proceeds to receive the query string. If the query string is valid, i.e., a valid SQL statement a new thread is spawned, the query thread. In the query thread a connection to PostgreSQL is opened on which the query operates. Just before the query is executed a cursor is defined and used. Whenever new results are available the Query Dispatcher sends them to the client. When the client no longer wants to be a part of BWHERE the "quit" string is sent to the client thread using the same connection as the query was sent on. Upon receiving the "quit" string the client thread sends a close signal to the query thread which terminates the query in PostgreSQL. Hereafter the client is disconnected from the BWHERE system. The Query Dispatcher only handles the routing of queries and their results. It is up to the client application to close the stream to TelegraphCQ upon disconnection from BWHERE.

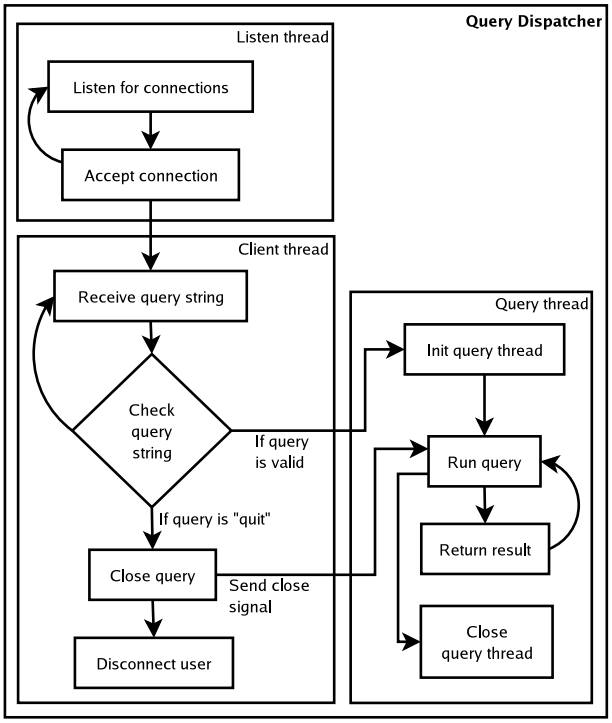


Figure 6.1: The Query Dispatcher.

Chapter 7

Experimental System - Server

In this chapter we describe the server side of Bwhere. *Before we discuss the design of the enhancements we make to PostgreSQL and TelegraphCQ, we need an analysis of the inner workings of these.*

Despite the fact that TelegraphCQ still is an early prototype, and therefore the implementation does not include all the intended functionality, we have designed Bwhere *as if TelegraphCQ was fully functional.*

After the presentation of the optimal system design, we describe how we have dealt with TelegraphCQ's limitations in the implemented prototype of Bwhere.

7.1 PostgreSQL

The chosen GIS for Bwhere, PostgreSQL, started as a research project at the Berkeley Computer Science Department and has evolved into the most advanced open source database [PostgreSQL, 2005]. PostgreSQL supports SQL92 and SQL99 and can be extended by the user in several ways, e.g., custom data types, functions, and operators.

In PostgreSQL all of the types needed for handling 0-,1-, and 2-cells exist, and they are the following:

Point

`point` A point in a plane consisting of a coordinate set.

Line

`line` An infinite line, represented by two points.

`lseg` A line segment, consisting of a start and end point.

`path` Either a closed or open path, depending on the input syntax. A path consists of a series of points (minimum two). In closed paths, first and last point are assumed to be connected.

Region

`box` A simple rectangular box, represented by two points that are opposite corners of the box.

`polygon` Essentially the same as a closed path, but is stored with some additional information and has more functions supporting it.

`circle` A circle in a plane. Represented by a point and a radius.

7.1.1 Enhancements to PostgreSQL

Given the above we here describe the how we will use PostgreSQL and which enhancements we need for Bwhere.

The data types in PostgreSQL that are closest to the definition of the 0–2-cells, point, line, and region, are `point`, `lseq`, and `polygon`, respectively. In Bwhere we will

use `point`, `lseq`, and `box`, simply because `box` is easier to work with, and conceptually it is not different from `polygon`, in that a rectangular box is a subcategory of a polygon.

Trajectories, which we also need, can not be directly mapped to a data type in PostgreSQL. According to the trajectory definition in Section 4.4.3 on page 26, a trajectory is a set of two points, ordered in time, and conforming to the point definition. This definition can be implemented in PostgreSQL, since PostgreSQL can be extended with user specific data types.

Concerning the enter and leave operators, we will adhere the definitions presented in Section 4.4.3 on page 26. Thereby we omit the meet predicate, which can be presumed to have taken place, as described earlier. The actual point and time of the meet event can therefore not be determined with this solution, but it is also of no interest in the context of BWHERE. We just want to detect that an enter or leave action has occurred, and not the actual time and place of the event. In PostgreSQL the enter and leave operators can be defined by combining boolean expressions containing the basic geometric data types, mentioned in Section 4.4.1 on page 24, with the geometric operators [PostgreSQL, 2005]. In the following, examples of combined expressions for the enter and leave operators are defined. The only geometric operator utilized is @ (contained or on).

Enter (Disjoint Inside):
(`<Previous point>` @ `<Area>` = False) AND
(`<Current point>` @ `<Area>` = True)

Leave (Inside Disjoint):
(`<Previous point>` @ `<Area>` = True) AND
(`<Current point>` @ `<Area>` = False)

As it can be seen in the example expressions, the *Inside* and *Disjoint* predicates, can be expressed by use of the @ (contained or on) operator, and combined to the enter and leave operators.

7.2 TelegraphCQ

In Section 2.2 on page 13 we selected TelegraphCQ as the DSMS for BWHERE. In this section we analyze TelegraphCQ further in order to gain knowledge on how TelegraphCQ works internally, thereby enabling us to modify it to fit our needs. First an overview of TelegraphCQ's architecture is given. Secondly, we look at how data from external sources are streamed into TelegraphCQ via *wrappers* and how we use wrappers in BWHERE. Finally, we examine how to define streams and how to query upon these streams. The main source of information for this section is [Krishnamurthy et al., 2003].

7.2.1 TelegraphCQ Architecture Overview

As TelegraphCQ is built in top of PostgreSQL it inherits a lot of functionality from PostgreSQL. In PostgreSQL, whenever a client connects, a new process is spawned, i.e., a process-per-connection model, which then handles the query from the client. This is illustrated in Figure 7.1 on the next page by the Postmaster which forks a new PostgreSQL server. The components marked in dark grey represents components which have been changed slightly in TelegraphCQ. The components marked in light grey represents components that have been changed significantly in TelegraphCQ. In

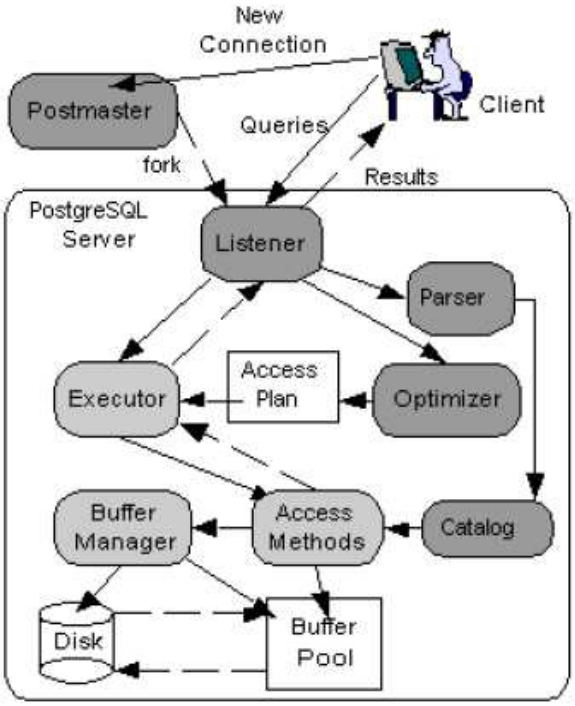


Figure 7.1: PostgreSQL architecture [Krishnamurthy et al., 2003].

order to accommodate streams, continuous queries, shared processing and adaptivity in PostgreSQL, the TelegraphCQ developers modified the PostgreSQL architecture into the one depicted in Figure 7.2 on the following page.

The TelegraphCQ Back End is a dedicated process that uses shared memory to execute shared continuous queries. In the Back End the query executor uses *eddies*[1] for tuple-routing and operator scheduling. It also uses *fjords*[2] for inter-operation communication. In the TelegraphCQ Front End they have kept the process-per-connection model which receives queries from, and returns results to, the client as well as handle non-continuous queries. The TelegraphCQ *Wrapper ClearingHouse (WCH)* handles the operators used to control the data ingress and makes tuples ready for processing.

7.2.2 Wrappers

When an external source streams data to TelegraphCQ the WCH process takes care of the data acquisition. The operations that the WCH is responsible for, are the following [Krishnamurthy et al., 2003].

- Accepting connections from external sources, and obtaining the stream name they advertise.
- Loading the appropriate user-defined wrapper functions.

[1] Refer to [Avnur and Hellerstein, 2000] for an in-depth explanation of eddies.
[2] Refer to [Madden and Franklin, 2002] for a detailed description of fjords.

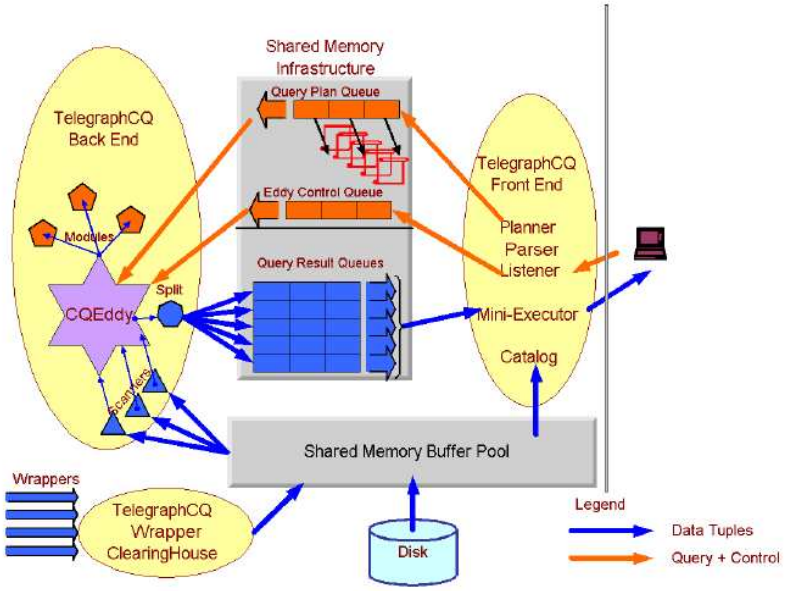


Figure 7.2: TelegraphCQ architecture [Krishnamurthy et al., 2003].

- Calling these wrapper functions when data is available on a connection.
- Processing result tuples returned by a wrapper according to the stream type.
- Cleaning up when a source ends its interaction with TelegraphCQ.

When a source sends data, the WCH accepts a connection and based on the name of the stream advertised by the source, loads the appropriate functions. When the wrapper has enough data to form a tuple the wrapper converts the tuple into the expected PostgreSQL data types defined by the stream. When converting a tuple the wrapper must return PostgreSQL data types that can be used to create the tuple in PostgreSQL format. The three functions of the wrapper are:

init: The initialization function will typically allocate the memory and states to be used during execution.

next: The next function is the worker which produces new records by processing data from the source an turning it into PostgreSQL data types.

done: The done function, frees memory and performs clean-up of the states.

The WCH also has the responsibility of making the data available to the rest of the system and is done by using the standard PostgreSQL buffer pool and putting the tuples in it. If the stream is an archived stream the tuples are flushed to disk. If the stream is unarchived it is not flushed to disk.

External Sources

The external source from which the stream receives its data can be defined by the user such as it fits the user's requirements. It is the task of the user-defined wrappers to handle the data. The only requirement imposed is that the first few bytes sent to TelegraphCQ contains the name of the stream so TelegraphCQ can identify the stream. Many sources can simultaneously push data to the same stream.

7.2.3 Wrapper for BWHERE

Our external sources are the mobile clients. Each mobile client sends a string containing comma separated values in the following format:

```
user_id, gps_time, latitude, longitude
```

Latitudes and longitudes have been converted into decimal degrees by the mobile client as they also did in AWHERE. An example string will look like this:

Example
`1,141924.017,57.0140083333334,9.98698333333334`. ■

Before sending the values we send the name of the stream, e.g., "Bwhere", to TelegraphCQ.

CSV++ Wrapper

The TelegraphCQ distribution comes with a generic wrapper, which expects data in a comma separated form, the *csvwrapper*. As the data from our external sources is comma separated we use a slightly modified version of the csvwrapper. We have modified it because the operators we use, enter and leave, takes a trajectory and a box, and the external source only sends the present position. So our wrapper, called *csvppwrapper*, takes the inputted position and the last inputted position from that external source and creates a two-point trajectory of them.

The last known position of a given external source is in memory in a hash table. The key for the hash table is presently the in-streamed `user_id`. This, however, may cause problems if the external source stops streaming without sending a "quit" signal, e.g., the mobile phone runs out of power. The problem arises when that user signs on next time, because the old position from last time still exists, and if the user has moved the system will receive a false trajectory, and may return false data.
The problem can happen in two situations:

1. The user was in a region that some of his/her contacts subscribes to when the battery ran dry. Then he/she leaves the region and does not sign on before he/she is in the region again. Then his/her contacts will not receive an enter.

2. Same situation as above, but he/she signs in again after an extended period of time, and outside the region. Then there will be a leave, but it will be delayed.

The problem could be handled by using a *session id* as key in the hash table. The session id should be generated by the client application in a manner that would make

sure that the id was unique[3]. If the mobile client, for some reason, crashed, it could at next start-up ask the user whether the previous session should be resumed or a new should be started.

The problem can also arises if we extend BWHERE with more operators, e.g., cross, but even then the problem is relatively small and we will not deal with it in BWHERE.

Creating Wrappers

The three user-defined functions, `init`, `next`, and `done`, that the data from the external source use, must be set up in TelegraphCQ before the wrapper itself is set up.

To declare the functions, the `CREATE FUNCTION` statement from PostgreSQL is used. When the functions have been created the wrapper is created by the `CREATE WRAPPER` statement. In our system the wrapper and its functions would be declared as follows.

```
CREATE FUNCTION Bwhere_init(integer) RETURNS boolean
AS 'libcsvpp.so', 'csvpp_init' LANGUAGE C;

CREATE FUNCTION Bwhere_next(integer) RETURNS boolean
AS 'libcsvpp.so', 'csvpp_next' LANGUAGE C;

CREATE FUNCTION Bwhere_done(integer) RETURNS boolean
AS 'libcsvpp.so', 'csvpp_done' LANGUAGE C;

CREATE WRAPPER BwhereWrapper (init=Bwhere_init,
                              next=Bwhere_next,
                              done=Bwhere_done);
```

7.2.4 Streams and Queries

In order for users to interact with streams in TelegraphCQ they have to define a stream via *Data-Definition Language (DDL)* statements in TelegraphCQ, create an external source to send data to the stream, create a wrapper which handles the received data, and finally issue a continuous query on the stream.

Creating Streams

New DDL statements for creating and dropping streams have been introduced in TelegraphCQ. The statements are `CREATE STREAM` and `DROP STREAM` and are similar to those used to create and drop tables. When defining a stream in TelegraphCQ, a requirement is imposed as the stream must have exactly one column of the type time. This column serves as a timestamp for the stream and is specified by the `TIMESTAMPCOLUMN` keyword. The timestamp is not a part of the data transmitted from the external source but is inserted into the stream by TelegraphCQ.

When the content of the stream has been defined it is necessary to specify whether the type of the stream is `ARCHIVED` or `UNARCHIVED`. An `ARCHIVED` stream will insert the received data into a relation with the same name as the defined stream. `UNARCHIVED` streams use shared memory queues to communicate results between the executor and the WCH. [TelegraphCQ, 2005]

[3] The session id could be based on the GPS time in seconds.

To illustrate how a stream is defined we define the stream to be used in BWHERE. As mentioned in Section 4.1 on page 19 we are not interested in old data, hence we create an unarchived stream called `Bwhere`. In order for the wrapper to know which stream it is associated with, it is added to the stream.

```
CREATE STREAM Bwhere ( user_id int,
                       gps_time varchar(15),
                       latitude double precision,
                       longitude double precision,
                       trajectory trajectory,
                       tcqtime TIMESTAMP TIMESTAMPCOLUMN
                     ) TYPE UNARCHIVED;

ALTER STREAM Bwhere ADD WRAPPER BwhereWrapper;
```

The `trajectory` is a trajectory stream, see Section 4.4.3 on page 26, defined in PostgreSQL as a user-defined data type and was explained in Section 7.1 on page 35.

Issuing Queries

After a stream has been created and the data flows into TelegraphCQ, continuous queries can be issued over these data. If a query is operating on one or more streams it is identified as being continuous and do not end until the users cancels it. In our example a query could look like this:

```
SELECT b.user_id, u.user_name
FROM Bwhere b, users u
WHERE b.user_id = u.user_id
WINDOW b ['10 minutes'];
```

The `WINDOW` clause defines a sliding window and is optional. It restricts the amount of data that the query evaluates over to the defined period of time.

Queries can be issued by clients which use supported interfaces such as ODBC and JDBC. As the queries are continuous they never end so the clients must use named cursors when declaring queries. If cursors are not used, PostgreSQL will buffer the results of the query until all results have been received, which will not happen since the query never ends.

7.3 Limitations of TelegraphCQ

In Section 2.2 on page 13 it was mentioned that TelegraphCQ does not support subqueries. This indicates that queries containing streams does not support the full `SELECT` syntax of SQL. In Section 7.2.4 we also saw the use of a `WINDOW` clause which indicates that the `SELECT` statement has been modified to accommodate streams. In this section we show the modified syntax of the `SELECT` statement, then list other limitations of TelegraphCQ and comment on the impact these might have on the BWHERE system. The modified `SELECT` statements has the following form:

```
SELECT <select_list>
FROM <relation_and_stream_list>
WHERE <predicate>
GROUP BY <group_by_expressions>
WINDOW stream[interval], ...
ORDER BY <order_by_expressions>;
```

The `WINDOW` clause may contain one window expression for each stream in the query. The interval of the windows can be any PostgreSQL interval data type.

The following list from [TelegraphCQ, 2005] shows the restrictions where the clauses in the `SELECT` statement does not behave like the PostgreSQL select statement.

- Windows may not be defined over PostgreSQL relations.
- `WHERE` clause qualifications that join two streams may only involve columns, not column expressions or functions.
- `WHERE` clause qualifications that filter tuples must be of the form `column OP constant`.
- The `WHERE` clause may contain `AND`, but not `OR`.
- Subqueries are not supported.
- `GROUP BY` and `ORDER BY` select clauses may only appear if the query also contains a window.

The fact that subqueries are not supported has some impact on the BWHERE system. As long as BWHERE is a prototype system the impact is not significant as a workaround can be made, however, such a solution is unacceptable for a finished system. In the following we will address these limitation and modify the design presented in Section 7.4.

During the analysis we discovered that TelegraphCQ only supports 32 simultaneous queries which means that we can only connect a maximum of 32 clients at any time. This limitation has a major effect on the scalability of the system as the ability to serve many clients is one of the goals of this project. A detailed description of this limitation is given in the following section.

7.3.1 Scalability of TelegraphCQ

In this section we will explain what the cause of the maximum number of queries limitation is, and comment on a solution on how to overcome it.

In a default installation of PostgreSQL the maximum amount of simultaneous users is set to 32. Two of these connections are reserved for super users. In order to allow more users the options `max_connections` and `shared_buffers` in the configuration file for PostgreSQL, `postgresql.conf`, is set to higher values, e.g., 64 and 128, respectively. The minimum value of `shared_buffers` is recommended to be `max_connections` x 2. If the `max_connections` option is set too high PostgreSQL might fail to start. Adjusting *system V IPC parameters* such as semaphores and shared memory enables PostgreSQL to start successfully.

```
uint16
AllocQueue()
{
        uint16          qno;

        SpinLockAcquire(shmResultQueueBitmapLock);
        for (qno = 0; qno < NUMRESULTQUEUES; qno++)
        {
                if (!(*shmResultQueueBitmap & (1 << qno)))
                {
                        *shmResultQueueBitmap |= (1 << qno);
                        break;
                }
        }
        SpinLockRelease(shmResultQueueBitmapLock);
        if (qno >= NUMRESULTQUEUES)
                elog(ERROR, "Out of result queues.");
        return qno;
}
```

Listing 1: Excerpt of `rqdest.c`.

In TelegraphCQ some settings need to be altered as well, in order to allow more clients. Most of these settings are set in the header files of the TelegraphCQ implementation. The default amount of users allowed to stream data into TelegraphCQ is set to 64. The maximum amount of simultaneous queries, the `NUMRESULTQUEUES` constant, is set to 32. When raising this number to, e.g., 64, we discovered that only 32 clients could connect. When client 33 tried to connect an error occurred yielding the following information: `Out of result queues`. This error was traced to the file `rqdest.c` which handles functions necessary for result delivery in TelegraphCQ. An excerpt of the file, Listing 1, shows in line 16, that if the `qno` variable becomes larger than or equal to the `NUMRESULTQUEUES` constant the error occurs. The reason for this error is because of the bit manipulation of the `shmResultQueueBitmap` variable. This variable is an unsigned 32 bit integer, `uint32`, in which a bit is set for every client that connects, ending up in a maximum of 32 clients.

To overcome this limit in TelegraphCQ, the handling of these result queues would need to be reimplemented in order to support more simultaneous queries. This, however, would affect other parts of TelegraphCQ and how it is integrated with PostgreSQL. As is would take too long time to implement a new solution, we chose to keep TelegraphCQ.

7.4 Optimal Design of BWHERE

With the knowledge from Section 7.1 and Section 7.2 we continue with the application data design for BWHERE. This design is "best-case" design, a design where we do not consider the fact that TelegraphCQ is still an early prototype including the limitations listed in the previous section and therefore does not include all the features that the people behind TelegraphCQ intended to give it.

7.4.1 Table Design

In Figure 7.3 the Entity-Relationship (E-R) diagram for BWHERE is depicted. Since streams are not part of the standard E-R specification, we have chosen to represent it by parallelogram. The attributes trajectory and TCQTIME are not streamed in, but added, as described earlier, by the *csvppwrapper* and are therefore depicted with dotted circles. Finally, the arrow denotes that a stream tuple relates to exactly one user, but user can have many stream tuples.

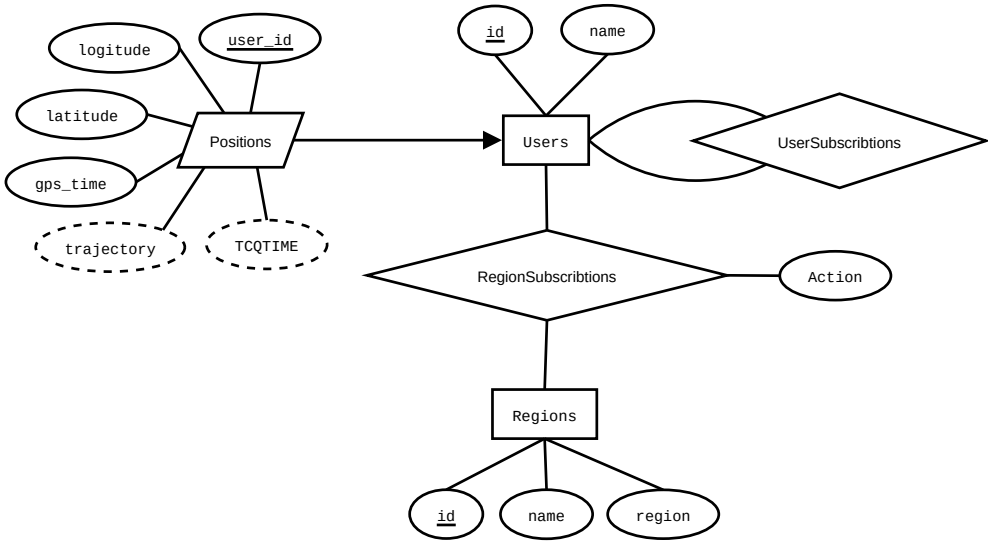


Figure 7.3: E-R-diagram for BWHERE.

7.4.2 User Query

In Listing 2 on the next page the SQL query that is sent by the mobile user is shown. The mobile client replaces <myUserId> with the user id the user has entered. The + and − operators represents enter and leave, respectively. The query returns the following

Contact name The name of the contact that the user subscribes to.

Action Whether this was an enter or leave.

Region name The name of the region, which the action applied to and which the user subscribes to.

Time The GPS time, when the action occurred at the contact.

Whether the just streamed in user id is among a given users contacts, is done by the sub-query, which returns the user ids of the contacts that the user subscribes to. To get the action and the region name we select only RegionSubscriptions that match this user's id. Further, the RegionSubscriptions region id must match Regions id and Users id must match the in-stream's user id. Finally, the trajectory, that the wrapper created, must either enter or leave the subscribed region. Because of the nature of enter and

```
SELECT u.name, rs.action, r.name, st.gps_time
FROM Users u, Positions st, RegionSubscriptions rs, Regions r
WHERE st.user_id IN (
        SELECT subscribee
        FROM UserSubscriptions
        WHERE subscriber = <myUserId> )
AND rs.user_id = <myUserId>
AND rs.region_id = r.id
AND u.id = st.user_id
AND ( (rs.action = 'e' AND st.trajectory + r.region)
    OR (rs.action = 'l' AND st.trajectory - r.region)
);
```

Listing 2: The query every user on the systems uses.

leave, both of them never occur at the same time, and therefore it is safe to put both of them in the same query, and not make one query for enter and one for leave. The RegionSubscriptions table includes the action for a given region and therefore either a enter or leave is returned.

7.5 Modified Design

Three of the limitations presented in Section 7.3 has an impact on the system design in the previous section; *no subqueries*, *no **OR***, and *filter must have the form column operator constant*. We will describe the impact and the solution in the following three sections.

7.5.1 No Subqueries

The first filter in the **WHERE** clause in our query (see Listing 2) uses a subquery to only select the users that `myUser` subscribes to.

To handle this we have let the mobile client form this section in the naive way, by replacing the sub-select by a number of **OR**s, e.g.,:

WHERE st.user_id = 12 **OR** st.user_id = 42 **OR** ....

This, however, means that the mobile client knows the ids of the contacts that the user of the mobile client subscribes to. By using the naive approach another problem surfaces as the user will not know which contact has entered or left an area[4]. Further, this work-around uses **OR**s, which is another limitation, one we will handle in the next section.

7.5.2 No OR

From boolean algebra theory we have DeMorgan's Theorem [Wakerly, 1999], which allows us to create an **OR** using **AND** and **NOT**. DeMorgan's Theorem in shown in

[4]This could be handled by using a query for every contact.

Equation 7.1.

$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{7.1}$$

Which can yield:

$$A + B = \overline{\overline{A} \cdot \overline{B}} \tag{7.2}$$

Which can expand to:

$$A + B + C... = \overline{\overline{A} \cdot \overline{B} \cdot \overline{C}...} \tag{7.3}$$

Therefore, we can replace any **OR** in our query. To increase the readability of the query we have create an SQL function, `dmOR`, that convert the expression. `dmOR` is defined as listed in Listing 3.

```
CREATE FUNCTION dmOR(boolean, boolean)
RETURNS boolean
AS 'SELECT ((NOT ($1)) AND (NOT ($2)))=FALSE'
LANGUAGE SQL;
```

Listing 3: Definition of the DeMorgan OR, `dmOR`.

7.5.3 Column Operator Constant

Fortunately, this restriction only applies to qualifications that filter the tuples, and not the qualifications that acts as joins. Therefore in our query this only affects the qualifications:

$$\text{rs.user\_id} = \text{<myUserId>}$$

and

$$\text{st.trajectory} +/- \text{ r.region}$$

The first one is already on the allowed form and the latter can be modified to to fit the form in this way:

$$(\text{st.trajectory} +/- \text{ r.region}) = \textbf{TRUE}$$

In this way we can create a functioning system, although not an optimal one, but given the unfinished nature of TelegraphCQ and the fact that BWHERE is in the prototype stage we find this acceptable.

Based on this design we implemented an experimental system with which we have conducted some experiments.

Chapter 8

Implementation

In this chapter we present the prototype implementation of BWHERE. *First we present the client side implementation. This is is followed by the Query Dispatcher which binds the client and server together. Finally the enter and leave operators are presented which constitutes the server side implementation in PostgreSQL.*

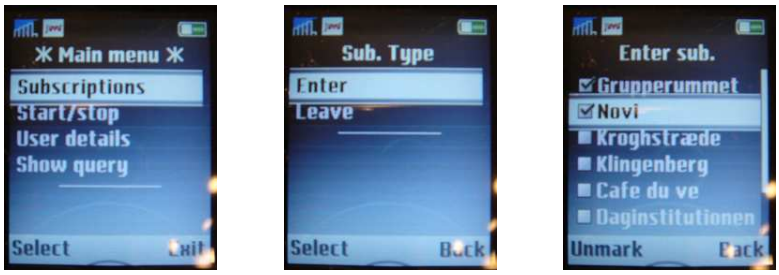
8.1 Client

The implementation of the client application will only be described with respect to the components that are different from the AWHERE implementation. As described in Section 6.1 on page 33, the subjects are "report-in and receive subscription updates" and "area specification and subscription". The implementation of these solutions are described in the following.

Concerning report-in and receive subscription updates, the former is implemented with a simple sleep function, that waits for a predefined fixed interval of 30 seconds. The interval can thereby not be changed while running. The receive subscription updates subject is left up to TelegraphCQ, since it provides requested updates as it receives new position information. The development source code of the client can be found in Appendix C on page 71.

The handling of area specification and subscription is implemented with predefined areas, as described in Section 6.1 on page 33. In order to specify areas and report in GPS positions, a GPS module is needed. The GPS modules at our disposal, were both utilizing Bluetooth communication. The mobile platforms available to us, two Sony Ericsson K700i mobile phones, have no support for the Bluetooth API [JSR82, 2002]. By use of an optional package, the Bluetooth API could be supported, but the access might be restricted on the specific platform. Therefore, we have decided to measured 192 GPS positions on a 45 minutes walk in the neighborhood, which we have used as reference and test data. From these data we have defined eight areas of 100 by 100 meters, by use of the program GPS Utility [GPSUtility, 2005]. All of the GPS positions have been converted to decimal degrees for easier handling. The route of 192 GPS points intersects (enters and leaves) all of the areas. The predefined areas and points have then been stored in a `RecordStore` in the client.

Concerning subscriptions, the user can select a subscription option from the main menu, as illustrated in Figure 8.1(a). In the subscription menu, Figure 8.1(b), the two categories "Enter" and "Leave" can be selected. Subsequently, the eight predefined areas will appear on both lists, as illustrated for enter in Figure 8.1(c). The selected areas will be stored for enter and leave, respectively, and the area information from the `RecordStore` will be utilized for generating the query. User subscriptions is handled via the subselect statement shown in the ideal query, see Listing 2 on page 45. Although this can not be expressed on the server, because of the limitations in TelegraphCQ, we have chosen to implement the ideal query on the client. The user interface source code of the client can be found in Appendix C.1 on page 71.



(a) Main menu (b) Subscription list (c) Enter list

Figure 8.1: BWHERE application.

For testing purpose, the 192 points can be accessed in the `RecordStore`, and used for simulating the 45 minutes walk, through the eight areas.

8.2 Query Dispatcher

The implementation of the Query Dispatcher is very similar to the design of the Query Dispatcher presented in Figure 6.1 on page 34. The source code of the Query Dispatcher can be found in Appendix B on page 67.

We chose to implement the Query Dispatcher in C as we wanted to minimize the risk of turning the Query Dispatcher into a performance bottleneck. The three treads of the Query Dispatcher, the listen thread, the client thread and the query thread, is implemented as follows.

The listener thread is implemented as the `main` function in Appendix B where it is the content of the `while` loop, line 53 to 56, that accepts a new connection, starts the client thread and then returns to the listen state.

The client thread is the `readQuery` function, line 64 to 96,found in Appendix B. The client thread is the only part that is not implemented exactly as presented in Figure 6.1 on page 34 since we do not make a check whether the query is a valid SQL statement. Instead we only see if the query differ from "quit" string and if so starts the query thread. If not it sets the `close` variable in the `mystruct` struct to 1. Further, a check has to be made to see whether at least one query is running before allowing the thread to be terminated. Currently, if a "quit" string is sent and no query is running the Query Dispatcher will not handle this correctly.

The query thread is implemented in the `connect_to_postgres` function found in Appendix B on page 68, line 107 to 190. This thread starts by initializing the connection to PostgreSQL and setting up the cursor for use. It then retrieves results in every pass of the `while` loop where it sends the result to the client via the `send_reply` function. The `send_reply` function uses the same connection to send results to client as the client thread uses to receive queries from the client.

8.3 PostgreSQL

The implementation of the enter and leave operators, and the trajectory data type is relatively straight forward and will therefore not be described in great detail.

PostgreSQLsupports programming languages for the user to write extensions in, e.g., C, Perl, and SQL. We chose to write in C, because, of the supported languages, it is the most well known to us, and because of its efficiency.

8.3.1 Operators

Listing 4 shows the enter function as it is implemented in PostgreSQL. Enter takes two arguments, `trajectory` and `box`, both in the form of pointers. We then determine if the oldest `point` in the `trajectory` is *inside* the `box`. This is done by calling the PostgreSQL function that encapsulates the *inside* predicate between `points` and `boxes` called `on_pb`[1]. Next we do the same with the newest `point` in the `trajectory`. If the old `point` was *disjoint* and the new `point` was *inside* we return `true`, else `false`.

```
#include "enter.h"

Datum
enter_tb(PG_FUNCTION_ARGS)
{
  Trajectory *traj = PG_GETARG_TRAJECTORY_P(0);
  BOX *b = PG_GETARG_BOX_P(1);

  bool old_in = DatumGetBool(DirectFunctionCall2(on_pb,
                                                 PointPGetDatum(&(traj->oldPoint)),
                                                 BoxPGetDatum(b)));
  bool new_in = DatumGetBool(DirectFunctionCall2(on_pb,
                                                 PointPGetDatum(&(traj->newPoint)),
                                                 BoxPGetDatum(b)));

  if (!old_in && new_in)
    PG_RETURN_BOOL(1);
  else
    PG_RETURN_BOOL(0);
}
```

Listing 4: C-code for the *enter* function.

The *leave* function is defined completely analogue to *enter*, and will therefore not be presented.

```
#ifndef __TRAJECTORY___
#define __TRAJECTORY___

#include "postgres.h"
#include <stdio.h>
#include "utils/geo_decls.h" // spatio functions

typedef struct traj_s {
  Point oldPoint;
  Point newPoint;
} Trajectory;


#define DatumGetTrajectoryP(X) ((Trajectory *) DatumGetPointer(X))
#define TrajectoryPGetDatum(X) PointerGetDatum(X)
#define PG_GETARG_TRAJECTORY_P(n) DatumGetTrajectoryP(PG_GETARG_DATUM(n))
#define PG_RETURN_TRAJECTORY_P(x) return TrajectoryPGetDatum(x)

PG_FUNCTION_INFO_V1(traj_in);
PG_FUNCTION_INFO_V1(traj_out);
PG_FUNCTION_INFO_V1(points_traj);

#endif
```

Listing 5: trajectory.h.

8.3.2 Trajectory

When defining a new data type, one specify, at minimum, two functions, one for converting an input string to the internal data representation, and one for the opposite operation. Further, one, also specifies the internal representation. Listing 5 shows the file `trajectory.h`, which include the definition of the internal representation of the data type `trajectory`.

The code for converting an input string to the internal representation is listed in Listing 6 on the facing page which is an excerpt of the file `trajectory.c`. The code for the opposite function is not included here, because it is just that, the opposite, and therefore not interesting.

8.3.3 SQL commands

The SQL commands needed for telling PostgreSQL how to use the function described in the before is shown in Listing 7 on page 52:

[1] `on_pb` is the function PostgreSQL invokes when the @-operator is used. The @-operator is overloaded, but which function is called with a specific combination of parameters, can be determined by examining the `pg_operator` table in the `pgcatalog` database.

```
#include "trajectory.h"

/* traj_in - convert a string to internal form.
 *
 * External format: "(f8, f8); (f8, f8)"
 */
Datum
traj_in(PG_FUNCTION_ARGS)
{
  char        *str = PG_GETARG_CSTRING(0);
  Trajectory *traj = (Trajectory *) palloc(sizeof(Trajectory));

  if (sscanf(str, " (%lf , %lf ) ; ( %lf , %lf ) ",
             &(traj->oldPoint.x),
             &(traj->oldPoint.y),
             &(traj->newPoint.x),
             &(traj->newPoint.y))
      != 4){
    traj->oldPoint.x=0;
    traj->oldPoint.y=0;
    traj->newPoint.x=0;
    traj->newPoint.y=0;
  }
  PG_RETURN_TRAJECTORY_P(traj);
}
```

Listing 6: trajectory.c.

```
--Trajectory
CREATE FUNCTION traj_in(cstring)
RETURNS trajectory
AS 'trajectory'
LANGUAGE C;

CREATE FUNCTION traj_out(trajectory)
RETURNS cstring
AS 'trajectory'
LANGUAGE C;

CREATE TYPE trajectory (internallength=32, input=traj_in, output=traj_out);

--Enter and leave
CREATE FUNCTION enter(trajectory, box)
RETURNS boolean
AS 'enter', 'enter_tb'
LANGUAGE C;

CREATE FUNCTION leave(trajectory, box)
RETURNS boolean
AS 'leave', 'leave_tb'
LANGUAGE C;

--Operators
CREATE OPERATOR + (leftarg = trajectory, rightarg = box, procedure=enter);
CREATE OPERATOR - (leftarg = trajectory, rightarg = box, procedure=leave);
```

Listing 7: SQL commands

Chapter 9

Software Test

In this chapter we present a test strategy for the Bwhere *system, which serves as a validation of the program. Acceptance of these tests are required to guarantee that the system conforms to the specifications. In addition, we describe the assessments we have conducted on the development system.*

In the following we present a test strategy for the Bwhere system. Because of Bwhere being a development system, an extensive test of the code will not conducted. In addition, the limitations of TelegraphCQ, as presented in Section 7.3, have demanded changes in the development system. These changes should not appear in a final version of the software, which is another reason for not conducting a completely test of the current version.

9.1 Test Strategy

According to [Biering-Sørensen et al., 1996] there are several ways and levels of testing software systems and the following approach is what we have estimated would be a thorough and sufficient test strategy.

The Bwhere system consist of the components illustrated in Figure 9.1.

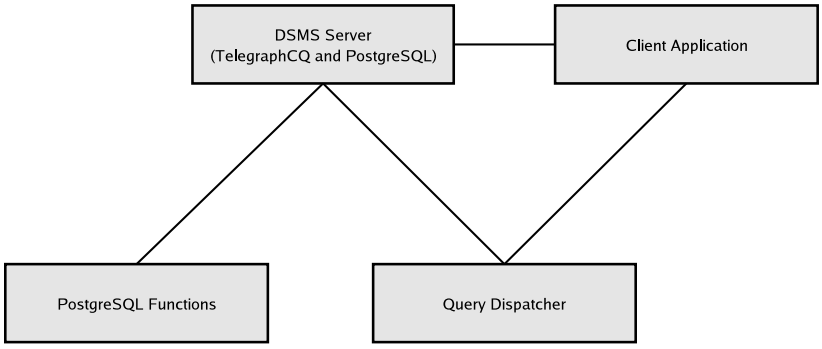


Figure 9.1: Components of the Bwhere system.

From Figure 9.1 we can see that the system consist of a four primary components, which contains a number of non-illustrated modules. The two main components are the DSMS server and the client application. From a hierarchically perspective, both of these components are at the highest level, whereas the Query Dispatcher is shared between them at the lowest level. At the lowest level we also have the PostgreSQL functions, which is depicted beneath the DSMS server. Despite the hierarchical organization, we suggest a top-down testing technique. The separate tests are described in the following.

Module Test Although limitations in TelegraphCQ are known, the TelegraphCQ module in the DSMS component should only be tested with a black-box test. Likewise,

the PostgreSQL module should also be tested with black-box test. The reason is that we are relying on existing systems, which we want to ensure the the general functionality of. The other three components should be tested with white-box tests, to ensure, among other factors, that the logical structure is correct and the modules behave correctly.

Module Integration Test The two activities involved in integration test are gathering the modules and testing that the modules work together as components. The result will be separate and well functioning components, also referred to as the four components i Figure 9.1. The modules should be gathered in a top-down style, implementing the main functionality and thereby extending with underlying modules. This might require program test *stubs* during integration, compensating for the non-implemented modules.

Component Integration Test The focus of the component integration test is to ensure the communication between the components are working as specified. In BWHERE, the DSMS server and client application should be brought together first, thereby ensuring the communication between the main components. Secondly, the Query Dispatcher should be integrated, and thereby enabling query handling functionality. Finally, the PostgreSQL functions component should be added, which will complete the top-down component integration.

Acceptance Test The final test of the complete program, the acceptance test, is intended to verify that the system meets the specified demands. Due to the project's goal of developing an experimental system, we have not made a comprehensive specification of demands. Therefore, we can not specify the tests to the executed in order to obtain acceptance for the system.

The presented test strategy is our recommendation for testing that the BWHERE system conforms to the specification. As mentioned, we have not conducted the tests, but only made some assessments on the system. The assessments are described in the following section.

9.2 BWHERE Assessment

In this section we describe some of the assessments we have conducted on the development version of the BWHERE system.

Client The client application has more functionality, but we have specifically tested the query generation functionality with a black-box test. From the user interface we have selected different numbers of subscription areas and types, including lower/upper range values. The assessment has revealed a few bugs, which have been corrected. The assessment has afterwards been approved.

Query Dispatcher The Query Dispatcher has been evaluated by initiating queries on the client application, and thereby checking via debug information on the screen, if the dispatcher handled the tasks as intended. The actual confirmation on this could be read out on the screen on the client application, when the result on the test query was returned. The assessment was approved.

Query Evaluation The query evaluation has been examined on the server by executing the actual queries from another computer. Thereby we could, among other factors, evaluate if the developed enter and leave functions were performing well. The assessment was approved.

The presented assessments have served as a fundamental evaluation of the key functionality, while developing the Bwhere system.

Chapter 10

Experiments

In this chapter we describe the experiments we have conducted and the results of these. The experiments have all been conducted on the prototype system. To demonstrate and prove the practical system, as stated in Chapter 3 on page 17, we have specified and performed some experiments with the developed implementation of the system. The experiments have been divided into two categories, "practical" and "scalability", to prove the functionality of the system and how it would scale, respectively.

10.1 Practical Experiment

In this section we describe how the practical experiment of the BWHERE system was configured, executed, and evaluated. In the AWHERE project, we did not conduct any experiments with real mobile phones, but only in a emulator on a PC. The purpose of this practical experiment was therefore, to evaluate the general functionality of the system, running in a real setup.

Experiment Setup The computer running the TelegraphCQ DSMS and PostgreSQL database was an Intel Pentium 3, with an 800 MHZ CPU and 512 MB RAM. The operating system was Ubuntu Linux, kernel 2.6.10. The two clients were Sony Ericsson K700i mobile phones, providing GPRS data communication and a J2ME environment with MIDP 2.0 and CLDC 1.1.

Adjustment Parameters In this experiment it is not the focus to adjust on parameters to validate if the system meets the demands, but just to prove functionality. We have only had the two mentioned mobile phones at our disposal, which limited the extend of the experiment. In addition the TelegraphCQ limitations with no subqueries and no `OR`, as mentioned in Section 7.3 on page 41, has forced us to show the system functionality with a predefined query, which makes the user subscribe to itself. The functionality with a query being generated could be applied, but not tested.

Measurement Factors Since the experiment was supposed to show the general functionality, the only measurement factor was to approve the answers received from the server, based on the query.

Experiment Execution Due to the above mentioned limitations in TelegraphCQ, the execution of this experiment was fairly simple. The mobile clients were executing an adjusted version of the client software, which sends a predefined subscription query to the server. The subscription area was defined in the query. The only adjustable parameter from the client was the user id, which had to be different for the two clients. To simulate the movement of the client, the 192 GPS test points were utilized and sent to the server with a predefined interval. The interval was set to 2000, 1000, 500, and 100 milliseconds, which worked fine in all cases.

From the experiment we can conclude that the general functionality of the client, i.e., connection setup, submit query, stream positions, and receive answers, were working fine. Concerning the server, it was able to execute the queries and return the answers to the respective clients. If the limitations of TelegraphCQ had not been present, the experiment could have shown more functionality, but the basics were proved, nevertheless.

10.2 Scalability Experiment

As described for the practical experiment, the configuration, execution, and evaluation of the scalability experiment, is described in this section. The purpose of this experiment was to evaluate the general scalability of the system, to illustrate if it can support a large number of clients. The experiment was compared with the previous scalability experiment from the AWHERE project, to have a performance reference.

Experiment Setup The computer running the TelegraphCQ DSMS and PostgreSQL database was the same as in Experiment 10.1 on the preceding page. In order to connect a large amount of clients to the system it is not feasible to use the mobile client as this would require as many mobile phones. A tailored version of the client application was therefore implemented in the standard edition of Java where each client was started as a thread herein. The system running the client application was an Intel Dual Xeon 2.80 GHz with 4 GB RAM. The operating system was RedHat Enterprise Linux 3, kernel 2.4.21.

Adjustment Parameters In Section 7.3 on page 41 we explained that TelegraphCQ limits us to a maximum of 32 users. As this hinders the scalability experiment with respect to the number of users we are able to connect, we can not show any results regarding this. We can, however, by using the same query as in Experiment 10.1 on the preceding page and continuously sending only two different positions, e.g., one inside an area and one outside the area, with a fixed interval, e.g., 30 seconds, measure the average time which the client receives results. This time should be close to the interval time. By reducing the interval but keeping the 32 clients, we can analyze when the workload of TelegraphCQ and PostgreSQL becomes too large.

Measurement Factors We based this experiment on the assumption that the workload of 32 clients with an interval 30 of seconds was the same as 64 users using a 60 second interval, etc. From the interval adjustment parameter, we could deduct how many possible clients the system should be able to handle.

The offset for these measurements were deducted from the AWHERE scalability experiment, in which the system was capable of handling 1600 users, subscribing to themselves. Calculating an average send interval time from that experiment and scaling the results down to 32 users, yields an maximal average receive interval of 2700 milliseconds. Based on this we have measured the average time using the following intervals:

```
0, 2, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 500, 1000, 2000, 4000
```

The intervals are shown in milliseconds as the `sleep` method in Java expects this as input. Note that when the interval 0 was used, the sleep method was not in use, i.e., commented out.

Experiment Execution As we implemented the client application using the standard edition of Java we could have bypassed the Query Dispatcher as a PostgreSQL API for Java exists. In order to show overall performance of BWHERE we chose not to bypass the Query Dispatcher as doing so would remove an important factor of BWHERE. TelegraphCQ and PostgreSQL was started followed by the Query Dispatcher. Then the client application was started running 32 threads were each client thread received 300 results on each interval. This resulted in 32 x 300 results which were used to measure the average on. In Figure 10.1, the actual time results are illustrated along with the theoretical time. Further, the maximum average send/receive interval from the AWHERE scalability experiment, is inserted as a reference. As we can see from the results in the figure, the theoretical and actual time split around 100 milliseconds, which indicate the maximal performance.

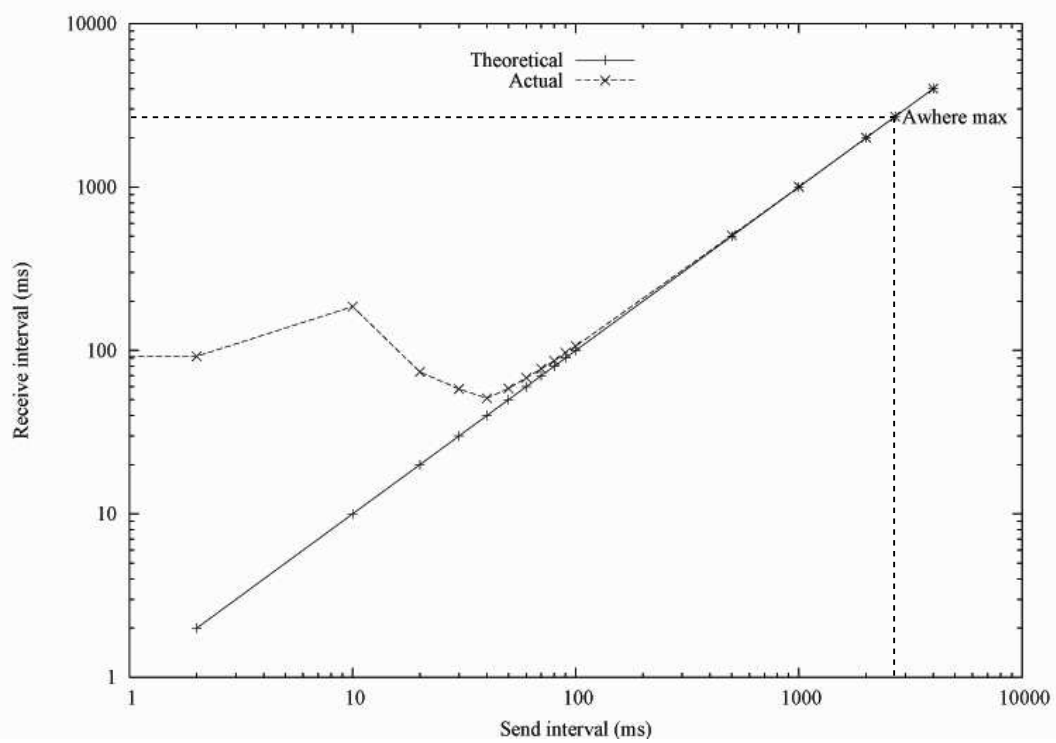


Figure 10.1: Scalability results.

This shows that BWHERE can send results to 32 clients every 100 milliseconds. If our assumption holds and we use the minimum and maximum interval of AWHERE, i.e., 30 seconds and 5 minutes, BWHERE can handle 9,600 and 96,000 clients, respectively. This, of course, is only hypothetical as factors such as available ports on the server and the amount of memory each client connection uses will prove it less. It does, however, indicate that BWHERE is capable of serving a reasonable number of clients and should easily be able to handle the potential

number of users, i.e., 4000, we estimated in Section 4.2 on page 20.

From the experiment we can conclude that although the current state of TelegraphCQ limits BWHERE to 32 users it is possible serve more users than the AWHERE system was capable of. Although we use the same query technique as in AWHERE, i.e., to subscribe to oneself, we are not seeing the same performance issues as we have based BWHERE on a well-known DBMS where it gains performance by PostgreSQLs query optimizer and a more developed DSMS than the AWHERE-DSMS, giving us the yet more optimization although it also provided limitations as well. Notice that the AWHERE experiment, used for comparison, were utilizing the RDE evaluation form, and a much faster PC (Intel P4, 2.66 GHz CPU, 512 MB RAM). A direct comparison between the two experiments can therefore not be drawn, but the comparison is still valid.

In case no limitations in TelegraphCQ were present, this experiment would have been performed in a different manner. Such an experiment would use a fixed interval of, e.g., 30 seconds, and the number of client connections would have been increased until the average would indicate a performance drop. This experiment would show a more accurate number of the amount of clients the system could handle.

The main foci of the experiments were to demonstrate the practical use of the BWHERE system, and show its scalability. From the practical experiment, we can conclude that the combination of technologies works well in a real setup. In addition, the scalability experiment illustrated that the system has an acceptable scalability, that conforms to the estimated number of users, i.e., 4000, described in Section 4.2 on page 20.

Chapter 11

Conclusion

The purpose of this chapter is to conclude the project, herein discussing the various areas which have been dealt with in this project. Finally, a discussion about improvements and future work is presented.

Mobility is becoming something people look for in services they use and hardware they own, e.g., mobile phones, SMS, MP3-Players, and laptops. Further, the next wave in computer innovation is expected to be pervasive computing, where computers surround us and service us every moment of the day. One indication of this tendency is the LBSs, and the recent Location-Aware Services, which provide the user with information depending on the location of the user at the time of the request.

In the predecessor to this project, AWHERE, we built a service providing users with the ability to know the location of their contacts. But the service only provided a distance from the user to the contact, as well as speed and direction of the contact. In this project we wished to add some of the functionality you find in a GIS to enable the users of the improved AWHERE, called BWHERE, to set-up more detailed queries and thereby receive more detailed results.

We have not only expanded AWHERE with basic spatial predicates, but also included complex composite spatiotemporal predicates.

This project has been an effort to combine the three worlds of technologies presented in the Introduction on page 4; the mobile environment, the GISs, and the world of streaming data. We have done this in two ways, by a theoretical approach and by a more practical approach, where we have developed an experimental system.

The theoretical solution was reached by analyzing the three different worlds, and further analyzing what was needed to combine them. The theoretical analysis showed that, given certain requirements to the three different worlds, a system combining the three worlds is a theoretical possibility.

After it was determined that the BWHERE system was a theoretical possibility, we proceeded to design and implement an experimental system.

During development of the experimental system, we discovered that there is a limitation on the number of possible queries TelegraphCQ allows. With this in mind we created a design that would operate on an implementation of TelegraphCQ that did not have this limitation and a design that will work on the actual TelegraphCQ system, although in a less optimal manner.

We have conducted two main experiments; a practical experiment and a scalability experiment. The goal of the practical experiment was to prove that a system such as BWHERE would function on actual real life mobile phones. As the experiment showed BWHERE will function on a mobile phone provided the phone software includes the required APIs, i.e., the Bluetooth API and potentially the JDBC optional package.

The scalability experiment intended to show how well the BWHERE system scales. The experiment concluded that BWHERE scales in an acceptable manner, and that, with a fully implemented TelegraphCQ and further optimizations to BWHERE, BWHERE will have a scalability that is capable of handling the estimated number of users, i.e., 4000.

For the reason presented in all of the above and because both theory and experiments revealed that such a combination of mobility, GIS, and streaming is indeed a viable possibility, we conclude that, although some corners in the experimental triangle need to mature, we have successfully proved that such systems can be realized.

11.1 Future Work

Based on the current status of the developed system of BWHERE, we present suggestions for future work.

TelegraphCQ Limitations The limitations in TelegraphCQ, presented in Section 7.3 on page 41, have imposed some restrictions on functionality in the BWHERE system. On the assumption that these limitations were dealt with in TelegraphCQ, the client could be extended in several ways. First, the limitation of 32 queries should be handled, to provide a much better scalability. Second, support for subqueries could result in more efficient queries. Third, an `OR` implementation would also contribute to more efficient queries, compared to the simulated `OR`, presented in Section 7.5.2 on page 45.

Client Application The client application is a development version, and due to this, the user interface also lacks some functionality. In the current user interface, illustrated in Appendix D on page 87, it is for instance not possible to specify the users you want subscription information from, and the result is presented in SQL format. This should of cause be handled in a final version of the client application. In addition, more dynamic functionality could be implemented, such as run-time adjustment of subscriptions.

Client Update Strategy Concerning update strategy, the current development version is based on a predefined report-in frequency, as described in Chapter 6.1 on page 33. As mentioned in the same chapter, a more efficient update strategy could be utilized, such as update when moved. More specifically, a specified threshold for change in movement will determine if it is necessary to report a new position to the server. In case the position has not changed more than the threshold, the position will not be send to the server. In order to indicate that the client is still active in the area of the threshold, a liveness signal will be send. The liveness signal will be send in case the client has not moved outside the specified threshold for a given period of time. When a liveness signal is send, it is assumed that the client position is unchanged, and the timestamp of the liveness signal will be the new timestamp of the unchanged position.

Another update strategy could be utilization of a prediction algorithm, running on both the client and the server, as described in [Civilis et al., 2004]. This strategy is based on predicting the next position of the client, and only reporting in when the position differs from a predefined threshold. Use of this strategy could reduce the data transfer between the client and the server, but will also require more processing on both sides. In addition, both processings would have to be performed almost simultaneously, to ensure valid data on the server.

Buffering Buffering of position values is another important task, since both the enter and leave operators rely on buffered data. In BWHERE the last position value

is buffered, and is stored although the client logs off and on again. The current implementation handles the "cold start" situation (no buffered value) by utilizing the position $(0, 0)$, which is at the equator in the Atlantic ocean. The reference point far away, ensures that a possible enter will be detected on locally defined areas when starting up.

This solution could be improved, by specifically deleting the last buffered value when logging off. When logging on, no buffered value will be available for detecting an enter. In this situation the enter function should detect if the single position value available, is inside any of the defined subscription areas. If it is inside, an enter should be detected, which would give the subscribing users the information that the current user has entered. This would provide enter detections on users that do not necessary enter as defined in Section 4.4.3 on page 26, but suddenly enter by appearing inside an area.

Sessions In Section 7.2.3 on page 39 we mentioned that in order to maintain the trajectory of a user, although it only is based on two values, sessions could be used. The sessions could be implemented via a unique key for the given session, and thereby allowing the user to resume a session in case of disconnection.

Area definition and subscription The area definition is currently implemented with predefined areas. This could be extended with a web interface, as in the related application "Trygghetsmobilen", described in Section 6.1 on page 33. In addition, a client interface for specifying areas could be provided.

Concerning subscriptions, they are activated when the user logs on to the system and runs for the duration of the time being online. This could be extended with subscriptions with specific day and time intervals, e.g., Monday through Friday from 8 a.m. to 4 p.m.

Appendix A

9-intersection Model

In the 9-intersection model describes the binary topological relation between two objects, A and B, in R^2 is based on the intersection of A's interior (A^o), boundary (δA), and exterior (A^-) with B's interior (B^o), boundary (δB), and exterior (B^-). The nine intersections between the six object parts describe a topological relation and can be concisely represented by a 3x3-matrix, called the 9-intersection. The 9-intersection matrix $\mathfrak{F}_9$ is shown in Equation A.1, where each intersection is characterized by a value empty ($\varnothing$) or non-empty ($\neg\varnothing$). [Egenhofer and Sharma, 1993]

$$\mathfrak{F}_9 = \begin{pmatrix} A^o \bigcap B^o & A^o \bigcap \delta B & A^o \bigcap B^- \\ \delta A \bigcap B^o & \delta A \bigcap \delta B & \delta A \bigcap B^- \\ A^- \bigcap B^o & A^- \bigcap \delta B & A^- \bigcap B^- \end{pmatrix} \tag{A.1}$$

The 9-intersections and their geometric representations for a region and a line are shown in Figure A.1 [Egenhofer and Sharma, 1993].

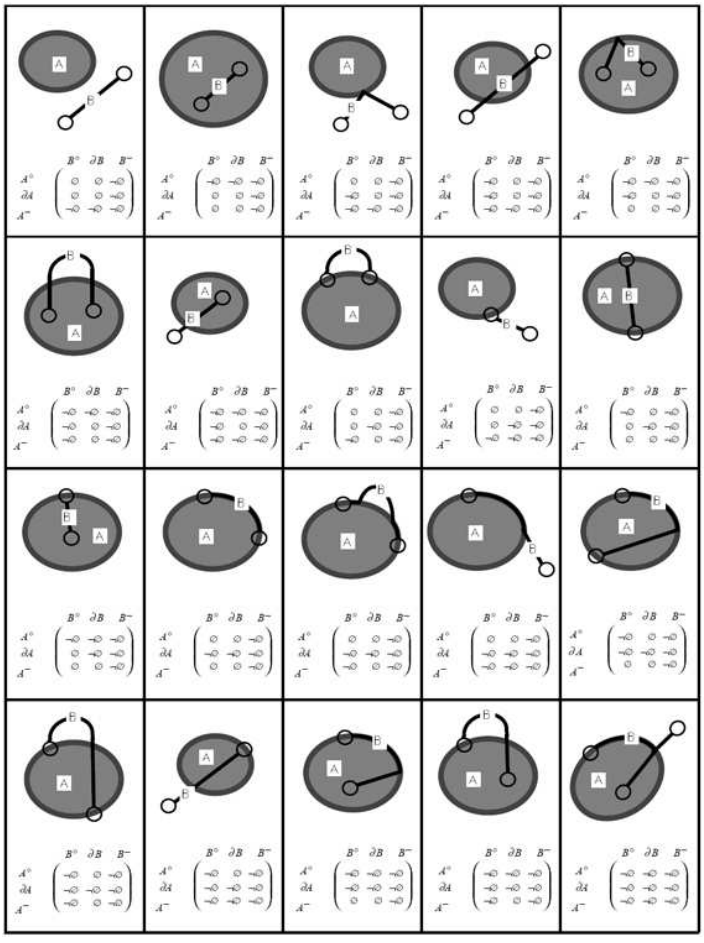


Figure A.1: A geometric interpretation of the 20 relations between a region and a line (one of them can be only realized if the line is non-simple).

Appendix B

Query Dispatcher

In this appendix the source code for the Query Dispatcher is listed.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <libpq-fe.h>

#define QH_PORT 5534

void *readQuery();
int send_reply();

void *connect_to_postgres();
static void exit_nicely();

struct mystruct{
  int aSocket;
  char *buffer;
  int close;
};

int main(){

  pthread_t mythread;
  int servSock, uSock;
  servSock = socket(PF_INET, SOCK_STREAM, 0);
  if(servSock <0){
    perror("cannot open socket ");
    return 1;
  }
  struct sockaddr_in sa,si;

  sa.sin_family = AF_INET;
  sa.sin_addr.s_addr = htonl(INADDR_ANY);
  sa.sin_port = htons(QH_PORT);

  int yes = 1;
  if (setsockopt(servSock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
  }

  if(bind(servSock, (struct sockaddr *) &sa, sizeof(sa))<0){
    perror("cannot bind socket ");
    return 1;
  }
```

```
  listen(servSock, 5);
  int len;
  while(1){
    len = sizeof(si);
    uSock = accept(servSock, (struct sockaddr *) &si, &len);
    pthread_create(&mythread, NULL, readQuery, (void *)uSock);
  }
  close(servSock);
  close(uSock);

  return 0;
}

void *readQuery(void *arg){

  ssize_t n = -1;
  char buf[1024];
  char *quit = "quit";
  int uSock = (int)arg;

  struct mystruct data;
  data.aSocket = uSock;
  data.close = 0;

  pthread_t postgres_thread;
  while(n != 0){
    n = recv(uSock, &buf, sizeof(buf), 0);
    if(n<0){
      perror("cannot receive from socket ");
      pthread_exit(NULL);
    }

    if(n!=0){
      int cmp = strncmp(buf, quit, 4);
      if(cmp != 0){
        data.buffer = buf;
        pthread_create(&postgres_thread, NULL, connect_to_postgres, &data );
      }
      else{
        data.close = 1;
        pthread_join(postgres_thread, NULL);
      }
    }
  }
  pthread_exit(NULL);
}

int send_reply(char* reply, int uSock){
  int n = send(uSock, reply, strlen(reply), MSG_NOSIGNAL);
  if(n<0){
    perror("Unable to send reply");
    return 1;
  }
  return 0;
}

void *connect_to_postgres(void *arg){

  struct mystruct *indata = arg;
```

```
int sock = indata->aSocket;
char* query = indata->buffer;
char *pghost, *pgport, *pgoptions, *pgtty;
char *dbName;
int nFields, i, j;
PGconn *conn;
PGresult *res;
dbName = "xstream";
pghost = "localhost";
pgport = "5535";
pgoptions = NULL;
pgtty = NULL;

conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);
if (PQstatus(conn) == CONNECTION_BAD)
  {
    fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
    fprintf(stderr, "%s", PQerrorMessage(conn));
    exit_nicely(conn);
  }

res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
  {
    fprintf(stderr, "query failed\n");
    PQclear(res);
    exit_nicely(conn);
  }
PQclear(res);

char declare_cursor[1024] = "DECLARE myportal CURSOR FOR ";

strcat(declare_cursor, query);
printf("DEC: %s\n", declare_cursor);

res = PQexec(conn, declare_cursor);
if (PQresultStatus(res) != PGRES_COMMAND_OK)
  {
    fprintf(stderr, "DECLARE CURSOR command failed\n");
    PQclear(res);
    exit_nicely(conn);
  }
PQclear(res);

static char temp[1024];
char *temp2;

while(indata->close == 0){

  res = PQexec(conn, "FETCH NEXT in myportal");
  if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
      fprintf(stderr, "FETCH NEXT command didn't return tuples properly\n");
      PQclear(res);
      exit_nicely(conn);
    }

  nFields = PQnfields(res);
  for (i = 0; i < PQntuples(res); i++)
```

```
    {
      for (j = 0; j < nFields; j++){
        temp2 = PQgetvalue(res, i, j);
        strcat(temp, temp2);
        strcat(temp, ",");
      }
      if(send_reply(temp, sock) != 0)
        indata->close = 1;

      memset(temp, 0, 1024);
    }
  }

  res = PQexec(conn, "CLOSE myportal");
  PQclear(res);

  res = PQexec(conn, "END");
  PQclear(res);

  PQfinish(conn);
  return 0;
}

static void exit_nicely(PGconn *conn)
{
  PQfinish(conn);
  pthread_exit(NULL);
}
```

Appendix C

Client Application

In this appendix the source code for the client application is listed.

C.1 UIMIDlet

```
// UIMIDlet.java

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.RecordStoreException;
import java.util.Hashtable;

public class UIMIDlet extends MIDlet implements CommandListener {

    // This is an instance of UIMIDlet to which we return when the
    // back button is pressed
    private static UIMIDlet instance = null;
    public List mainList;
    private List subOptionsList;
    private List enterSubList;
    private List leaveSubList;
    //Command (name, type, priority), priority 0 = highest
    private final static Command CMD_EXIT = new Command("Exit", Command.EXIT, 0);
    private final static Command CMD_BACK = new Command("Back", Command.BACK, 0);
    private String query;
    private String testQuery;

    // UIMIDletStream stuff
    private boolean isPaused;
    private Client client;

    // RecordStore stuff
    private static final String kUser = "user";
    private static final String kPassword = "password";
    private Store mPreferences;
    private Form mForm;
    private Form queryForm;
    private TextField mUserField;

    // Static area storage
    private Store staticAreas;

    // Test data storage
    private Store testData;

    // Constructor
    public UIMIDlet() {
        instance = this;
        mainList =
            new List("* Main menu *", List.IMPLICIT, mainOptions, null);
        subOptionsList =
```

```
            new List("Sub. Type", List.IMPLICIT, subOptions, null);
        enterSubList =
            new List("Enter sub.", List.MULTIPLE, subscriptionOptions, null);
        leaveSubList =                                                          50
            new List("Leave sub.", List.MULTIPLE, subscriptionOptions, null);

        mainList.addCommand(CMD_EXIT);
        subOptionsList.addCommand(CMD_BACK);
        enterSubList.addCommand(CMD_BACK);                                      55
        leaveSubList.addCommand(CMD_BACK);

        // RecordStore stuff
        try {
            mPreferences = new Store("preferences");                            60
        }
        catch (RecordStoreException rse) {
            mForm = new Form("Exception");
            mForm.append(new StringItem(null, rse.toString()));
            mForm.addCommand(new Command("Exit", Command.EXIT, 0));             65
            mForm.setCommandListener(this);
            return;
        }

        mForm = new Form("User details");                                       70

        if (mPreferences.get(kUser).equals("No entry")) {
            mPreferences.put(kUser, "1");
            mUserField = new TextField("User ID", mPreferences.get(kUser),
                                       8, TextField.NUMERIC);                   75
        }
        else {
            mUserField = new TextField("User ID", mPreferences.get(kUser),
                                       8, TextField.NUMERIC);
        }                                                                       80

        mForm.append(mUserField);
        mForm.addCommand(CMD_BACK);
        mForm.setCommandListener(this);
                                                                                85
        // Load static areas
        loadStaticAreas();

        // Reload selected subscriptions
        setSelectedStaticAreas();                                               90

        // Load test data
        loadTestData();

        // Show query                                                           95
        testQuery = generateQuery();
        queryForm = new Form("Query");
        queryForm.append(testQuery);
        queryForm.addCommand(CMD_BACK);
    }                                                                           100

    private Displayable display = null;

    // Main menu options
    private static final String[] mainOptions = {                               105
```

```
                "Subscriptions",
                "Start/stop",
                "User details",
                "Show query"
                };
// Settings menu options
private static final String[] subOptions = {
                "Enter",
                "Leave"
                };

// Subscription menu options
private static final String[] subscriptionOptions = {
                "Grupperummet",
                "Novi",
                "Kroghstræde",
                "Klingenberg",
                "Cafe du ve",
                "Daginstitutionen",
                "NORK",
                "Nykredit"
                };

// Signals the MIDlet to start and enter the Active state
protected void startApp() {
    isPaused = false;
    Display.getDisplay(this).setCurrent(mainList);
    mainList.setCommandListener(this);
}

// Signals the MIDlet to stop and enter the Paused state
protected void pauseApp() {
    isPaused = true;
}

// This method quits the current midlet
public static void quit() {
    instance.destroyApp(true);
    instance.notifyDestroyed();
    instance = null;
}

public void commandAction(Command c, Displayable d) {
    if (d.equals(mainList)) {
        // in the main list
        if (c == List.SELECT_COMMAND) {
                int option = ((List)d).getSelectedIndex();
                switch (option) {
                    case 0: // Subscription options
                        try {
                            Display.getDisplay(this).
                                setCurrent(subOptionsList);
                            subOptionsList.setCommandListener(this);
                        }
                        catch(Exception err) {
                            Alert a = new Alert("Error");
                            a.setString("Failed in handling the command " +
                                "at position: " + option);
```

```
                    a.setTimeout(Alert.FOREVER);
                    Display.getDisplay(this).setCurrent(a);
                }
                break;

            case 1: // Start/stop
                try {
                    client = new Client(this);
                    client.start();
                }
                catch(Exception err) {
                    Alert a = new Alert("Error");
                    a.setString("Failed in handling the command " +
                        "at position: " + option);
                    a.setTimeout(Alert.FOREVER);
                    Display.getDisplay(this).setCurrent(a);
                }
                break;

            case 2: // User details
                try {
                    Display.getDisplay(this).setCurrent(mForm);
                    mForm.setCommandListener(this);
                }
                catch(Exception err) {
                    Alert a = new Alert("Error");
                    a.setString("Failed in handling the command " +
                        "at position: " + option);
                    a.setTimeout(Alert.FOREVER);
                    Display.getDisplay(this).setCurrent(a);
                }
                break;

            case 3: // Show query
                testQuery = generateQuery();
                System.out.println("Number of items: " + queryForm.size());

                // Delete item 0 which is testQuery
                queryForm.delete(0);
                // Append the new value of testQuery
                queryForm.append(testQuery);
                System.out.println("Query: " + testQuery);
                try {
                    Display.getDisplay(this).setCurrent(queryForm);
                    queryForm.setCommandListener(this);
                }
                catch(Exception err) {
                    Alert a = new Alert("Error");
                    a.setString("Failed in handling the command " +
                        "at position: " + option);
                    a.setTimeout(Alert.FOREVER);
                    Display.getDisplay(this).setCurrent(a);
                }
                break;
        }
    }
}

if (d.equals(subOptionsList)) {
```

```
    // in the settings list
    if (c == List.SELECT_COMMAND) {
        int option = ((List)d).getSelectedIndex();
        switch (option) {
            case 0: // Enter
                    try {
                        Display.getDisplay(this).
                            setCurrent(enterSubList);
                        enterSubList.setCommandListener(this);
                    }
                    catch(Exception err) {
                        Alert a = new Alert("Error");
                        a.setString("Failed in handling the command " +
                            "at position: " + option);
                        a.setTimeout(Alert.FOREVER);
                        Display.getDisplay(this).setCurrent(a);
                    }
                    break;

            case 1: // Leave
                    try {
                        Display.getDisplay(this).
                            setCurrent(leaveSubList);
                        leaveSubList.setCommandListener(this);
                    }
                    catch(Exception err) {
                        Alert a = new Alert("Error");
                        a.setString("Failed in handling the command " +
                            "at position: " + option);
                        a.setTimeout(Alert.FOREVER);
                        Display.getDisplay(this).setCurrent(a);
                    }
                    break;
        }
    }
}

if (d.equals(enterSubList) || d.equals(leaveSubList)) {
    // If "back" while in subscription list
    if (c == CMD_BACK) {
        try {
            Display.getDisplay(this).setCurrent(subOptionsList);
            subOptionsList.setCommandListener(this);
        }
        catch(Exception err) {
            err.printStackTrace();
        }
    }
}
else if (c == CMD_BACK) {
    instance.display = null;
    Display.getDisplay(this).setCurrent(mainList);
    mainList.setCommandListener(this);
}

if (c == CMD_EXIT) {
    destroyApp(false);
    notifyDestroyed();
}
```

```
}

// Signals the MIDlet to terminate and enter the Destroyed state.
protected void destroyApp(boolean unconditional) {
    mainList = null;
    subOptionsList = null;
    display = null;
    // Save the user name and password.
    mPreferences.put(kUser, mUserField.getString());

    // Save the user selected enter subscriptions
    for(int i=0; i<8; i++){
        String key = "Enter" + subscriptionOptions[i];
        // If area is checked on subscription list
        if (enterSubList.isSelected(i)){
            // Save name of area with the value "checked"
            mPreferences.put(key,"checked");
        } else{
            // Save name of area with the value "unchecked"
            mPreferences.put(key,"unchecked");
        }
    }

    // Save the user selected leave subscriptions
    for(int i=0; i<8; i++){
        String key = "Leave" + subscriptionOptions[i];
        // If area is checked on subscription list
        if (leaveSubList.isSelected(i)){
            // Save name of area with the value "checked"
            mPreferences.put(key,"checked");
        } else{
            // Save name of area with the value "unchecked"
            mPreferences.put(key,"unchecked");
        }
    }

    try { mPreferences.save(); }
    catch (RecordStoreException rse) {
        rse.printStackTrace();
    }

    enterSubList = null;
    leaveSubList = null;
}

// Load names and coordinates of area boxes
private void loadStaticAreas() {
    try {
      staticAreas = new Store("Static Areas");
      // Abbreviations: LLC = Lower Left Corner
      //                URC = Upper Right Corner
      //                NI = North Indicator
      //                EI = East Indicator

      //           ("name","LLC NI, LLC EI, URC NI, URC EI")
      staticAreas.put("Grupperummet","57.01400833333334,9.986983333333333," +
        "57.0149,9.988623333333333");
      staticAreas.put("Novi","57.01213,9.986833333333333," +
        "57.013018333333335,9.98848");
```

```
        staticAreas.put("Kroghstræde","57.01367666666667,9.982308333333334," +
          "57.014586666666666,9.983955");
        staticAreas.put("Klingenberg","57.01450333333333,9.97712," +
          "57.015388333333334,9.978751666666666");
        staticAreas.put("Cafe du ve","57.01523666666667,9.976728333333334," +
          "57.016128333333334,9.978361666666666");
        staticAreas.put("Daginstitutionen","57.015345,9.98132," +
          "57.01623,9.98294");
        staticAreas.put("NORK","57.01639333333333,9.988818333333333," +
          "57.017273333333335,9.99045");
        staticAreas.put("Nykredit","57.015251666666664,9.986283333333333," +
          "57.016146666666664,9.987923333333333");
    }
    catch (RecordStoreException rse) {
      rse.printStackTrace();
    }
}

// Load test data
private void loadTestData() {
    try {
      testData = new Store("Test data");

      testData.put("1","063740.983,57.014425,9.988026666666666");
      testData.put("2","063750.982,57.01437,9.987835");
      testData.put("3","063800.982,57.01423833333333,9.987785");
      testData.put("4","063820.990,57.01396833333333,9.987653333333334");
      testData.put("5","063830.990,57.01384,9.98765");
      testData.put("6","063840.989,57.01371,9.987653333333334");
      testData.put("7","063850.988,57.01353666666667,9.987666666666666");
      testData.put("8","063940.986,57.013145,9.98771");
      testData.put("9","063950.685,57.012933333333336,9.98768");
      testData.put("10","064000.989,57.012816666666666,9.98772");
      testData.put("11","064010.990,57.012685,9.987711666666666");
      testData.put("12","064020.990,57.01253166666667,9.9877");
      testData.put("13","064030.989,57.012565,9.987658333333334");
      testData.put("14","064040.990,57.012573333333336,9.98765");
      testData.put("15","064050.989,57.01255833333333,9.98762");
      testData.put("16","064100.989,57.012535,9.987608333333334");
      testData.put("17","064110.989,57.01253666666667,9.987631666666667");
      testData.put("18","064130.989,57.01253666666667,9.987673333333333");
      testData.put("19","064140.988,57.01255166666667,9.987691666666667");
      testData.put("20","064150.989,57.012568333333334,9.987733333333333");
      testData.put("21","064210.989,57.01255333333334,9.987733333333333");
      testData.put("22","064220.691,57.012541666666664,9.987721666666667");
      testData.put("23","064230.988,57.01253,9.987675");
      testData.put("24","064300.986,57.0125,9.987216666666667");
      testData.put("25","064310.988,57.012501666666665,9.987056666666666");
      testData.put("26","064320.988,57.012515,9.986723333333334");
      testData.put("27","064340.987,57.01253666666667,9.98609");
      testData.put("28","064350.987,57.01254,9.985796666666667");
      testData.put("29","064400.987,57.012548333333335,9.98546");
      testData.put("30","064410.987,57.01251666666667,9.985316666666666");
      testData.put("31","064420.987,57.01248833333333,9.98499");
      testData.put("32","064440.987,57.01248,9.984406666666667");
      testData.put("33","064450.986,57.01246833333333,9.984205");
      testData.put("34","064500.985,57.012545,9.984048333333334");
      testData.put("35","064510.987,57.01271,9.984013333333333");
      testData.put("36","064520.986,57.01285,9.98393");
```

```
testData.put("37","064530.985,57.012946666666664,9.983753333333333");
testData.put("38","064540.985,57.012945,9.983433333333334");
testData.put("39","064550.984,57.012993333333334,9.983183333333333");
testData.put("40","064600.984,57.01316166666667,9.98322");
testData.put("41","064610.983,57.01330333333333,9.983278333333333");
testData.put("42","064620.982,57.013515,9.9833");
testData.put("43","064630.982,57.0136,9.983158333333334");
testData.put("44","064640.981,57.01372666666666,9.983268333333333");
testData.put("45","064650.981,57.01380833333333,9.98326");
testData.put("46","064700.980,57.01392,9.983191666666666");
testData.put("47","064710.980,57.014158333333334,9.983121666666667");
testData.put("48","064720.979,57.01414666666667,9.983125");
testData.put("49","064730.978,57.014165,9.98312");
testData.put("50","064740.978,57.014183333333335,9.983115");
testData.put("51","064750.977,57.014205,9.983136666666667");
testData.put("52","064800.977,57.014226666666666,9.983156666666666");
testData.put("53","064840.984,57.014685,9.983208333333334");
testData.put("54","064850.983,57.014813333333336,9.98321");
testData.put("55","064900.983,57.01488333333333,9.983091666666667");
testData.put("56","064910.982,57.014876666666666,9.982758333333333");
testData.put("57","064920.981,57.014853333333335,9.982438333333333");
testData.put("58","064930.981,57.01484833333333,9.982133333333334");
testData.put("59","065010.991,57.015078333333335,9.981441666666667");
testData.put("60","065020.991,57.01503833333334,9.981443333333333");
testData.put("61","065030.990,57.015033333333335,9.981431666666667");
testData.put("62","065040.990,57.014965,9.981066666666667");
testData.put("63","065050.989,57.01494666666667,9.98075");
testData.put("64","065100.989,57.014955,9.980443333333334");
testData.put("65","065110.988,57.01497333333333,9.980146666666666");
testData.put("66","065120.988,57.014988333333335,9.979846666666667");
testData.put("67","065130.987,57.01499166666667,9.979553333333333");
testData.put("68","065140.986,57.014936666666664,9.979285");
testData.put("69","065150.986,57.014968333333336,9.979031666666666");
testData.put("70","065200.985,57.01515,9.979075");
testData.put("71","065210.985,57.01526666666667,9.978951666666667");
testData.put("72","065220.984,57.015245,9.97864");
testData.put("73","065230.983,57.01519833333333,9.978345");
testData.put("74","065240.983,57.0152,9.978041666666666");
testData.put("75","065250.982,57.01527166666666,9.977895");
testData.put("76","065300.990,57.015283333333336,9.97788");
testData.put("77","065310.989,57.01519166666667,9.977906666666666");
testData.put("78","065320.988,57.01500333333333,9.977863333333334");
testData.put("79","065330.988,57.01492666666667,9.977911666666667");
testData.put("80","065340.989,57.01494666666667,9.97794");
testData.put("81","065350.989,57.01494,9.977938333333332");
testData.put("82","065400.989,57.01493833333333,9.97793");
testData.put("83","065410.989,57.01494,9.977936666666666");
testData.put("84","065420.989,57.01495333333333,9.9779");
testData.put("85","065440.988,57.01509,9.977816666666667");
testData.put("86","065450.987,57.01522833333333,9.977988333333334");
testData.put("87","065500.987,57.015388333333334,9.978038333333334");
testData.put("88","065510.986,57.0155,9.978");
testData.put("89","065520.985,57.01561,9.977795");
testData.put("90","065530.985,57.01566166666667,9.977531666666666");
testData.put("91","065540.987,57.015681666666666,9.977543333333333");
testData.put("92","065550.987,57.015695,9.977576666666666");
testData.put("93","065610.988,57.01562166666667,9.978051666666667");
testData.put("94","065620.987,57.01551666666667,9.978296666666667");
testData.put("95","065630.987,57.01551166666667,9.978623333333333");
```

```
testData.put("96","065640.986,57.01549333333333,9.97894");
testData.put("97","065650.986,57.015505,9.979193333333333");
testData.put("98","065700.985,57.0155,9.979548333333334");
testData.put("99","065710.984,57.015496666666664,9.979868333333334");
testData.put("100","065720.984,57.01549833333333,9.980193333333334");
testData.put("101","065730.983,57.015503333333335,9.980491666666667");
testData.put("102","065740.983,57.015535,9.980715");
testData.put("103","065750.982,57.015613333333334,9.980853333333334");
testData.put("104","065800.982,57.01566666666667,9.981125");
testData.put("105","065810.981,57.01567,9.981428333333334");
testData.put("106","065820.980,57.015701666666665,9.981741666666666");
testData.put("107","065830.980,57.01573833333333,9.981971666666666");
testData.put("108","065840.979,57.015753333333336,9.982181666666667");
testData.put("109","065910.986,57.015888333333336,9.981895");
testData.put("110","070150.992,57.015785,9.982105");
testData.put("111","070200.992,57.01577833333333,9.982116666666666");
testData.put("112","070210.984,57.01582833333333,9.982146666666667");
testData.put("113","070220.992,57.015993333333334,9.982201666666667");
testData.put("114","070230.991,57.016146666666664,9.982328333333333");
testData.put("115","070240.990,57.01621166666666,9.982541666666666");
testData.put("116","070250.990,57.016223333333336,9.982716666666667");
testData.put("117","070300.991,57.016036666666665,9.983011666666666");
testData.put("118","070310.991,57.01603333333333,9.983246666666666");
testData.put("119","070320.991,57.01603333333333,9.983395");
testData.put("120","070330.991,57.016065,9.983591666666667");
testData.put("121","070340.686,57.016065,9.983576666666666");
testData.put("122","070350.990,57.016025,9.983738333333333");
testData.put("123","070400.990,57.01600166666667,9.98394");
testData.put("124","070410.989,57.01603166666666,9.984121666666667");
testData.put("125","070430.988,57.01601333333333,9.984505");
testData.put("126","070440.987,57.01603,9.984451666666667");
testData.put("127","070450.991,57.016035,9.984445");
testData.put("128","070500.990,57.01603166666666,9.984448333333333");
testData.put("129","070510.989,57.016011666666664,9.984453333333333");
testData.put("130","070520.990,57.01601,9.984431666666667");
testData.put("131","070530.989,57.016011666666664,9.984448333333333");
testData.put("132","070540.989,57.016018333333335,9.984426666666666");
testData.put("133","070600.988,57.01601,9.98451");
testData.put("134","070610.989,57.01602333333334,9.984436666666667");
testData.put("135","070620.685,57.016011666666664,9.984453333333333");
testData.put("136","070630.990,57.016016666666665,9.984453333333333");
testData.put("137","070650.989,57.01602,9.984441666666667");
testData.put("138","070700.988,57.01600166666667,9.984443333333333");
testData.put("139","070710.989,57.016015,9.984461666666666");
testData.put("140","070720.692,57.016015,9.984476666666668");
testData.put("141","070730.989,57.01602166666667,9.984456666666667");
testData.put("142","070750.987,57.01602,9.984506666666666");
testData.put("143","070800.989,57.01601333333333,9.984461666666666");
testData.put("144","070810.989,57.016015,9.984463333333334");
testData.put("145","070820.988,57.01602166666667,9.984473333333334");
testData.put("146","070830.989,57.016016666666665,9.98449");
testData.put("147","070840.988,57.01602333333334,9.984496666666667");
testData.put("148","070850.988,57.016103333333334,9.984661666666666");
testData.put("149","070900.988,57.016238333333334,9.984893333333334");
testData.put("150","070910.987,57.01638666666667,9.985103333333333");
testData.put("151","070920.691,57.016465,9.98555");
testData.put("152","070930.987,57.016398333333335,9.9858");
testData.put("153","070940.986,57.016333333333336,9.986106666666666");
testData.put("154","071000.691,57.01636333333333,9.986431666666666");
```

```
        testData.put("155","071010.987,57.01627666666667,9.986996666666666");
        testData.put("156","071020.987,57.01629666666667,9.987246666666667");
        testData.put("157","071030.986,57.016288333333335,9.987553333333333");
        testData.put("158","071040.986,57.016288333333335,9.987843333333334");
        testData.put("159","071050.985,57.016288333333335,9.988131666666666");
        testData.put("160","071100.985,57.016288333333335,9.988421666666667");
        testData.put("161","071120.986,57.01677166666666,9.98813");
        testData.put("162","071130.987,57.01668166666666,9.988493333333333");
        testData.put("163","071140.987,57.01668333333333,9.98882");
        testData.put("164","071150.986,57.01667166666667,9.98912");
        testData.put("165","071200.986,57.016655,9.98945");
        testData.put("166","071210.986,57.0168,9.989593333333334");
        testData.put("167","071320.985,57.01664,9.989406666666667");
        testData.put("168","071330.986,57.01660666666667,9.989198333333333");
        testData.put("169","071340.986,57.01660666666667,9.988918333333332");
        testData.put("170","071350.985,57.01654,9.988633333333333");
        testData.put("171","071400.985,57.01638166666667,9.988626666666667");
        testData.put("172","071410.985,57.016235,9.98857");
        testData.put("173","071420.985,57.01623333333333,9.98827");
        testData.put("174","071430.984,57.01621333333333,9.988026666666666");
        testData.put("175","071440.983,57.016185,9.987915");
        testData.put("176","071450.983,57.01616833333333,9.987745");
        testData.put("177","071500.688,57.01577666666667,9.987518333333334");
        testData.put("178","071510.992,57.01569833333333,9.987386666666668");
        testData.put("179","071540.687,57.015705,9.987121666666667");
        testData.put("180","071550.992,57.01568,9.987108333333333");
        testData.put("181","071600.992,57.01567333333333,9.987105");
        testData.put("182","071610.991,57.015663333333336,9.987106666666667");
        testData.put("183","071630.991,57.01556,9.987246666666667");
        testData.put("184","071640.991,57.01551833333333,9.987628333333333");
        testData.put("185","071650.991,57.01539833333333,9.987806666666666");
        testData.put("186","071730.991,57.01501,9.98772");
        testData.put("187","071740.991,57.01486833333333,9.98773");
        testData.put("188","071750.991,57.014723333333336,9.987706666666666");
        testData.put("189","071800.991,57.01457166666667,9.987725");
        testData.put("190","071810.990,57.01441,9.987736666666667");
        testData.put("191","071820.990,57.014361666666666,9.987863333333333");
        testData.put("192","071830.991,57.014363333333336,9.987871666666667");
    }
    catch (RecordStoreException rse) {
      rse.printStackTrace();
    }
}

// Set the "check marks" in the subscription list of the
private void setSelectedStaticAreas() {
    // Check if recordStore is empty to avoid "cold start" situation
    if (mPreferences.isEmpty() == false) {
        for(int i=0; i<8; i++){
            if (mPreferences.get("Enter" +
                subscriptionOptions[i]).equals("No entry")) {
            }
            else {
                if (mPreferences.get("Enter" +
                    subscriptionOptions[i]).equals("checked")) {
                    enterSubList.setSelectedIndex(i, true);
                }
            }
            if (mPreferences.get("Leave" +
```

```
                subscriptionOptions[i]).equals("No entry")) {
            }
            else {
                if (mPreferences.get("Leave" +
                    subscriptionOptions[i]).equals("checked")) {
                    leaveSubList.setSelectedIndex(i, true);
                }
            }
        }
    }
}

public String generateQuery() {
//      Designed query
        query = "SELECT u.name, rs.action, r.name, st.gps_time " +
            "FROM Users u, Positions st, RegionSubscriptions rs, Regions r " +
            "WHERE st.user_id IN ( " +
            "SELECT subscribee " +
            "FROM UserSubsciptions " +
            "WHERE subscriber = " + mUserField.getString() + ") " +
            "AND rs.user_id = " + mUserField.getString() + " " +
            "AND rs.region_id = r.id " +
            "AND u.id = st.user_id " +
            "AND ((rs.action = 'e' AND st.trajectory + r.region) OR " +
            "(rs.action = '1' AND st.trajectory - r.region));";

/*
        // PREDEFINED TEST QUERY
        query = "SELECT rs.action, asq.gps_time, reg.region_name, asq.tcqtime "+
                "from asquare asq, region_subscription rs, regions reg " +
                "where asq.user_id = " + mUserField.getString() +
                " AND rs.user_id = asq.user_id " +
                "AND rs.region_id = reg.region_id " +
                "AND (asq.trajectory + box '((57.0149,9.98862333333333)," +
                "(57.0140083333333,9.98698333333333))') = TRUE " +
                "AND (reg.area ~= box '((57.0149,9.98862333333333)," +
                "(57.0140083333333,9.98698333333333))') = TRUE " +
                "AND rs.action = 'E'";
 */
        return query;
    }

    // Return list of selected enter subscription areas
    private Hashtable getEnterAreas() {
        Hashtable selectedEnterAreas = new Hashtable();

        // Extract selected area
        for(int i=0; i<8; i++){
            if (enterSubList.isSelected(i)){
                selectedEnterAreas.put(subscriptionOptions[i],
                subscriptionOptions[i]);
            }
        }
        return selectedEnterAreas;
    }

    // Return list of selected subscription areas
    private Hashtable getLeaveAreas() {
        Hashtable selectedLeaveAreas = new Hashtable();
```

```
        // Extract selected area
        for(int i=0; i<8; i++){
            if (leaveSubList.isSelected(i)){
                selectedLeaveAreas.put(subscriptionOptions[i],subscriptionOptions[i]);
            }
        }
        return selectedLeaveAreas;
    }

    public String getUid(){
        return mUserField.getString();
    }

    public boolean isPaused(){
        return isPaused;
    }

    public Store getStore(){
        return testData;
    }
}
```

C.2 Client

```
// Client.java

import java.io.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class Client extends Thread implements CommandListener{
    private UIMIDlet parent;
    private Form mainform;
    private Command exitCommand = new Command("Quit", Command.EXIT, 1);
    private StringItem replyUI;
    private StringItem exception;
    private String url = "130.225.192.132", uid;
    private SocketConnection dbconn;
    private DataInputStream dis;
    private DataOutputStream dos;
    private String query;
    private boolean done = false;
    private Streamer stream;
    private Store data;

    public Client(UIMIDlet c){
        parent = c;
        uid = parent.getUid();
        data = parent.getStore();
        query = parent.generateQuery();

        stream = new Streamer(url, uid, data);
        stream.start();
```

```
        mainform = new Form("Client");
        replyUI = new StringItem("Reply: ", "" );

        mainform.append(replyUI);

        exception = new StringItem("Exception: ", "");
        mainform.append(exception);

        mainform.addCommand(exitCommand);
        mainform.setCommandListener(this);

        Display.getDisplay(c).setCurrent(mainform);
    }

    public void commandAction(Command c, Displayable s){
        if(c == exitCommand){
            stop();
            Display.getDisplay(parent).setCurrent(parent.mainList);
            }
        }

    public void stop(){
        done = true;
    }

    public void run(){
        try{
            dbconn = (SocketConnection)Connector.open("socket://"+url+":5534");
            dis = dbconn.openDataInputStream();
            dos = dbconn.openDataOutputStream();

            dos.write(query.getBytes());

            byte[] reply = new byte[1024];

            while(!done){
                dis.read(reply);
                replyUI.setText(new String(reply).trim());
            }

            dos.write(new String("quit").getBytes());

            stream.setDone();

            dis.close();
            dos.close();
            dbconn.close();
        }
        catch(IOException ioe){
            exception.setText(ioe.toString());
        }
    }
}
```

C.3 Store

```
// Store.java

import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;
import javax.microedition.rms.RecordStoreException;

public class Store {
    private String mRecordStoreName;
    private Hashtable mHashtable;

    public Store(String recordStoreName) throws RecordStoreException {
        mRecordStoreName = recordStoreName;
        mHashtable = new Hashtable();
        load();
    }

    public int getSize(){
        return mHashtable.size();
    }

    public String get(String key) {
        if(mHashtable.containsKey(key) == true){
            return (String)mHashtable.get(key);
        }
        else {
            return "No entry";
        }
    }

    public void put(String key, String value) {
        if (value == null) value = "";
        mHashtable.put(key, value);
    }

    public boolean isEmpty() {
        if (mHashtable.isEmpty()){
            return true;
        }
        else return false;
    }

    private void load() throws RecordStoreException {
        RecordStore rs = null;
        RecordEnumeration re = null;

        try {
            rs = RecordStore.openRecordStore(mRecordStoreName, true);
            re = rs.enumerateRecords(null, null, false);
            while (re.hasNextElement()) {
                byte[] raw = re.nextRecord();
                String pref = new String(raw);
                // Parse out the name.
                int index = pref.indexOf('|');
                String name = pref.substring(0, index);
                String value = pref.substring(index + 1);
```

```
                put(name, value);
            }
        }
        finally {
            if (re != null) re.destroy();
            if (rs != null) rs.closeRecordStore();
        }
    }

    public void save() throws RecordStoreException {
        RecordStore rs = null;
        RecordEnumeration re = null;
        try {
            rs = RecordStore.openRecordStore(mRecordStoreName, true);
            re = rs.enumerateRecords(null, null, false);

            // First remove all records, a little clumsy.
            while (re.hasNextElement()) {
                int id = re.nextRecordId();
                rs.deleteRecord(id);
            }

            // Now save the preferences records.
            Enumeration keys = mHashtable.keys();
            while (keys.hasMoreElements()) {
                String key = (String)keys.nextElement();
                String value = get(key);
                String pref = key + "|" + value;
                byte[] raw = pref.getBytes();
                rs.addRecord(raw, 0, raw.length);
            }
        }
        finally {
            if (re != null) re.destroy();
            if (rs != null) rs.closeRecordStore();
        }
    }
}
```

C.4 Streamer

```
// Streamer.java

import java.io.*;
import javax.microedition.io.*;

public class Streamer extends Thread{
    private SocketConnection streamSocket;
    private String url, uid;
    private Store store;
    private DataOutputStream dos;
    private String wrapper = "0012csvppwrapper";
    private boolean done = false;

    public Streamer(String url, String uid, Store store){
```

```
        this.url = url;
        this.uid = uid;
        this.store = store;
    }

    public void setDone(){
        done = !done;
    }

    public void run(){
        String streamString = "";
        try{
            streamSocket =
                (SocketConnection)Connector.open("socket://"+url+":5533");
            dos = streamSocket.openDataOutputStream();
            dos.write(wrapper.getBytes());

            for(int i = 1; i<=store.getSize();i++){
                streamString = uid + "," + store.get(Integer.toString(i));
                streamString = streamString.concat("\n");
                dos.write(streamString.getBytes());
                this.sleep(10);

                if(i==192){
                    i = 1;
                }
            }
            dos.close();
            streamSocket.close();
        }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
        catch(InterruptedException ie){
            ie.printStackTrace();
        }
    }
}
```

Appendix D

Client User Interface

In this appendix a navigational diagram of the user interface of the development client application is illustrated.

In Figure D.1 the arrows show the possible ways of navigating the client menu. Labels on the arrows describe software buttons (without quotation marks), and menu options (with quotation marks).

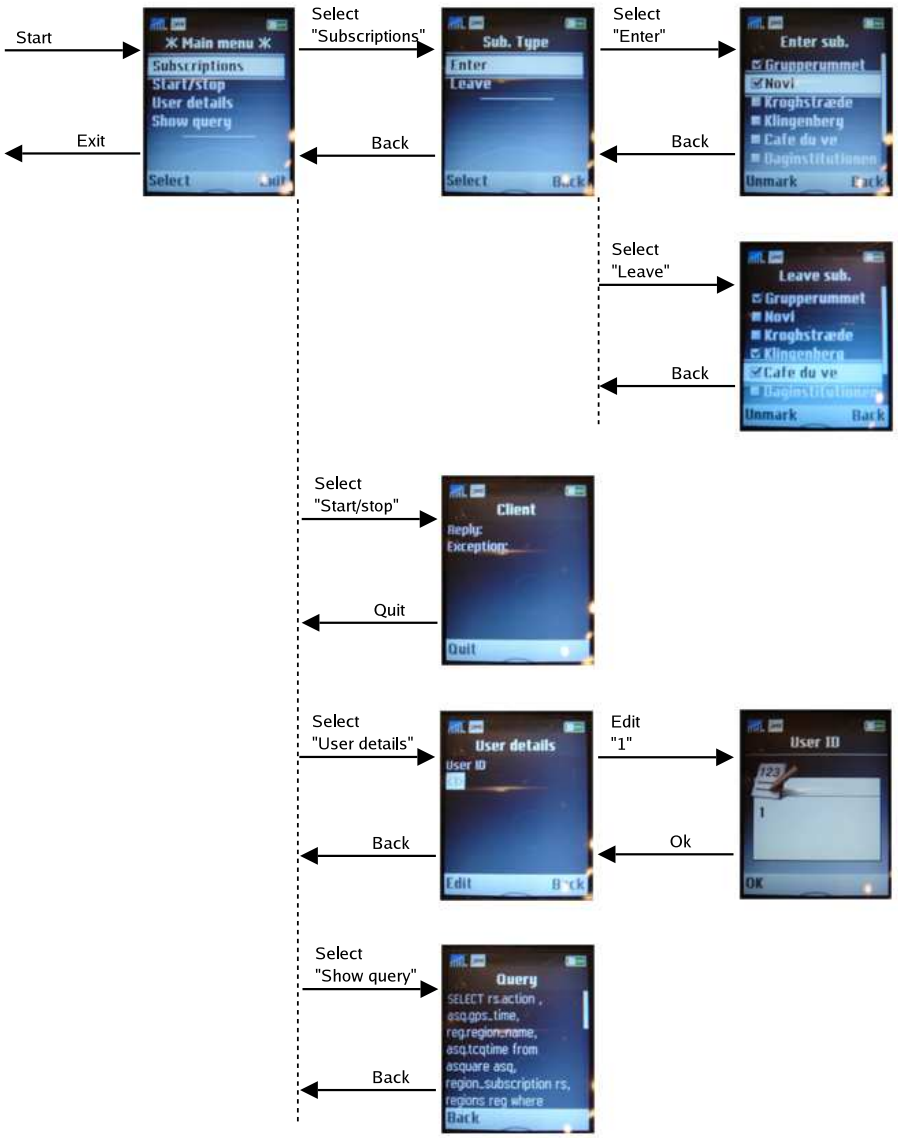


Figure D.1: Navigational diagram of developmen client application.

Appendix E

Summary

This project is named "BWHERE - Composite Spatiotemporal Query Operators in a Streaming and Mobile Context", and extends our last project "AWHERE - Handling of Data Streams in a Mobile Context".

The purpose of this project is to examine the three worlds of technologies, namely mobile environment, streams, and GISs. These worlds are the lower plane in Figure E.1. The upper plane represents the experimental plane with the three concrete systems that we have based BWHERE on.

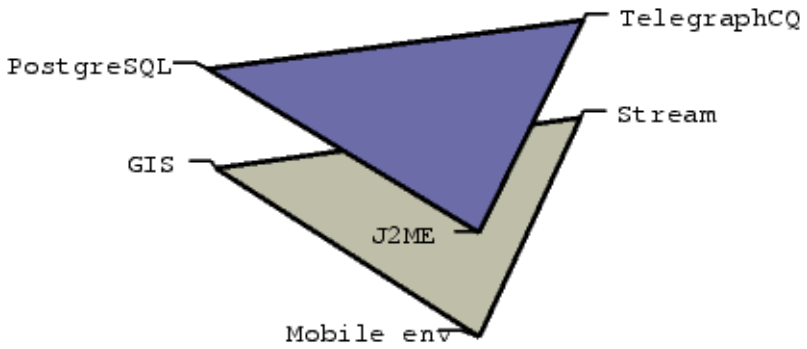


Figure E.1: The two-plane model.

Based on the figure and a preliminary analysis we split the main foci of this project in these main goals:

1. A theoretical solution to the problem of combining the three worlds of technologies. A solution that is based on the spatiotemporal model and the trajectory model. Further, a solution that includes both basic and composite spatial and spatiotemporal predicates.

2. An experimental system design and implementation, based on the theoretical solution. The experimental system will be implemented with the use of the following technologies: J2ME, GPS, GPRS, TelegraphCQ, and PostgreSQL.

3. Proving that the experimental system can function with a real mobile phone in a satisfactory way.

4. Proving that the experimental system scales in a acceptable manner according to an estimation of a reasonable number of users.

The project addresses these issues by providing a theoretical and experimental solution. The theoretical solution examines the worlds, and provides a set of composite spatiotemporal query operators for the streaming and mobile environment. Subsequently,

the result is applied to a experimental solution, with the convergent technologies from the three worlds, i.e., TelegraphCQ, PostgreSQL, and J2ME. A practical implementation of the composite spatiotemporal query operators, enter and leave, binds the worlds of technologies together.

Both theory and experiments revealed that such a combination of mobility, GIS, and streaming was a viable solution, although some of the technologies involved needs maturity.

Bibliography

[Aurora, 2005] Aurora (2005). http://www.cs.brown.edu/research/aurora/. Valid as of January 1st, 2005.

[Avnur and Hellerstein, 2000] Avnur, R. and Hellerstein, J. M. (2000). Eddies: continuously adaptive query processing. pages 261–272.

[Biering-Sørensen et al., 1996] Biering-Sørensen et al. (1996). *Håndbog i Struktureret Programudvikling*. Teknisk Forlag A/S.

[Bille et al., 2005] Bille, K. R., Jensen, A. S., Maach, M., and Søndergaard, J. (2005). Awhere - handling of data streams in a mobile context. Master's thesis, AAU - Aalborg University, http://www.cs.auc.dk/library/cgi-bin/detail.cgi?id=1105501345.

[CBB, 2005] CBB (2005). Cbb mobil. http://www.cbb.dk/. Valid as of June 7th, 2005.

[CeBIT, 2005] CeBIT (2005). Cebit 2005 trade fair.

[Civilis et al., 2004] Civilis, A., Jensen, C. S., Nenortaite, J., and Pakalnis, S. (2004). Efficient tracking of moving objects with precision guarantees. Technical report, http://www.cs.aau.dk/DBTR/DBPublications/DBTR-5.pdf.

[COUGAR, 2005] COUGAR (2005). http://www.cs.cornell.edu/database/cougar/. Valid as of January 1st, 2005.

[Cranor et al., 2003] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. (2003). Gigascope: A stream database for network applications. page 5.

[DST, 2004] DST (2004). Mobiltelefoner og abonnenter i 2002-2003. Danmarks Statistik http://www.dst.dk/.

[Egenhofer and Herring, 1992] Egenhofer, M. J. and Herring, J. R. (1992). Categorizing binary topological relations between regions, lines, and points in geographic databases.

[Egenhofer and Sharma, 1993] Egenhofer, M. J. and Sharma, J. (1993). A critical comparison of the 4-intersection and 9-intersection models for spatial relations: Formal analysis.

[Erwig et al., 1998] Erwig, M., Güting, R. H., Schneider, M., and Vazirgiannis, M. (1998). Abstract and discrete modeling of spatio-temporal data types. Technical report, Fernuniversität Hagen and Athens Univ. of Economics and Business, http://web.engr.oregonstate.edu/ erwig/papers/ModelingSTDT_ACMGIS98.pdf.

[Erwig and Schneider, 1999] Erwig, M. and Schneider, M. (1999). Visual specification of spatio-temporal developments.

[Golab and Özsu, 2003] Golab, L. and Özsu, M. T. (2003). Data stream management issues - a survey. Technical report, School of Computer Science, University of Waterloo, Waterloo, Canada.

[GPSUtility, 2005] GPSUtility (2005). Gps utility ltd. http://www.gpsu.co.uk/. Valid as of June 7th, 2005.

[JSR169, 2004] JSR169 (2004). Jsr-000169 jdbc optional package for cdc/foundation profile. Java Community Process: http://jcp.org/.

[JSR82, 2002] JSR82 (2002). Jsr 82: Java apis for bluetooth. Java Community Process: http://jcp.org/.

[Krishnamurthy et al., 2003] Krishnamurthy, S., Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., andFred Reiss, S. R. M., and Shah, M. A. (2003). Telegraphcq: An architectural status report.

[Madden and Franklin, 2002] Madden, S. and Franklin, M. J. (2002). Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*.

[MobiLife, 2005] MobiLife (2005).

[NiagaraCQ, 2005] NiagaraCQ (2005). http://www.cs.wisc.edu/niagara/. Valid as of January 1st, 2005.

[OpenCQ, 2005] OpenCQ (2005). http://disl.cc.gatech.edu/CQ. Valid as of January 1st, 2005.

[Patroumpas and Sellis, 2004] Patroumpas, K. and Sellis, T. (2004). Managing trajectories of moving objects as data streams. Second Workshop on Spatio-Temporal Database Management (STDBM 04).

[Pfoser et al., 2000] Pfoser, D., Jensen, C. S., and Theodoridis, Y. (2000). Novel approaches to the indexing of moving object trajectories. page 12.

[PostgreSQL, 2005] PostgreSQL (2005). Postgresql.

[Silbershatz et al., 2002] Silbershatz, A., Korth, H. F., and Sudarshan, S. (2002). *Database System Concepts*. McGraw-Hill, New York, USA, fourth edition.

[StatStream, 2005] StatStream (2005). http://cs.nyu.edu/cs/faculty/shasha/papers/statstream.html. Valid as of January 1st, 2005.

[STREAM, 2004] STREAM (2004). *STREAM: The Stanford Data Stream Management System*. Stanford University.

[TelegraphCQ, 2005] TelegraphCQ (2005). http://telegraph.cs.berkeley.edu. Valid as of January 1st, 2005.

[Tribeca, 1998] Tribeca (1998). Tribeca: A system for managing large databases of network traffic. page 13.

[Wakerly, 1999] Wakerly, J. F. (1999). *Digital Design: Priciples and Practices*. Pearson US Imports and PHIPEs.

[ZEEKIT, 2005] ZEEKIT (2005). Zeekit as.

Glossary

Notation	Description
CLAS	Continuous Location-Aware Service
CLDC	Connected, Limited Device Configuration
DBMS	Database Management System
DDL	Data-Definition Language
DSMS	Data Stream Management System
GIS	Geographical Information System
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communications
J2ME	Java 2 Platform, Micro Edition
LBS	Location Based Service
LCQL	Limited Continuous Query Language
MIDP	Mobile Information Device Profile
POI	Point Of Interest
RDE	Repeated Discrete Evaluation
WCH	Wrapper ClearingHouse