Toward A Method of Grouping Server Data Fragments for Improving Scalability in Intermittently Synchronized Databases

Wai Gen Yee*, Michael J. Donahoo†, Shamkant B. Navathe*

Abstract

We consider the class of mobile computing applications with periodically connected clients. These clients wish to share data; however, due to the expense of mobile communication, they only connect periodically – and not necessarily synchronously – to a common network. Traditionally, a continuously-connected server, containing an aggregate of client data, facilitates sharing amongst clients by allowing the clients to upload local updates and download updates submitted by other clients. The server computes and transmits these updates on a client-by-client basis; consequently, the complexity of these operations is on the order of the number of clients, limiting scalability. Recent research proposes exploiting client data overlap by grouping updates according to how the data is shared amongst clients (data-centric) instead of on a client-by-client basis (client-centric). Each client downloads updates for the relevant set of groups. By grouping, update operation distribution is computed only once per group, irrespective of the number of clients downloading a particular group's updates. Additionally, we may gain bandwidth scalability by employing broadcast delivery since, unlike the case in the per-client approach, multiple clients may be interested in a group's updates. Clearly, group composition directly affects the scalability of this approach. Given a relative cost of resources such as server processing, bandwidth, and storage space, we focus on developing a group derivation approach that significantly improves the scalability of the resources. We construct a formal specification of this problem and discuss the intractability of an optimal solution. Based on observations from the specification, we derive a heuristically based approach and evaluate its efficacy with respect to the client-centric approach. We run experiments on an implemented system that demonstrates that as the amount of overlap increases between client subscriptions, the data-centric approach with groups generated by our heuristic-based algorithm yields significant cost reduction when compared to the traditional client-centric approach.

1 Introduction

Advances in portable computing technologies are enabling information processing to reach new frontiers. Data access is no longer limited to those in the office. Instead, enabled by portable computing devices, the mobile office is an emerging paradigm where workers can perform information manipulation tasks in virtually any setting. In the fixed office context, only continuous connectivity is considered so decisions basically center around issues such as bandwidth and availability. In mobile offices, we add another dimension to the decision making process: connectivity-type. Connectivity-type may range from con-

*College of Computing, Georgia Institute of Technology - {waigen, sham}@cc.gatech.edu
†Baylor School of Engineering and Computer Science - Jeff_Donahoo@Baylor.edu

tinuous connectivity as provided by cellular and satellite technologies to intermittent connectivity that is purely end-user directed. The major factor affecting decisions on connectivity-type is cost.

Since continuous, wireless connectivity is costly, we consider data sharing applications for intermittently connected mobile offices. Sharing data with only intermittent connectivity is suited for applications that require high availability, but not necessarily strict consistency between shared data values such as sales, particularly in pharmaceuticals, consumer goods, industrial parts; insurance and financial consulting and planning; real estate or property management or maintenance activities, etc. To allow sharing in such applications, a continuously connected server manages a central database containing the union of all shared client data. The central database is partitioned into "mixed fragments" (horizontal and vertical partitions as described in [18]) as required by the application. Clients subscribe to subsets of these fragments, and, in doing so, maintain local replicas of them, on which they can perform transactions. The clients also maintain a log of the updates to their local replicas. When the clients have network access, such as in the office, they connect and send the log of their updates to the server. At periodic checkpoints, the server gathers all client update logs and applies them to its database. Changes sent by one client need to be propagated to all clients sharing the changed data item. When the server processes the client update logs, it creates a set of logs that can be downloaded by the clients to update their local copies. Each time a client connects, it downloads the update logs generated by the server since its last connection and applies the changes in those logs to its local data copy. In this way, the states of the clients and the server can be "synchronized." We call database systems which support these capabilities "intermittently synchronized databases (ISDB)."

Typical commercial products [4, 5] take a "client-centric" approach in managing sharing among clients. With respect to the server, each client is independently defined by the fragments to which it subscribes. The server "dumbly" generates update logs based on these client definitions, and each client downloads its unique update log. Unfortunately, the complexity of update log creation and transmission is directly related to the number of clients the server is supporting. Each client that is added to the system requires more server processing and transmission bandwidth, limiting the scalability of the number of simultaneously supportable clients. We instead advocate the "data-centric" approach to update management. In this approach, a tunable number of fragment "groups" is created, based on the original client-fragment subscription patterns. Previous work demonstrates that managing updates according to data sharing groups instead of clients can significantly increase scalability[14]. Instead

of each client downloading update logs based on its individual fragments subscriptions, it downloads update logs based on a set fragment groups which "covers" its original fragment subscriptions. Potential advantages of data-centric grouping over client-centric grouping include:

- Reduced server processing cost - As the number of groups is tunable, fixing the number of update logs generated controls server processing cost.
- Reduced disk storage cost - Data-centric fragment grouping can reduce storage cost by reducing redundancies that may exist in client-centric grouping. For example, multiple clients independently subscribing to identical fragments can instead subscribe to a single group containing those fragments.
- Reduced transmission cost - Individual groups can be designed so that increasing numbers of clients become interested in them. Their resultant update logs become good candidates from broadcast or multicast transmission. (For brevity, we only consider unicast transmission in this paper.)

Previous work in data-centric grouping (or aggregation) describes the necessary extensions to the traditional solution, such as multigroup update log merging; however, it does not address the issue of the optimal set of groups to create for a particular set of clients. In this paper, we offer a method of grouping fragments based on a measure of "inter-client sharing," which basically describes how many fragments each client has in common with other clients in its subscription. To make this discussion more tangible, we provide two examples. The first is a concrete data-centric grouping example. The second relates our work to commercial ISDBs.

University Database Example Consider a modified version of the University database example from [14] in Figure 1. The server database contains all data concerning student enrollment. We have three types of clients, each of which wants a different subset of the server data. Note that there may be many instances of a particular client type. In the client-centric approach, the server creates and transmits an update log for each client, so, 10,000 undergraduate students require 10,000 update log creations and transmissions from the server. Figure 1 also gives one possible grouping for the University Database by defining four groups in the "Groups" section and showing the assignment of these groups to clients in the "Clients" section under the "Group Subscription" column. Note that irrespective of the number of students, the server only needs to create four update logs. This example also demonstrates that clients may receive superfluous data which must be prefiltered as

in undergraduates receiving graduate courses. Details of assuring integrity and security for this approach are in [14] and, therefore, not repeated here.

Server Schema

Students(StudID, Name, Class[1])
Enroll(StudID, CrsID, Grade)
Courses(CrsID, Name, Class[1])

Clients

Client Type	Client Schema	Group Subscription
Undergrads	Undergraduate data minus Grades	UStuds, ACrs, AEnr
Grad Students	Graduate data minus Grades	GStuds, ACrs, AEnr
Professors	All data	GStuds, UStuds, ACrs, AEnr

Groups

Group	Group Dataset
UStuds	Students where Class = UNDER
GStuds	Students where Class = GRAD
ACrs	Courses (all data)
AEnr	Enroll (all data)

Figure 1: University Database with One Possible Grouping

The University Database example hints that the set of groups ("grouping") can have a significant effect on the relative scalability; consequently, it is important to select the correct grouping for each set of clients.

Commercial Product Example Sybase's *SQL Remote* is a typical example of software which integrates ISDB functionality into a DBMS. The main component of Remote is its *Message Agent*, which runs on both the client and the server, and is in charge of (i) capturing changes to replicated portions of the database (based on monitoring the database's log), (ii) preparing update logs containing the changes based on information it contains in a *Data Distribution Model* (DDM), (iii) sending these update logs to their respective recipients, and (iv) receiving and incorporating changes contained in the update logs from others. The DDM contains group definitions: at the server, the DDM contains an entry corresponding to each client, and the replicas that each possesses; at the client, the DDM indicates which parts of its local database are replicas of server data.

The work presented in this paper cleanly fits into SQL Remote's operational model. It is designed

[1]Class $\in \{UNDER, GRAD\}$

to fit into a layered ISDB architecture, isolating the effects of aggregation to "contiguous levels" of operation. This results in functional transparency from the perspective of other levels. If we consider ISDB operation to start from the point at which the server decides to generate update logs, and end when some client assimilates update data from the server, then the aggregation functionality is isolated to some well-defined levels between these two endpoints.

Regardless of architecture–either client-centric or data-centric–the server collects updates from each client, orders them, and tries to apply them to its local database. Conflicts are resolved, and transactions that need to be undone are reported. The result is an ordered list of transactions that have to be applied to the global database. In the client-centric case, the server generates an update log for each individual client which is received and installed by that client. In the data-centric case, update logs are generated for a group of fragments. Each clients get the requisite number of update logs as defined by the client's subscription to groups. Each client must perform a "merge" operation on the updates received from multiple update logs to combine them, and filter out redundant or inapplicable data. This results in a single update file, customized for each client, identical to that of the client-centric case. Fig. 2 describes the differences.

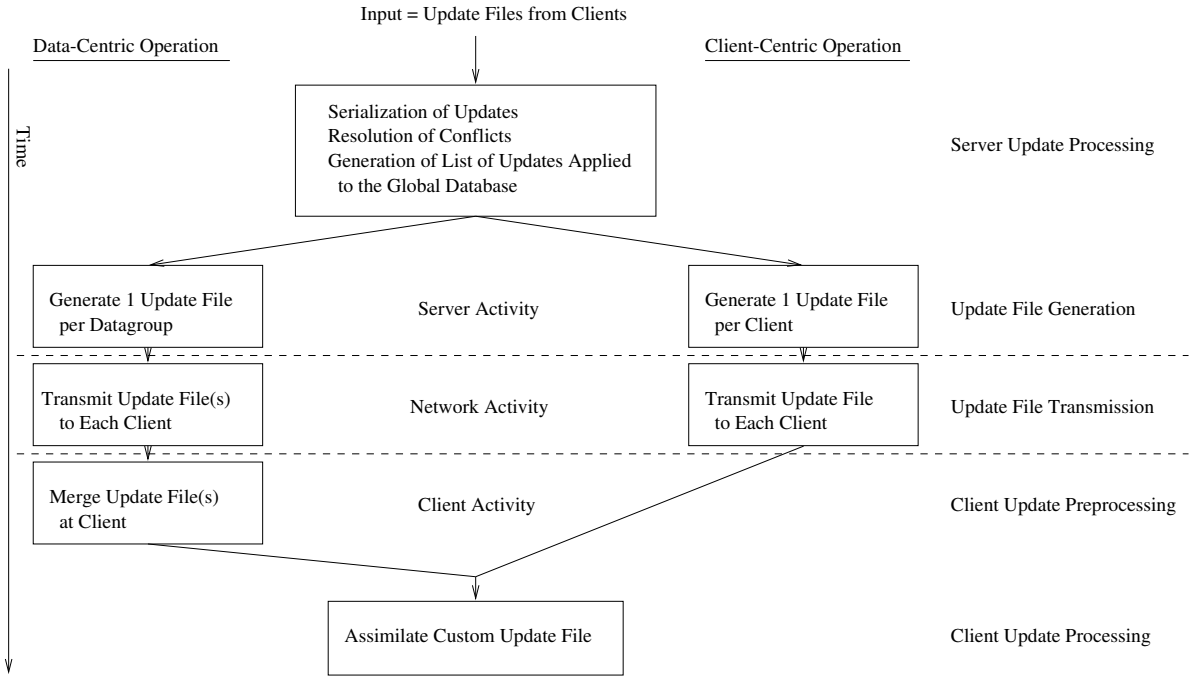


Figure 2: A Comparison Between Data-Centric and Client-Centric Synchronization Activity Over Time. A Data-Centric ISDB system operates differently only in group generation, group transmission, and requires a merge operation.

Hence, incorporating data-centric aggregation into ISDB software, such as SQL Remote, is straightforward–all that must be done is modify the contents of the DDM to contain data for aggregated groups instead of individual clients, and add a merge process to combine multiple update logs at the client. Details of the necessary server extensions and the merge operation are detailed in [14], and are not repeated here. The rest of the complexity of "intermittent synchronization" is taken care of by the ISDB system.

We begin this paper by presenting a formal model specification of the grouping problem in Section 2. This formal problem specification provides the groundwork for possible solutions. Unfortunately, the problem of fragment creation and allocation is related to other fragment allocation problems such as file partition allocation and distributed database design; therefore, we believe the derivation of the optimal solution to be computationally intractable. Since the grouping problem is intractable, in Section 3, we propose an approximation algorithm based on spanning trees. We evaluate the computational complexity of our algorithm and compare its performance against the client-centric approach. We then describe previous work related to the grouping problem in Section 4. Finally in Section 5, we set out some conclusions based on our observations of experimentation.

2 Grouping Model

The specific grouping employed by the server profoundly affects the overall performance of the system. To determine the desired grouping, we need to know the data, expressed as predicates on the server schema, relevant to each client. We want to design groups that reduce the scalability limiting redundancy created by the overlap between client data interests in the client-centric approach. The creation of a grouping relates to fragmentation design and allocation in traditional distributed databases[18, 20] so we shall draw on many of the ideas in that body of work.

2.1 Initial Design

Typical ISDB systems allow administrators to define vertical and horizontal partitions of tables of a relational DBMS to which clients can subscribe. Sybase describes this process as employing "data qualifications similar to the *where* clause in a SQL *select* command, making fine-granularity replication possible."[8] In this paper, we use the word *fragment* to describe these "mixed" data partitions, (and

denote them $F = \{F_1, \ldots, F_T\}$), and assume the knowledge of which client subscribes to which fragments.

Note that a more formal approach to initial fragment design would consider the substantial amount of work done in both horizontal and vertical fragmentation[2, 17]. These techniques have been combined into "mixed fragmentation" models, which have been implemented in prototype design tools[18]. Typically, the 80/20 rule always applies where 20% of transactions account for 80% of the traffic in database applications. However, fragment design is beyond the scope of this paper.

2.2 Grouping Problem

Formally, we specify the grouping problem as follows. Let $C = \{C_1, \ldots, C_N\}$ where C_i is the set of predicates describing the data requirements of client i. These predicates specify the subscription of the clients to fragments of the server schema. To perform data-centric aggregation, we need two parts: 1) a set of groups; and 2) a mapping of clients to groups. Each element in the set of groups defines the information contained in a data-centric group. In the University Database example of Figure 1, the set of groups is $\{UStuds, GStuds, ACrs, AEnr\}$, and the mapping is given in the "Group Subscription" column of the "Clients" section. The server will periodically generate an update log for each group, containing operations necessary to updating the parts of the database described by the group. The client-to-groups mapping indicates which groups each client must subscribe to in order to receive updates for all the data it shares with the other ISDB users.

Let $G = \{G_1, \ldots, G_M\}$ where G_j is a set of predicates describing the data associated with group j, and let $\Phi(i) \rightarrow A$ where $\Phi(i)$ is a function computing the groups from G associated with client i ($A \subseteq G$). We define a single required constraint on a grouping, called the Correctness Constraint: the union of all data represented by a client's group subscription is a superset of the client's data subscription. That is, let $\bigtriangledown()$ be a relational operator that correctly combines fragments as in [20].

$$\bigtriangledown(\Phi(i)) \supseteq \bigtriangledown(C_i), \forall i, 0 \leq i \leq N \tag{1}$$

Constraint 1 only guarantees that the subset of groups assigned to a client contains at least all of the data to which the client subscribes. This constraint does not consider any of the relevant performance metrics. In fact, a single group containing all client subscriptions satisfies Constraint 1.

Three cost factors are considered in evaluating the performance of a particular grouping: server processing, server storage, and network transmission. We compute these quantities on a per-average-

operation basis. That is, for each metric, we compute the cost of handling one average operation. Server processing measures the amount of time (seconds) required to compute the set of updates resulting from an average operation for all groups. Server storage quantifies the space (bytes) needed to record the average operation in the relevant group update logs. Network transmission evaluates the bandwidth (bytes) consumed in transmitting the typical operation in the relevant group update logs to the client(s). Since each quantity assesses a different type of resource consumption, we must compute relative costs. That is, for a particular system, we must determine how many seconds of server processing is "equivalent" to the storage of a byte. To accomplish this, we normalize each metric to a standard measure of resource cost called *resource tokens*. For example, to normalize server processing cost, we multiply by α, the resource tokens per second. By normalizing all three metrics, we derive common units for each metrics, allowing us to sum the metrics for a total cost as in

$$\text{Total Cost} = \alpha(\text{Server Cost in secs.}) + \beta(\text{Storage Cost in bytes}) + \gamma(\text{Networking Cost in bytes}) \quad (2)$$

2.3 Cost Model

To derive a concrete cost, we need a model of average operations; therefore, we first derive absolute costs on a per-operation basis. For each group, let $\Omega(G_j)$ be a function returning the percentage of operations that update group j. The server cost measures the number of seconds required to determine the applicability of an operation to each group, made by evaluating the operation for each fragment of a group. To process an operation, the server must evaluate its applicability to each group; therefore, the server cost is the product of the number of groups and the average number of seconds to evaluate the average group, V_s, as in

$$\text{Server Cost} = V_s|G| \quad (3)$$

The storage cost measures the number of bytes required to store an operation in the relevant update logs. If we are given the average number of bytes per operation, V_d, we determine the number of bytes stored for an operation as

$$\text{Storage Cost} = \sum_{G_j \in G} \Omega(G_j) V_d \quad (4)$$

The unicast networking cost measures the number of bytes required to transmit the appropriate operations individually to all clients. The networking cost calculation is similar to the storage cost calculation,

except that a group may be accessed by multiple clients and, therefore, the group's operations may be transmitted multiple times. Given G and Φ, the network bandwidth cost for an average operation is the sum of the bytes for each group the operation is in, over all clients:

$$\text{Unicast Transmission Cost} = \sum_{i=1}^{N} \sum_{G_j \in \Phi(i)} \Omega(G_j) V_d \tag{5}$$

Since the number of operations applying to each fragment may vary widely, we include $\Omega(G_j)$ in the computation of storage and networking cost. Effectively, this function provides an *a priori* prediction of system operation. We use database statistics to determine the proportion of operations that apply to each fragment, and define $W = \{W_1, \ldots, W_T\}$ where W_i is the percentage of operations applying to fragment i. W can the be used to predict system operation. V_s and V_d are similarly computed based on database statistics.

2.4 Grouping Approach

A grouping approach determines both the set of groups, G, and the function, $\Phi(i)$, mapping client i to the set of groups in G. These groups are combinations of fragments such that the objective function (Equation 2) is optimized. Clearly, there are an intractible number of grouping solutions: Given F, $G = \{G_1, \ldots, G_M\} \subseteq 2^F$.

While there are many similarities between distributed database design and our problem of grouping design, we are addressing a very different problem. In distributed databases, we compose fragments to create a single fragment allocation for each client based on access probabilities. Clients can access data at other clients so the design criteria necessary for correctness requires that the original relation can be reconstructed in a distributed fashion from the client fragments. In ISDBs, clients must have locally all of the data relevant to their needs. In addition, each client potentially accesses multiple data groupings instead of a single allocation of fragments.

To demonstrate some dimensions of this problem, we give some extreme solutions which minimize various combinations of the above-mentioned cost factors. For example, at one extreme we can create a single group containing the entire server database, minimizing server processing and storage space; however, clients sharing only a small portion of the server database must be transmitted many superfluous operations. At another extreme, we can create a datagroup for each fragment, simultaneously minimizing

storage and transmission costs.[2] However, the resultant number of groups may result in an extremely high server processing cost. Intermediate solutions such as "client-centric" (i.e., one specialized group per client) and "data-centric" (i.e., groups defined by accounting for common interests of multiple clients) constitute intuitively sensible compromises.

3 Grouping Approximation Algorithm

The grouping problem, as specified in the previous section, is related to the class of problems containing distributed file fragment allocation and distributed database design. Unfortunately, these types of allocation problems have been shown to be computationally intractable[7], thus, we provide an approximate solution with a heuristic-based algorithm.

In this section, we first give a high level description of our algorithm, and discuss its underlying concepts. We then give its formal specification, and demonstrate its execution on the University example described in Fig. 1. We end the section by describing experimental results that demonstrate the performance effects in terms of server processing, disk usage, and data transmission using the data-centric approach as "data sharing" amongst clients increases.

3.1 Basic Concepts

Our approximation algorithm is based on the following heuristics:

1. The groups should form a partition of the server database fragments. That is, each fragment should appear exactly once in the set of groups. This disjointness constraint may be relaxed in further reorganization of groups.

2. Formation of the groups should proceed by selection of the larger sets of shared fragments, removing the fragments from the subscription of each client, and repeating the process.

One consequence of the first heuristic is that each update operation is only saved in one update log. Hence, there is no duplication in disk usage, and storage cost is optimized. However it is clear that problem specifications exist where these heuristics would produce suboptimal results; however, we believe that our approach, based on these heuristics, performs well, especially relative to the client-centric approach.

[2] $\Phi()$ is trivial in both extreme cases.

We begin by constructing a graph to represent interclient sharing. For each client, we construct a graph node. For each pair of clients that share fragments, we construct an edge between their corresponding nodes. Recall that each fragment, F_i, has a corresponding percentage of operation applicability, W_i. We attribute a weight to each edge that is the sum of all W_is for the shared client fragments. If the clients represented by two nodes, have fragments $F_i, \ldots F_j$ in common, then the weight of the edge is $W_i + \cdots + W_j$. In the worst case, we have $O(N^2)$ edges in our graph, where N is the number of clients.

The second heuristic instructs us to iteratively group together sets of fragments which are shared most. To accomplish this, we compute a special type of maximal spanning tree within the graph. The edge with the highest weight indicates the two clients with the greatest amount of sharing; therefore, we begin by selecting the two nodes that are the endpoints of the most heavily weighted edge, and coalescing them into a single node (called the *root* node). The fragment subscription of the root node is equal to the intersection of the fragment subscriptions of the two selected nodes. For each edge attached to this new, aggregate node, we recompute the edge weights as the sum of W_is for the fragments in the intersection of the fragments for the new root node and the node at the other end of the edge. We then select an edge incident to the root with the maximum edge weight, and repeat the coalescing process of the root and the node at the other end of the selected edge. The root node represents a special type of maximum spanning tree in the original graph. At some point, we stop coalescing nodes and create a group from the fragments subscribed to by the current root node. Any client subscribing to any of the fragments in the root node subscribes to this new group. We remove the fragments of the root node from the subscription of all clients and rebuild the tree with the new subscriptions. Then we repeat the same process with the new client subscriptions.

There are two decision variables unspecified by our approach thus far: the number of coalescing operations to perform in determining a spanning tree and the number of spanning trees to compute. The root node represents the fragments that are shared between *all* the currently coalesced nodes. Clearly, as we continue to coalesce, the number of shared fragments in the root node can only decrease. Longer chains of coalescing operations increase the probability of a greater number of smaller groups. From Equation 3, we know that more groups results in greater server cost; however, intuitively, a large number of small groups may have higher "granularity," and better fit a client subscription, potentially reducing storage and networking costs.

Our objective function (Equation 2) provides a means to specify the tradeoff between server, storage, and networking costs with α, β, and γ, respectively. As mentioned above, our heuristic partitions the data, resulting in optimal storage costs. Because of this, storage cost is no longer constitutes a group design decision, and we remove the β term from the objective function. If α is greater than γ, then server processing time is relatively worth more than storage or transmission (in terms of the normalized cost unit). Since coalescing creates potentially greater server processing (more groups) but smaller network transmission cost (better fit of groups with clients), we must make the decision of continuing coalescing based on the relative importance of server processing and networking transmission costs. To determine when to stop coalescing, we must determine a baseline expectation for sharing. This baseline should be relative to the typical client subscription sharing. That is, if an average client shares 20% of its data with other clients, then a group containing 10% of the shared data between two clients may be worth generating. However, if an average client shares 80%, a group containing only 10% of the shared data between two clients may not be worth server processing time.

We first must determine a measure of sharing between clients. Let $C' = \{C'_1, \ldots, C'_N\}$ where C'_i is the set of fragments from F that define the minimal superset of client i's fragment subscription ($\bigtriangledown(C_i) \subseteq \bigtriangledown(C'_i)$). We begin our definition of sharing by defining a function to calculate the **pairwise sharing between two clients**:

$$P(i,j) = \frac{\sum\limits_{F_l \in (C'_i \cap C'_j)} W_l}{\sum\limits_{F_l \in (C'_i \cup C'_j)} W_l}$$

This gives us a ratio of shared to total data. Now we define sharing to be the **average of all pair-wise sharing functions**:

$$S = \frac{\sum\limits_{i=1}^{N-1} \sum\limits_{j=i+1}^{N} P(i,j)}{\begin{pmatrix} N \\ 2 \end{pmatrix}}$$

We can approximate S by selecting random pairs and determining their average pair-wise sharing if the number of pairs (determined by N) becomes too great.

We need a lower threshold of sharing in the groups. Since increasing the length of the spanning tree usually decreases sharing, we will no longer add nodes to the spanning when the addition of new nodes decreases the sharing of the new group. Therefore, we base our threshold on the average sharing, S,

between clients. Recall that consideration of server processing minimizes the number of groups, while consideration of networking cost increases the number of groups. Our threshold should also consider the relative importance of server to networking costs which can be computed as the ratio of α to $\alpha + \gamma$. Given this, we determine not to add an edge unless the new weight of the edge is greater than or equal to

$$\text{Threshold} = \frac{S\alpha}{\alpha + \gamma}$$

Also, we will not add an edge with zero weight. This relationship limits the length of the spanning tree if an additional node coalescing creates sharing below the threshold defined by α and γ. This relationship also limits the number of groups we create since eventually we will not be able to find even a single edge satisfying this relationship.

We may still have fragments that have not been allocated so we need another approach to gather the remaining fragments into disjoint groups. To handle this "clean-up," we assume that each remaining client[3] has a disjoint set of fragments. We then numerically compute the optimal number of groups into which we can divide the remaining fragments.

3.2 Approximation Algorithm Specification

We now formally describe our approximation algorithm by describing its two main components: FindGroups and Clean-Up. FindGroups is responsible for finding the set of shared groups derived by developing the spanning trees until sharing falls below the threshold. Clean-Up is applied after FindGroups to group all of the non-shared (assumed) fragments not grouped by FindGroups. Before entering the FindGroups phase, we compute the fragmentation set, F; sharing, S; client fragment subscription, C'_i; and $Threshold$.

FindGroups:

1. Create a graph, $Y = (V, E)$, where $V = \{1, \ldots, N\}$ and $E = \{(i,j) \mid (i,j \in V) \wedge (i \neq j) \wedge (C'_i \cap C'_j \neq \emptyset)\}$. For each edge, (i,j), assign the weight

$$W(i,j) = \sum_{F_l \in C'_i \cap C'_j} W_l$$

2. Find the edge, (i,j), with $Max(W(i,j))$. If $W(i,j) < Threshold$ then Clean-Up.

[3]The shared grouping may be sufficient to cover some clients. These clients would not be included in the clean-up.

3. Coalesce nodes i and j in Y.

   (a) Create a new node, R, where $C'_R = C'_i \cap C'_j$, and remove (i, j) from E.

   (b) For every edge incident to node i or j, let k be the "other" endpoint of the edge ($k \notin \{i, j\}$), remove edge (i, j). If $C'_R \cap C'_k \neq \emptyset$, create a new edge, (k, R), with weight $W(k, R)$.

   (c) Remove i and j, and add R to V ($V = V - \{i, j\} \cup R$).

4. Find the edge, (k, R), incident to R with the maximum weight.

5. If $W(k, R) > Threshold$ then go to Step 3.

6. Let R be the set of nodes coalesced in the root. Create a group with fragments

$$FG = \bigcap_{i \in R} C'_i$$

7. Remove the fragments in FG from all clients ($\forall i, C'_i = C'_i - FG$).

8. If $\exists i$, such that $C'_i \neq \emptyset$, call FindGroups for the client subscription C'_i.

University Database Example: Consider the University database example from Figure 1. Let us focus on the distribution of Students and Enroll. To illustrate the algorithm, let us populate a simple database with 2 undergraduate students, each enrolling in 3 classes, and 1 graduate student enrolling in 2 classes. We have 2, 1, and 2 undergrad, graduate, and professor clients, respectively. We assume the access probabilities are similar to the frequencies in the current database. Our fragmentation schema consists of six fragments:

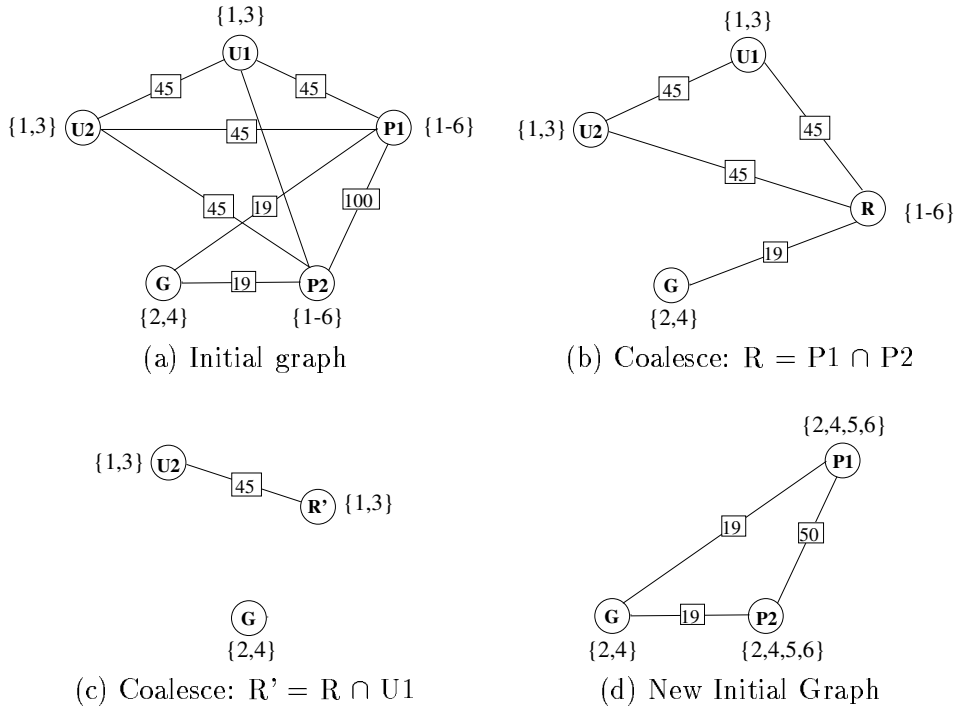
FID	Fragment	W_i	Fragment Description
1	StudU	18%	Undergraduate student data
2	StudG	10%	Graduate student data
3	EnrollU	27%	Undergraduate enroll data minus grades
4	EnrollG	9%	Graduate enroll data minus grades
5	GradesU	27%	Undergraduate grade data
6	GradesG	9%	Graduate grade data

For this example, there are a total of 11 tuples (3 students and 8 enroll tuples). Fragment 1 corresponds to 2 of those tuples so $W_1 = 2/11 = 18\%$. The sharing value, S, for this scenario is 41.8%; therefore, if we let $\alpha = 1$ and $\gamma = 2$, the break point is 13.9%. Figure 3(a) gives the initial graph generated by the FindGroups phase. Nodes with Ux, G, and Px represent undergrad, graduate, and professor clients, respectively. There are no edges between Ux and G nodes because they do not share any fragments. The boxed values represent the edge weight for the associated link. The sets near each node represent the fragment IDs of the client's subscription. FindGroups begins by selecting the heaviest link, which is between the two professor client nodes, and coalesces P1 and P2 into a single node (R) as shown in Figure 3(b). Figure 3(c) coalesces R and U1. Now R's subscription is the *intersection* of the subscriptions of P1, P2, and U1. Node G is not connected to the graph because it does not have any fragments in common with any other nodes in the graph. The next step (not shown) coalesces R and U2 without changing the subscription of R. Since there are no more edges, we have completed the first iteration of FindGroups. The fragments in the subscription of R comprise the first group, $G_1 = \{1, 3\}$. Figure 3(d) gives the initial graph for the next iteration of FindGroups. The new graph does not have any Ux nodes because G_1 covers their subscription. Also, the client subscriptions do not include fragments 1 and 3 because they are already contained in G_1. The next FindGroups iteration will coalesce P1, P2, and G, creating $G_2 = \{2, 4\}$. The initial graph of the final iteration contains only P1 and P2 which will be coalesced, creating $G_3 = \{5, 6\}$. Since all clients are covered by some combination of $\{G_1, \ldots, G_3\}$, we do not need the Clean-Up phase.

The Clean-Up phase of our approximation algorithm assumes that none of the fragments are shared. Given this assumption, we can numerically derive the optimal grouping of the remaining fragments. Since we are trading off server processing with transmission cost, it may not necessarily be optimal to create a separate group for each client. Let N' be the number of clients with fragment subscriptions unfulfilled by the groups created in the FindGroups phase, and $\hat{D}$ be the average fragment access probability for all remaining clients.

$$\hat{D} = \frac{\sum_{i=1}^{N'} \sum_{F_l \in C'_i} W_l}{N'}$$

We need to find the optimal number of groups, X, to contain the remaining disjoint fragments. To



(a) Initial graph (b) Coalesce: R = P1 ∩ P2

(c) Coalesce: R' = R ∩ U1 (d) New Initial Graph

Figure 3: Sharing Graph for University Database Example

compute X, we consider the objective function (Equation 2) for *only* the remaining fragments under the disjoint fragment assumption and without the storage cost since it is already optimized by making the groups disjoint. The resulting objective function is

$$\alpha V_s X + \gamma \frac{N'}{X} \hat{D} V_d N' \tag{6}$$

We want to solve for the value of X that minimizes this equation. To do this, we take the derivative of Equation 6 with respect to X, set the derivative equal to 0, and solve for X

$$X = \sqrt{\frac{\gamma V_d \hat{D} N'^2}{\alpha V_s}} \tag{7}$$

Clean-Up:

1. Determine $\hat{D}$ and then X. Round X to nearest integer.

2. Divide the fragments evenly (according to their W_i values) among the X groups.

The Clean-Up phase distributes the remaining fragments evenly among the disjoint groups. We add these disjoint groups to the set of groups derived in the FindGroups phase. The clients to groups mapping

($\Phi(i)$) is derived by selecting the set of groups containing the fragments (from C_i') for each client.

The complexity of our algorithms is $O(TN^3)$ Major computations (such as those of P and W) iterate over all pairwise combinations of the N nodes, performing computations on each of their $O(T)$ fragments. We deduce the final complexity, noticing that FindGroups iterates a maximum of N times.

3.3 Performance

Our experiments are designed to show the relative performance in terms of server processing, storage, and transmission costs between the data-centric grouping we propose in this paper, and the client-centric grouping that is performed in commercial ISDBs with respect to increasing degrees of interclient data sharing. Intuitively, we know that the data-centric approach performs best when all clients share the same data and worst when there is no sharing[4]. We desire a non-application-specific mechanism to evaluate the relative performance of client to data-centric for varying "amounts" of sharing in the continuum between "all-data-shared" and "no-data-shared." That is, how does increasing the amount of data shared between the average client affect the performance of data over client-centric using our heuristic to determine grouping.

3.3.1 Experimental Framework

In our experiments, we simulate $N = 100$ clients, each subscribing to a subset of $T = 50{,}000$ tuples of a universal relation (containing a single attribute). All clients subscribe to a fixed set of M tuples, and each is allocated D tuples, which are uniquely its own, so that $D \times N + M = T$. The degree of sharing is varied from 0% to 100% in intervals of about 20%, and is achieved by varying the values of D and M–for example, an increase in M results in an increase in S, the degree of sharing. We run two sets of experiments to test the effects of varying values of the cost coefficients, α and γ. In the first set of experiments, $\alpha = \gamma = 1$, meaning that equal weight is given to both server processing and transmission costs. In the second set of experiments, $\alpha = 1$ and $\gamma = 5$, which should result in a greater number of smaller groups, reducing transmission costs.

In accordance with the model developed in Sec. 2, we base performance on the following metrics: server processing, storage space, and transmission (unicast) cost. Server processing cost is a measure of the total time required to distribute the data contained in the universal relation to update logs. Storage

[4]Recall that the client-centric approach is a special case of the data-centric approach, so the ratio of client to data-centric performance need never be less than 1.

space cost is the number of bytes required to store the update logs on disk. Transmission cost is the sum of the number of bytes of the update logs that must be transmitted to every client in order to satisfy Constraint 1. For each of these metrics, instead of reporting absolute numbers, we report performance as a *ratio* of client-centric to data-centric costs. Hence, ratios above 1 indicate that the data-centric approach is more effective than the client-centric approach, and visa-versa.

These experiments were conducted on a 200MHz PC with 64MB of RAM and a 2GB hard drive, running Windows NT Server. Software used includes Sybase SQL Anywhere as our DBMS and a production version of non-disclosed commercial software similar to SQL Remote and modified with the data-centric extensions described in [14]. More details of our experimental setup are described in [6].

3.3.2 Results

We vary sharing, S, as shown in Table 1. The corresponding values of M ("matching" tuples), D ("different" tuples), and the number of groups generated under that data-centric approach are also given. (S values are not exact multiples of 20 because it is computed from values of D and M, which must be integers.) As shown in Fig. 4, as sharing increases, the ratios of client-centric to data-centric costs of all three metrics–server processing (**Processing**), storage (**Disk**), and transmission (**Unicast**)–increase, indicating the increasing effectiveness of the data-centric approach. In fact, the ratios are plotted on a log scale, which indicates that data-centric server processing performance actually increases *exponentially* with increased sharing. The transmission cost ratio is below 1 when sharing is at 0%, but approaches 1 as sharing approaches 100%.

Table 1: Client-Data Associations and Grouping. ($\alpha = 1$)

Actual Sharing (S)	Matching (M)	Different (D)	Groups	
			$\gamma = 1$	$\gamma = 5$
0%	0	500	11	21
16.7%	200	498	11	21
41.5%	700	493	11	21
60.7%	1500	485	11	21
79.9%	3700	463	11	21
100%	50000	0	1	1

These trends remain the same regardless of whether $\gamma = 1$ or $\gamma = 5$. However, $\gamma = 5$ results in a larger number of smaller groups, so server processing cost is higher, and transmission cost is lower in

this case. Since the groups are partitioned, disk cost is optimized regardless of the value of γ.

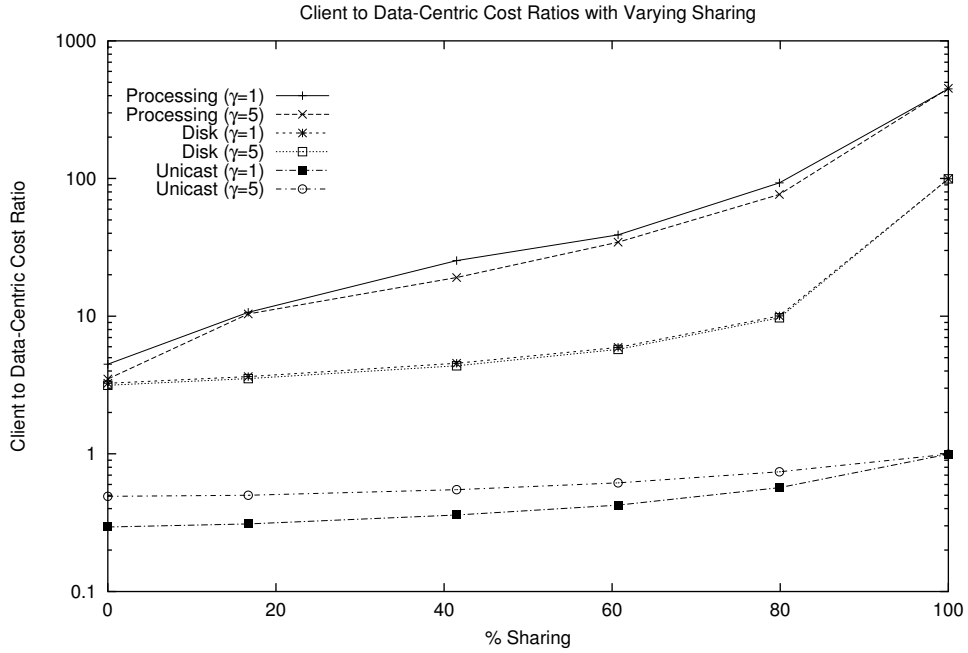


Figure 4: Sharing Performance - Increasing ratios indicate increasing effectiveness of data-centric grouping.

3.3.3 Discussion

Using a universal relation containing a single attribute allows us to focus on the specific issue of sharing versus performance in an application independent manner. This is similar to work done on universal relations[16]. Our experimental results, which we are about to discuss, are therefore generic and unbiased.

We consider two groupings: one for $\alpha = \gamma = 1$ and one for $\alpha = 1, \gamma = 5$. (We will refer to these groupings by γ value only, since α is fixed.) Table 1 contains the values of M, D, the actual S, and the number of groups generated by both the FindGroups and Clean-Up phases for both groupings[5]. In all executions of our approach, the FindGroups phase created a single, shared group. Incrementing D by one actually moves ND tuples from the shared group to the different groups; therefore, changing the sharing value requires only a minor change in D as shown in the table. This sensitivity also manifests itself in determining the number of groups. Note that D and $\hat{D}$ are tightly related in this case ($\hat{D} = D/T$). Since the value of D needs only minor changes to have a major effect on the sharing value and $\hat{D}$ is the only variable in the equation for X (Equation 7), the total number of groups does not change when

[5]A shared group is created even with a zero sharing value ($M = 0$). This group carries meta-data (administrative) information to inform the clients how data is shared.

sharing changes until $S = 100$ as shown in Table 1. However, the amount of data assigned to each group is changing significantly.

Table 2 presents the ratios of client to data-centric time and space requirements. Of course, ratios greater than 1 indicate the data-centric performed better than client-centric, and the higher the ratio the better data-centric performed relative to client-centric. We present the derivation of the performance ratios for $S = 0$ and $\gamma = 1$ to demonstrate their calculation. For server processing cost, the client and data-centric server processing times are 6 minutes, 21 seconds and 1 minute, 25 seconds, respectively, so the performance ratio is 4.48. The storage space cost for client and data-centric are 3200KB and 982KB, respectively, so the storage space performance ratio is 3.26. The transmission cost for client and data-centric as 3200KB and 10900KB, respectively, so the performance ratio is 0.294.

Table 2: Client to Data-Centric Cost Ratios. ($\alpha = 1$)

	Server Processing		Storage Space		Transmission Cost	
S	$\gamma = 1$	$\gamma = 5$	$\gamma = 1$	$\gamma = 5$	$\gamma = 1$	$\gamma = 5$
0%	4.48	3.50	3.26	3.15	0.294	0.492
16.7%	10.68	10.38	3.64	3.52	0.310	0.500
41.5%	25.33	19.11	4.55	4.36	0.360	0.549
60.7%	38.95	34.45	5.94	5.74	0.424	0.615
79.9%	92.98	76.62	10.05	9.73	0.568	0.741
100%	447.66	452.52	99.47	99.47	0.995	0.995

From Table 2, we observe that, as sharing increases, the client-centric performance degrades because it is performing greater and greater amounts of duplicate work. Figure 4 gives a graph of the performance ratios (y-axis) from Table 2 for various sharing values (x-axis) with $\gamma = 1$ and $\gamma = 5$. Note that, as expected, increasing the sharing greatly increases the relative performance of the data-centric approach for server processing. The y-axis is log scaled so the server performance increase is actually exponential (not linear). Increasing sharing also decreases the required storage space for the data-centric approach relative to client-centric as seen by the increase in the storage curve. Because the data-centric approach sends superfluous data that the clients must filter, using the data-centric approach is more expensive with respect to transmission cost than client-centric. This is demonstrated by the transmission cost performance ratio always being less than 1; however as sharing increases the transmission performance ratio approaches 1 for 100% sharing. It is also interesting to note from Figure 4 that even with an overall sharing of 0%, data-centric still performs better than client centric for server processing. This is because

the grouping saves significantly on server processing (10 groups in data-centric aggregation versus 100 clients in client-centric aggregation).

We present two data-centric groupings ($\gamma = 1$ and $\gamma = 5$) to demonstrate the tradeoffs inherent in Equation 2. By setting $\alpha = \gamma = 1$, we give equal weight to processing a tuple as to transmitting it. When we increase α, we are increasing the weight of cost for transmitting a tuple over processing it. In the data-centric approach, we increase transmission efficiency by decreasing the number of superfluous tuples sent to receivers. We reduce the number of superfluous tuples by creating more groups, giving a better fit of each data-centric group to client needs. Note from Table 1 that increasing γ from 1 to 5, almost doubles the number of groups from 11 to 21. The effect of increasing α is decreasing the individual log sizes for the D tuple update logs; however, since increasing γ increases the number of groups, from Equation 3 we know this increases the server processing cost. This is confirmed by the higher server processing cost for $\alpha = 5$ in Table 2. Figure 4 demonstrates these observations for varying share values. Note that the curve for server processing cost for $\gamma = 5$ is slightly worse than the same curve for $\gamma = 1$; however, the transmission cost curve for $\gamma = 5$ is slightly better than the same curve for $\gamma = 1$. This demonstrates how we can use the parameters of Equation 2 to tune the performance for a particular system.

4 Related Work

Intermittently synchronized databases (ISDBs) naturally fall into a class of applications related to both view (and replica) maintenance and mobile database systems. The server of an ISDB is in charge of maintaining the views of its clients. As in classical view materialization work, updates are propagated to views via refresh processes. Refreshment of views may either be immediate[10]–within the transaction that updates the base table–or, as in ISDBs, deferred[3]. View maintenance work, however, typically deals with issues of consistency and speed of view refresh. One early work deals with replica maintenance on mobile computers[9], but focuses on reduction of deadlock and reconciliation of *lazy replication*. It proposes the master copy replication scheme that exists in most commercial ISDBs today. However, it does not deal with the scalability of update log generation.

Mobile database research takes a more system-wide view and considers the costs incurred by the components in a distributed and even disconnected architecture[12]. Issues such as cell-handoff, power consumption of components and resource location in a wide area are considered. Although an ISDB

is an instance of a mobile database system, our work is not focused on these topics. However, ISDBs share the issues of centralized processing and update propagation with mobile databases and broadcast databases[1]. An ISDB imposes a high workload on the server, as it must collect updates from clients, maintain their views, and return update logs to them. However, an ISDB has different applications because of different means of communication with the server. Unlike much other mobile database work, communication is not assumed to be wireless nor asymmetric.

We identify server-side processing to be a problem, and frame it as a database design problem, in which predefined fragments (horizontal and/or vertical partitions[2, 18, 19] of the database) are allocated to nodes (which, in the ISDB case, are called update logs). Allocation research typically assumes a fixed network of varying reliability and capacity, in which resources (e.g. files or fragments) are assigned to nodes with tradeoffs in reliability, read/update costs, and storage costs[15], but does not consider the ISDB case in which clients can naturally work while partitioned from the server, and communication is mediated via update log transfers.

Some work suggests client caching as a remedy for partitioning or high bandwidth costs[21], and others[11] study dynamic allocation techniques to reduce communication costs. These works focus on availability and communication costs, assume high participation between the client and the server during synchronization phases, which we consider unscalable as the number of clients increases.

Our reference architecture is actually taken from existing commercial DBMSs which include a feature known as "replication." Such an architecture assumes a more passive role by the server, in which it pre-packages update "objects" in order to disseminate updates. Related works [13, 14] recognize possible savings in server or network costs by considering receiver interests in the delivery of updates, but these works leave out the problem of designing the update objects. To our knowledge, ours is the first work that analyzes the cost components of this architecture.

Finally, just as patterns in client subscription can be exploited to reduce update log generation costs, we also see an opportunity in exploiting these patterns to reduce update log dissemination costs. Effective use of bandwidth becomes an issue as the volume of update data increases. Grouping data can have the side-effect that many clients share interests in common update objects, and multicasting or broadcasting them becomes more feasible.

5 Conclusions

ISDB systems on the market today handle update dissemination in a client-centric manner, by generating update logs on a client-by-client basis. Unfortunately, the computational complexity of the system increases directly with the number of clients. Previous research proposes an alternative approach, called data-centric aggregation, that focuses on *how* the data is shared among clients, rather than *who* gets what data. By aggregating updates according to data sharing, server processing complexity is related to the number of data groupings, which we control; therefore, we can adapt the grouping to achieve greater scaling. Data-centric aggregation provides additional mechanisms for improving scalability because it offloads server processing onto the network and individual clients. In addition, data-centric aggregation creates data aggregates which are amenable to multicast because several clients access the same group, unlike the client-centric approach which generates client-specific update logs.

We start this paper by describing the architecture of a typical ISDB system, and showing how simply it can be retrofitted with data-centric extensions. Incorporating these extensions can be done in a manner transparent to the application. We then discuss technical issues related to data-centric aggregation.

The specific grouping can greatly affect the performance of the new approach. We develop a formal specification of the grouping problem including a cost model that combines various performance measures important to system scaling. Ongoing research is using this model to provide various solution specifications of the grouping problem. Using this model, the grouping problem has been expressed in terms of a mathematical optimization problem[22]. As in similar treatments of the allocation problem[15], there are nonlinear terms in both the objective function and the constraints, and the solution must be in integer form. Using the cost model, we can evaluate the relative scalability of various approaches. Unfortunately, the grouping problem falls into the class of problems of fragment allocation including the file allocation problem and distributed database design; therefore, the optimal grouping problem is believed to be computationally intractable. We develop a simple heuristic-based approach using maximal spanning trees that derives a grouping based on the specified tradeoff between server processing and network transmission costs.

To evaluate our approach, we develop a formal definition of the amount of sharing in a particular system. Intuitively, as sharing increases, the relative scalability of the data-centric approach improves. We engineer a standard mechanism for generating data with a specific sharing value and use our heuristic approach to generate the data-centric groupings for various sharing levels. Given these mechanism, we

execute performance testing to evaluate the effects of increasing sharing on the relative performance of the data-centric approach. We found that in our model data-centric always outperforms client-centric, even with zero sharing, and that linearly increasing the degree (amount) of sharing exponentially increases the data-centric performance, both server processing and storage space, relative to client-centric performance.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1995.

[2] S. Ceri, S. B. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Transactions on Software Engineering*, 9(3), July 1983.

[3] L. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1996.

[4] Oracle Corp. Oracle mobile and embedded products. Technical report, Oracle Corp., http://www.oracle.com/mobile, 1999.

[5] Sybase Workplace Database Division. Sql remote: Replication anywhere. Technical report, Sybase Corp., http://www.sybase.com/products/system11/workplace/remote2.html, 1999.

[6] M. J. Donahoo. *Application-based enhancement to network-layer multicast.* PhD thesis, Georgia Insitute of Technology, May 1998.

[7] K. P. Eswaran. Placement of records in a file and file allocation in a computer network. *IFIP Conference Proceedings*, pages 304–307, August 1974.

[8] A. Gorelik, Y. Wang, and M. Deppe. Sybase replication server. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Proceedings of the IEEE International Conference on Data Engineering*, 18(2), June 1995.

[11] Y. Huang, P. Sistla, and O. Wolfson. Data replication for mobile computers. *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1994.

[12] T. Imielinski and B. R. Badrinath. Wireless computing: Challenges in data management. *Communications of the ACM*, October 1994.

[13] B. Levine, J. Crowcroft, C. Diot, J. Garcia-Luna-Aceves, and J. Kurose. Consideration of receiver interest in content for IP delivery. *Submitted for publication*, July 1999.

[14] S. Mahajan, M. J. Donahoo, S. B. Navathe, M. Ammar, and S. Malik. Grouping techniques for update propagation in intermittently connected databases. *Proceedings of the IEEE International Conference on Data Engineering*, February 1998.

[15] S. Mahmoud and J. S. Riordion. Optimal allocation of resources in distributed information networks. *ACM Transactions on Database Systems*, 1(1):66–78, March 1976.

[16] D. Maier and J. D. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Transactions on Database Systems*, 8(1):1–14, March 1983.

[17] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4), 1984.

[18] S. B. Navathe, K. Karlapalem, and M. Y. Ra. A mixed fragmentation methodology for the initial distributed database design. *Journal of Computers and Software Engineering*, 3(4), 1996.

[19] S. B. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1989.

[20] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 2nd edition, 1998.

[21] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *Proceedings of the IEEE Workshop on Workstation Operating Systems*, September 1989.

[22] W. G. Yee, S. Navathe, A. Datta, and S. Mitra. A mathematical optimization approach to improve server scalability in intermittently synchronized databases. Technical Report GIT-CC-99-18, Georgia Institute of Technology, July 1999.