liquid: Context-Aware Distributed Queries

Jeffrey Heer, Alan Newberger, Christopher Beckmann, and Jason I. Hong

Group for User Interface Research, Computer Science Division
University of California, Berkeley
{jheer,alann,beckmann,jasonh}@cs.berkeley.edu

Abstract. As low-level architectural support for context-aware computing matures, we are ready to explore more general and powerful means of accessing context data. Information required by a context-aware application may be partitioned by any number of physical, organizational, or privacy boundaries. This suggests the need for mechanisms by which applications can issue context-sensitive queries without having to explicitly manage the complex storage layout and access policies of the underlying data. To address this need, we have developed liquid, a prototype query service that supports distributed, continuous query processing of context data. This paper articulates the current need for such systems, describes the design of the liquid system, and presents both a room-awareness application and notification service demonstrating its functionality.

1 Introduction

One important aspect of the evolving ubiquitous computing vision is the development of context-aware computing [11], in which sensor networks and other data sources are leveraged to provide computing systems with an increased awareness of a user's physical and social environment. Sensed and inferred context data can then be exploited to provide enhanced computing services [5]. For example, consider a scenario in which a student busy working on a paper and in need of feedback may use a context-aware notification service to inform them when their advisor is in the same building and interruptible [10].

Until recently, most technical work on context-aware computing has focused on low-level architectural support [5,9,14], primarily concerned with the acquisition and storage of context data. For example, in Dey et al.'s Context Toolkit (CTK), context widgets are associated with sensors and used to collect and aggregate sensed data [5]. Client applications can then subscribe to these widgets over the network to access and monitor context data. Hong's Context Fabric (Confab) [9] provides the abstraction of information spaces (*infospaces*), logical storage units using the tuple space paradigm [6] that serve as context repositories for individual entities in an environment (e.g., people, locations, and objects). Similar to the CTK, Confab users can set up subscriptions to specific infospaces to monitor changes in context data (e.g., subscribing to a room infospace to monitor its current occupants).

While this work greatly supports the acquisition and organization of context, the means of monitoring context can be much improved. Unfortunately, the nature of context data provides some inherent challenges to achieving this. First, we expect context data to be distributed—organized by a number of logical and physical separations, including organizational and privacy boundaries. Second, context is dynamic—objects are moved around, people walk in and out of rooms, temperatures fluctuate, and activities begin and end. This requires systems capable of dynamically monitoring and re-routing connections as necessary.

While many context-aware architectures touch on these issues, providing distributed storage and subscription services for context data, they still place an unnecessary burden on context-aware application developers. Using current architectures, the previous notification scenario would require an application to manage a host of subscriptions to remote repositories, maintain multiple network connections, and perform all intermediate data processing. Duplicating this level of work for all desired context-aware applications is simply unacceptable. Those systems that do attempt to simplify context access, namely the CTK's Situation Abstraction [5], do so at the cost of centralizing all related context data, limiting scalability. What is needed are general, adaptive, and decentralized infrastructure services that both scale to wide-level deployment and simplify application design. As a result, we believe a distributed query service for context data is the logical next step for context-aware infrastructures.

Many of the pieces necessary for creating such a service have been well investigated in the database literature. Distributed databases (e.g., [15,16]) support the separation of data across multiple nodes in a network. Recent work in streaming databases [2,3,4,13] breaks the relatively static model of traditional systems, viewing data as a (possibly infinite) sequence of items, and can thus monitor data that changes over time. Finally, semi-structured databases (e.g., [12]) illustrate techniques for dealing with data of varying and dynamic structure. Although each of these individual areas have been well researched, the needs of context-aware systems lie in their intersection, which unfortunately has yet to be fully realized in the database community.

To address these needs, we built liquid, a query service that supports context-aware applications. Building on emerging work in the field of streaming databases, liquid supports distributed, decentralized query processing over continuously changing context data, stringing queries across various context repositories, re-routing queries as changing context dictates, and providing streaming results back to query issuers. liquid not only collects results from distributed context repositories, but distributes the actual *processing* of the query across the network, decentralizing query processing and improving scalability. As such, liquid helps to fill a current void in the design space of both database systems and context-aware computing infrastructures.

2 liquid Design and Implementation

The liquid query service is built atop an existing context-aware storage infrastructure, the Context Fabric (Confab) [9]. In this section we describe the design of Confab as relevant to liquid, and present the design of liquid itself.

2.1 Context Fabric

The Context Fabric is a distributed context-aware infrastructure with services to support the acquisition and retrieval of context data. Using Confab, people, places and things (*entities*) are assigned network-addressable logical storage units called Information Spaces (*infospaces*) that store context data. Sources of context data, such as sensor networks, can populate infospaces to make their data available for use and retrieval. Applications retrieve and manipulate infospace data to accomplish context-aware tasks. Infospaces provide an abstraction with which to model and control access to context data. Infospaces can be both distributed across a network or managed on a centralized server. We expect some combination of both to be typical of a Confab deployment. Confab is implemented in Java, using HTTP for network communication.

The basic unit of storage in an infospace is the *context tuple* (tuple). Elements of interest common to all tuples are *type*, a textual name describing the relationship of a tuple to the containing infospace's entity; one or more *values*, each identified by name; and an optional *entity-link* denoting the address of an infospace for an entity described by the tuple. Infospaces can store tuples containing arbitrary data, many of which may describe other entities related to the original infospace. Such tuples' entity-link elements refer to the infospace of the other entity. For instance, the infospace for a specific room may contain numerous tuples of type 'occupant', each with values denoting a name and email of an occupant of the room and an entity-link referring to the infospace that hold tuples on behalf of that occupant. Context tuples are represented externally as XML documents, and are the basic data unit processed by liquid queries.

2.2 liquid

liquid is a query processing service, implemented in Java, that deploys on top of the Context Fabric. The processing of a liquid query involves receipt of the query over the network, translation of the received query into a query plan, execution of that plan (including, if necessary, the dispatch of sub-queries to remote query processing units), and the return of any results to the query issuer. Both queries and query results are represented in XML and communicated over HTTP.

A core concept of liquid is the entity type path, which is illustrated in Figure 1. A type path specifies a tuple to be retrieved at the end of a sequence of infospaces, each infospace being specified by its type relation to the infospace prior in the sequence. For example, to get the occupants of the room you're currently in, the entity type path of this data, relative to your own personal infospace, would be "location.occupant". This scheme, similar in spirit to type

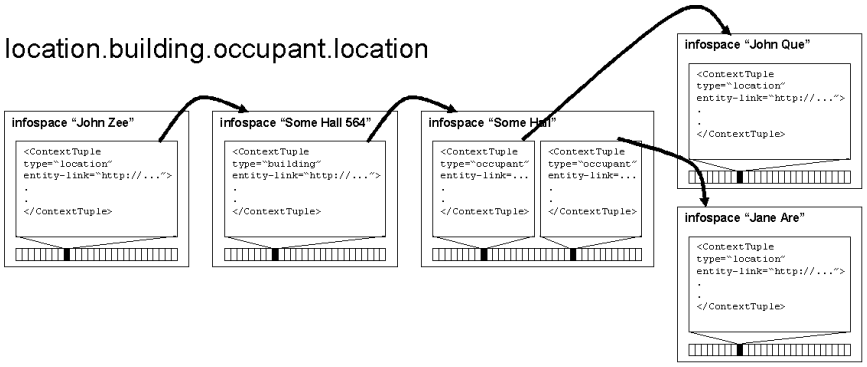

Fig. 1. The evaluation of an entity type path relative to a root infospace. This path returns the specific locations of people in the same building as the query issuer.

paths used in semi-structured databases like [12], specifies data across a logically separate and possibly distributed set of storage repositories.

liquid queries are issued in a declarative format (currently in XML) which specifies the tuples to be retrieved as a set of type paths. This format organizes the query into partitions that can be serviced at individual infospaces. For instance, a query retrieval path of "location.occupant.age", designed to retrieve the ages of people in the same room as the query issuer, would be split into 3 partitions: one retrieving "age" tuples, which is contained within another partition retrieving "occupant.age" tuples, which is in turn contained in a partition retrieving "location.occupant.age" tuples. Each partition can also contain conditions that tuples in the retrieval path must satisfy. When queries are received by a liquid node, the system determines which partition must be executed locally, and uses it to generate a *query plan*. A query plan is a structured tree of *operators* that perform query processing and partition forwarding to remote infospaces. liquid operators use the standard iterator model of query processing. Interested readers are referred to [7] for an introduction to these concepts.

Query execution consists of the management and evaluation of the generated query plans. Queries are first registered with the *query execution manager*, which holds a registry of live queries and oversees query scheduling. Upon receiving a query, the execution manager associates the query with its own *executor*, an object which oversees the evaluation of a single query, requesting new results from the query operator tree and returning them to the query issuer. The executor returns results to an issuer through a callback provided by liquid's network communication layer. Our current implementation uses a single-thread of execution per query, which the execution manager oversees using a thread pool.

The basic unit of query processing passed between operators is the *result item*. The result item is a time-stamped collection of context tuples, each indexed by their entity type path. Result items also maintain a status flag that indicates

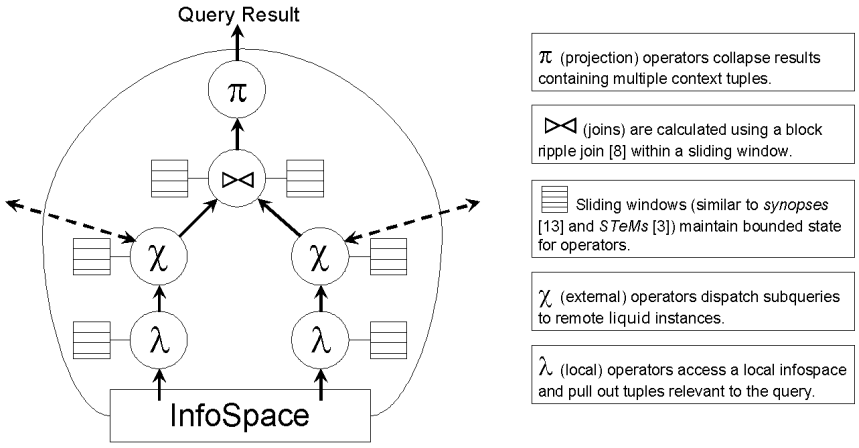

Fig. 2. An example liquid query operator tree. Dotted arrows indicate remote queries.

if an item has been *inserted*, *deleted*, *updated*, *exited*, or *expired*. *Inserted* or *deleted* items have either been added or removed from the scope of the query (e.g., a person walks into a monitored room, and then walks out). *Updated* and *exited* items are those whose content has been updated, either in such a way that the item still meets the query conditions (*updated*) or that it no longer does (*exited*). Finally, *expired* items are those that the system can no longer track due to bounded resources (see the discussion on windows below).

2.3 Operators and Windows

Currently, liquid supports the traditional database operators of selection (σ), projection (π), and joins ($\bowtie$). We also introduce two new operators specific to liquid: local (λ) operators, which are responsible for interfacing to a local infospace, and external (χ) operators, which are responsible for managing and dispatching remote sub-queries to other liquid nodes. The operator tree also features windows, caches which keep a history of tuples while maintaining bounded operator state. Figure 2 shows an example of an actual liquid operator tree.

Windows provide operator state within bounded resources, akin to synopses in [13] and state modules (SteMs) in [2]. liquid windows support result item insertion and removal, item searching, and automatic item eviction. Windows issue callbacks to their associated operators to inform them of eviction events. This functionality is used to keep query issuers apprised of which results are being actively monitored and which have *expired*.

liquid employs a number of operators to service queries. Local (λ) operators are responsible for retrieving context data from underlying infospaces. The λ operator monitors the local infospace for tuple insertions, deletions, and updates. A λ operator also maintains a window to keep a history of retrieved tuples,

primarily to handle *exited* tuples. Items evicted from the window are flagged as *expired* and added to the queue of items to be passed up the operator tree.

External (χ) operators retrieve data from remote infospaces by querying other liquid nodes. Result items are pulled from a child operator, and the entity-links of these items are used to issue queries to liquid instances at other infospaces. The query results are then streamed back, enqueued into the χ operator through a callback from the network layer. The operator keeps a registry of running queries, indexed by the result item that spawned it. *Updated* items arriving from the child operator may cause running queries to be canceled and then reissued as necessary. *Deleted*, *exited*, and *expired* items cause the corresponding queries to be canceled. Accordingly, χ operators maintain their own window, allowing the results from canceled queries to be effectively "recalled" by passing the proper *deleted*, *exited*, or *expired* items up the tree.

Selection (σ) operators filter items based on a given condition, while projection operators (π) collapse results items by removing unnecessary context tuples. These operators are stateless and hence quite easy to implement. Join ($\bowtie$) operators evaluate two incoming streams and merge the contents of two result items if they meet a specified condition. Each stream has an associated window, over which the join is computed using a variant of the block-ripple join [8]. Join operators have also been designed to carefully manage the cases when *deleted*, *exited*, and *expired* items must be propagated up the operator tree. This results in the desirable property that query issuers are guaranteed to be notified whenever a result item goes "stale".

3 Applications

As a first evaluation of the liquid system and its impact on context-aware application design, we've built a simulation system and two applications showcasing the system's functionality. We are in the process of setting up instrumented spaces in which to explore real-world context-aware scenarios, but in the meantime we have built a location simulator to support system testing and application design. The system provides a complete model of our building on our campus, and represents any number of people and objects located within the building. The system allows users to simulate movement of people and objects through a web-based interface (Figure 3), or can be programmed to automatically simulate activity. The simulator is a surrogate for an underlying sensing infrastructure, populating and updating infospaces as simulated context changes over time.

3.1 Application 1: Co-occupant Awareness

We built the first application to provide augmented awareness within a room or space. For a given person, the application provides the e-mails and webpages (if available and allowed by privacy policies) for all people currently in the same room as the query issuer. As people enter and exit the room, this result is automatically updated, and as the query issuer moves from place to place, the

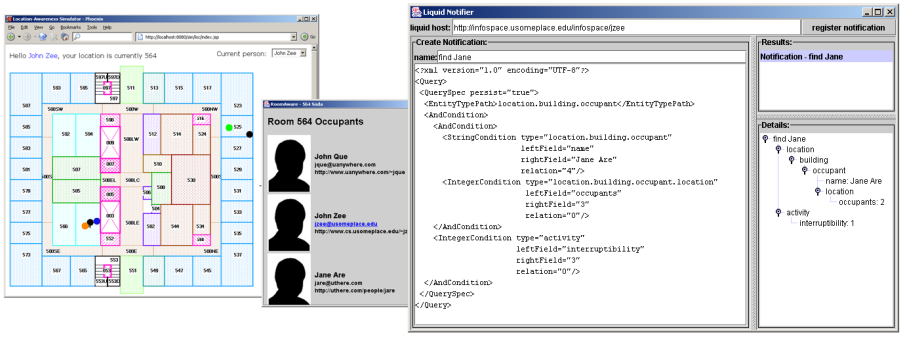

Fig. 3. Application screenshots of the location simulator (left), the room awareness application (center), and the notification service interface (right).

query is automatically re-routed to the correct infospace for the new location. This application showcases liquid's ability to both handle distributed data (i.e., across infospaces for both people and locations) and respond to the dynamic nature of context. The user interface for this application is shown in Figure 3.

3.2 Application 2: Notification Service

Next, we built a context-aware notification service, in the spirit of CybreMinder [5], that allows the user to specify conditions they would like to be notified of. This specification is performed using the interface in Figure 3, which currently shows a running query for the "advisor hunting" scenario mentioned in the introduction. The application issues user-provided queries and provides notification to the user as query results arrive. The processing of the notification conditions is completely provided by the liquid infrastructure (e.g., by computing joins), leaving the application developer responsible only for generating the appropriate queries and displaying the results. The evaluation of these queries is dispersed throughout the infrastructure, supporting scalability and reuse of computation through both caching and sub-plan sharing.

4 Future Work and Conclusion

Despite the success of our system, there is still much work to be done. First, our modeling scheme of context data is preliminary. We are in the process of determining the right balance between simplicity and expressive power in how context-aware architectures model the surrounding world. Next, a more natural query language is needed to further simplify the burden of application designers. Privacy issues with context data are also a serious concern. While access control is handled by the underlying Context Fabric, liquid will need to consider authentication and encryption in the evaluation of queries. Another major area

is optimization and evaluation, including query scheduling and parallelism, subplan sharing, window size adjustments [13], and dynamic operator optimization [1,2]. An important benefit of incorporating query processing into context-aware computing systems is the potential for improved efficiency of the infrastructure, allowing informed resource management and avoiding replicated work. It is our hope that by identifying these system needs, the ubicomp and database fields might inform each other, providing both new challenges and application domains for database researchers and new systems and techniques for the designers of ubiquitous computing environments.

In conclusion, this paper presented liquid, a continuous query processing engine for supporting context-aware applications. To this aim, liquid supports persistent, streaming queries over distributed data repositories, and dynamic query re-routing in response to changing context. Current context-aware computing architectures do not efficiently support these services, often relying on context-aware application designers to implement them on their own. liquid hopes to simplify and enhance the creation of context-aware applications by moving these services into the available infrastructure.

5 Acknowledgments

We are indebted to the guidance and insights of Anind Dey, Eric Brewer, James Landay, and Jennifer Mankoff. This work was supported by an NSF ITR for context-aware computing. The first author was supported by an NDSEG fellowship.

References

1. R. Avnur and J. M. Hellerstein. "Eddies: Continuously Adaptive Query Processing." In SIGMOD 2000.
2. S. Chandrasekaran, et al. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World." CIDR 2003.
3. J. Chen, D. DeWitt, F. Tian, and Y. Wang. "NiagaraCQ: A Scalable Continuous Query System for Internet Databases." In SIGMOD (2000).
4. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. "Scalable Distributed Stream Processing." CIDR 2003.
5. A. K. Dey. "Providing Architectural Support for Building Context-Aware Applications." Ph.D. thesis, December 2000, College of Computing, Georgia Institute of Technology.
6. D. Gelernter. "Generative communication in Linda." ACM Transactions on Programming Languages and Systems. 7(1), 1985.
7. G. Graefe. "Query Evaluation Techniques for Large Databases." ACM Computing Surveys 25 (2): 73-170, 1993.
8. P. J. Haas and J. M. Hellerstein. "Ripple joins for online aggregation." In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, SIGMOD 1999.
9. J. I. Hong. The Context Fabric. http://guir.berkeley.edu/cfabric.
10. S. E. Hudson, J. Fogarty, et al. "Predicting Human Interruptibility with Sensors: A Wizard of Oz Feasibility Study." CHI 2003.
11. Human-Computer Interaction Journal: Special Issue on Context-Aware Computing, Vol. 16, No. 2-4, 2001.
12. J. McHugh, et al. "Lore: A Database Management System for Semistructured Data." SIGMOD Record, 26(3):54-66, September 1997.
13. R. Motwani, et al. "Query Processing, Resource Management, and Approximation in a Data Stream Management System." CIDR 2003.
14. B. Schilit. "System architecture for context-aware mobile computing." Unpublished doctoral dissertation, Columbia University, 1995.
15. M. Stonebraker, et al. "Mariposa: a wide-area distributed database system." VLDB Journal 5, 1 (Jan. 1996), p. 48-63.
16. R. Williams, et al. "R*: An Overview of the Architecture." Technical Report RJ3325, IBM Research Lab, San Jose, CA, December 1981.