The Internet Spatial Spreadsheet: Enabling Remote Visualization of Dynamic Spatial Data and Ongoing Query Results over a Network

Glenn S. Iwerks
Computer Science Department, Center for Automation Research, and Institute for Advanced Computer Studies, University of Maryland at College Park
iwerks@umiacs.umd.edu

Hanan Samet[*]
Computer Science Department, Center for Automation Research, and Institute for Advanced Computer Studies, University of Maryland at College Park
hjs@umiacs.umd.edu

ABSTRACT

Moving object databases store and process data for objects that change location frequently. Materialized views maintained over time must be updated to reflect changes due to the motion of objects in their environment. To visualize view query results, displays must be updated to reflect the change. In this paper we present the Internet Spatial Spreadsheet (ISS) as a means to organize, query, and visualize changing spatial data in a network environment such as the Internet. The goal of the ISS is to keep client visualizations of query results up to date with the server state. This is accomplished by pushing the minimal set of spatial data needed for rendering query results on the client. Incremental changes to query results are subsequently transmitted to the client as the database is updated to keep the visualization current. Additional constraints in the network environment such as firewall limitations are also considered.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management; H.2.8 [**Database Management**]: Database Applications—*spatial databases and GIS*

General Terms

ALGORITHMS,DESIGN

Keywords

spatial databases, GIS, client server, visualization

[*]This work was supported in part by the National Science Foundation under grants EIA-99-00268, IIS-00-86162, and EIA-00-91474.

1. INTRODUCTION

Internet geo-spatial applications have become prevalent and practical through increased bandwidth and spatial database research. The Internet facilitates data sharing through standards, and distributed public geo-spatial data sources. Now a newer type of spatial data is becoming more prevalent. As mobile networks, sensor networks, and improved remote sensing techniques evolve, there is an increasing need to process and visualize moving object data that is frequently updated. This combined with network environments, such as the Internet, leads to new challenges in how to manage many updates to query results on a server and simultaneously visualizing the changes on a remote client. To address this problem we present the Internet Spatial Spreadsheet (ISS). The ISS is a means to organize, query, and visualize dynamic spatial data. It is a client-server system design to operate in a network environment while supporting frequent updates to the spatial data and simultaneous visualizations for the user.

The rest of this paper is organized as follows. Section 2 reviews some related work. An overview of the ISS architecture is given in Section 3. Section 4 presents a simple example to illustrate the operation of the ISS. Server side query processing is described in Section 5. Network communication is discussed in Section 6. Section 7 describes how data is presented to the user. Section 8 contains concluding remarks.

2. RELATED WORK

2.1 Spreadsheet for Images

Spreadsheets for Images [8] applies the spreadsheet concept to the image processing domain. In this case, the spreadsheet is a means of data visualization. Each cell in the spreadsheet contains graphical objects such as images and movies. Formulas for processing data can be assigned to cells. These formulas can use the contents of other cells as inputs. This ties the processing of data in the cells together. When a cell is modified, other cells that use it as input are updated. A similar capability is provided by the CANTATA programming language used with the KHOROS system [9].

2.2 SAND Browser

The SAND Browser is a front end graphical user interface for the SAND [2] spatial-relational database. The query results are displayed graphically. This gives the user an intuitive interface to the database to help the visualization of the data and the derivation of additional information from it. However, such a system does have limitations. In the SAND Browser, one primitive operation is processed at a time. A primitive operation is a query invoking one simple unary or binary query operation such as select, project, join, etc. When the user wants to make a new query, the results of the previous operation are lost unless saved explicitly in a new relation. As a result, there is no simple and implicit way to compose complex queries from primitives. In [6] we presented alternatives to the SAND Browser designed to overcome some of these limitations while maintaining ease of use and an intuitive interface.

2.3 Original Spatial Spreadsheet

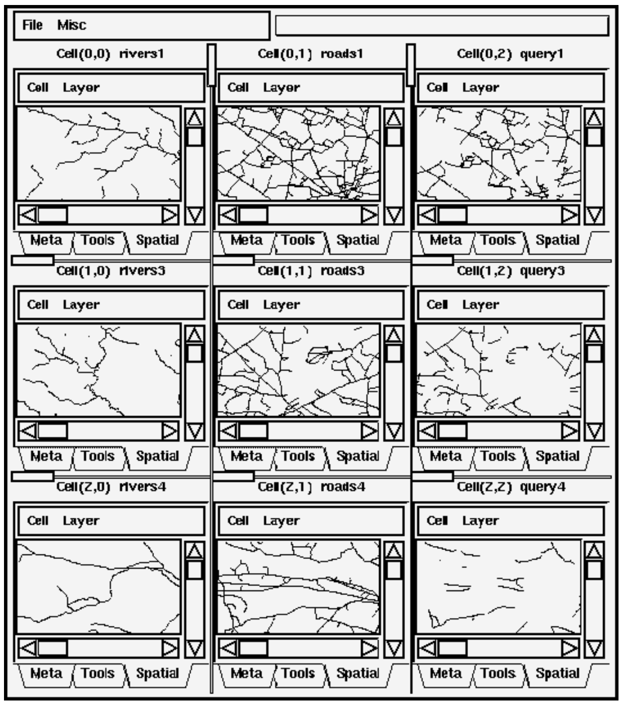

Figure 1: The original Spatial Spreadsheet: Cells display spatial data contained in base relations and query results associated with each cell.

The power of a spreadsheet resides in its ability to organize data, formulate operations on that data quickly through the use of row and column operations, and to propagate changes in the data throughout the system. The original Spatial Spreadsheet described in [6] attempts to combine a spreadsheet paradigm with a spatial database management system, creating a new way to conceptually organize spatial data, pose queries on that spatial data, and view the results (see Figure 1). In particular, the Spatial Spreadsheet provides a way to organize base relations and query results in a manner that is intuitively meaningful to the user.

The original Spatial Spreadsheet operates as a front end to a spatial database system called SAND [2] running as a single process on one machine. Although the original Spatial Spreadsheet is useful as a means to organize, visualize, and query spatial data, it was not designed to handle dynamic spatial data, or operate as a remote client.

3. INTERNET SPATIAL SPREADSHEET

The *Internet Spatial Spreadsheet* (ISS) extends the concept of the original Spatial Spreadsheet [6] to support dynamic spatial data, and operate over a network.

3.1 ISS Server

Conceptually, the ISS server manages a set of spreadsheets each consisting of a set of cells organized in rows and columns. Each cell manages the processing of a single relation. A cell's relation may be a base relation found in the database schema, or a materialized view. In the ISS, materialized view cells are also called *query cells*.

Incremental view maintenance techniques [4, 5, 10] have been extensively studied to efficiently maintain materialized views when changes occur. These techniques rely on the assumption that a relatively small number of tuples in an input relation are affected by any given transaction. This assumption is also known as the *heuristic of inertia* [5].

The data for each relation or materialized view is contained in three tables or relations. The first table is the main table containing the state of the entire base relation or query result. The other two tables contain pending updates to the main table. These are called the insert differential table, and the delete differential table. The main table is stored on disk. The differential tables are assumed to be small, so they are stored in main memory.

Consider a relation r. Let relation i_r be the set of tuples inserted into r during transaction Φ. The insertion update to r is expressed as $r' = r \uplus i_r$ where r' is the state of r after transaction Φ. Let relation d_r be the set of all tuples deleted from relation r during Φ. The deletion update to r is expressed as $r' = r - d_r$. By combining these two expressions we get $r' = (r \uplus i_r) - d_r$. The parentheses show the appropriate precedence needed in case a tuple is inserted and deleted during the same transaction. Symbols i_r, and d_r denote the insert differential table, and delete differential table of relation r, respectively.

Incremental view maintenance algorithms are written by substituting $(r \uplus i_r) - d_r$ for r' in the query expression. For example, the update for the selection query $\sigma r'$ becomes $\sigma((r \uplus i_r) - d_r) = (\sigma r \uplus \sigma i_r) - \sigma d_r$. In this way the selection need only be applied to i_r (d_r), and the result inserted to (deleted from) the current query result σr. This results in an incremental update to the view rather than recomputing the view from scratch (see [5, 7] for more details).

Each cell of the ISS manages a main relation table, and two differential tables to support incremental view maintenance (see Figure 2). When a base relation is open in a cell, the cell handles the processing of updates to that base relation. An update is a combination of one or more insertions or deletions to a base relation that take place during a single database transaction. These updates then propagate to query cells managing the materialized views. The update propagation algorithm described in Section 5 is differ-

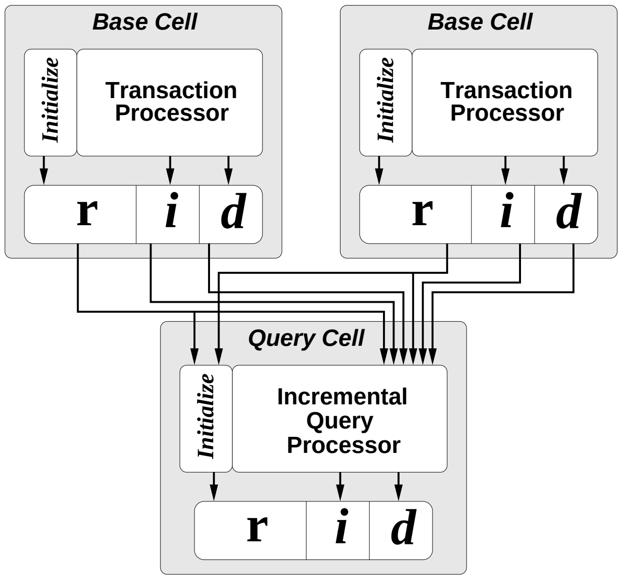

Figure 2: ISS data flow: r indicates a base relation, or materialized view. Tables *i* and *d* are insert and delete differential tables respectively. The query cell's view (a binary operation in this case) is initially computed using the base relations of each input cell. It is then incrementally updated when input cells are updated. Changes in the query cell's view are stored in its own differential tables. This allows query cells to be composed with other query cells. After the updates propagate through the system, the differential tables are applied to the cell's relation r and the differential tables are cleared for the next transaction.

ent from that of the original Spatial Spreadsheet presented in [6]. The original Spatial Spreadsheet did not update materialized views incrementally, but instead recomputed the query results from scratch.

3.2 ISS Client

The ISS client runs remotely on a separate machine. The conceptual architecture of the client mirrors the server in that it also has a set of cells arranged in rows and columns. There is a one-to-one correspondence between the cells of a given client and the cells on the server. Cells in the client handle user interactions with the data. This includes query formulation, and spatial data visualization.

Instead of rendering an image on the server and transmitting it to the client, the server transmits the geometry information for the spatial data to the client, then the client renders the image on the client. Multiple perspectives, or different points of view, may be rendered simultaneously without increasing the load on the server. Each cell can have a different perspective through zooming and panning of individual cell displays. Additionally, when the perspective changes, the information needed to render the image is already on the client. If rendering were done on the server machine, a new image would need to be transmitted to the client each time the perspective changed. This would not only increase network traffic, but also increase the load on the server with work not directly related to query processing.

Although broadband Internet access is becoming more prevalent, improvements in processor speeds, main memory capacity, and graphics hardware are progressing even faster. This leads us to believe that network bandwidth rather than client machine processing and graphics capability is a greater constraint.

In our approach we allow for clients running behind firewalls. We assume only the ability to operate a client web browser that accesses external web sites from the client machine using the HTTP[3] protocol. We do not assume that any other means of communication through a firewall may be available. The HTTP client pull model presents some interesting challenges when there is a need to push data to the client from the server. This happens when base relations are updated thereby requiring data to be sent to the client to update the visualization displays.

4. EXAMPLE

We will use the following query to illustrate the concepts presented in the following sections. Consider two relations $r(R)$ and $s(S)$ where schema $R = \{id, loc, type\}$, loc is a 2D point, id is a unique object identifier, and $type$ is a number. The schema of relation s is the same, $R = S$. As an example, consider the materialized view defined below. In our notation we denote a tuple in relation s as τ_s. For tuple $\tau_s \in s$ we denote the value of attribute α_0 in τ_s as $\tau_s[\alpha_0]$. The join of two tuples τ_r and τ_s is their concatenation is written $\tau_r\tau_s$.

$$Q = \{\tau_r\tau_s : \tau_r \in r \wedge \tau_s \in s \\ \wedge \, \mathsf{Distance}(\tau_r[loc], \tau_s[loc]) \leq 2 \wedge \tau_s[type] = 1\}.$$

This query returns all pairs of objects in r and s that lie within 2 distance units of each other, where all the objects from s are of type 1. Suppose that the initial states of r and s at time t_0 are as shown in Figures 3 and 4, respectively. Now, suppose that object y is moving at a constant velocity, while object a moves and then stops as shown in Figure 5. Intermittent updates to the database change the current known locations of y and a.

id	*loc*	*type*
a	$(2, 2)$	1
b	$(3.5, 5)$	1
c	$(6, 2)$	2

Figure 3: Base relation r at time t_0.

id	*loc*	*type*
x	$(3, 1)$	2
y	$(5, 3)$	1
z	$(6, 1)$	1

Figure 4: Base relation s at time t_0.

Now consider the example query Q. The result for time t_0 is shown in the first row of Figure 6. The locations of the objects participating in the join are indicated by the ovals in Figure 5a. Note that although object a is within distance 2 of object x, the pair is not included in the query result because the type of x is not 1.

Now suppose that at time $t_0 + 1$ minutes, the s relation is updated by deleting tuple $\{y, (5, 3), 1\}$ and inserting tuple $\{y, (4, 3), 1\}$, and the r relation is updated by deleting tuple

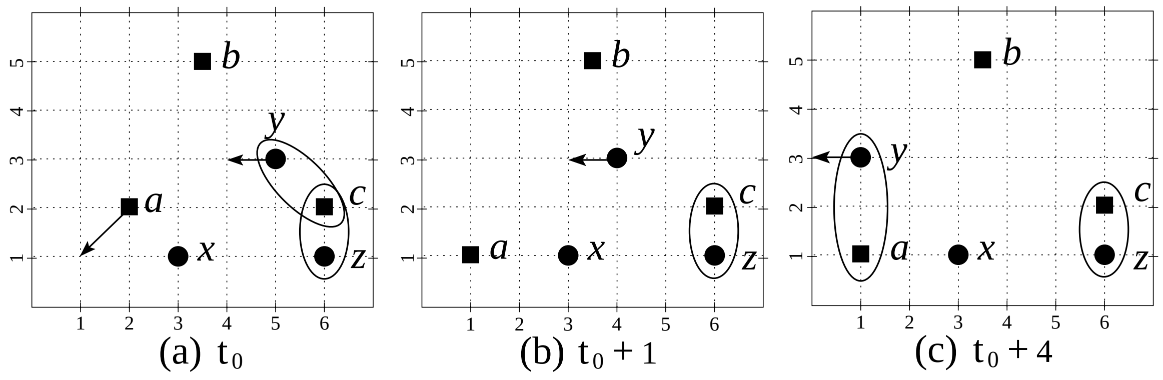

Figure 5: Graphical representation of the state of relations r (locations denoted by the ■ symbol) and s (locations denoted by the • symbol) at times (a) t_0, (b) $t_0 + 1$, and (c) $t_0 + 4$. The ovals show pairs of objects included in the query result shown in Figure 6. Arrows show the direction of motion of moving objects.

$\{a, (2,2), 1\}$ and inserting tuple $\{a, (1,1), 1\}$. The resulting change in the query result is shown in the second row of Figure 6, and graphically by the oval in Figure 5b. Now suppose that after 3 more minutes, at time $t_0 + 4$, an update changes object y's location from $(4,3)$ to location $(1,3)$. The join result after the update is shown in the third row of Figure 6, and corresponding ovals in Figure 5c.

at time:	*r.id*	*r.loc*	*s.id*	*s.loc*
t_0	*c*	$(6,2)$	*y*	$(5,3)$
	c	$(6,2)$	*z*	$(6,1)$
$t_0 + 1$	*c*	$(6,2)$	*z*	$(6,1)$
$t_0 + 4$	*a*	$(1,1)$	*y*	$(1,3)$
	c	$(6,2)$	*z*	$(6,1)$

Figure 6: Result of example query Q. The first row shows the initial result at time t_0. The second row shows the result at time $t_0 + 1$ after the deletion of tuple $\{y, (5,3), 1\}$, and insertion of tuple $\{y, (4,3), 1\}$ in relation s, and the deletion of tuple $\{a, (2,2), 1\}$ and insertion of tuple $\{a, (1,1), 1\}$ in relation r. The result at time $t_0 + 4$ is shown in the last row after tuple $\{y, (4,3), 1\}$ is replaced with tuple $\{y, (1,3), 1\}$ in s. Tuple $\{x, (3,1), 2\}$ from relation s does not appear in the join result because its *type* attribute is not equal to 1.

5. CELL UPDATE PROPAGATION

If any base relations used in the definition of the view are updated, then the view needs to be updated to reflect the change to keep query results current. To update materialized views after a transaction, it is often more efficient to reevaluate the query in terms of changes to the base relations instead of reevaluating the query from scratch.

When a transaction updates a base relation, the tuples to be inserted are entered into the insert differential table of the base relation before the transaction commits. Similarly, tuples to be deleted are entered into the base relation's delete differential table. When some view v is defined in terms of some view or base relation r, we say that v is a *child* of r, and r is a *parent* of v. When the transaction commits, the contents of the differential tables for a given view, or base relation, are fed to its children. Updates for each subsequent child materialized view are computed and fed to its own children, and so forth. The graph of parent-child relationships must be acyclic. When the propagation is complete, the differential tables are applied to their associated main tables and the differential tables are cleared.

Cell Marking: To ensure correct order of execution, a simple cell marking algorithm is employed. Before a transaction takes place, all cells are marked *clean*. When a base relation is updated it is marked *dirty*. After all the base relations affected by the transaction are processed, and before the updates are propagated to their children, each child of a dirty base relation is marked dirty as well. Next, all of their children's children are marked dirty, and so on. This process continues recursively until no more views can be marked.

To illustrate, consider the example query Q given in Section 4. To pose this query in the ISS, the user first opens the base relations in their own cells. Figure 7 shows relation s open in cell (0,0), and relation r open in cell (1,0). Now suppose that the user wants to know where all the objects of type 1 are located before the join is performed. To do this the user can create a view using the query $\sigma s = \{\tau_s : \tau_s \in s \wedge \tau_s[type] = 1\}$. In Figure 7, σs is in cell (0,1). Finally, to see which objects are within distance 2 of each other, the user creates a view in cell (1,1) using the query $r \bowtie \sigma s = \{\tau_r \tau_{\sigma s} : \tau_r \in r \wedge \tau_{\sigma s} \in \sigma s \wedge \mathsf{Distance}(\tau_r[loc], \tau_{\sigma s}[loc]) \leq 2\}$. At this point the view $r \bowtie \sigma s$, shown in cell (1,1) of Figure 7, is equivalent to the example query Q given in Section 4.

Now suppose, once the spreadsheet is set up, a transaction arrives at time $t_0 + 1$ minutes updating both relations r and s. When this occurs, the marking algorithm marks cell (0,0) and cell (1,0) as dirty. Once all base relations are marked, it then marks their children dirty, cells (0,1) and (1,1).

In the next step of update propagation, all cells managing base relations are marked clean. Then all query cells are placed on a queue and examined one at a time. When a cell is popped off the queue, if all its parents are clean, then it is marked clean and the incremental view maintenance for that cell is performed. If all its parents are not yet clean, then it is placed back on the end of the queue. No updates for a given materialized view are computed until all its parents are marked clean. If instead of using this cell marking algorithm

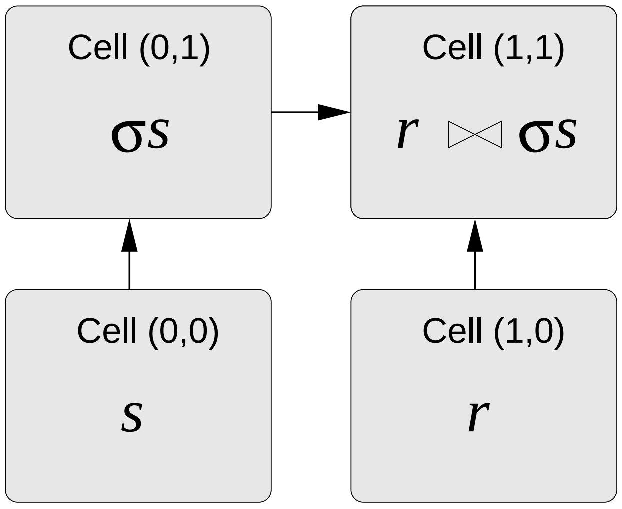

Figure 7: Update propagation example: The example query Q from Section 4 is broken up into two views. Supposing the user wants to see where all the objects of type 1 are located before the join is performed, a view is created as a selection on relation s using the query $\sigma s = \{\tau_s : \tau_s \in s \wedge \tau_s[type] = 1\}$, in cell (0,1). To see which objects are within distance 2 of each other, the user then creates the view $r \bowtie \sigma s = \{\tau_r\tau_{\sigma s} : \tau_r \in r \wedge \tau_{\sigma s} \in \sigma s \wedge \mathsf{Distance}(\tau_r[loc], \tau_{\sigma s}[loc]) \leq 2\}$ in cell (1,1). Arrows indicate data update flow from parent to child.

we simply iterate through the cells and update a view after any parent is updated, then incorrect results may occur. For instance, without cell marking in our example, cell (1,1) can be updated as a result of the update to cell (1,0), but before cell (0,1) is updated. Since cell (0,1) is also a parent of cell (1,1) this could lead to incorrect results.

6. PUSHING DATA FROM SERVER TO CLIENT USING HTTP

After update propagation, and before applying the differential tables to the main tables, the spatial data of the non-empty differential tables are transmitted to the client for incremental update of the data visualization.

Pushing data from a server to a client in an Internet environment can present additional challenges when the client is behind a firewall. Many users may want to run their clients from behind a firewall for security reasons. Firewalls help prevent attack from outside by closing off communication ports through the firewall. A firewall configuration may allow clients to initiate connections through particular ports to servers on the outside. For example, port 80 is sometimes open for web browsers to access web servers outside the firewall. Opening ports so that connections may be initiated from outside the firewall is not desirable because it can make the system more vulnerable to attacks from outside.

Although access from clients inside a firewall to the outside may vary from firewall to firewall, many firewall administrators at least allow web browsing from inside the fire wall. This is done via the HTTP protocol through a specific port either directly or via a proxy. The Hypertext Transfer Protocol (HTTP) [3] is designed to pass Hypertext Markup Language (HTML) [1] documents between web servers and clients. HTTP tunneling is a technique used to pass arbitrary data back and forth between clients and servers by placing it in an HTTP wrapper, using the HTTP protocol to send the data through the firewall.

The HTTP protocol enforces a client pull model. HTTP is a request/responce protocol where the request originates from the client. The HTTP session has two phases. In the first phase, the client sends data to the server. In the second phase the server receives data from the client. Once the second phase starts, no more data may be sent to the server during that session. When the second phase is complete, there is no more activity until a new request is issued by the client. The protocol does not provide any way for servers to send data without an initial client request.

The HTTP client pull model is a problem in the ISS. Updates to the base relations on the server side from external sources other than the client require the server to push data to clients to update visualization displays. Since HTTP does not support server initiated sessions, the server can not unilaterally push data to a client. To get around this, a client polls the server for updates in a continuous never ending HTTP session that quickly moves to the second phase and stays there. This is called the ISS *polling session*. If for some reason the connection is terminated (e.g., the proxy closes the connection after a time out period), then the client immediately establishes a new polling session.

Multi-threading can be used to enable the client to continually poll the server and still send data to the server at the same time allowing more than one HTTP connection or session to be active at a time between a client and server. Multi-threading is supported in most popular general purpose computer systems today, either directly by the operating system, or through programming libraries.

The threads handling the polling session are called the ISS *push threads*. There is both a client side push thread and a server side push thread (see Figure 8). The client push thread initiates an HTTP polling session with the server. The server spawns its own server side push thread when contacted by the client to service the polling session. Other threads running in the server handle processing of data. The other threads push data to the client by placing the data on a queue. When data is placed on the queue, the server push thread wakes up, pops the data off the queue, and transmits it to the client. On the client side, the client push thread receives the data and places it on a queue. Another thread in the client, called the *data processing thread*, processes each item on the queue in turn. Having a separate thread to receive data and place it on the clients queue enables data to be transmitted while processing data already received in parallel.

If the data pulled off the queue is an update to a relation, or a query result, then the appropriate visualization displays are updated. To accomplish this, each cell controlling its own display window registers a callback with the communication event thread to be invoked when data is received from the server. Callback procedures or methods are registered with the data processing thread to be invoked when particular kinds of data are transmitted from the server. The data

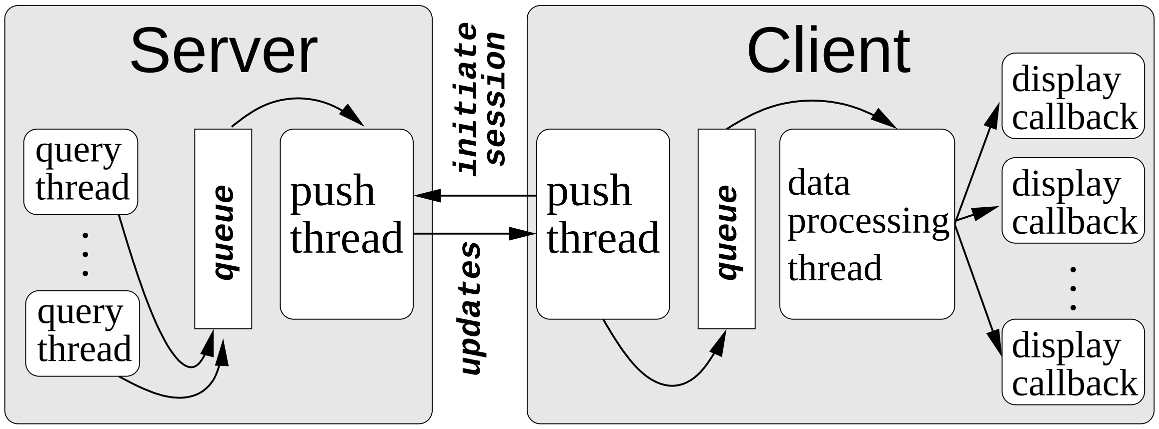

Figure 8: ISS client-server polling session. The client initiates the HTTP session. Updates generated in the server query threads are placed on a queue in the server. The server's push thread pops updates off the queue and transmits them to the client. The client push thread receives the updates and places them on the client's queue. The client data processing thread pops the data off the queue and executes the display callbacks. Each display callback is associated with a particular spreadsheet cell in the client.

processing thread invokes the appropriate callbacks depending on the nature of the data it pops off the queue.

Let us now consider again the example from Section 4 set up in the spreadsheet shown in Figure 7. Recall that relation s is updated at time t_0+1 by deleting tuple $\{y, (5,3), 1\}$ and inserting tuple $\{y, (4,3), 1\}$. The selection predicate on the view in cell (0,1) is $\tau_s[type] = 1$. The update propagation algorithm computes $d = \{y, (5,3), 1\}$, and $i = \{y, (4,3), 1\}$ for cell (0,1). Before moving on to the next cell, data from d and i are placed on the server's push queue by the query thread processing the transaction. The data needed for rendering in this case is the object id and old location, $\{y, (5,3)\}$, and the object id and new location, $\{y, (4,3)\}$. The server's push thread reads this data off the queue, and then transmits this information to the client where the client's push thread places the update information for cell (0,1) on its own queue. The data processing thread of the client in turn pops this data off the queue. It invokes all callbacks that are registered for update data on cell (0,1). These callbacks take the update information as an argument and use it to update data visualization displays.

7. INTERACTIVE VISUALIZATION

The spatial attributes of base relations and query results are transmitted to the client to be rendered. This enables interactive visualization without the need to transmit data from the server again when the perspective changes and a new rendering is needed. The user can pan and zoom without any further interaction with the server.

In the ISS, every client cell owns its own display window to display spatial data from the corresponding cell on the server. The data used to create the rendering is saved in a separate data structure to be used for rendering when the perspective changes. Any cell can also display data belonging to other cells. This allows the user to overlay data from different cells in the same display window. Although this other data is owned by the other cells, it is not rendered by other cells. This is because each cell may have a different perspective (e.g. different zoom or pan). Each cell renders data according to its own perspective.

In a network based environment, the spatial data to be displayed must fit in main memory on the client. This requirement can be relaxed if the client is allowed to write files to the local machine. Non-spatial data, other than object ids, are not transmitted to the client except in small amounts when the user wants to know the details about a particular object.

8. CONCLUSION

As a proof-of-concept we implemented the algorithms and techniques described here in JAVA (client and server front end), Tcl (server interface), and C/C++ (database server engine). The implementation demonstrated the ability of this approach to achieve the goal of keeping client visualizations of spatial data up-to-date as materialized view results change. This was accomplished in the Internet environment constrained by firewall security. We have not performed any empirical studies of the ISS yet. Future work includes a comparison of this approach with a more conventional approach, such as rendering images on the server and transmitting the finished image to the client for display. In particular, we want to investigate how the database update rate and data set size affect network load and response time.

9. REFERENCES

[1] T. Berners-Lee and D. Connolly. Hypertext markup language–2.0. Technical Report RFC 1866, Network Working Group, November 1995.

[2] C. Esperança and H. Samet. Spatial database programming using SAND. In *Proceedings of the Seventh International Symposium on Spatial Data Handling*, volume 2, pages A29–A42, Delft, The Netherlands, August 1996.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1. Technical Report RFC

2616, The Internet Society, Network Working Group, June 1999.

[4] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD Conference*, pages 328–339, San Jose, CA, May 1995.

[5] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C., May 1993.

[6] G. Iwerks and H. Samet. The spatial spreadsheet. In *Visual Information and information Systems: Third International Conference, VISUAL'99*, volume 1614, pages 317–324, Amsterdam, The Netherlands, June 1999. Springer-Verlag.

[7] G. Iwerks and H. Samet. Incremental view maintenance of spatial joins. Technical Report CS-TR-4179, University of Maryland, College Park, MD, August 2000.

[8] M. Levoy. Spreadsheets for images. In *Proceedings of the SIGGRAPH'94 Conference*, pages 139–146, Los Angeles, CA, July 1994.

[9] J. Rasure and C. Williams. An integrated visual language and software development environment. *Journal of Visual Languages and Computing*, 2(3):217–246, September 1991.

[10] N. Roussopoulos and H. Kang. Principles and techniques in the design of adms±. *IEEE Computer*, 19:19–25, December 1986.