Federated Spatial Cursors

Nazario Cipriani, Matthias Grossmann, Daniela Nicklas, Bernhard Mitschang

[1]Universität Stuttgart, Institute of Parallel and Distributed Systems
Universitätsstraße 38, 70569 Stuttgart, Germany

{cipriani, grossmann, nicklas, mitschang}@ipvs.uni-stuttgart.de

Abstract. *The usage of small mobile devices for data-intensive applications becomes more and more self-evident. As a consequence we have to consider these devices and their inherent characteristics in future system designs, like the limitations of memory and communication bandwidth. For example, when querying data servers for information, a mobile application can hardly anticipate the size of the result set. Our approach is to give more control over the data delivery process to the application, so that it can be adapted regarding its device status, the costs and availability of communication channels, and the user's needs. This paper introduces a flexible and scalable approach by providing spatially federated cursor functionality. It is based on an open federation over a set of loosely coupled data sources that provide simple object retrieval interfaces.*

1. Introduction

With the growth of online accessible data and information systems, the need for integration architectures is increasing. As can be seen in the Web 2.0 trend, more and more information is provided by autonomous data sources, like web sites, Wikis, or web services. To uniformly integrate this information for application use is a very cumbersome, and, at large, almost impossible task. However, when focusing on a certain application domain, we can exploit common characteristics to provide integrated views: e.g., WWW search engines integrate their search results based on rankings that represent the relevance to the user's query. In the Nexus project, we target the upcoming application domain of location-based information services and pervasive computing. Here, new data-intensive applications emerge, which support their users with the right information at the right time and right place, i.e., providing on demand what fits best to the user's current situation [Dey and Abowd 1999]. They often rely on large-scale information systems, where the data is scattered across a multitude of data sources ranging from web sites over digital libraries and geo-information systems to sensors and other stream-based sources. Our integration approach is based on an open federation over a set of loosely coupled data sources that provide simple object retrieval interfaces.

In contrast to conventional distributed database systems, the partitioning of the information is unknown. There is no closed-world assumption, since data providers can dynamically connect and disconnect from the platform. Also, there can be multiple representations of real-world entities in several data providers. Our open federation differs also from conventional federated database systems: since it is based on simple object retrieval and does not provide the full-fledged SQL function set, it does not have to materialize the whole result set within the federation layer when integrating the results from the data providers.

This allows us to develop a scalable algorithm for object retrieval that works on partial results from data providers. It is based on the concept of a federated cursor. Cursors are a long-known database concept that allows an application to piece-wise retrieve tuples of a result set [Date 2000]. This is especially beneficial if the applications run on resource-limited devices, which typically retrieve information over a costly wireless communication channel. Such devices are often used in the areas of location-based services and pervasive computing.

The remainder of the paper is structured as follows: we overview related work in Chapter 2. Chapter 3 describes the Nexus platform which provides an integrated view over geographic data sources in the application domain of location-based and context-aware services. In Chapter 4, we introduce the cursor concept, and in Chapter 5 we present a flexible and efficient federation strategy also covering histograms. Our prototype implementation is evaluated in Chapter 6. Finally, in Chapter 7 we conclude the paper with an outlook on future work in this area.

2. Related Work

There has been some work addressing the problem of efficiently processing and incrementally retrieving partial results. [Haas et al. 1999] try to speed up data intensive applications needing fine-grained object access by loading the cache of the system with relevant objects. The decision of what objects are relevant is taken by the frequency applications access objects. However, this technique does not consider multiple representations of the same object containing incomplete or partial information distributed over several data sources. In this case one has to find and fetch all representations of an object in order to get a complete and consistent object representation.

Garlic [Josifovski et al. 2002] is a platform for federated data management of relational data sources based on IBM DB2. For the incremental retrieval of the result set two possibilities are described. One is to materialize the entire result of each data source. The other is the retrieval of the data using the cursor mechanism. Each time *Fetch* is invoked, one data element a time is retrieved from the data source. Here, no possibility of sophisticated retrieval of the result sets is mentioned. The possibility of incomplete partial results is also not taken into consideration.

In Disco [Tomasic et al. 1996] the problem of dealing with unavailable data sources is addressed. The selected approach uses a *partial evaluation* semantics to return partial answers to queries. Here the portions of the query that could not be answered are mapped back to OQL and integrated with the portions of the query being answered by data providers. It is neither described how exactly the results are retrieved from the data providers nor how the partial results of the portions of the query are integrated to a single answer.

Information Manifold [Levy et al. 1995, Levy et al. 1996] deals with the efficient query processing in a distributed environment involving a large number of data sources. They use descriptions of the data sources for a given query to identify relevant sources, query these sources and finally collect the complete result from these partial results. The query processing engine tries to recognize sources providing redundant information and prunes them. No integration of the partial results or further computations are made. This has to be done by the inquiring application. Also, no further reflection on alternative retrieval mechanisms were made.

There also exist several mediator-based systems like TSIMMIS [García-Molina et al. 1997] or MedMaker [Papakonstantinou et al. 1996]. However, we focus on location-aware applications using location-based data and provide domain-specific operators and optimizations.

3. The Nexus Platform

The Nexus platform is a federated open system for location-based applications [Nicklas and Mitschang 2004]. As depicted in Figure 1, the Nexus architecture is built up in three tiers: applications, a federation tier containing Nexus nodes and a service tier consisting mainly of context servers, which provide stored or sensed data. Context servers must implement a predefined interface, through which they are contacted by Nexus nodes, and they must register at the Area Service Register (ASR), announcing the area they offer data for. Otherwise the implementation of a context server is not restricted, thus it can easily be tailored to the needs of different kinds of data like positions of vehicles (high update rates) or geometries of buildings (large data volumes) [Grossmann et al. 2005]. Being an open system, adding new context servers to the Nexus platform is not restricted. In particular, it is possible that the data of a new context server overlaps with existing ones in both its service area and content, which can lead to multiply represented objects (MReps). When integrating different result sets from different context servers, Nexus nodes try to detect such multiple representations based on location-based criteria and merge them into a single object [Volz and Walter 2004, Volz 2006]. In the following, the term *federation* is used as a synonym for Nexus nodes.

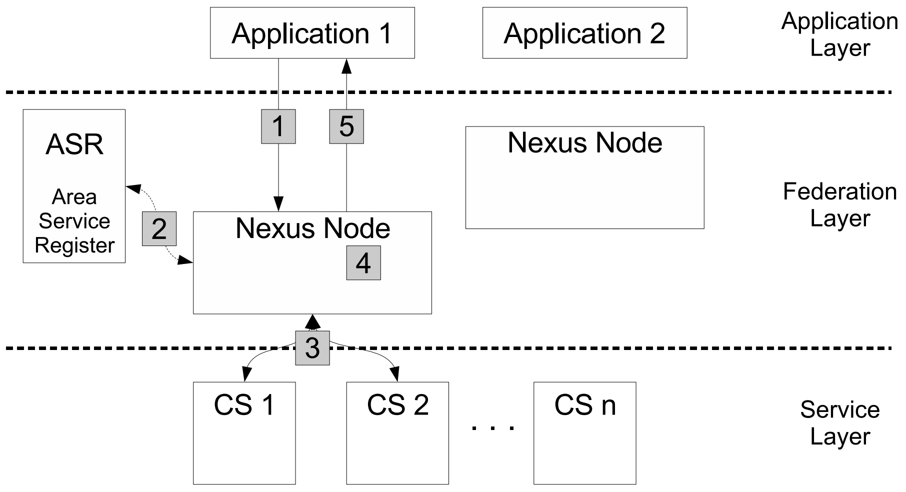

Figure 1. Overview of the original Nexus architecture

The Nexus platform uses a request-response protocol in which queries typically contain a spatial restriction. The processing model is depicted in Figure 1:

1. An application sends a query like *Menu and position of all restaurants closer than 1 mile to my current position* to an arbitrary Nexus node.
2. The Nexus node determines the relevant context servers by an ASR lookup based on the spatial restriction and the queried object type. In the example above the

spatial restriction corresponds to *closer than 1 mile to my current position* and the object type to *restaurants*.

3. The Nexus node forwards the query to those context servers. The context servers process the query and send back their results.
4. The Nexus node integrates the context servers' results. It detects and merges multiply represented objects (MReps). For this, domain-specific methods are used that exploit the spatial structure of the data: only objects in a spatial vicinity are considered candidates for being MReps.
5. The Nexus node returns the integrated result to the application.

Initially, this processing model computes and forwards the full result to the requesting application, which has no control over the data transfer. Obviously, when transferring large results, this can overburden resource-limited devices. For this reason we developed a better suited processing model based on the federated cursor concept.

4. The Federated Nexus Cursor Concept

The main idea for this federated cursor approach is to

- request only context servers that actively contribute
- process only the necessary result data
- support temporarily disconnected applications
- support mobility of applications (free choice of Nexus node).

In the past, cursors were used to bridge the so-called "impedance mismatch" between database systems and programming languages. The cursor allows conventional programming languages to cope with tuple-wise processing by providing a pointer to the actual tuple to be worked up.

The idea has been retained but applied to the domain of a federated, context-aware platform. Here, the cursor concept is used for retrieving partial results of spatial queries in order to prevent memory overflow and to save communication bandwidth, thus bridging the "resource mismatch" between often resource-limited mobile devices and the 'unlimited' server infrastructure.

Using a cursor, an application does not have to wait until the entire result is transferred before processing it. Depending on the type of connection there may be unwanted disconnections: the larger the result, the higher the risk that the full result never reaches the querying application. Also, certain network paths may be expensive.

One way to overcome this problem is to partition the spatial query into many small disjoint sub-queries and post each sub-query subsequently. However, this approach requires knowledge about the results' size of each sub-query which (without the necessary information) cannot be clearly predicted. Thus, the way the initial query should be partitioned is unclear. With a cursor no partitioning is necessary. Instead the result is partitioned with respect to the application needs, which is especially useful if the result is suitably ordered.

In the subsequent chapters the general concept of a federated, status-conscious cursor is introduced, which is used to efficiently retrieve objects over distributed data sources by exploiting the spatial nature of the data.

4.1. Query Processing Sequence

Up to now, a Nexus application posts a spatial query and receives an answer consisting of the query result. The new cursor-based processing model is three-phased. This is analogous to the cursor processing as described in [Date 2000]. The application has to post a query associated with a cursor on the query's result. After that, the application can start to piece-wise process the result. In the end, the result is deleted, if either its lifetime has expired or the application signals that it is no longer needed.

4.1.1. Phase 1: Initialization Phase

In the initialization phase preparations for the next phase (delivery phase) are made. The necessary steps are as follows (cf. Figure 2):

1. An application sends a spatial query to an arbitrary Nexus node and additionally asks the federation to create a cursor on the query's result.
2. The Nexus node determines the relevant context servers by an ASR lookup based on the spatial restriction and the object type in the query.
3. The Nexus node forwards the query to those context servers, which process the query and send their results back. Additionally, the federation sends back an ID (called Nexus Session Locator, see below) of that cursor to the application.

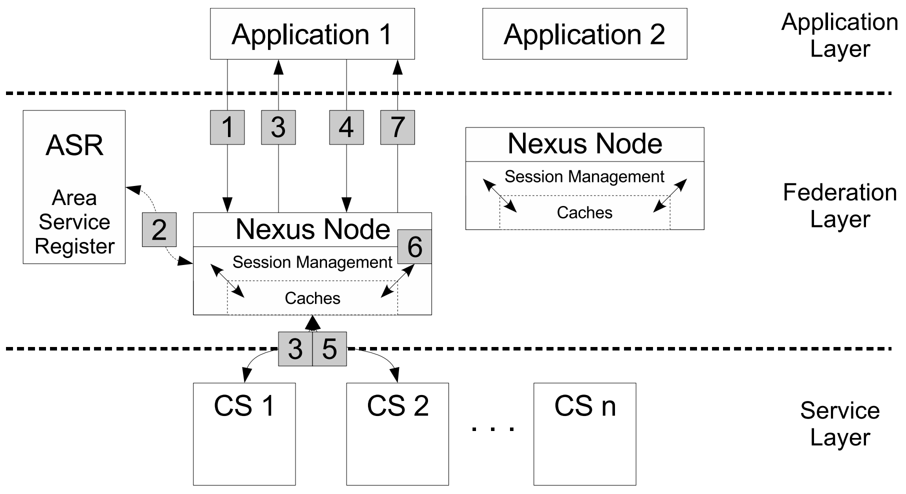

Figure 2. Overview of the new architecture

4.1.2. Phase 2: Delivery Phase

If the initialization phase has been successfully completed, the application is able to send cursor operations on its cursors to the federation to give access to the result data piece by piece. This phase is called delivery phase. The necessary steps are as follows:

4. The application posts a *next* operation stating the next elements pertaining to a certain result identified by an ID.

5. The federation looks for the result belonging to the ID and prepares the objects that go to the result set. Objects have to be retrieved from the context servers (if they are not already in cache).
6. Multiply represented objects have to be detected and merged. This operation can reduce the number of objects.
7. The result set is sent back to the application.

The application repeats the delivery phase until the end is reached or it does not need any further elements and decides to finish the retrieval. If the application signals that decision, the federation enters the termination phase.

4.1.3. Phase 3: Termination Phase

This final phase is entered if the lifetime of the result has expired or if the application signals the federation that it does not need more elements. The resources connected to the ID are released.

4.2. Session Management

For identifying sessions within the Nexus platform we introduce the so-called Nexus Session Locator (NSL). A NSL consists of two parts: a basic service part, which encodes the Nexus node the session was created on and thus holds the session information and a session identifier (SID). The hosting node is encoded within the NSL to support distributed session management: since we want to support mobile devices, an application can change its Nexus node. Using the NSL, a Nexus node that receives a cursor query for a cursor that it does not host can easily forward that query to the correct node. If a mobile device changes the Nexus node during operation, the new node has to retrieve the specific application information from the relevant Nexus node encoded in the NSL in order to be able to process the request adequately. For this, there are two possibilities: one is to transfer all relevant information to the new node (incrementally or at once) and to replace the part of the NSL storing the host with the new host address. The other is to always forward the query to the original node.

5. Federated Processing Strategies

After introducing the general federated cursor concept, we discuss federated processing strategies. In order to optimize query processing, the context servers should support a cursor concept, too. It is an optional feature of a context server. Without that functionality the entire result from each context server has to be transferred to the federation. In Figure 2, the context servers are also extended by a session management in order to be able to hold application specific information. In that way, the results can be kept at each context server locally and objects just needed will be transferred to the federation.

To provide optimal response times for applications, the federation should precache partial results [Haas et al. 1999]. There exist several ways to do this. The naive approach is to query all relevant context servers and cache all results locally. This approach has the advantage that there is no more communication overhead between the federation and the context servers and long latencies for query answering are avoided. But it suffers from high memory consumption within the federation layer and a long initialization phase since the results of all context servers must be fetched.

To reduce the memory consumption at the federation layer, spatially portioned queries can be sent to the context servers. Here the initialization phase consists of the non-trivial problem of partitioning the query. Objects may be queried that currently are not needed and in worst case never needed and it must be taken care that all multiply represented objects are present for the merge operation at processing time.

Both solutions sketched above are not recommendable. One suffers from memory consumption in the federation layer. The other suffers from communication overhead between the federation and context servers and could also miss information for some objects. So there is a trade off between memory consumption and the system load.

5.1. Cache Histograms

A major feature of the Nexus federation is the merging of multiply represented objects (MReps). To correctly perform this operation also in the cursor mode, we have to pre-cache partial results in a way that all candidates for a MRep-merge are present whenever this operation is carried out. The naive way would be to pre-cache the whole result at federation level. However, this introduces an often unnecessary memory usage at federation level and communication overhead between federation and context servers, particularly if an application does not retrieve the whole result.

We use cache histograms to solve that problem in an efficient way. A single cache histogram represents the query-dependent frequency distribution of the resulting objects based on a sorting criterion (i.e., the distance from a geographical point). Cache histograms are provided by each context server. A cache histogram consists of a set of cache histogram entries. Each cache histogram entry consists of a bucket value which indicates how the partial result of a context server was sorted and the amount of occurrences of that bucket within that partial result. A bucket here refers to a discrete point in the sorting domain and not to an interval as usual.

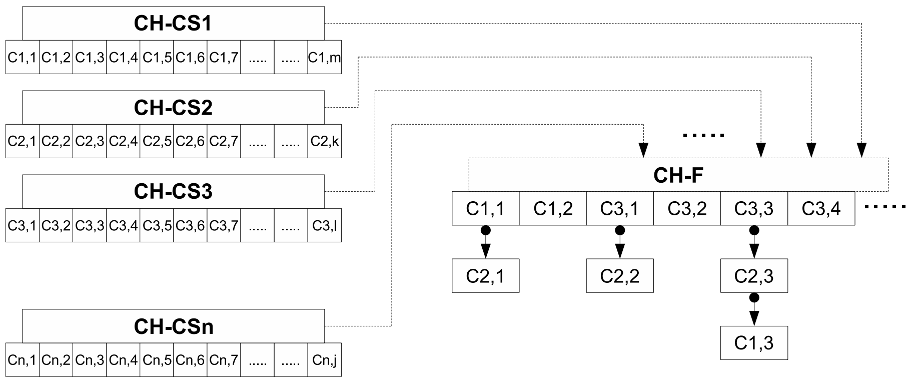
CH-CS1
C1,1 C1,2 C1,3 C1,4 C1,5 C1,6 C1,7 .... C1,m
CH-CS2
C2,1 C2,2 C2,3 C2,4 C2,5 C2,6 C2,7 .... C2,k
CH-CS3
C3,1 C3,2 C3,3 C3,4 C3,5 C3,6 C3,7 .... C3,l
CH-CSn
Cn,1 Cn,2 Cn,3 Cn,4 Cn,5 Cn,6 Cn,7 .... Cn,j
.....
CH-F
C1,1 C1,2 C3,1 C3,2 C3,3 C3,4 .....
C2,1
C2,2
C2,3
C1,3

Figure 3. Federated cache strategy using cache histograms

As shown in Figure 3, each context server delivers a cache histogram (CH-CS1 to CH-CSn) which is spatially sorted. $C_{a,b}$ corresponds to a cache histogram entry and gives the value of the cache entry and its frequency of occurrence. In our example the value refers to the distance of an object to the reference point, e.g., $C_{1,1}$, with a value of

<17,5>, addresses 5 nearest objects from context server 1 with a distance of 17 to that reference point.

If cache histograms supplied by the corresponding context servers are not already sorted by the bucket value, the federation has to do it by itself. That may occur if the cache histogram is created before sorting the partial result or the context server does not support sorting at all. Usually the context servers support result sorting. If no sorting criterion is specified, the federation has to sort it. The federation now merges the cache histograms delivered by each data context server into a federated cache histogram in order to get an overall overview (CH-F) of all data sources involved in the incremental retrieval process. The most important information at this point is the order in which the context servers should be queried, what context servers have to be queried and the quantity of objects (which is encoded in the cache histogram entries) to query the context servers for.

Since there may be multiple representations for the same real world entity, there can be objects with the same sorting value in different cache histogram. In that case these entries are stored as a linked list as shown in Figure 3. Elements in the linked list potentially represent the same object. All these objects must be transferred to the federation in order to guarantee a lossless merge. Whether two or more entries in the linked list represent the same object is decided by the federation's merger algorithm. Here $C_{1,1}$ and $C_{2,1}$ got the same bucket value and are thus stored as linked list. Taking for example $C_{1,1}$ with a value of <17,5> and $C_{2,1}$ with a value of <17,3>, the federation would first ask context server 1 for the next 5 objects and then context server 2 for the next 3 objects each with a distance of 17 to the reference point.

Listing 1 shows the cache histogram algorithm in pseudocode. It is used by the federation to build up a federated cache histogram.

Listing 1. The cache histogram algorithm

```
// application sends query to system
receive application query

// determine the relevant context servers
ask ASR for relevant context servers

// answer
send NSL to application

// send query to all necessary sources
for each context server do
    forward query
    // get cache histograms from each context server
    receive the cache histogram
    // eventually sort them
    if cache histogram not sorted
        sort cache histogram

// merge the cache histograms
merge cache histograms to federated cache histogram
```

5.2. The Retrieval Process Using Cache Histograms

Internally, the cursor is split in an horizontal (H) and vertical (V) component. The H component traverses the cache histogram from left to right, the V component from top to bottom. The algorithm is shown in Figure 4 for a *next* operation. The initial state of the algorithm is displayed in the upper left. The H component corresponds to the current cursor position. The V component indicates the position within the linked list of elements with the same bucket value.

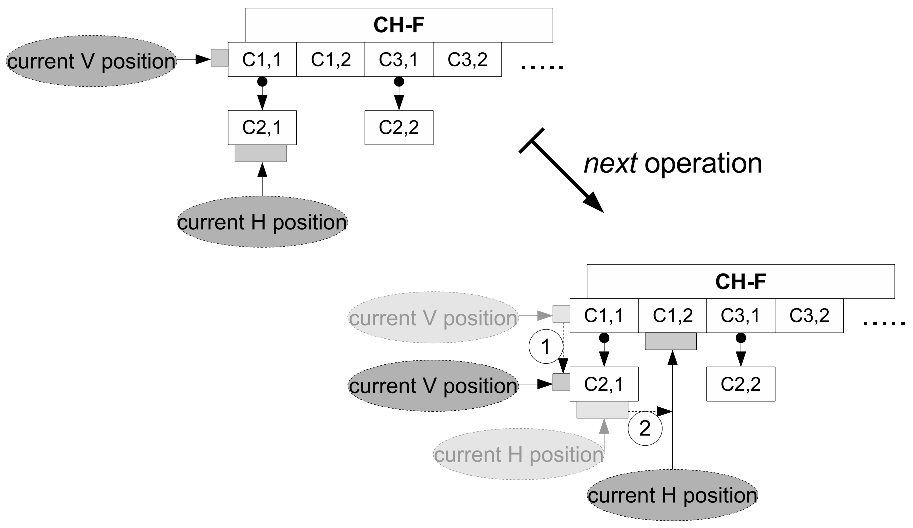

Figure 4. One cache histogram retrieval step

On the bottom right side the next two steps of the algorithm are displayed. First all elements in the linked list have to be processed (step 1), to ensure that all representations of the same object are retrieved.

The next step is to move the H-position by one to the right (step 2). If the linked list here has also got more than one element then step 1 is repeated. Otherwise step 2 is repeated.

The algorithm ends filling the federation caches if the amount of objects needed to answer the previously posted query is reached or if there are no more objects to retrieve. In the last case, a message stating that there are no more objects is sent to the inquiring application.

The algorithm works in an efficient way in terms of memory consumption and network load because only relevant context servers are queried for objects, irrelevant context servers are not considered as the federation only retrieves objects as a result of *next* operations. Furthermore no multiply represented objects are missed. Listing 2 shows a simplified version of the retrieval process using cache histograms.

Listing 2. Retrieval process algorithm using cache histogram

```
// application requests next N objects
K := number of objects in output buffer
PL := []
do N - K times
    // output buffer does not contain enough objects
    if V-component points to cache histogram entry
        P := context server in current cache histogram entry
        M := bucket size in current cache histogram entry
        if P in PL
            increment number of objects to fetch from P by M
        else
            append P to PL
            set number of objects to fetch from P to M
        move V-component one step down
    else
```

```
        move H-component one step right
OL := []
for each P in PL do
    retrieve the given number of objects from P
    append objects to OL
merge objects in OL
append OL to output buffer
remove first N objects from output buffer
send removed objects to application
```

6. Experience and Evaluation

Considering scenarios where mobile devices are forced to piecewise retrieve result sets due to memory limitations of the device, extending the Nexus platform by cursors is clearly an enhancement. Without a cursor, such devices would have to send the same query multiple times to a Nexus node, receive the complete result set each time but only process the appropriate subset and ignore the rest, which is obviously inferior wrt. overall query processing time, data volume transferred and overall energy consumption. In order to assess the overhead involved with the cursor concept, we conducted a suite of experiments to show that the additional overhead caused by the cursor management and histogram calculations is comparatively small.

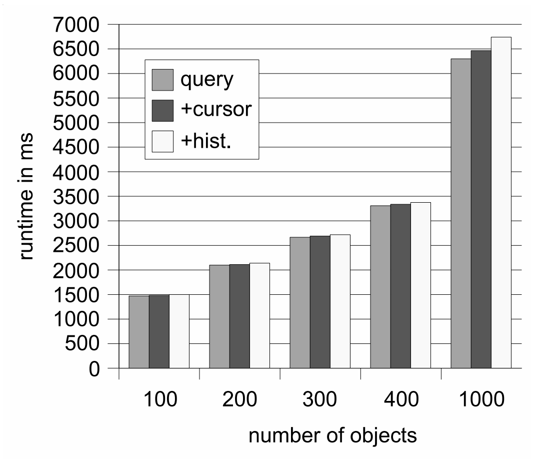

Figure 5. Run times for different result set sizes

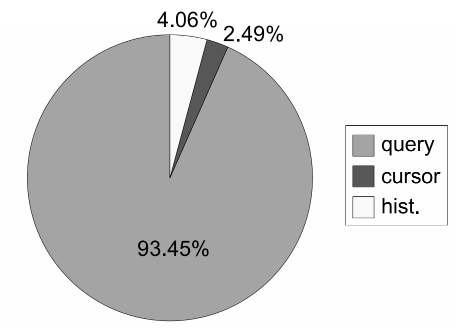

Figure 6. Runtime fractions for cursor management and histograms of 1000 objects query

The context server used for the experiments is implemented in Java and was running on a SUN Blade 2000 with two 1.2 GHz UltraSPARC III CPUs and 6GB of RAM. IBM DB2 8.1.3 was used as the backend system for storing the data. The database contained 3380 objects in total. Figure 5 shows the runtimes of a nearest-neighbor-query with sorting by distance from a reference point. We varied the number of objects to retrieve between 100 and 1000. *query* refers to query processing alone, *+cursor* additionally creates a cursor and *+hist.* furthermore computes a histogram. Figure 6 shows the fractions of the runtime required for processing the query, creating a cursor and computing a histogram for the 1000 objects query. The extra overhead is below 7%. This fraction is even lower for smaller result sets, approximately between 0.5% and 2%.

7. Conclusion

In this paper, we introduced a generic federated cursor concept, explained the underlying idea, and applied it to an integration architecture, the Nexus platform. The main idea is to request just those context servers that actively contribute and to process only necessary result data in order to reduce memory consumption and transmission volume over the network.

Multiple object representation is an additional problem to deal with since no information should be missed in order to achieve a lossless merge of multiple representations. It was solved by introducing the novel approach of cache histograms representing a query-dependent frequency distribution of the resulting objects based on some sorting criterion.

Finally, temporarily disconnected applications due to faulty mobile connections are supported. This problem was solved using sessions. In this way applications are able to reconnect at later time and at arbitrary connection points.

Our prototype evaluation and measurements indicated that the overhead introduced by the cursor concept is in the low percentage range. This is clearly acceptable in view of the benefits the cursor concept introduces to the applications and the federation layer, e.g. availability, disconnection and partial evaluation.

However, the approach suffers from assumptions that may not always be correct. First, it assumes that stored values of objects are exact. That implies that the ordering is always the same for each object and the corresponding bucket, but that is not always correct. For example, if the objects are sorted by distance to the application's reference point there could be some divergence between the position values or between the calculated distances to that reference point. The introduction of an interval could eliminate the error and minimize communication overhead between federation and context servers. Such an interval can be calculated on statistical values the federation (or some other component) has to collect in advance. Here the problem consists of dynamically finding convenient ranges for each bucket.

References

Date, C. J. (2000). *An Introduction to Database Systems*. Addison Wesley Longman, 17th edition.

Dey, A. and Abowd, G. (1999). Towards a better understanding of context and context-awareness. Technical Report GIT-GVU-99-22, Georgia Tech GVU.

García-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J. (1997). The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132.

Grossmann, M., Bauer, M., Hönle, N., Käppeler, U.-P., Nicklas, D., and Schwarz, T. (2005). Efficiently managing context information for large-scale scenarios. In *3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005), 8-12 March 2005, Kauai Island, HI, USA*, pages 331–340. IEEE Computer Society.

Haas, L. M., Kossmann, D., and Ursu, I. (1999). Loading a cache with query results. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 351–362, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Josifovski, V., Schwarz, P., Haas, L., and Lin, E. (2002). Garlic: a new flavor of federated query processing for DB2. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 524–532, New York, NY, USA. ACM Press.

Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996). Querying heterogeneous information sources using source descriptions. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 251–262, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Levy, A. Y., Srivastava, D., and Kirk, T. (1995). Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems - Special Issue on Networked Information Discovery and Retrieval*, 5(2):121–143.

Nicklas, D. and Mitschang, B. (2004). On building location aware applications using an open platform based on the NEXUS augmented world model. *Software and System Modeling*, 3:303–313.

Papakonstantinou, Y., García-Molina, H., and Ullman, J. (1996). MedMaker: A mediation system based on declarative specifications. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 132–141. IEEE Computer Society.

Tomasic, A., Raschid, L., and Valduriez, P. (1996). Scaling heterogeneous databases and the design of Disco. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 449–457, Washington, DC, USA. IEEE Computer Society.

Volz, S. (2006). An iterative approach for matching multiple representations of street data. In *Proc. of the JOINT ISPRS Workshop on Multiple Representations and Interoperability of Spatial Data*, volume XXXVI Part 2/W40.

Volz, S. and Walter, V. (2004). Linking different geospatial databases by explicit relations. In *Proceedings of the XXth Conference of ISPRS '04*.