Grouping Techniques for Update Propagation in Intermittently Connected Databases*

Sameer Mahajan[+†] Michael Donahoo[+] Shamkant Navathe[+] Mostafa Ammar[+]
Sanjoy Malik[‡]

[+] College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA

[‡] Advanced Technology Development Center
Georgia Institute of Technology
Atlanta, GA 30332, USA

Abstract

We consider an environment where one or more servers carry databases that are of interest to a community of clients. The clients are only intermittently connected to the server for brief periods of time. Clients carry a part of the database for their own processing and accumulate local updates while disconnected. We call this the Intermittently Connected Database (ICDB) environment. ICDBs have a wide variety of applications including sales force automation, insurance claim processing, and mobile work forces. Our focus in the paper is on the problem of update propagation at the server in ICDBs and the associated processing at the clients. The typical client-centric approach involves the communication and processing of updates and transactions on a per-client basis, ignoring the overlap of data between clients. The complexity of this approach is on the order of the number of connecting clients, thereby limiting the scalability of the server. We propose a data-centric approach which clusters data into groups and assigns to each client one or more of these groups. The proposed scheme results in server processing complexity on the order of the number of groups, which we control. We propose various techniques for grouping and discuss the processing required at the clients to enable the grouping approach. While the client-centric approach is expected to significantly degrade with the increasing number of clients, we expect that a properly designed grouping scheme will sustain a number of clients that is significantly larger. A prototype has been developed and performance studies are currently in progress.

1 Introduction

The growing trend towards mobile computing has resulted in the adaptation of many traditional applications to intermittently connected [1, 2, 3, 4] environments, including replicated database views. Consider a database environment where a server database system shares portions of data with a set of intermittently connected clients. Since the connectivity is intermittent, the clients maintain a copy of the shared data in their local database. Updates made to this local database are logged and propagated to the server database when the client connects. The data shared between the server and some client, X, may also be shared with another client, Y; therefore, changes to that data at Client X should be reflected at Client Y. Since the clients are only intermittently connected and cannot directly send changes to other clients, the server acts as a conduit for updates by forwarding updates to data shared among multiple clients. In fact, the server is responsible for tracking client updates to shared data and batching those updates for dissemination to the other clients which share the data. Consider the relational example in Figure 1. The schema at the server consists of three relations (Students, Enroll, and Courses), portions of which need to be replicated to the types[1] of clients defined in Client Data. Note that a Professor shares with the Undergrads the undergraduate tuples from the Students relation. When Professor Jane updates the value of an undergraduate tuple, she records the update in a log. Upon server connection, Professor Jane sends her log of updates to the server. The server executes the undergraduate tuple change and batches the transaction for dissemination to all Undergrad clients. When an Undergrad client connects, it accesses the change log at the server and updates its local database.

Currently, ICDB systems treat each client individually, creating per-client update files, containing only the updates relevant to each particular client. That is, for each client, the server prepares a client-specific update file. We call this the "client-centric" approach because it aggregates database changes based on the data needed by each client. Unfortunately, the processing and sending of each client-specific file is expensive in terms of server processing and network bandwidth consumption; therefore, the server processing

*Supported by the Georgia Research Alliance under a Technology Development Investment Grant.

†Currently working at **Informix Software Inc.**, Portland, Oregon 97205.

[1]Each client type can represent a large number of actual clients. For example, there may be thousands of undergraduate students.

Server Schema

Students(StudID, Name, Phone#, Class*)
Enroll(StudID, CrsID, Grade)
Courses(CrsID, Name, Class*)

* Class $\in \{UNDER, GRAD\}$

Client Data

Client Types	Client Schema
Undergrads	Students, Enroll, and Courses with only undergraduate data minus the grades from Enroll and Phone# from Students
Grad Students	Students, Enroll, and Courses with only graduate student data minus the grades from Enroll and Phone# from Students
Professors	Students, Enroll, and Courses (All data)
Registrar	Enroll minus Grades

Figure 1: University Database Example

load is on the order of the number of clients. That is, the server incurs additional cost for **each and every** client, so the number of clients able to be served is limited.

We propose exploiting the overlap of data shared between various clients to increase the scalability of the server. We accomplish this with data-centric, rather than client-centric, processing by grouping data according to how it is shared between clients. Reconsider the University Database example of Figure 1. Note that much of the data is shared between several clients. We can leverage this sharing knowledge by generating update files based not on what individual clients need but on what subsets of the data are common to groups of clients. For example, the undergraduate tuples of the Students relation are shared by all Undergrad and Professor clients. In the data-centric approach, we consider grouping all of this data together and processing changes only once, unlike the client-centric approach which would require processing a change for every Undergrad and Professor client. Figure 2 provides one possible grouping for the University Database[2]. This grouping defines only five groups to service all of the Professor, Undergrad, Grad Students, and Registrar clients. In data-centric processing, the server need only create an update file per data group. Unlike the client-centric approach which builds an update file for each client, the data-centric approach builds update files for data groupings and requires the clients to merge the correct set of update files to retrieve the needed updates. For example, the Undergrad clients must access the Under-Students, All-Courses, and All-Enroll update files and merge those files together to get all relevant changes. Note that the Undergrad clients, in fact, receive superfluous updates for graduate courses and enrollment, as well as grades for undergraduate enrollment; therefore, Undergrad clients are responsible for filtering out unwanted updates. With our data-centric approach, the server must maintain only five update files, irrespective of the number of clients. The data-centric approach reduces the complexity of update file maintenance from on the order of the number of clients to on the order of the number of groups, thereby significantly increasing the scalability of server processing.

Groups

Group	Group Dataset
Under-Students	Students where Class = UNDER minus Phone#
Grad-Students	Students where Class = GRAD minus Phone#
Students-Phone	Students(StudID, Phone#)
All-Courses	Courses (all data)
All-Enroll	Enroll (all data)

Client Group Assignments

Client Type	Groups Joined
Undergrads	Under-Students All-Courses All-Enroll
Grad Students	Grad-Students All-Courses All-Enroll
Professors	Grad-Students Under-Students Students-Phone All-Courses All-Enroll
Registrar	All-Enroll

Figure 2: Group Assignment for University Database

We have achieved a considerable increase in server processing scalability by changing to the data-centric approach. Recall that we are also concerned with the scalability of network bandwidth utilization. Under the client-centric approach, a similar type of unnecessary duplication of effort occurs in the network because the same data, contained in different update files, is being transmitted by the server to different clients. For example, if two professors connect at the same time, both will be sent the same data, and since each professor has his own update file, the same data will be sent twice on the network. We propose reducing network duplication by utilizing multicast[5] to send the same shared data once to a set of clients. In the University Database example, if multiple Professor clients attempt to simultaneously[3] access the

[2] This example merely demonstrates the concepts and is not meant to represent a grouping which minimizes server processing or network bandwidth.

[3] Discussion of how client request synchronicity is accomplished beyond the scope of this paper.

Under-Students update file, the server could send this file **once** using multicast, and the network would handle delivering a copy to each client. This concept of aggregating and multicasting data bound for individuals has been tried in the context of web pages[6] and broadcast databases[7]. In this paper, we focus exclusively on processing scalability; consequently, the issue of using multicast communication to further improve the performance of ICDBs is the subject of ongoing research.

Section 2 describes related work in this area. In Section 3, we address various grouping options and select one for study. Section 4 presents mechanisms for implementing a data-centric server. Section 5 describes the problems and our proposed solutions for filtering at the clients. Section 6 presents the problems of merging the group update files into a single update to ensure properly ordered operation execution at the clients. Section 7 provides an analysis of our approach, and Section 8 presents conclusions and future work.

2 Related Work

Effectively, we are performing view maintenance for intermittently connected data systems. View maintenance has been a long standing research problem in database systems. The incremental techniques and algorithms are classified and thoroughly surveyed in [8]. Most of the techniques [9, 10, 11, 12, 13, 14, 15, 16] make use of complete information (i.e. base relations as well as materialized views) for the maintenance. They are almost directly applicable for our server processing. The difference is that the intermittent connectivity forces us to refresh the views on-demand regardless of the query-issued times [17, 18], rather than immediately after each update [19, 20] or to defer view updates until issuing a related query to the view [21, 22]. Sometimes it is not possible to have all the information (e.g. in case of the intermittently connecting clients in the system under consideration). A view is not always maintainable for a modification by using only the view. Some work [11] concentrates on self-maintainable views, while other work [23] looks at how additional information in terms of some base relations along with the view can facilitate view maintenance. There are some other approaches [24, 25] that study propagation of updates on a materialized view to other materialized views. This work does not address the issue of multiple differential update files, as required by our approach.

Related work in aggregation of data for broadcast has been performed in many areas. This approach has been used to reduce bandwidth consumption of web servers[26, 6]. In [27], the authors analyzed data aggregation techniques for broadcasting data in Videotex systems. Our approach of dissemination of updates from server to the clients is similar to that in broadcast disks [28]. The motivation for this related work is to operate in asymmetric network environments where downstream (server to client) bandwidth is much higher than upstream bandwidth. We look at this approach mainly in terms of scalability provided at the server to support a higher number of clients. Broadcast disks also do not support the updates performed at the clients.

3 Data Grouping Techniques

We must determine the best method of deriving the data groups (called "Data Grouping") based on metrics such as server processing and network bandwidth consumption. After Data Grouping, the clients must determine which data groups they must join such that they get a minimal superset ("Client Set") of the changes they need. Data grouping can be achieved in different ways. We divide the grouping techniques into two categories, based on when the Data Grouping is derived: Dynamic and Static. In static grouping, the Data Grouping is determined *a priori* and remains the same irrespective of which clients are connected. In dynamic grouping, the number and makeup of the groups in the Data Grouping depends on constantly changing attributes such as the current set of transactions and/or connected clients. Within each category of grouping, we consider various approaches:

Static

One-Group-Set: The Data Grouping is a single group receiving **all** transactions. All Client Sets consist of this group.

Admin-Set: The Data Grouping and Client Sets are defined in advance by the administrator.

Dynamic

Transaction-Set: In this Data Grouping, a separate group exists for each transaction. A Client Set consists of the groups with applicable transactions.

Minimal-Set: This Data Grouping and the associated Client Sets contain the minimal number of groups which insure each transaction is sent at most once by the server and only to the applicable clients.

Dynamic grouping techniques help to reduce network bandwidth consumption and client filter processing by sending only data relevant to the current set of connected clients; however, these techniques require much more server processing because the server must dynamically calculate and coordinate groupings to handle any combination of clients and data. In the Minimal-Set case, the server must group updates for dissemination for each connecting set of clients. Since each group represents a set of transactions to be sent to a set of clients, the Minimal-Set is actually a subset of the Cartesian product of the transactions and power set of clients. This problem is related to the Exact Cover Set problem which is known to be NP-Complete; however, useful approximations exist[29]. A Transaction-Set server must communicate with each client individually to handle sending the transactions only to the relevant clients.

Static grouping significantly reduces server processing requirements by handling update dissemination independent of the connected clients; however, if only some of the clients connect, the server may transmit a much larger update file than needed. (The extreme is one client connecting in the One-Group-Set approach.) We believe the bottleneck in our data-centric system will be the server processing because determining to which data groups a transaction applies is an expensive operation and all of the processing is performed at the server. Consequently, we choose Static Grouping, and, as One-Group-Set is simply a special case of Admin-Set, we will focus on the Admin-Set approach in this paper.

4 Server Architecture

We assume the existence of a server which, given the sharing model and a set of transactions, can construct an update file for each client. The server processes the updates[30] received from the clients in some serial order[4] and creates an update file for every client, based on its knowledge of data sharing. This update file (called a "delta" file) for each client contains all the operations on the data shared with the client. We make the following assumptions about the delta files created by the server:

- A delta file consists of a sequence of transactions, each of which is identified by a globally unique sequence number indicating the order of transaction execution at the server.
- Each transaction consists of a sequence of atomic operations each bearing a unique sequence number indicating the order of operation execution within the transaction.
- Atomic operations apply only to a single fragment.
- Each transaction identifies the client on which it was originally executed.

These assumptions enable the following:

- Using the transaction and sequence numbers, we can reconstruct the order of operation execution at the server, despite fragmentation of operations in delta files.
- Since each atomic operation operates on a single fragment, there are no interrelation dependencies among these statements, except for foreign key dependencies.
- We can arrange the interrelational operations on relations with no foreign key dependencies in any order. Intrarelation operations must preserve the serial order at the server.
- Interrelational operations on relations with foreign key dependencies must be ordered as follows (Assume that the foreign key is from a relation called the "parent relation" and the primary key is from a relation called the "child relation."):
  - An insert into the parent relation should precede the corresponding insert into the child relation.
  - A delete in child relation should precede the corresponding delete in the parent relation.

[4]Details of the actual processing performed at the server are beyond the scope of this paper. We assume that the server executes these updates in a correct order.

We assume that the server operates under these assumptions, which is exactly suited for the client-centric approach, to realize our data-centric approach. To accomplish this, for each data group we simply describe to the server a "fictitious client" whose dataset matches the dataset for the group. The server does not distinguish fictitious from real clients; instead, it simply generates update files in the same manner for either client type. In the example of Figure 2, a fictitious client would be created for the Under-Students group. From the server's perspective, an actual client wanting only the undergraduate tuples from the Students relation exists. Note that by using this idea of fictitious clients, we can have a server which can simultaneously provide client-centric and data-centric update files; consequently, if we have a specific client, X, that needs updating more often than the other clients, we can keep client X as client-centric and implement the remaining clients as data-centric. When we wish to update client X, we can just generate an update for it specifically, instead of generating a set of group files which every client must access.

5 Client Filtering

Recall that under our data-centric approach, clients may be receiving a superset of the data they share; therefore, we must provide a mechanism for discarding superfluous data. In the example of Figure 2, Undergrad clients receive the Grades data from the All-Enroll group which must be filtered. We consider the following filtering issues and propose appropriate solutions:

Reflection Filtering: Since the server now processes delta files for groups rather than individual clients, the delta files received by a client may contain transactions originally submitted by that client. The client needs to identify and filter such "reflected" transactions. To carry out reflection filtering on a client, we simply go through the delta files, received on various groups in which the client participates, and remove those transactions whose creator ID matches the client's ID.

Schema Mapping: Clients must apply synonym conversions for relation and attribute names. If an undergrad user wishes to rename the Students relation to UnderGrads in her local database, the client filtering must convert references to Students into UnderGrads references.

In general, schema mapping can get very complex and work has been done to address the related problems[31, 32]; therefore, in this paper, we do not discuss how the schema mapping or translation can be performed. We only assume that it is performed at the respective clients. This saves some processing on the server, and it also makes it easier to handle the conflicting schemas of individual clients within a group. Currently, we plan to implement only the simple "name equivalence" schema mappings.

Intrarelation Filtering : Superfluous tuple data may be sent to the client, which should be filtered out. Filtering from a relation based on data internal to the relation is termed Intrarelation Filtering. In our University example, undergraduate clients receive Grade data from the Enroll relation which they are not authorized to view[5]. Eliminating the Grade information requires dropping data corresponding to a particular attribute. Undergraduate clients also receive all of the data from the Courses relation; however, they are only authorized to view the undergraduate course information. Filtering graduate student courses from undergrad clients requires dropping tuples with attribute Class = GRAD.

The relation, the attribute (for vertical filtering), and the values of the tuples (for horizontal filtering) that appear in the individual operations need to be analyzed for the elimination. We retain only those operations which:

- operate on relations whose equivalent (as determined by schema mapping) relations exist in the local schema of the client, and
- update the attributes whose equivalent attributes exist in the client schema, and
- operate on the value ranges of these attributes as they exist in the local schema.

Interrelation Filtering: Not all superfluous data can be filtered by data within a relation. Data in a server relation may be shared with a client based on data in another relation. For example, undergraduate clients receive only the Enroll relation data for undergraduates; however, to determine if an Enroll tuple is for an undergraduate, the classification of the course must be determined using the Courses relation. In our grouping example, an undergraduate client would receive **all** of the Enroll relation tuples in the All-Enroll group. Undergraduate clients must use the Courses relation to determine which Enroll tuples should be filtered out.

Clients with relations whose sharing is determine by other relations must ensure that they have the necessary data to perform the filtering. We consider the following options for replicating the necessary data to the clients:

Parent Requirement: The clients could guarantee that the sharing of any child relation implies the sharing of the necessary data from the parent relation(s), thus ensuring the existence of the necessary data. For example, including the Enroll relation at a client would require inclusion of the Courses relation.

Replication Key Denormalization: Another option is to eliminate the need for Interrelational Filtering by denormalizing every relation so that it includes the key from the parent relations. For example, the All-Enroll group contains the Enroll relation which is a child of the Courses relation; therefore, the server need only append each Enroll tuple with the Class value from the Courses relation. The resulting relation in the All-Enroll group is derived by

$$\pi_{StudID,CrsID,Grade,Class}(Enroll \bowtie Courses)$$

We take the Parent Requirement approach as it keeps the additional server processing to the minimum. Additional client data required by the Parent Requirement approach can be hidden using local DBMS security and views.

[5]Dealing with security issues such as preventing clients from unauthorized data viewing is beyond the scope of this paper.

6 Client Merging

Our data-centric approach requires that clients receive an update file from each group; however, to ensure the integrity of the client data, these transactions and operations within the update files must be properly ordered. In fact, clients must merge the update files together into a single update file to execute on the local client's database. We consider the necessary merging requirements for the clients and present a merge algorithm based on the output of the server specified in Section 4.

6.1 Merging Requirements

Duplicate Discarding: Some groupings may result in multiple groups with duplicate data, requiring clients to detect and discard duplication. In our example, Professor clients join the Grad-Students, Under-Students, and Students-Phone groups. Consider a transaction deleting a student (Assume student 1000 is an undergrad.):

```
DELETE FROM Students WHERE StudID = 1000;
```

This transaction is sent to Professor clients **two** times (Students-Phone and Under-Students groups). Professors must detect duplication and carry out a single deletion against the local data.

Operation Merging: Operations received from multiple update files may be broken up when placed in groups. Clients must ensure that these operations are merged such that integrity is maintained. Consider the following sequence of operations at the server,

```
INSERT (1000, J. Smith, 555-5555, UNDER)
INTO Students;
DELETE FROM Students WHERE StudID = 1000;
```

This sequence of operations would be broken up into the following sequences for groups Under-Students and Students-Phone defined in Figure 2, respectively:

```
INSERT (1000, J. Smith, UNDER)
INTO Students;
DELETE FROM Students WHERE StudID = 1000;
```

and

```
INSERT (1000, 555-5555) INTO Students;
DELETE FROM Students WHERE StudID = 1000;
```

A professor client must merge the transactions from the Under-Students and Students-Phone such that 1) the inserts are merged into a single insert, 2) duplicate deletes are eliminated and 3) the merged insert is executed before the delete.

Referential Integrity Ordering: When merging transactions, clients must be aware of referential integrity constraints. Executing transactions out of order at the client may result in rejection of operations at the client which were allowed at the server, leading to degradation of client data integrity. Consider two inserts executed at the server:

```
INSERT (6555, Queuing Theory, GRAD)
INTO Courses;
```

followed by,

```
INSERT (1001, 6555) INTO Enroll;
```

These two inserts would be received by graduate student clients from the groups All-Courses and All-Enroll, respectively. Observe that Courses.CrsID is a foreign key of Enroll.CrsID; consequently, the order of the inserts must be preserved, specifically the tuple for the Courses relation must be inserted prior to the Enroll tuple insertion. In general some ordering needs to be preserved among operations on different relations that are related by a referential integrity constraint.

6.2 Merging Algorithm

We present an algorithm for merging a set of data-centric update files from the server into a single file which can be executed against a client's local database. Our goal is to merge the files into a single update file such that the merged file's execution produces the same results as produced when the transactions were originally applied to the server. Recall that the transactions of the delta files are all uniquely identified by a global sequence number, and the operations within each transaction are uniquely identified by a transaction-scoped sequence number. Consequently, each operation can be uniquely identified and ordered by a (Transaction ID, Operation ID) pair. Consider merging two delta files D_1 and D_2. Since the delta files are ordered by Transaction ID (primarily) and Operation ID (secondarily), we can use a sorted merge algorithm to join the two files in the proper execution order. Note that some (Transaction ID, Operation ID) pairs may appear in both delta files. This results from the issues discussed in the descriptions of Duplicate Discarding and Operation Merging previously. For duplicates, we simply discard all but one copy of the operation. For operation merging, we create a merged operation containing the data from all operations with the same (Transaction ID, Operation ID) pair values.

Consider the example update file[6] from a client of Figure 3 based on the example of Figure 2. When the server processes these transactions, it generates the updates files in Figure 4 for the Under-Students and Students-Phone groups. In merging Under-Students and Students-Phone, a Professor client recognizes that operation 1 of transaction 1 is duplicated and keeps only one copy. The professor client then merges operation 1 of transaction 2 into a single operation. Finally, the professor eliminates the duplicated operation 2 of transaction 2. The resulting delta file is the same as the original transaction set.

```
Begin T1
   Op1: DELETE FROM Students
        WHERE StudID = 1000;
End T1

Begin T2
   Op1: INSERT (1000, J. Smith, 555-5555,
        UNDER) INTO Students;
   Op2: DELETE FROM Students
        WHERE StudID = 1000;
End T2
```

Figure 3: Example Transaction Set

The elimination of duplicates and merging of split operation is easily handled by the merging process. In addition, since the order of the original delta file preserves the referential integrity and the merge restores the original order, the merge process satisfies the referential integrity ordering requirement. Note that we use sequence numbers to completely restore the server transaction execution ordering. Consider two transactions one of which updated the Students relation and the other the Courses relation. These transactions can be executed in any order at the client. We reestablish the server order because it does not require us to determine if an ordering is correct. We simply reuse the server's order of execution which is assumed to be correct.

[6] We assume student 1000 to be an undergrad.

Under-Students:

```
Begin T1
   Op1: DELETE FROM Students
        WHERE StudID = 1000;
End T1

Begin T2
   Op1: INSERT (1000, J. Smith, UNDER)
        INTO Students;
   Op2: DELETE FROM Students
        WHERE StudID = 1000;
End T2
```

Students-Phone:

```
Begin T1
   Op1: DELETE FROM Students
        WHERE StudID = 1000;
End T1

Begin T2
   Op1: INSERT (1000, 555-5555)
        INTO Students;
   Op2: DELETE FROM Students
        WHERE StudID = 1000;
End T2
```

Figure 4: Figure 3 Transaction Set Update Files

7 Grouping Performance

We present a simple model for analyzing the server resource (processing bandwidth and disk space) gains from data-centric grouping for update propagation. A complete picture requires consideration of the trade-offs between server, client, and network resources, which is the subject of ongoing research. Note that since client-centric grouping can really be thought of as a special case of data-centric grouping, the performance of the data-centric approach need be no worse than client-centric grouping. For our analysis, consider a model of n clients each of which provides, on average, m operations to the server for replication. Also, assume that the operations from a particular client must be replicated by the server, on average, to half the other clients and that the data-centric approach for an application requires, on average, k groups. Utilizing this model, we look at server resource consumption with respect to the number of clients.

With n clients and m operations per client, the server is required to handle $m \cdot n$ operations in the incoming update files for all clients. Each operation must be analyzed for each delta file. In the traditional client-centric approach, there is a delta file per client, requiring $m \cdot n^2$ steps for replication. The data-centric approach has only k delta files, thereby requiring only $m \cdot n \cdot k$ steps for replication. This means increasing the number of clients for the client-centric approach requires a polynomial increase in server processing but only a linear increase for the data-centric approach. Note that the data-centric approach performs better than client-centric when $k < n$. Since we typically choose a number of groups such that $k << n$, the performance gain can be substantial.

With the operations of each client affecting, on average, $\frac{n}{2}$ of the n clients, each client generates $\frac{n}{2} \cdot m$ replication operations. In the client-centric approach, the server maintains an exchange file per client; therefore, the total storage space required on the server is $n \cdot \frac{n}{2} \cdot m$. The data-centric grouping approach maintains the exchange files on per group basis; therefore, the storage requirement on the server is roughly $k \cdot \frac{n}{2} \cdot m$. Thus the data-centric approach again provides linear as opposed to polynomial storage requirement with respect to the number of clients.

8 Conclusions and Future Work

We propose a novel approach of grouping to address scalability of update propagation in Intermittently Connected Databases (ICDBs). We observed that grouping provides a viable mechanism for scaling server processing because it ships some of the server processing to individual clients. An efficient approach for client adaptation is developed in the paper. We provide a complete preliminary analysis of the issues of client processing in case of the data grouping and demonstrate how data-centric processing may be accomplished at the server and clients.

In this paper, we do not address the tradeoffs of network and client processing. These issues need to be formulated and investigated further. The problem of how to form groups for optimal performance is challenging. Additional techniques of grouping can also be explored to see their impact, and the details of the solution with the multiple servers may also be interesting. A combination of grouping and client-centric approach is worth investigating under the scenario of intermittently connected databases that remain disconnected while the information is being disseminated by the server. We are currently developing a testbed to experimentally investigate some of these issues.

References

[1] J. J. Kistler, *Disconnected Operation in a Distributed File System.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1993. CMU-CS-93-156.

[2] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the CODA file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 3–25, Feb. 1992.

[3] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, "Exploiting weak connectivity for mobile file access," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, (Copper Mountain Resort, CO), Dec. 1995.

[4] M. Satyanarayanan, J. Kistler, L. Mummert, M. Ebling, P. Kumar, and Q. Lu, "Experience with disconnected operation in a mobile computing environment," in *Proceedings of the USENIX*

Symposium on Mobile and Location-Independent Computing, (Boston, MA), June 1993.

[5] S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, pp. 85–110, May 1990.

[6] R. J. Clark and M. H. Ammar, "Providing scalable Web services using multicast," in *Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments (SDNE 95)*, June 1995. To appear in Computer Networks and ISDN systems.

[7] G. E. Herman, G. Gopal, K. C. Lee, and A. Weinrib, "The datacycle architecture for very high throughput database systems," in *SIGMOD 1987*, pp. 97–103, Dec. 1987.

[8] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques and applications," *IEEE Data Engineering Bulletin*, vol. 18, pp. 3–18, June 1995.

[9] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," *VLDB*, 1991.

[10] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," *ACM SIGMOD*, 1995.

[11] A. Gupta, H. V. Jagdish, and I. S. Mumick, "Data integration using self-maintainable views," *Technical Memorandum 113880-941101-32*, Nov. 1994.

[12] A. Gupta, I. S. Mumick, and V. Subrahmaniam, "Maintaining views incrementally," *ACM SIGMOD*, pp. 157–167, 1993.

[13] J. V. Harrison and S. Dietrich, "Maintenance of materialized views in a deductive database: An update propagation approach," *Workshop on Deductive Databases, JICSLP*, 1992.

[14] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering queries using views," *PODS*, pp. 95–104, 1995.

[15] B. Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and N. Sudkamp, "Sql/xnf - processing composite objects as abstractions over relational data," *IEEE Data Engineering Conference*, 1993.

[16] X. Qian and G. Weiderhold, "Incremental recomputation of active relational expressions," *IEEE TKDE*, vol. 3, pp. 337–341, 1991.

[17] M. E. Adiba and B. G. Lindsay, "Database snapshots," *VLDB*, pp. 88–91, Oct. 1980.

[18] B. G. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, "A snapshot differential refresh algorithm," *ACM SIGMOD*, pp. 53–60, June 1986.

[19] J. A. Blakeley, P. Larson, and F. W. Tompa, "Efficiently updating materialized views," *ACM SIGMOD*, pp. 61–71, May 1986.

[20] S. Horowitz and T. Teitelbaum, "Generating editing environment based on relations and attributes," *ACM Trns. Programming Languages Syst.*, vol. 8, Oct. 1986.

[21] E. R. Hanson, "A performance analysis of view materialization strategies," *ACM SIGMOD*, pp. 440–453, May 1987.

[22] N. Roussopoulos and H. Kang, "Principles and techniques in the design of ADMS," *IEEE Computer*, Dec. 1986.

[23] H. V. Jagdish, I. S. Mumick, and A. Silberschatz, "View maintenance issues in the chronicle data model," *PODS*, pp. 113–124, 1995.

[24] A. Segev and W. Fang, "Currency-based updates to distributed materialized views," *ICDE*, pp. 512–520, 1990.

[25] A. Segev and J. Park, "Updating distributed materialized views," *TKDE*, vol. 1, pp. 173–184, June 1989.

[26] K. Almeroth and M. Ammar, "Scalable delivery of web pages using cyclic best-effort (UDP) multicast." To appear in Infocom 98, 1998.

[27] J. W. Wong and M. H. Ammar, "Analysis of broadcast delivery in the Videotex system," *IEEE Transactions on Computers*, vol. 34, pp. 863–866, Sept. 1985.

[28] S. Acharya, M. Franklin, and S. Zdonik, "Disseminating updates on broadcast disks," in *VLDB*, (Bombay, India), Sept. 1996.

[29] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[30] V. R. Narasayya, "Distributed transactions in a mobile computing system," in *IEEE Workshop on Mobile Computing, Systems and Applications*, (Santa Cruz, CA, USA), Dec. 1994.

[31] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," in *ACM Computing Surveys*, Dec. 1986.

[32] S. Navathe, S. Mahajan, and E. Omiecinski, "Rule based database integration in HIPED: Heterogeneous intelligent processing in engineering design," in *Proceedings of International Symposium on Cooperative Database Systems for Advanced Applications*, (Kyoto, Japan), Dec. 1996.