Query Merging:
Improving Query Subscription Processing in a Multicast Environment *

Arturo Crespo, Orkut Buyukkokten, and Hector Garcia-Molina

Stanford University

Abstract

This paper introduces techniques for reducing data dissemination costs of query subscriptions in a multicast environment. The reduction is achieved by merging queries with overlapping, but not necessarily equal, answers. The paper formalizes the query-merging problem and introduces a general framework and cost model for evaluating merging. We prove that the problem is NP-hard and propose exhaustive algorithms and three heuristic algorithms: the Pair Merging Algorithm, the Directed Search Algorithm and the Clustering Algorithm. We develop a simulator, which uses geographical queries as a representative example, for evaluating the different heuristics and show that the performance of our heuristics is close to optimal.

KEYWORDS: Query Processing, Data Dissemination, Query Merging, Query Subscriptions, Multicast of Query Results, Geographical Queries.

1 Introduction

With information dissemination (information push), data is delivered from a set of *producers* to a (typically) larger set of *consumers*. Examples of dissemination-based applications include information feeds (e.g., stock and sports tickers of news wires), traffic information systems, electronic newsletters, and entertainment delivery [15]. We focus on a type of dissemination system where the consumers in advance submit *subscriptions* defining their interests. Each subscription may include one or more queries over the

*A one-page extended abstract of this paper appeared in ICDE 2000.

data that the producers hold or generate. The producers run the queries periodically, disseminating information of specific interest to the consumers. Systems such as Pointcast [31], Marimba [29], Backweb [3], and Airmedia [2] are examples of subscription-based dissemination.

Subscription-based dissemination services are well suited to users' needs, but can be very expensive. As a real world example, a 1996 study that monitored Internet traffic found that more than 17% of the HTTP Internet traffic involved PointCast [19]. Additionally, PointCast "pulls" saturated company networks so much that large corporations have limited or even outlawed the use of PointCast on their desktop PCs [30]. In this paper we study a novel technique that can significantly reduce traffic and server loads. The overconsumption of resources in subscription-based dissemination services is the result of three factors. First, the network is point-to-point (i.e., the answers to each query subscription are transmitted separately to each consumer). Second, each query is processed independently. And third, previous work do not make full use of the processing power of clients. Instead, clients are considered "dumb" processes that are unable to perform any post-filtering of data they receive.

The overhead of point-to-point dissemination can be reduced by using a multicast network. For example, consider the case where n clients issued exactly the same query in their subscriptions. A subscription service using a point-to-point network will process and transmit the answers to those queries n times, while a multicast-based service will establish a "channel" for the answer and will transmit the answer only once [19].

However, in many applications, it is unlikely that a large number of clients will issue exactly the same query, preventing us from fully exploiting the advantages of a multicast network. In this paper, we present algorithms for efficient use of a multicast network for such applications. We achieve this by considering merging not only identical queries but also queries with answers that overlap significantly. By merging these queries, the server has to process fewer queries and the amount of information sent may be reduced. (As we will see later, in some cases, merging queries might *increase* the data sent.) On the negative side, the merged answers may contain some data that is irrelevant to a client. As a result, a client needs to make use of its processing power and apply a post-filtering *extraction query* over the received data in order to obtain the answer to its original query. For example, say we merge queries $q_1 : \sigma_{2 \leq A \leq 40} R(A)$ and $q_2 : \sigma_{3 \leq A \leq 41} R(A)$ into $q_3 : \sigma_{2 \leq A \leq 41} R(A)$. The server can then process this single query and send the result, $ans(q_3)$, to the clients that issued q_1 and q_2. The q_1 client will need to *extract* the q_1 answer from $ans(q_3)$ by applying the extraction query $q_1 : \sigma_{A \leq 40}(q_3)$. Similarly, the q_2 client applies its own extraction query to eliminate all elements less than 3. By merging q_1 and q_2, we reduce both the amount of work done by the server to process the query and the amount of information sent to the clients; however, this is at the expense of having to set up a multicast channel and having to post-process the messages at the clients.

In this paper we address the query merging tradeoffs. We present a framework for studying query merging (sometimes called logical-channel building) and its costs. We present a variety of algorithms for merging, some optimal, and some heuristic. We study the complexity of the algorithms. We use a simulation tool to evaluate their performance (i.e., time required for merging, and dissemination costs saved).

We start by presenting a motivating example (Section 2). We then specify our problem more formally (Section 3) and define our cost model (Section 4). Next, a specific scenario using geographic queries is considered (Section 5). The algorithms are presented in Sections 6 and 7 and their evaluation in Section 8.

2 Motivating Example

In this section, we motivate our techniques for reducing data dissemination cost by using the DARPA Battlefield Awareness and Data Dissemination initiative (BADD), which funded this work. The goal of BADD is to develop an operational system that delivers to combat troops an accurate, timely, and consistent picture of the battlefield and provides access to key transmission mechanisms and worldwide data repositories. Figure 1 outlines the relevant components of the BADD architecture.

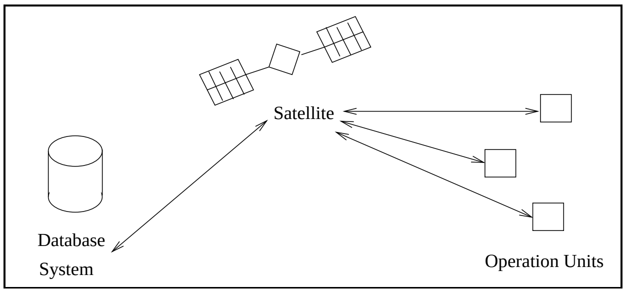

Figure 1: The BADD Scenario

In BADD, a database receives new information (e.g., satellite images, intelligence reports) from a set of *information sources*. The database also receives queries from *operational units*, answers the queries (on an on-going basis), and disseminates the answers. The operational units are limited capacity computers that can perform simple operations on the data received.

A common request in the BADD environment is information (troop presence, weather, topography, etc.) about a geographical area. For these requests, sources typically associate a geographical location with each object. For example, if the data source is a relational database, it may have the schema *R(longitude, latitude, attributes)*, where the pair *(longitude, latitude)* identifies the location, and *attributes* describes the object. This database can be visualized as in Figure 2(a). The dots in the figure represent the objects that have a given longitude and latitude. As stated before, operational units will query this database for objects inside a geographical area. For simplicity, we will assume that such area is a rectangle, defined by two coordinates (c_1, c_2) and (c_3, c_4). The queries over the database will have the form: $\sigma_{(c_1 \leq latitude \leq c_3) \wedge (c_2 \leq longitude \leq c_4)} R$. Figure 2(b) illustrates one of these queries. Although we will use this simple scenario as a running example, we want to stress that our algorithms can handle more complicated queries and database schemas.

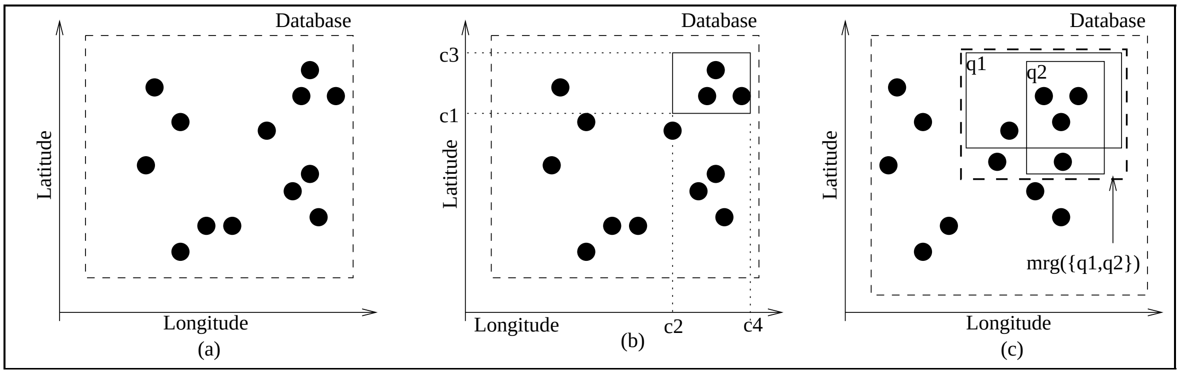

Figure 2: A Sample BADD Database and Queries

In Figure 2(c) we illustrate the merging of two BADD queries q_1 and q_2. Since these queries are very similar, it may be advantageous to merge them into a single query $mrg(\{q_1, q_2\})$. Note that the answer to the new query will contain objects that were not in the answer of q_1, or in the answer of q_2, or both. The operational units that receive the answer of $mrg(\{q_1, q_2\})$ must be able to derive from it the answers to q_1 and q_2.

There are many other applications that can benefit from query merging. For instance, a stock-market information service can use our setup. Specifically, the system will receive information (e.g., stock prices, news wires, press releases) from a set of information sources. Users can subscribed to the database according to their interest (e.g., certain stocks or stocks with some level of capitalization). A broadcast system would not be as effective since clients will receive much more information than they are interested in. On the other hand, processing each query independently will consume significant server and network resources. The solution is to merge similar queries and disseminate their results to the appropriate clients.

In this paper we do not consider security issues when merging query answers. For example, a client may not have the right to access all the data in a merged query. There are many ways of dealing with this problem. We may only merge queries from clients that are at the same security level. We can also encrypt each content of each object, so only clients that have the key to the object can see it. (However, in that case, we cannot use the encrypted attribute for the extraction procedure.)

3 Problem Specification

In this section, we formalize the query merging problem by presenting a detailed conceptual model. The objective of our approach is to reduce the cost of answering a set of query subscriptions made by clients to a server. We attempt to reduce the cost by finding a (possibly) different set of queries, with lower processing and transmission costs, from which the clients can derive the answers to their original queries.

3.1 Conceptual Model

The conceptual model for a query subscription service is shown in Figure 3. In this model, we have a set of clients, $C = \{c_1, ..., c_n\}$, that require information. The information need of c_i is described by a set of subscriptions. Each subscription consists of a query and its timing requirements (e.g., how often it should be run). For simplicity, we assume that all subscriptions have identical timing requirements. Thus, we can view the subscriptions of client c_i simply as a set of queries Q_i. We will assume that Q_i is relatively small but that its queries will be processed periodically (and answers sent to the client) over a long period of time. We call Q the set of all queries received by the server.

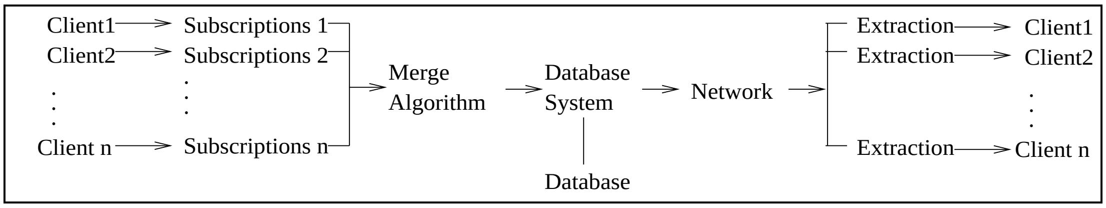

Figure 3: Subscription Service Conceptual Model

Clients send their sets of queries to a *server*. The server periodically processes the queries against a database, and sends *answers* to the clients. Before processing queries, the server runs a merge algorithm that combines "similar" queries. The output of the merge process is a collection $\mathcal{M} = \{M_i\}$ where each M_i contains the queries that are merged. The queries in each M_i are merged into a single query, $mrg(M_i)$. We use $ans(q)$ to represent the answer to query q. Thus, the server generates $ans(mrg(M_i))$ for each M_i in $\mathcal{M}$. For completeness, we require that $\cup_i Q_i = \cup_i M_i$. Similarly, we require that $ans(q) \subseteq ans(mrg(M_i))$ for every $q \in M_i$. We call the difference between the answer to the merged query sent to a client, $ans(mrg(M_i))$, and the original query, $ans(q)$, the *irrelevant information for q* sent to the client.

To illustrate these concepts, say client c_1 submits queries $Q_1 = \{x, y\}$, and c_2 submits $Q_2 = \{z\}$. The server may merge them into $M_1 = \{x, z\}$ and $M_2 = \{y\}$. Then the server runs $mrg(M_1)$ and $mrg(M_2)$ against the database and generates $A_1 = ans(mrg(M_1))$ and $A_2 = ans(mrg(M_2))$. Note that A_1 needs to be sent to both c_1 and c_2 and that each must apply an *extractor* to obtain the desired answer. For example, the extractor, e, that c_1 applies to A_1 should yield $e(A_1) = ans(x)$. Thus, when the server sends A_1 out, it must include a *header* containing the following information:

- A list of clients that should receive A_1.
- For each such client c, one or more pairs, (e, q), where e is an extractor and q is a query identifier. The extractor e is what client c needs to apply to obtain the answer to its original query q.

Note that more than one (e, q) pair is needed if multiple c queries are involved in A_1. If clients do not need to know what queries generated answers, then the query identifiers are not required. In our example, the information sent with A_1 would be $c_1 : (e_x, x)$ and $c_2 : (e_z, z)$. Client c_1 then applies $e_x(A_1)$ to obtain its answer to query x while c_2 applies $e_z(A_1)$ to obtain its answer.

There are many options for implementing extractors. For example, the server could tag each individual answer object with the identifier of the query that generates the object, or with the identifier of the client that should receive the object. Then each extractor only needs to look for the appropriate tags. In some cases, the extractor for a query is the original query itself. In particular, this happens when queries only have selections and projections. In any case, the extractor needs to be installed on the client machine. Above, we assumed that extractors were generated by the server and sent with answers. However, if the client can deduce its extractors (e.g., if the extractor is the original query itself), then the server need not send them.

Note that our basic model does not specify what kind of network is used to send queries to the server and answers to the clients. In the next section, when we discuss cost, we will introduce a multicast network into the model. This model is extended in Section 7 to include multiple channels.

4 The Cost Model

As we saw in Section 2, merging queries may or may not improve the performance of the system. To decide if merging is beneficial, we need an accurate and precise cost model that allows us to compare the "cost" of merging queries versus the cost of not merging those queries. In this section, we analyze the cost elements involved in our model and we integrate them in a cost formula.

As described in Section 3.1, the server receives a set of queries Q and outputs a set $\mathcal{M}$ where each of its elements is a set of queries to be merged. The query merging problem is to find the set $\mathcal{M}$ with the minimum cost. The input for the problem is a cost function $cost()$, a merge procedure $mrg()$, and a set of queries Q. The output is a collection $\mathcal{M}$ such that the total cost, $cost(\mathcal{M})$, is minimized.

The cost of processing the queries and sending the answers back is represented by the total resources consumed. The total resources consumed are the sum of all the resources used by the server, the network, and the clients. The costs involved in our model can be summarized as follows:

- Server cost to run the merging algorithm and to process the merged queries.
- Cost of transmitting the answers of the merged queries.
- Client cost of applying the extraction procedure.

In order to compute the resources used, we need to estimate the size of the query answers and the cost for computing them. Such estimate can be obtained using well-known database system techniques [28]. We use $cost(q)$ to denote the estimated cost of retrieving q's answer. The estimated total cost of retrieving all the answers (equal to $cost(mrg(M_1)) + cost(mrg(M_2)) + ... + cost(mrg(M_m)))$ will be denoted as $cost(\mathcal{M})$. We will denote the estimated size of q's answer as $size(q)$. The total size of all the answers (equal to $size(mrg(M_1)) + size(mrg(M_2)) + ... + size(mrg(M_m)))$ will be denoted as $size(\mathcal{M})$ (note that this is the total amount of data that the server needs to transmit to the clients).

As stated before, clients extract the answers to each of their queries *independently* by applying an extraction query to the messages they receive. The independence assumption means that a client may do some redundant work. For instance, consider a client that submits two queries that are then merged by the server. The client receives a single message containing both answers. It processes the message once to extract the answers for the first query, and then again to get the second set of answer objects. In extracting the first answer, the client will consider some objects as irrelevant, even though they will be later found to be relevant for the second query. We believe this independent processing model is the most realistic since clients are expected to be relatively simple and unable to process different queries concurrently. We will denote the size of the irrelevant information for query q_i by $u_i = size(mrg(M_j)) - size(q_i)$, provided $q_i \in M_j$. We will call $U(Q, \mathcal{M})$ the sum of all u_i ($U(Q, \mathcal{M}) = \sum_{q_i \in Q}(u_i)$).

The resources used by each component of the system can be computed as follows:

- Server cost: If the complexity of the merging algorithm is low, its cost will be insignificant in a subscription service. Therefore we will ignore the cost of executing the merging algorithm.

  The other component of the server cost is the time for computing and retrieving the answers from the database. The cost of computing the query answers will be denoted as $K_A \cdot cost(\mathcal{M})$, where K_A is a proportionality constant.

$Cost_{server} = K_A \cdot cost(\mathcal{M})$

- Network cost: Our network model assumes a multicast medium; namely, one where we can establish channels that allow sending data from one server to many clients. (In Section 7 we consider a multicast network with a *fixed* number of channels.) The network cost will be proportional to two factors. First, the network resources consumed are proportional (by factor K_T) to size of the data being transmitted ($size(\mathcal{M})$). Since we expect the size of the header and the size of the queries to be very small compared to the size of the data, we will ignore these when computing the size of an answer message. Second, in some cases we may need to establish network connections or "logical channels" for each M_i set. Messages then just include a logical channel id, and clients can subscribe to one or more channels. The cost of maintaining logical channels (e.g. table space in the routers, or operating system connection overhead) is proportional (by a factor K_M) to the number of merged queries transmitted. Thus,

  $Cost_{network} = K_T \cdot |\mathcal{M}| + K_M \cdot |\mathcal{M}|$.

- Client cost: As answers are multicast, clients need to spend resources receiving the information they want plus the irrelevant information added by the merging algorithm. These resources are proportional to the amount of data received. The total amount of relevant data received by the clients is $\sum_{q_i \in Q} size(q_i)$, while the total amount of irrelevant data is $U(Q, \mathcal{M})$. (Recall that queries are processed independently at each client, as we discussed earlier.) Therefore, the cost at the clients is equal to $K_U \cdot (\sum_{q_i \in Q} size(q_i) + U(Q, \mathcal{M}))$, where K_U is a proportionality constant. Note that when comparing merging alternatives, we can ignore the cost of listening to the relevant data since this cost does not depend on the merging algorithm and it will cancel out in the comparison. (Of course, when computing actual costs we need to consider both costs.) In the following expression we focus only on the differential cost between merging strategies.

  $Cost_{clients} = K_U \cdot U(Q, \mathcal{M})$.

Using the three cost components, we can compute the total cost as:

$Cost_{total} = Cost_{server} + Cost_{network} + Cost_{clients}$

$Cost_{total} = K_A \cdot cost(\mathcal{M}) + K_M \cdot |\mathcal{M}| + K_T \cdot size(\mathcal{M}) + K_U \cdot U(Q, \mathcal{M})$.

5 Geographic Queries

The framework and cost model presented so far is very general. The parameters in the cost model allow us to handle a wide range of capabilities in the servers and in the clients. We can model scenarios with clients ranging from very simple palm devices to sophisticated field computers. Similarly, we can handle servers having the functionality of a simple file system, to servers supported by a full fledged database. However, due to the limited space available in this paper, we will focus on a particular scenario. To illustrate how the merge procedure may operate and how the cost model can be used, we will use geographic query example presented in Section 2.

5.1 The Merging Procedure for Geographic Queries

As before, we consider the database to be a single relation R, that has position attributes (e.g., "latitude" and "longitude"), as well as other attributes describing that position. A geographic query has the form $\sigma_{(c_1 \leq latitude \leq c_3) \wedge (c_2 \leq longitude \leq c_4)} R$.

In Figure 4 we illustrate three different merge procedures that can be used in this geographic query scenario. In the figure, the solid lines represent the queries, and the dotted line represents the result of the merge procedure. Figure 4(a) shows the *bounding rectangle merging procedure*, the merging procedure introduced in Section 2. This procedure merges a set of 2-dimensional selection queries into a single 2-dimensional selection query. We can visualize this merged query as the smallest rectangle that bounds the original queries. The bounding rectangle merging procedure is very simple (and therefore fast to execute). Additionally, it is easy to extract the answers to the original queries from the answers to the merged query, as we just need to re-apply the original geographical query on the received answers. However, a disadvantage is that the answer includes objects that will be irrelevant to some or all of the input queries.

There are other possible merge procedures for the geographic query scenario. Figure 4(b) shows the *bounding polygon merging procedure*. This procedure also generates a single merged query, but, the query may have disjunctions. Although, the merge query contains less irrelevant information than the bounding rectangle merging procedure, irrelevant information is still present (the area of the polygon outside each query is irrelevant to the query). We can again use the original query as the extraction function for this merging procedure. Figure 4(c) shows a merge procedure that completely eliminates irrelevant information. However, five "merged" queries are generated. A client implementing the extraction function for this merging procedure needs to combine the answers to the five merged queries in order to find the answer to the original query.

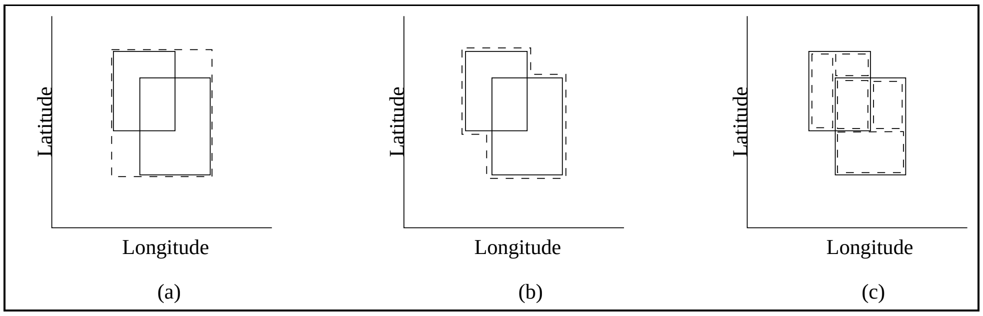

Figure 4: Three Different Merge Procedures

We close this section with a comment about our definition of "similar" queries. We are assuming that similar queries produce similar answers. In general, this assumption may not be true. Queries that look very similar may have different answers. To illustrate, consider the queries in Figure 4. We may have decided to merge the two queries because of the large overlap of the ranges of their selections. However, it may happen that the actual result of the queries is completely disjoint. Nevertheless, we believe that in practice queries with overlapping selection conditions will on the average have significantly overlapping results. It may also be possible to use data statistics and histograms to predict which queries will have more overlapping results. In summary, there are many choices for merge procedures that trade off complexity of the merged query, complexity of the extractor and amount of irrelevant information added.

In this paper, we have assumed that $|mrg(M)| = 1$. However, our model can be extended to the case when $|mrg(M)| > 1$ by taking the union of the answers in $mrg(M)$ to create a single answer.

5.2 Cost Model for Geographic Queries

In the geographic scenario, the queries are only selection and projection queries. In this case no intermediate results are generated to compute the answers. Therefore, with the use of clustering indices, the cost of the server is directly proportional to the number of messages and the size of the answers:

$Cost_{server} = k_1 \cdot |\mathcal{M}| + k_2 \cdot size(\mathcal{M})$

By incorporating k_1 and k_2 into the values of K_M and K_T, we can simplify the cost model to:

$Cost_{total} = K_M \cdot |\mathcal{M}| + K_T \cdot size(\mathcal{M}) + K_U \cdot U(Q, \mathcal{M})$. In Section 8 we illustrate how values for the model parameters can be obtained in a particular scenario.

5.3 The 2-Query Merging Problem for Geographic Queries

The 2-Query Merging Problem, is the special case of the query merging problem when $|Q| = 2$. Our geographic query example is convenient for illustrating why the 2 query merge problem is simple, but why it is hard for more than 3 queries.

In the 2-Query Merging Problem we want to decide if it is worthwhile to merge two queries q_1 and q_2 into a merged query q_3. For compactness, in the following discussion, let us denote $size(q_i)$ as S_i. Therefore, the cost of processing and transmitting queries q_1 and q_2 separately will be $K_M + K_T \cdot S_1$ and $K_M + K_T \cdot S_2$ respectively, for a total cost of $2K_M + K_T(S_1 + S_2)$. If we merge the queries into a single query q_3, the total cost will be $K_M + K_T \cdot S_3 + K_U \cdot U(Q, \mathcal{M})$, where $U(Q, \mathcal{M}) = 2 \cdot S_3 - S_1 - S_2$. We derive the $U(Q, \mathcal{M})$ term in the following way: if we send a message with only $ans(q_1)$, the client receives an answer with size S_1. If we send a message with $ans(q_3)$ instead, the client will receive an answer of size S_3. The difference $(S_3 - S_1)$ is the size of the irrelevant results received by the client. We can use a similar derivation for the other client and conclude that the size of the irrelevant information for the other client is $(S_3 - S_2)$. Therefore the total size of irrelevant information is $U(Q, \mathcal{M}) = 2 \cdot S_3 - S_1 - S_2$.

From these expressions, it is easy to derive a decision rule that tells us exactly when it is beneficial to merge two queries (this is, if the second cost we computed is less than the first cost). Therefore, it is beneficial to merge q_1 and q_2 if $K_M + K_T \cdot [S_1 + S_2 - S_3] + K_U \cdot [S_1 + S_2 - 2 \cdot S_3] > 0$.

Unfortunately, the general problem ($|Q| > 2$) is significantly harder, since there are many ways to combine a set of queries into merged queries. For example, if we have three queries as input, it could be the case that it is not worthwhile merging any pair of them, but it is worthwhile merging the three queries into a single query. On the other hand, it could be the case that it is worthwhile merging one specific pair, but not the other pair and not the three queries. In conclusion, we would have to consider all possible ways to partition the input queries into subsets. For each possible partition we compute a cost, and then we pick the partition with minimum cost. This approach leads to an exponential algorithm. In fact, in Appendix I we show that the query merging problem is NP-hard.

Let us use our geographical database scenario to show a case when merging three queries is optimal, although merging any pair is not. In Figure 5, we show three queries over our geographical database.

In the following discussion, we will assume that the answer of a query over each square unit in the diagram has size S and that we are using the bounding rectangle merging procedure. Therefore, $size(q_1) = size(q_2) = 2S$, $size(q_3) = S$, and $size(mrg(q_1, q_3)) = size(mrg(q_2, q_3)) = size(mrg(q_1, q_2)) = size(mrg(q_1, q_2, q_3)) = 4S$. Since there are three queries, there are five ways to merge then: we can merge 2 of them (3 combinations), we can merge all of them, or we can keep them separately. By deriving the

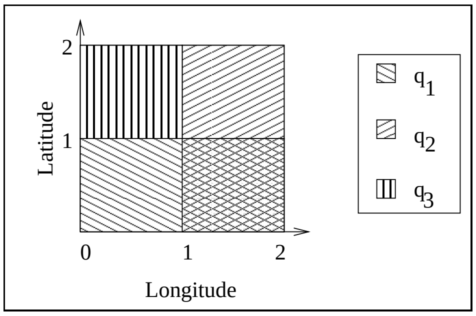

Figure 5: 3-Query Merging Example

costs of all the five possible ways of merging the queries, we conclude that merging all of the queries is advantageous, although merging any pair is not, when all of the following equations are satisfied:

$$S > \frac{K_M}{4K_U} \quad \text{and} \quad S > \frac{K_M}{5K_U + K_T} \quad \text{and} \quad S < \frac{2K_M}{7K_U - K_T}.$$

These equations are satisfiable; for instance, if we pick $S = 1$, $K_M = 10$, $K_T = 9$, and $K_U = 4$ all the equations will be true.

6 Algorithms for the Query Merging Problem

In this section we introduce heuristic algorithms for the query merging problem. However, we first briefly summarize two exhaustive algorithms that serve as reference points. We will compare the performance of all algorithms in Section 8.3.

6.1 Exhaustive Algorithm

An exhaustive algorithm for solving the query merging problem is presented in the first part of Appendix II. Unfortunately, exhaustive approaches have a doubly exponential complexity on the number of queries. This high complexity order makes exhaustive algorithms impractical for all but the smallest $|Q|$.

There exists a better algorithm for exhaustively solving the query merging problem when the cost model ensures the *single-allocation property*: each q_i in the solution is in one and only one element of $\mathcal{M}$. The single-allocation property means that if we want to process a set of queries $\{q_1, q_2, q_3, q_4\}$, we do not need to consider merged queries such as $\mathcal{M} = \{\{q_1, q_2, q_3\}, \{q_1, q_4\}\}$ where a q_i (in this case q_1) is in more than one element of $\mathcal{M}$. In the second part of Appendix II, we present the Partition Algorithm that exploits this property and has a complexity of $O(n^n)$

Although this may seem as a small improvement over the general exhaustive algorithm of the previous section; it significantly extends the values of $|Q|$ for which we can use an exhaustive algorithm. For example, if we have a limit of 5 minutes, we could find the optimal solution for up to $|Q| = 12$ using the Partition Algorithm, but only up to $|Q| = 4$ using the Exhaustive Algorithm.

6.2 Pair Merging Algorithm

The Pair Merging Algorithm takes a greedy approach to solve the query merging problem. The foundations of this algorithm are two simplifying assumptions. First, we assume that the cost model has the single-allocation property (as defined in Section 6.1). Second, and more important, we assume that pair-wise decisions (i.e., deciding which pairs of queries to merge) will lead to the correct global solution. The second assumption is in general incorrect (as shown in Section 5.3). However, the assumption allows us to efficiently obtain solutions that, in practice, are very close to the real "optimal" solution (see Section 8.3). At the end of this section, we will show that the complexity of the algorithm is $O(|Q|^2)$.

The Pair Merging Algorithm maintains a set of sets of queries. Initially, each set contains each single query. Then for all pairs of sets, the algorithm computes the change in the total cost if each pair is merged. The pair that produces the largest positive decrease in cost is chosen and the sets are replaced by their union. The algorithm continues picking and merging sets until no merging of any pair decreases the total cost.

We evaluate the cost achieved by merging sets using our cost model. For instance, for the geographic cost model we are using as our running example (Section 5.2), we can use a generalization of the formula in Section 5.3 for solving the 2-query merging problem. In particular, it can be shown that when merging two sets, M_a and M_b, containing the queries $\{q_{a_1}, q_{a_2}, ..., q_{a_p}\}$ and $\{q_{b_1}, q_{b_2}, ..., q_{b_r}\}$ respectively, the expression for solving the 2-query merging problem is:

$Cost_{sep} - Cost_{merge} = K_M + K_T \cdot (size(mrg(M_a)) + size(mrg(M_b)) - size(mrg(M_a \cup M_b))) + K_U \cdot \{p \cdot size(mrg(M_a)) + r \cdot size(mrg(M_b)) - (p+r)size(mrg(M_a \cup M_b))\}$.

Note, that we can obtain the expression for solving the 2-query merging problem given in Section 5.3 by making $size(mrg(M_a)) = S_1$, $size(mrg(M_b)) = S_2$, $size(mrg(M_a \cup M_b))) = S_3$, and the number of queries in each set equal to one ($p = r = 1$).

The Pair Merging Algorithm, as presented, needs to compute the cost of doing all possible merges in every step. However, in each step, only two of the sets (the ones that we decide to merge) have changed. The other sets remain the same so we can use all the computation involving them in the next step. Specifically, in step k of the algorithm, there are $(|Q| - k + 1) \cdot (|Q| - k)/2$ possible pairs; of those, only $|Q| - k - 1$ are new pairs. The rest were all candidate pairs that were considered in the previous iteration. Note, that the fact that those candidates were not chosen in a previous iteration, does not preclude them to be chosen later (as long as they have a positive benefit). To avoid computing the costs again for those sets, in each step, we save all computed costs in a *profit table*. Before computing the cost of merging a set, we check in the profit table to see if the cost was already computed; if it was, we take it from the table; otherwise, we compute it and add it to the table. After selecting the pair of sets to be merged, we remove all entries of the profit table that are related with those sets. Using the profit table, the number of cost model evaluations is $|Q|^2 + \sum_{i=1}^{|Q|-1} i - 1$. Therefore, the complexity of the algorithm is $O(|Q|^2)$.

6.3 Directed Search Algorithm

The Pair Merging Algorithm works in only one direction, that is, it starts with a set $\mathcal{M}$ where all the queries are single elements, and tries to merge those sets as much as possible. A potential weakness of this approach is that the Pair Merging Algorithm can be trapped into a local minimum of the cost function and miss the global minimum. This weakness is not unique of the Pair Merging Algorithm. Under a general cost function there are no polynomial algorithms that can avoid this weakness. However, in this section, we introduce the Directed Search Algorithm, a variation of the Pair Merging Algorithm that attempts to ameliorate this weakness.

The Directed Search Algorithm is based on two changes to the Pair Merging Algorithm. First, in addition to merging sets in $\mathcal{M}$, the Directed Search Algorithm may split one set in $\mathcal{M}$ into two sets, one containing only one element and another with the remaining elements. The rationale behind this change is that splitting sets, allows the algorithm to "undo" a bad decision made earlier. We limit one of the sets to have only one element to reduce complexity. If we allow a more general splitting function, the splitting step of the algorithm becomes exponential. The second change is to use multiple initial states and choose the one that leads to the minimum cost. In this way, if one initial state leads to a local minimum from which we cannot escape, there is a good chance that a different initial state will avoid that minimum.

Choosing the set of initial states, $\mathcal{T}$, is critical for the performance of the algorithm. The best set of initial states would be one that allows the algorithm to explore as much as possible of the search space. In our experiments, we included the initial states where all the queries are separate ($|\mathcal{M}| = |Q|$), the initial state where all queries are merged ($|\mathcal{M}| = 1$), as well as random partitions of Q. The algorithm is presented in Figure 6.

Input : $Q = \{q_1, q_2, ..., q_m\}$.
Output : Solution for the query merging problem.
Create set $\mathcal{T}$ of initial states.
For each initial state $\mathcal{S}$ in $\mathcal{T}$:
    $\mathcal{M} = \mathcal{S}$
    Repeat:
        For each pair of sets M_i, M_j in $\mathcal{M}$:
            Compute the cost of $(\mathcal{M} - M_i - M_j) \cup (\{M_i \cup M_j\})$.
        For each query q of each set S in $\mathcal{M}$:
            Compute the cost of $(\mathcal{M} - S) \cup \{S - \{q\}\} \cup \{\{q\}\}$.
        Perform the operation that reduces the total cost the most.
    Until no beneficial operation is possible.

Figure 6: Directed Search Algorithm

The worst case complexity of the algorithm is $O(|\mathcal{T}||Q|^{|Q|})$. This is because it is possible for the algorithm to explore the entire search space. Nevertheless, the algorithm is guaranteed to finish, as it only advances when there is a lower cost option. Although, the worst case performance is exponential, in our experiments this bound was never reached. Furthermore, in the experiments the algorithm showed a polynomial average running time.

6.4 Clustering Algorithm

The Clustering Algorithm takes a "divide and conquer" approach to the query merging problem. The foundation of the algorithm is the definition of a "distance" metric between queries. Basically, if the distance between two queries is "far enough," we can ignore all combinations of merged queries that contain those queries. In this section, we will describe the algorithm, independently of the distance metric. In the following section, we will define the distance metric and introduce concrete examples.

A graphical intuition of the Clustering Algorithm is presented in Figure 7. In the figure there are five queries, q_1 to q_5. Queries q_1, q_3, and q_5 are very close together, and therefore are good candidates for merging among themselves. However, queries q_2 and q_4 are far and it may not make sense to even consider merging with them. Specifically, the Clustering Algorithm works by computing the "distance" between each pair of queries and if this distance is below a certain threshold, it puts the two queries in the same cluster. In the figure, the algorithm may start by finding that the distance between the q_2 and q_4 is below the threshold and therefore should be in the same cluster (drawn as a dotted line). Then, the algorithm may find that q_1 and q_3 are also below the threshold and will put them in the same cluster. Then, the algorithm may find that the distance between q_5 and q_3 is also below the threshold, but because q_3 already belongs to a cluster, the algorithm adds q_5 to that cluster, instead of creating a new one. If q_5 were close to several clusters (that is, q_5 is close to at least a member of each of those clusters), a single cluster would be formed containing q_5 and all of those clusters.

After this, the algorithm cannot find more pair of queries with distances belong the threshold and the clustering phase ends. In the following phase, we need to apply within each cluster any of the algorithms previously studied to merge its queries.

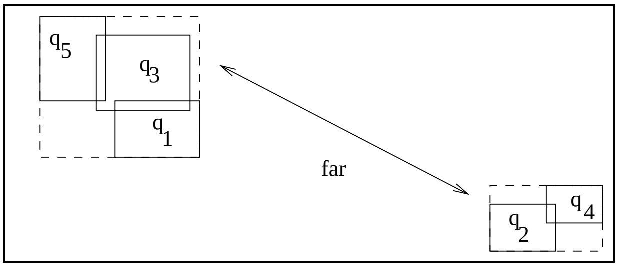

Figure 7: Clustering Algorithm Scenario

As the Clustering Algorithm considers pairs of queries, its performance is the greater of $O(|Q|^2)$ and $O(mrg_{alg}(|Q'|))$, where mrg_{alg} is the performance of the merging algorithm used in the second phase and Q' is the cluster with the maximum number of queries. Obviously, the clustering algorithm works well only if there are clusters in the data. There are two cases for non-clustered data: queries will either be very close together or they will be very far apart. In the first case, the algorithm will identify just a single cluster that contains all the queries (and therefore will not improve the running time of the second phase). In the second case, the algorithm will indicate that the are are not opportunities for merging queries, and the second phase need not be run.

In the next section, we will define the distance metric. Depending on this metric, the Clustering Algorithm behaves as an exact or as a heuristic algorithm. Due to limited space, we will only present a heuristic distance metric. In Appendix III, an exact metric is presented.

6.4.1 A Heuristic Distance Metric

Given queries q_1 and q_2, we want to determine if it will ever be advantageous to place them in the same partition, even if q_1 or q_2 have already been merged with other queries. To check, we can compute the maximum benefit that may be obtained by combining q_1 (or a merged query containing it) and q_2 (or a merged query containing it). In doing so, we will assume that costs not directly related to q_1 and q_2 are not affected by the decision (this may be false). If the maximum benefit is non-negative, we place q_1 and q_2 in the same cluster; otherwise, we leave them separate (though they may be eventually be combined due to another query that is close to both). This gives us a heuristic rule, but as we will see, our experimental results show it performs very well.

To illustrate, consider the cost model for geographic queries. To obtain the maximum possible benefit, we consider three components:

- $K_M \cdot |\mathcal{M}|$: By merging two queries together, we reduce the size of $|\mathcal{M}|$ by one. Note that we are ignoring the possible effects on $\mathcal{M}$ after merging the two queries (which may reduce $|\mathcal{M}|$ even more).
- $K_T \cdot size(\mathcal{M})$: In the best case (this is when the result of q_1 is contained in the result q_2), the benefit $size(\mathcal{M})$ will be at most $min(size\{q_1\}, size\{q_2\})$.
- $K_U \cdot U(Q, \mathcal{M})$: In the best case, $U(Q, \mathcal{M})$, the benefit will be reduced by $2 \cdot size(mrg(\{q_1, q_2\})) - size(\{q_1\}) - size(\{q_2\})$.

Therefore, we should leave queries in separate clusters when:
$K_M + K_T \cdot min(size\{q_1\}, size\{q_2\}) - K_U \cdot 2 \cdot size(mrg(\{q_1, q_2\})) - size(\{q_1\}) - size(\{q_2\}) < 0$.

7 Query Merging in a Multicast Network With a Fixed Number of Channels

In Section 4 we assumed a multicast network with an unlimited number of channels. However, in some cases, there may be a limit in the number of channels. This limitation may be the result of the physical properties of the transmission mechanism (e.g., a satellite network like the one described in the BADD scenario) or the result of network management decisions (e.g., we are only allowed to use certain number of channels). In this section, we extend our network model to one that disseminates data through a fixed number of multicast channels.

7.1 Problem Description

As described in Section 3.1, the server receives a set of queries Q and outputs a set $\mathcal{M}$ where each of its elements is a set of queries to be merged. The answers for each merged query, $ans(mrg(M_i))$, is then allocated to a multicast channel where the interested client would read it. However, we have now a limited number of channels available, $\mathcal{C} = \{Ch_1, Ch_2, ..., Ch_z\}$, to disseminate answers. If $|\mathcal{M}| \leq |\mathcal{C}|$ then we proceed as before. Otherwise, we need to assign more than one merged query answer to each channel. When listening to a channel, a client needs to check all the transmitted data to identify the merged query answers of interest. Discarding merged queries that were intended for other clients increases significantly the overhead of listening to a channel, so we will only require a client to listen to one channel. We call the problem of assigning channels to merged query answers the *channel allocation* problem.

A first approach to improve query subscription processing in this environment would be to solve the Query Merging Problem first and then the channel allocation problem. However, as we will see in the next example, this approach does not lead to an optimal solution.

Let us suppose that we have three clients c_1, c_2, and c_3 that have subscribed to query sets $\{q_1, q_2\}$, $\{q_3, q_4\}$, and $\{q_5\}$ respectively. Suppose that after running the merging algorithm, we obtained the set $\mathcal{M} = \{M_1, M_2, M_3\}$ where $M_1 = \{q_1, q_5\}$, $M_2 = \{q_2, q_3\}$, and $M_3 = \{q_4\}$. Let us consider a system with two channels, $\mathcal{C} = \{Ch_1, Ch_2\}$. We will describe in detail the cost model for channel allocation, but for now, let us assume that we want to send the minimum number of query answers through each channel. Under this criteria, an optimal allocation would be to send $ans(mrg(M_1))$ and $ans(mrg(M_2))$ through Ch_1, and $ans(mrg(M_2))$ and $ans(mrg(M_3))$ through Ch_2. Note that this solution requires each client to listen to only one channel (c_1 needs to listen only to Ch_1, c_2 only to Ch_2, and c_3 only to Ch_1).

Although this solution meets all our criteria, there is a better solution if we do not start with the merged queries. In fact, note that the answers of q_3 in $ans(mrg(M_2))$ are not required by any client on Ch_1. Similarly, the answers to q_2 in $ans(mrg(M_2))$ are not required by any client on Ch_2. Thus, we could reduce the number of query answers transmitted by not merging q_2 and q_3 and simply transmitting $ans(q_2)$ in Ch_1 and $ans(q_3)$ in Ch_2.

In conclusion, we cannot consider the query merging problem and the allocation problem in isolation. Therefore, we need to develop new algorithms that are able to do channel allocation and query merging at the same time. We also need a new cost model to be able to evaluate different allocation alternatives and to identify savings from merging queries.

We will call the combined allocation and query problem, the query merging and allocation problem (QMA). Formally, the QMA problem is to find (i) channel allocation $[C_1, C_2, ..., C_z]$ where C_i is the set of clients that are allocated to channel Ch_i, and (ii) for each channel the set $\mathcal{M}_i = \{M_1, M_2, ...\}$ where each M_i is the set of queries of clients in Ch_i that are merged, such that the cost of answering the queries of each client is minimized.

7.2 Cost model for a Network with Limited Number of Multicast Channels

As a first step for solving the QMA problem, we need to define precisely the cost model. As before, the cost includes the server cost to run the merging algorithm and to process the merged queries, the cost of transmitting the answers of the merged queries, and the client cost of applying the extraction procedure. But now, we also have to consider the client cost of going through the data sent on the channel to identify the merged query answers of interest.

We will compute the total cost as the sum of the cost of each channel (this sum may be weighted if channels have different costs or properties). Given this way of computing the total cost, we can concentrate on computing the cost of a single channel. Let us assume we have allocated clients $C_i = \{c_1, c_2, ..., c_p\}$ to channel Ch_i. Let Q_c denote the set of queries sent by client c and that a merging algorithm over the queries of those clients resulted in the set $\mathcal{M}_i$. Then, the cost of the channel will be the four factors discussed in Section 4 (with weights K_A, K_M, K_T and K_U) and a new factor (with weight K_S) representing the cost of *sharing* the channel with multiple merged queries. We will model this sharing cost through the amount of data contained in the queries in which each client is not interested. Specifically, the amount of data contained in the merged answers that a client c receives through channel Ch_i is $size(\mathcal{M}_i)$. However, the amount of data in the merged query answers that client c is interested is only $\sum_{q_j \in Q_c} size(q_j)$. Therefore, we can compute the cost of sharing a channel as $K_S(size(\mathcal{M}_i) - \sum_{q_j \in Q_c} size(q_j))$.

As a result, the overall cost associated with channel Ch_i will be:

$$Cost_i = Cost_{server} + Cost_{network} + Cost_{clients} + Cost_{sharing}$$

$$Cost_i = K_A \cdot cost(\mathcal{M}_i) + K_M \cdot |\mathcal{M}_i| + K_T \cdot size(\mathcal{M}_i) + K_U \cdot U(Q, \mathcal{M}_i) + K_S \cdot \sum_{c \in C_i} (size(\mathcal{M}_i) - \sum_{q_j \in Q_c} size(q_j)).$$

And the total cost would simply be the sum over all the channels:

$$Cost_{total} = \sum_{1 \leq i \leq ch} Cost_i$$

7.3 Algorithms for Channel Allocation

In this section, we will introduce an exhaustive algorithm and a heuristic algorithm to solve the QMA problem. As we saw in the previous subsection, we need to consider the allocation and merging problem simultaneously. Therefore, the algorithms iterate between an allocation phase and a merging phase. During the merging phase, we can use any of the algorithms of Section 6.

7.3.1 Exhaustive Algorithm for Channel Allocation

The exhaustive solution of the QMA problem is to evaluate all possible allocations of clients to channels Given clients $C = \{c_1, c_2, ..., c_p\}$, we generate all cases by building a search tree where the leaves represent all possible allocations of clients to channels. Then, we compute the cost of each of those allocations and choose the one with the lowest cost. The algorithm is presented in Figure 8.

In the algorithm, we associate with each node N of the search tree a list of sets of clients $N.L =$

```
Input : C = {c_1, c_2, ..., c_p}, ch, merging procedure.
Output : Optimal allocation of clients to channels.

Create root node N at level 1 with N.L = [{c_1}].
For h = 1 to |C| - 1
        For each node N at level h with N.L = [C_1, C_2, ..., C_|N.L|]
                Create children N_0, ..., N_{|N.L|-1} of N with N_i.L = N.L
                For each child N_i
                        Add c_{h+1} to the ith element of N_i.L
                If |N.L| ≤ ch then
                        Create child N_|N.L| of N with N_|N.L|.L = N.L
                        Append {c_{h+1}} to N_|N.L|.L
For each leaf node N
        Run merging procedure.
        Evaluate the cost for the allocation N.L.
The leaf with the minimum cost is the optimal allocation.
```

Figure 8: The Exhaustive Algorithm for Channel Allocation

$[C_1, C_2, ..., C_m]$ where C_i is the set of clients allocated to channel Ch_i. The sets of clients associated with the nodes at level h contain all possible combinations for the clients c_1, c_2, ..., c_h allocated to the channels.

We compute the cost of answering the queries sent by clients C_i through channel Ch_i by using our cost model. The total cost at leaf N is computed by summing up each individual cost associated with the sets of clients $N.L$. The leaf with the minimum cost gives the allocation that is the optimal solution.

If there are $|C|$ clients, ch channels, and the complexity of the merging procedure is O_M, the complexity of this algorithm is $O(ch^{|C|}O_M)$ since each client can be allocated to ch channels and for each allocation we run the merging algorithm once.

7.3.2 The Channel Merging Algorithm

The Channel Merging algorithm is a heuristic algorithm for solving the QMA problem. The algorithm starts with an initial allocation of clients to channels. Then it reallocates the clients by moving them between channels in the direction of decreasing total cost. The algorithm uses the hill climbing technique in which we move through the search space in the direction that reduces the cost most. It may not find the optimal solution since it can get trapped at a local minimum in the search space.

The algorithm is divided in two parts. First, we create an initial allocation where clients that have more overlapping query answers are in the same channel. Second, we attempt to decrease the cost of that allocation by moving clients to other channels.

```
Input : C = {c_1, c_2, ..., c_p}, ch
Output : Initial allocation of clients to channels.

L = {}
CCh = 0
For each pair c_a and c_b in C (c_a ≠ c_b)
        Calculate Cost_Δ of c_a and c_b.
        Store the triplet [c_a, c_b, Cost_Δ] in L.
While L is not empty:
        Pick triplet [c_a, c_b, Cost_Δ] in L with maximum Cost_Δ.
        Allocate clients c_a and c_b to channel CCh + 1.
        Delete from L all triplets that have either c_a or c_b.
        CCh = CCh + 1(mod ch)
While there is a client c_i that is not assigned to any channel:
        Allocate c_i to channel CCh + 1.
        CCh = CCh + 1(mod ch).
```

Figure 9: The Algorithm for Producing the Initial Allocation

To find the initial allocation, we first compute the potential saving of allocating each pair of clients to the same channel. The saving, $Cost_{\Delta}$, of allocating clients c_a and c_b to the same channel is $Cost_{\Delta} = Cost_{\{c_a\}} + Cost_{\{c_b\}} - Cost_{\{c_a,c_b\}}$, where $Cost_S$ is the cost when the clients in S are all allocated to a single channel (ignoring all other clients that may be allocated to that channel). This cost is computed with the formula described in Subsection 7.2. We choose the pair, say c_a and c_b, with the highest savings and we allocate them to the first channel. Then, we remove from consideration all pairs that have c_a or c_b as members, and, among the remaining pairs, we choose the one with the highest $Cost_{\Delta}$ and we assign that client pair to the second channel. We continue doing this iteratively, advancing one channel at at time (and returning to channel 1 when no more channels are available), until we have considered all clients. The algorithm to initialize the allocation is presented in Figure 9. After finding the initial allocation, we decrease the total cost further by iteratively making changes to that allocation with the algorithm presented in Figure 10.

If there are $|C|$ clients and the merge algorithm has a complexity of O_M, then the first phase of the algorithm (which computes the initial allocation) has a complexity of $O(|C|^2)$ as the algorithm considers all pairs of clients. The worst case complexity of the second phase of the algorithm is $O((O_M|C|)^{ch})$ as it is possible for the algorithm to explore the entire search space. Nevertheless, the algorithm is guaranteed to finish, as it only advances when there is a lower cost option. Although, the worst case performance is exponential, in our experiments this bound was never reached. Furthermore, we speed up the algorithm

Input : $C = \{c_1, c_2, ..., c_p\}$, ch, merging procedure
Output : Allocation of clients to channels.

Use algorithm in Figure 9 to obtain an initial allocation $L = \{L_1, ...L_{|L|}\}$,
where each L_i is a set of clients allocated to channel Ch_i.
Repeat:
  $S = \{\}$
  For each client c_i and channel Ch_j ($c_i \in L_k$, $k \neq j$):
    Run a merging procedure among the queries from clients in L_k.
    Run a merging procedure among the queries from clients in L_j.
    OldCost = Cost of L_k + Cost of L_j
    Run a merging procedure among the queries from clients in $L_k - \{c_i\}$.
    Run a merging procedure among the queries from clients in $L_j \cup \{c_i\}$.
    NewCost = Cost of $L_k - \{c_i\}$ + Cost of $L_j \cup \{c_i\}$
    Saving = NewCost$-$OldCost
    Insert triplet $[i, j,$Saving$]$ in S.
  Find triplet in S with the highest positive Saving.
  If there is such a triplet then move client c_i to channel Ch_j.
Until there are no triplets in S with positive Saving.
Allocate clients in L_i to channel Ch_i.

Figure 10: The Channel Merging Algorithm

significantly by caching the cost of each channel allocation (without this optimization we need to merge and evaluate each allocation $4|C|ch$ times per iteration, but with the optimization we only need to do it $8|C|$ times per iteration).

8 Performance Evaluation

In this section, we test the efficiency of the algorithms developed for the case of a single channel. (Due to space limitations, we cannot present the performance evaluation of the multiple channel version of the algorithms.) In order to test the efficiency of the algorithms developed, a simulator has been implemented for geographic queries. It simulates an environment in which the queries are given on a two-dimensional database (see Figure 11). The database elements consist of two search attributes and the queries received by the server are range queries. The simulator consists of three main modules. The first module provides input to the simulator. The user specifies certain parameters like the dimensions of region covered in database, the size and number of queries, the cost parameters (K_M, K_T, K_U) and the merge algorithm used. Given these parameters, this module generates a set of queries which is used as an input for the algorithms. The second module runs one of the algorithms described previously (i.e., Pair Merging, Directed Search, or Clustering algorithms). Finally, the last module evaluates the savings of the heuristic algorithms and quantifies their deviation from the optimal solution.

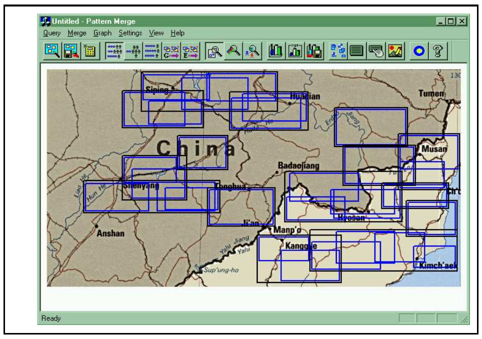

Figure 11: Screen-shot of the Simulator

8.1 Generating Input

In most environments, it is quite likely that the input given by clients generates a pattern which creates groups of queries that are located near each other. Some portions of the database are likely to be accessed more frequently than others. For instance if the database specifies atmospheric conditions, the portions of the database denoting the regions with higher population are more likely to get queries. Similarly, in a battlefield situation, the number of queries in regions which have more combat troops will be much higher. Therefore, rather than generating random input queries, a clustering effect has been added to the input generating module, in which some queries tend to create small clusters.

Queries are generated in two ways; randomly and using clustering. The parameter cf is the ratio of the number of queries generated using clustering to the number of total queries. So $cf|Q|$ gives the number of queries generated using clustering and $(1 - cf)|Q|$ gives the number of queries generated randomly. The parameter sf is the ratio of number of queries in a cluster to the number of queries generated using clustering. We can compute the number of queries in a cluster and the number of clusters by $sf \cdot cf \cdot |Q|$ and $1/sf$ respectively. For each cluster, a cluster origin is generated randomly in the database. The distances of the queries in a cluster to the origin is computed using a normal distribution $N(0, df^2)$ where df is the density of the clusters. The direction of each query from the cluster origin is generated randomly (i.e., a random value between $0°$ and $360°$). Another input parameter gives the size of the queries. The minimum and maximum ranges for both attributes are given. The size of each query is selected randomly in these ranges.

8.2 Cost Parameters

To select values for our cost parameters K_M, K_T and K_U, we need to consider the specifics of the system we are studying, e.g., how costly it is to transmit data, relative to the other costs. The parameters impact not just the solution obtained, but how hard it is to find the optimal solution. For instance, there are some values for which it is trivial to find the optimal solution (consider $K_M = 1$, $K_T = 0$, $K_U = 0$). For other values, algorithms such as Pair Merging may not find the optimal solution.

In this subsection we briefly illustrate how these parameters can be estimated in a given scenario. Please keep in mind that this is simply an illustration. We measure costs in dollars, since this makes it easier to compare processing and network costs.

At the server, we can first estimate the dollar cost of one second of processing as follows. We estimate that the server and its database system cost $\$100,000$, and this system will be amortized for 2 years, giving a cost of $\$50,000$ per year. In addition, operating costs are $\$50,000$ per year, say. Thus, the cost per server second is

$C_S = (50,000[dollar/year] + 50,000[dollar/year])/31,536,000[sec/year]$

$C_S = 0.003171[dollar/sec]$

(In this and the expressions that follow we show the units in square brackets.) We test the server with queries that yield no answer (null queries), and discover that the server can process 1 query per second. We then test queries with different result sizes and discover that each additional answer object adds 1/100 second to the query time. Thus, we estimate the dollar cost at the server as

$Cost_{server} = C_S[dollars/sec] * (1[sec/query] * |\mathcal{M}|[query] + 0.01[sec/object] * size(\mathcal{M})[object])$

For the network, we compute the dollar cost per megabyte transmitted using numbers for a DSL service provided by Stanford. A DSL modem including installation cost is \$1165, or \$48.50 per month if we amortize over 2 years. The monthly fee is \$235, and the maximum bandwidth is 1.1Mbs. This gives us a dollar cost per MB of

$C_N = (48.50[dollar/month] + 235[dollar/month])/((1.1Mbs/8[b/B]) * 2592000[sec/month])$

$C_N = 0.0007956[dollar/MB]$

Using the results of [8], we estimate that setting up the connection for each query answer consumes 100KB. We also estimate that each answer object is 1KB in size. Hence, the network cost is

$Cost_{network} = C_N[dollar/MB] * (0.1[MB/query] * |\mathcal{M}|[query] + 0.001[MB/object] * size(\mathcal{M})[object])$

For computing client costs, we assume the client is a handheld device, and that the major cost is due to battery use. Using the PalmPilot as an example, a daily usage of 30 minutes results in a battery lifetime of 2 months. In other words, we get 1800 minutes of processing per battery change. We estimate the cost of batteries (including labor) at \$5. Thus, the cost per second is

$C_C = 5[dollars/replacement]/(1800[min/replacement] * 60[sec/min])$

$C_C = 0.0000463[dollar/sec]$

By performing experiments, we estimate that processing each answer object, whether useful or not, takes 0.1 second of processing on the client device. Thus, the client cost is

$Cost_{client} = C_C[dollar/sec] * 0.1[sec/object] * (\sum_{q_i \in Q} size(q_i)[object] + U(Q, \mathcal{M})[object])$

Combining the constants we have estimated, we obtain the following values for the overall cost model: $K_M = 0.003251$, $K_T = 0.000032$, $K_U = 0.00000463$.

As we have stated, our main objective in this section has been to illustrate the process by which one can estimate these constants. The performance experiments that must be performed to estimate costs, and the actual values obtained, can of course vary, but in the end, one can obtain cost proportionality constants that make it possible to compare the costs incurred at each stage of the multicast process.

8.3 Experiments

In this section we study the performance of the Pair Merging, Directed Search, and Clustering algorithms. Sample geographical queries were generated and both the exhaustive and heuristic algorithms were used to evaluate the results. Obviously the exhaustive algorithms give the optimal solution for a given set of queries. In order to evaluate the efficiency of our algorithms, we wish to address the following questions:

- What is the probability that the heuristic algorithms find the optimal solution?
- If the algorithms do not find the optimal solution, how far are the solutions to the optimal ones?
- In what scenarios does query merging pay off, and what are the potential gains?

Since we want to focus on how well our algorithms perform, rather than on predicting performance of a particular system, we select a scenario where it is particularly hard to find the optimal solution, and where we will stress the algorithms. This scenario was obtained by running the simulator over many different sets of parameters, and selecting the values (for cf, sf, df, K_M, K_T, K_U) where the heuristic algorithms found solutions further from the optimal. Thus, the results we will present here are *pessimistic* for the heuristic algorithms. As we will see, the results are rather good for the high-stress scenario, so this means the algorithms will perform even better in almost any other scenario.

The input parameters and their base values are given in Figure 12. The values used for K_M, K_T, K_U are close to those illustrated in Section 8.2, but adjusted slightly to stress the algorithms more. Incidentally, note that only the ratio between K_M, K_T, K_U matters when comparing algorithms. If we multiply each of

Cost Parameters	
Parameter	*value* $\times 10^{-5}$
K_M	365
K_T	3.25
K_U	0.16

Query Generating Parameters	
Parameter	*value*
cf	0.80
sf	0.30
df	100
$\lvert Q\rvert$	100
Maximum Query size	40x40
Minimum Query size	20x20

Other Parameters	
Parameter	*value*
Database Size	
$\lvert Q\rvert \leq 12$	100x100
$\lvert Q\rvert > 12$	400x400
Sample Size	10000
$\lvert \mathcal{T}\rvert$	50

Figure 12: Base values

these values with a constant factor, the overall cost would change but the goodness of the solutions would not change. In our experiments, we use two different database sizes depending on the number of queries. When having a small number of queries, we used a smaller database size, so queries were not so dispersed that merging was never advantageous. The *Sample Size* specifies the number of times the simulator was run to generate the results shown in the graphs.

In Figure 13, we can see the sensitivity of the Pair Merging Algorithm to the cost parameters K_M, K_T and K_U. The y-axis gives us the resulting number of merged queries ($|\mathcal{M}|$). The parameter values on the x-axis are normalized. For instance, in the central graph in Figure 13, the number 1.5 on the x-axis indicates that K_T was chosen to be 1.5 times the base value, i.e, $1.5 \cdot 0.0000325 = 0.00004875$. As we can see from these graphs, the algorithm seems to be most sensitive to K_M since the overall cost is directly related to $|\mathcal{M}|$. We can also observe that for high values of K_T, a change in this parameter does not have a great influence on the algorithm since the network cost dominates.

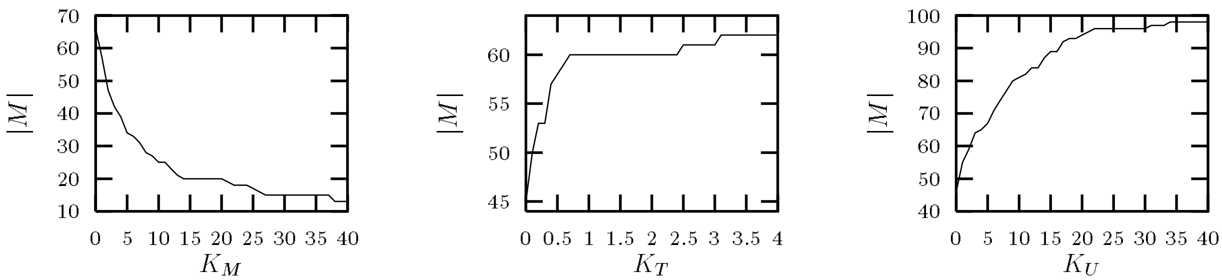

Figure 13: Sensitivity Analysis

In Figure 14, the performance of the Pair Merging and the Directed Search algorithms are compared. This graph gives the fraction of the runs where the Directed Search Algorithm performs better than the Pair Merging Algorithm. The x-axis indicates the number of initial states for the Directed Search Algorithm.

For instance, if we use 50 initial states, Directed Search finds a better solution than Pair Merging in about 20% of the cases, and in the remaining 80% of the cases both find the same solution. Note that Pair Merging never finds a better solution since it considers a subset of the merge configurations considered by Directed Search. Thus, the figure quantifies how much better Directed Search becomes as we increase the number of initial states ($|\mathcal{T}|$).

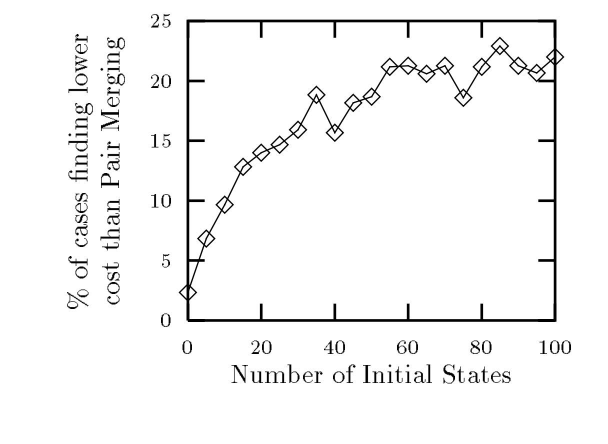

Figure 14: Performance of Directed Search Algorithm Compared to Pair Merging Algorithm

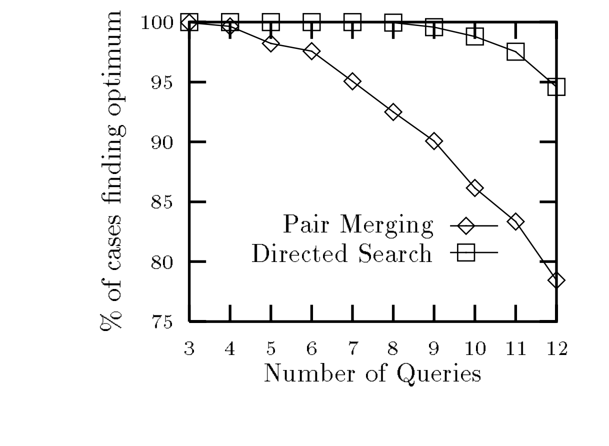

Figure 15: Percentage of Cases Finding the Optimal Solution

Figure 15 shows the fraction of runs where the Pair Merging and Directed Search algorithms find the optimal solution (as found by the Partition Algorithm). We only consider 12 queries or less as the Partition Algorithm must evaluate as many combinations as the $|Q|$th Bell number. For $|Q| = 12$, this is 4,213,597 combinations, and beyond that it grows to unmanageable sizes. We omitted the trivial case $|Q| = 2$, as both algorithms are guaranteed to find the best solution.

As we can see, the chances of reaching the optimal solution decreases as the number of queries increase in both algorithms. The Directed Search Algorithm is more likely to reach the optimal solution. Extrapolating the curves, the results seem to imply that for large numbers of queries, it will be very unlikely that the optimal solution is found with heuristic algorithms. This is bad news, except that our next graphs will show that the deviation from optimal is very small.

To compute the deviation from optimum, let us say that the cost of disseminating a given set of queries without any merging is $Cost_{initial}$. Let $Cost_{optimal}$ be the optimal cost obtained by an exhaustive algorithm, and let $Cost_{heuristic}$ the cost reached by a heuristic algorithm. We measure the distance of the heuristic solution to the optimal solution as follows:

$$Distance = 100 \cdot \frac{Cost_{heuristic} - Cost_{optimal}}{Cost_{initial} - Cost_{optimal}}$$

This formula gives the deviation from optimum, relative to the maximum costs that may be saved through merging. For instance, a value of 0.0% indicates the solution is optimal, and a value of 100.0% indicates the cost is the same as with no merging.

Figure 16 shows the distance of the solutions. As expected, the distance for Pair Merging and Directed Search increase as we increase the number of queries. The Directed Search Algorithm has distances equal to zero for $|Q| \leq 7$ because it is acting almost as an exhaustive algorithm. Recall that, in our experiments, we used 50 initial states for the Directed Search Algorithm. The total number of cases for $|Q| = 3$ to 7 ranges between 5 and 877; thus, there is a high probability that Directed Search will be able to search the whole space. As the number of queries increase, the Directed Search Algorithm is no longer able to do an exhaustive search and its distance increases.

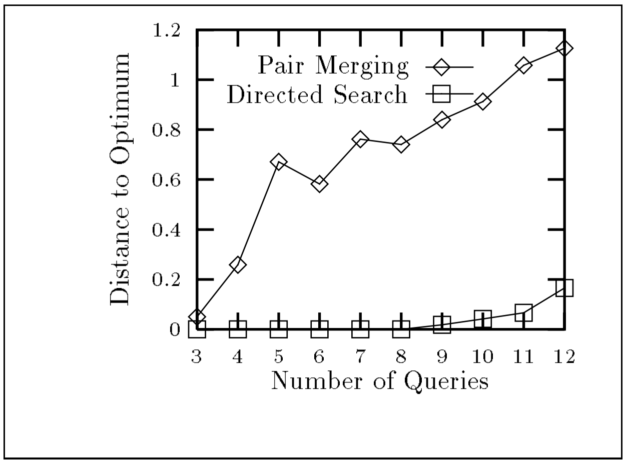

Figure 16: Distance to Optimum of the Pair Merging and Directed Search Algorithms

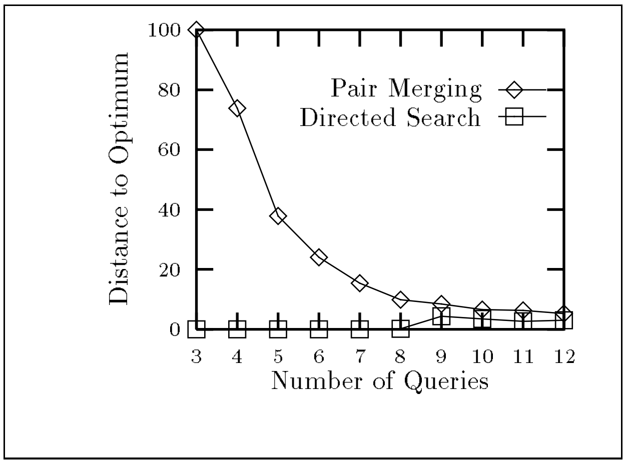

Figure 17: Distance of Suboptimal solutions to Optimum

Although we would need more data points to confirm this, we hypothesized that the the curves in Figure 16 have a logarithmic growth. So we expect that the heuristic algorithms will continue to have small distances even for relatively large number of queries. Furthermore, recall that we selected our base parameters to stress our algorithm, so they are likely to perform much better in most other scenarios. Each data point in Figure 16 gives the distance averaged over all 10,000 runs.

In Figure 17, on the other hand, we show the average distance for those runs where the optimal solution was *not* found. (Note the change of scale.) The distance for Directed Search *increases* a bit around 9 queries, but this is simply because at this point the algorithm stops being close to exhaustive. Beyond 9 queries, the distance for Directed Search should start decreasing. To see this, recall that the distance of Directed Search will always be lower or equal to that of Pair Merging. Since Pair Merging shows a clear decreasing trend, Directed Search must also be decreasing. This is good news, for it predicts that distances (errors) will be small for large numbers of queries. As we increase the number of queries, the search space becomes huge, but the number of solutions that are very close to the optimal one also grows rapidly, so the heuristic algorithms have an excellent chance of finding a very good solution. Again, recall that our base scenario is one where it is hard to find the optimal solution, so in many other cases, distances will be even smaller.

In Figure 18 we can see the total cost after merging obtained by the Pair Merging, Directed Search and the Clustering with Heuristic Distance Metric algorithms. Since the costs given by the Pair Merging and Directed Search algorithms are very close, we showed them as a single line. To compute the cost of the Clustering Algorithm, the Pair Merging Algorithm was run on each of the clusters generated. The *no merge cost* is the cost of processing the queries without any merging. If our cost parameters are based on dollars (see Section 8.2), then the costs in Figure 18 are in dollars. The actual costs shown are small (a few dollars), but keep in mind that this is for a small number of queries. As the number of queries grows, so will the savings introduced by query merging. Furthermore, if the multicast is repeated say every hour, then the savings will be multiplied by 24×365 in a year, and we can see that the savings can be significant.

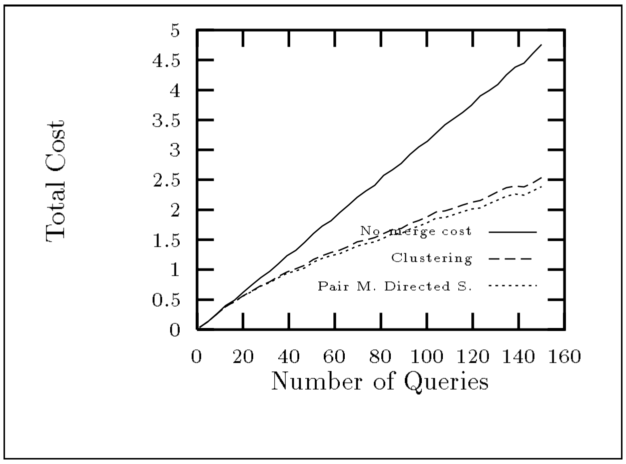

Figure 18: Final Total Cost

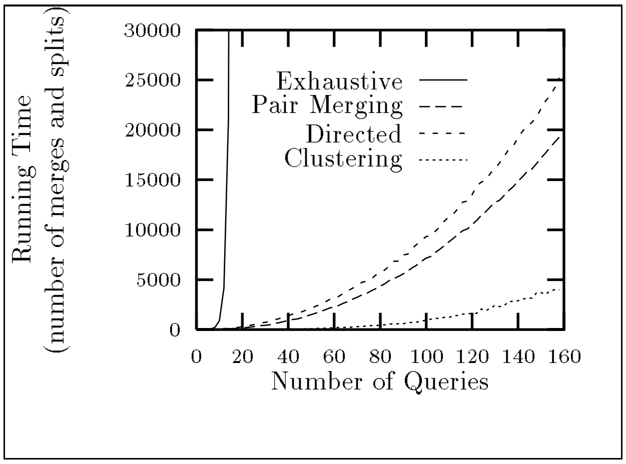

Figure 19: Running Time of Algorithms

The Clustering Algorithm does not yield better solutions, but we expect it have a smaller running time, since the merging algorithm runs on clusters with smaller number of queries. This is quantified in Figure 19, which shows the running times of the algorithms, measured by the number of merge/split operations. For reference, a merge/split operation took about 1msec using unoptimized code on a Pentium 200 computer. Recall that the complexity of the exhaustive algorithm is $O(|Q|^{|Q|})$, whereas the complexities of the Pair Merging and Directed Search algorithms is $O(|Q|^2)$. As we can see, the Clustering Algorithm is much faster as the number of queries increases, while still finding solutions that are close to those of the other algorithms. Thus, the Clustering Algorithm seems to be a good choice for scenarios with many queries.

In our final experiment we attempt to quantify when query merging pays off and by how much. Clearly, the most critical factor is the amount of overlap between submitted queries. If the queries are mostly disjoint, there will be little advantage to merging; it there is significant overlap, we expect significant gains. To get a sense for these gain, we can vary parameter cf, which controls the fraction of the queries that exhibit clustering. When $cf = 1$ all queries are clustered, and when $cf = 0$, all queries are independent. Figure 20 shows the *total* costs incurred in processing the queries when merging is used and when it is not used, as a function of cf. Notice that even when queries are independent, merging introduces savings because there is still some random overlap among queries. As cf increases, the overlap increases, and the savings grow. Again, keep in mind that the savings of 1 to 2 dollars seen, will increase if there are more queries (100 were used for this experiment), and the savings will occur *every time* results are multicast.

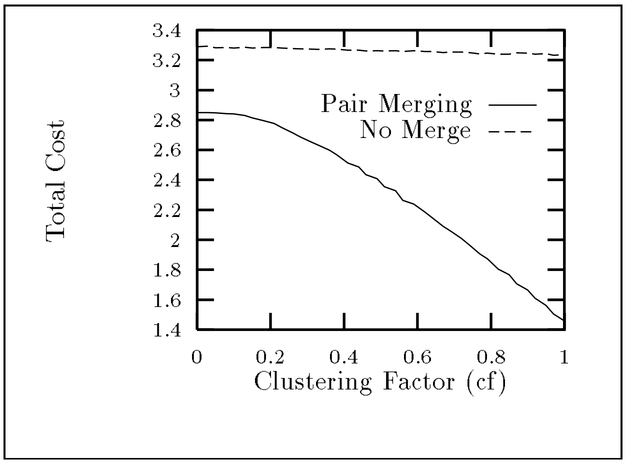

Figure 20: Total Cost Before and After Pair Merging

9 Related Work

The Data Dissemination Problem has been studied by a number of projects [17, 6, 21, 12, 1, 16, 11, 5]. However, none of them attempt to reduce costs by automatically merging similar queries.

The Query Merging Problem is also related to Client-side caching in client/server configurations [22]. In this approach, data is loaded into each client cache as answers to other queries are broadcast by the server. When a client is ready to make a query, it first checks in its own cache to see if the cache already contains the answer. The difference with our work, is that in the client-caching approach, queries are not known, so the server cannot optimize the global cost.

Our research is also related with the Semantic Query Optimization problem [23]. The goal of the Semantic Query Optimization problem is to use semantic knowledge (such as integrity constraints) for transforming a query into a form that may be answered more efficiently than the original version. The main difference with our work is that in semantic query optimization only one query at a time is optimized. This limits the opportunities for improvement versus our work, where we considered a sets of queries.

There are a number of data dissemination products and services in the market [7, 31, 3, 29]. However, as far as we know they do not attempt to do any real query merging. Most of these products are very simple, requiring clients to maintain their subscriptions and to "pull" from the server any new information. Servers normally unicast the results to each client, making this approach non-scalable and resulting in a very high cost.

The Cellular Telephony and Telecommunication research community has also consider the problem of improving the bandwidth use on broadcast channels [20][9][25]. The difference between this effort and our work is the level of abstraction. While the telephony community focus on random memory page requests (and therefore, there is little information available to the optimizer), our work focus on queries and query answers which allows us to have more sophisticated schemes.

The BADD problem [24, 13] has generated a wealth of research in the data dissemination arena. References [4] and [27] have proposed multicast protocol, that can be used as a low level support to our algorithms. Deployment of Internet services through a satellite broadcast channel has been studied in [26] and "smart information push" by [14]. Reference [33] extends the client-side caching by considering caches not only at the client, but also at intermediate locations "close" to the clients. Finally, in [34] the data staging problem is described and heuristics to solve it are presented.

The query merging problem in a geographical database is closely related to the polygon covering problem [10], and to the set covering problem [32]. However, the special characteristics of the Query Merging Problem make it difficult to directly use the well known solutions to those problems.

10 Conclusions

In this paper we have studied the Query Merging Problem. We presented a very general framework and cost model for evaluating merging, and we presented a variety of merging algorithms. To illustrate and experimentally evaluate performance, we considered geographic queries as a representative example. Our results show that dissemination costs can be significantly decreased by applying a merging algorithm, and that heuristic algorithms work well.

Choosing which algorithm to use depends on the number of query subscriptions, the time available, and the precision required. If we have a small number of queries, we can use the Partition Algorithm (the practical limit is twelve queries when running in a typical workstation). If the running time is critical (e.g., in a scenario where queries and subscriptions change dynamically and hence the merge sets must be recomputed on the fly), then using the Clustering Algorithm before applying any merging algorithm improves the running time significantly. If finding the best solution is important and the number of queries is large, the Directed Search Algorithm should be used. When using the Directed Search Algorithm, the number of initial states can be chosen according to the running time limitations.

Acknowledgments: This work was partially supported by DARPA, as part of the BADD Technical Enhancements Program and the NASA Interchange for Joint Research NCC 2-5347.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: data management for asymmetric communication environments. In *ACM SIGMOD*. ACM, May 1995.

[2] Airmedia. At http://www.airmedia.com/.

[3] BackWeb. At http://www.backweb.com/.

[4] F. Bauer and M. Brown. A reliable multicast transport protocol for a global broadcast service-based network. In *MILCOM 97*, volume 2, pages 988–92. IEEE, 1997.

[5] A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time for distributed information systems. In *Proceedings of ICDE'96*, March 1996.

[6] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, J. Raitz, and A. Weinrib. The datacycle archictecture. *CACM*, 35(12), December 1992.

[7] A. Chan. Transactional publish/subscribe: the proactive multicast of database changes. In *ACM SIGMOD*, June 1998.

[8] J. C.-I. Chuang and M. A. Sirbu. Pricing multicast communication: a cost based approach. Technical Report 98354, Carnegie Mellon University, 1998.

[9] M. Corson and J. Macker. Global quality of service-based and reliable data dissemination via asymmetric direct broadcast satellite channels. In *19th Pacific Telecommunications Conference*, Jan 1997.

[10] J. C. Culberson and R. A. Reckhow. Covering polygons is hard (preliminary abstract). In *29th Annual Symposium on Foundations of Computer Science*. IEEE, oct 1988.

[11] S. Dao and B. Perry. Efficient dissemination of information on the internet. *Bulletin of the Technical Committee on Data Engineering*, 19(3):48–54, September 1996.

[12] S. Dao and B. Perry. Information dissemination in hybrid satellite/terrestrial networks. *Bulletin of the Technical Committee on Data Engineering*, 19(3):12–18, September 1996.

[13] R. Douglass, J. Mork, and B. Suresh. Battlefield awareness and data dissemination (badd) for the warfighter. In *Proceedings of the SPIE*, volume 3080, pages 18–24, 1997.

[14] J. Dukes-Schlossberg, Y. Lee, and N. Lehrer. Iids: intelligent information dissemination server. In *MILCOM 97 Proceedings*, volume 2, pages 635–9. IEEE, 1997.

[15] M. Franklin and S. Zdonik. Dissemination-based information systems. *Bulletin of the Technical Committee on Data Engineering*, 19(3):20–30, September 1996.

[16] M. Franklin and S. Zdonik. A framework for scalable dissemination-based systems. In *OOPSLA '97. Object Oriented Programming Systems Languages and Applications*. ACM, Oct 1997.

[17] D. Gifford. Polychannel systems for mass digital communication. *CACM*, February 1990.

[18] H. Gould. Research bibliography of two special number sequences. *Mathematica Monongaliae*, 12, 1971.

[19] J. Graham-Cumming. Hits and miss-es: A year watching the web. In *Sixth International World Wide Web Conference*, 1997.

[20] Q. Hu, D. L. Lee, and W.-C. Lee. Optimal channel allocation for data dissemination in mobile computing environments. In *18th International Conference on Distributed Computing Systems*, May 1998.

[21] T. Imielinski and B. Badrinath. Mobile wireless computing. *CACM*, 37(10):18–28, Oct 1994.

[22] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1), January 1996.

[23] J. J. King. Quist: A system for semantic query optimization in relational databases. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 510–517. IEEE Computer Society Press, 1981.

[24] M. Lazaroff and P. Sage. Any information, anywhere, anytime for the warfighter. In *Proceedings of the SPIE*, volume 3080, pages 35–42, 1997.

[25] W.-C. Lee, Q. Hu, and D. L. Lee. Channel allocation methods for data dissemination in mobile computing environments. In *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing*. IEEE, Aug 1997.

[26] R. Lindell, J. Bannister, C. DeMatteis, M. O'Brien, J. Stepanek, M. Campbell, and F. Bauer. Deploying internet services over a direct broadcast satellite network: challenges and opportunities in the global broadcast service. In *MILCOM 97*. IEEE, 1997.

[27] N. Lu. Network interface to tactical communications. In *MILCOM*. IEEE, 1997.

[28] M. V. Mannino, P. Chu, and T. Sager. Statiscal profile estimation in database systems. *ACM Computing Surveys*, 20(3), September 1988.

[29] MARIMBA. *Castanet*. At http://www.marimba.com.

[30] B. Nadel. A kinder, gentler pointcast. *PC Magazine*, 15(18), 1996.

[31] S. Ramakrishnan and V. Dayal. The pointcast network. In *ACM SIGMOD*, June 1998.

[32] H. Salkin and J. Saha. Set covering: algorithms, results and codes. In *Bulletin of the Operations Research Society of America*, volume 20, suppl.2, Nov 1972.

[33] T. Stephenson, B. DeCleene, G. Speckert, and H. Voorhees. Badd phase ii. dds information management architecture. In *Proceedings of the SPIE*, volume 3080, pages 49–58, 1997.

[34] M. Tan, M. D. Theys, H. J. Siegel, N. B. Beck, and M. Jurczyk. A mathematical model, heuristic, and simulation study for a basic data staging problem in a heterogeneous networking environment. In *Proceedings of the 7th International Computing Workshop (HCW'98)*. IEEE, 1998.

Appendix I: Complexity of the Query Merging Problem

In this section we will prove that the Query Merging Problem is NP-hard. We will prove it by showing that an algorithm that solves the Query Merging Problem can also solve the Set Covering Problem, a well known NP-complete problem.

Definition of the Minimum Set Covering Problem (MSCP): Given a collection $\mathcal{L}$ of subsets of a finite set C, a set cover is a subcollection $\mathcal{L}' \subseteq \mathcal{L}$ such that the union of all subsets in $\mathcal{L}'$ is equal to C and $\mathcal{L}'$ has minimum size. For example, given, $\mathcal{L} = \{\{1,2\},\{2,3\},\{1\}\}$ and $C = \{1,2,3\}$, the solutions of the MSCP are $\mathcal{L}' = \{\{1\},\{2,3\}\}$ and $\mathcal{L}' = \{\{1,2\},\{2,3\}\}$.

In order to solve an instance of the MSCP using the Query Merging Problem, we need to map the input of the MSCP into the Query Merging Problem as follows:

- The set of queries Q will be equal to C
- The constants in the cost function, $Cost_{total} = K_M * |\mathcal{M}| + K_T * size(\mathcal{M}) + K_U * U(Q,\mathcal{M})$, will be $K_M = 0$, $K_U = 0$, and $K_T = 1$.
- We define symbols $\lambda_0, ...\lambda_{n-1}$ representing each element of L, and the symbol λ_n representing any set not in L. We define the merge function as:

$$mrg(S) = \begin{cases} \lambda_i & \text{if } S \text{ is the ith element of } L \\ \lambda_n & \text{otherwise} \end{cases}$$

Additionally, we will assume that the database manager will be able to process only the symbols $\lambda_0, ...\lambda_{n-1}$. The function $size(q)$ will be defined as 1 if the queries can be processed ($q = \lambda_0, ...\lambda_{n-1}$) or ∞ ($q = \lambda_n$) if not.

Under this assumption is easy to see that $cost(\mathcal{M})$ is transformed into:

$$cost(\mathcal{M}) = \begin{cases} \infty & \text{if } \exists M \in \mathcal{M} \text{ such that } M \notin \mathcal{L} \\ |\mathcal{M}| & \text{otherwise} \end{cases}$$

With this input, an algorithm that solves the query merging problem, will find a collection $\mathcal{M} = \{M_i\}$ such that the union of all M_is is C and $cost(\mathcal{M})$ is minimized. Therefore, $\mathcal{M}$ is a solution of the MSCP as minimizing $cost(\mathcal{M})$ means that the size of $\mathcal{M}$ is minimum and all $M_i \in \mathcal{L}$.

This proves that the query merging algorithm is NP-hard with an arbitrary assignment of the K_M, K_T and K_U proportionality constants, as well as an arbitrary $size()$ function. However, there be might be values of the proportionality constants for which the problem is polynomial. Specifically, if $K_T = K_U = 0$, then the problem has a trivial solution: merge all queries into a single one.

Appendix II: Exhaustive Algorithms for the QM Problem

General Algorithm

A general exhaustive approach for solving the query merging problem is presented in Figure 21. First, we form the superset of Q, $\mathcal{S}(Q)$, which contains all possible sets formed by combining elements of Q. Note that $\mathcal{S}(Q)$ contains all potential merges of the queries in Q. Then, we generate the superset of $\mathcal{S}(Q)$, $\mathcal{S}(\mathcal{S}(Q))$. The set $\mathcal{S}(\mathcal{S}(Q))$ is a superset of all the possible solutions to the query merging problem. An element of $\mathcal{S}(\mathcal{S}(Q))$ is a solution only if it provides a total cover of Q. That is, the element includes all the queries in Q. After eliminating all the elements that are not solutions, the final step of the algorithm is to use the cost model to evaluate the remaining elements of $\mathcal{S}(\mathcal{S}(Q))$. The solution with the lowest cost is the optimal solution. The Exhaustive Algorithm has a doubly exponential complexity on the number of queries. Step 1 generates a set $\mathcal{S}(Q)$ with $2^{|Q|}$ elements. Then, step 2 generates a total of $2^{2^{|Q|}}$ elements. Therefore the algorithm will have a cost of $O(2^{2^{|Q|}})$.

Input : $Q = \{q_1, q_2, ..., q_m\}$

Output : Optimal solution for the query merging problem

Generate $\mathcal{S}(Q)$, the superset of the set Q.

Generate $\mathcal{S}(\mathcal{S}(Q))$, the superset of $\mathcal{S}(Q)$.

$A = \{x \mid x \in \mathcal{S}(\mathcal{S}(Q))$ and x is a total cover of $Q\}$

Evaluate the cost of each element of A using the cost model.

The element of A with the minimum cost is the optimal solution.

Figure 21: The Exhaustive Algorithm

Partition Algorithm

The partition algorithm can be used when the cost model ensures the *single-allocation property*: each q_i in the solution is in one and only one element of $\mathcal{M}$. In fact, the cost model presented in this paper ensures the single-allocation property. We can prove this by showing that given any candidate solution $\mathcal{M}$ that does not follow the single-allocation property, we can always build a new set $\mathcal{M}'$ that follows the single-allocation property and has the same or lower cost. The candidate solution $\mathcal{M}'$ is formed by eliminating all but one of the duplicate queries in $\mathcal{M}$. Obviously, the new $\mathcal{M}'$ is a valid solution of the problem (if the original solution was also valid); and, it has the same or lower cost because eliminating a duplicate q_i, reduces or keeps constant all costs in our model.

Basically, the Partition Algorithm generates each possible candidate solution. To ensure that each candidate solution is generated only once, the algorithm uses a search tree as an auxiliary data structure. In Figure 22, we present an example of the Partition Algorithm with input $Q = \{q_1, q_2, q_3\}$. We build the tree by considering one query at a time. At the end of each iteration, the leaves of the tree will contain all possible partitions of the queries considered up to that point. In the figure, in Step 0, the tree only has a single node with an empty list. In Step 1, we consider all partitions that can be created with q_1. In this case, the only possible partition is $\{q_1\}$. In Step 2, we consider adding q_2 to the current partition. There are only two partitions, one that has q_1 and q_2 as separate queries and another, that merges them together. Similarly, in Step 3, we generate the partitions that include q_3. After all partitions are generated, in Step 4, we compute the cost of each one, and select the minimum cost partition (which is the optimal solution).

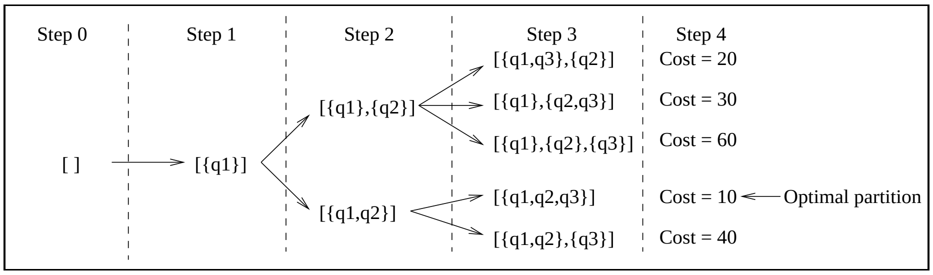

Figure 22: An Example of the Partition Algorithm

Formally, the algorithm associates to each node N of the search tree a list of partition of the queries $N.\mathcal{L} = [M_1, M_2, ...]$. The lists of partitions associated to the nodes at level h contain all possible partitions for the queries q_1, q_2, ..., q_h. At each step of the algorithm, for each node, we create as many children as elements are in the list of partitions plus one. Each of the children has the parent partition, but with the new element considered in that step, added to one of its partitions. The additional child, adds the new element as a separate partition. The algorithm is presented in Figure 23. The number of cases that the Partition Algorithm needs to consider is equal to the *Bell Number* which for large values of n is $O(n^n)$ [18].

Input : $Q = \{q_1, q_2, ..., q_m\}$.
Output : Optimal solution for the query merging problem.

Create root node N at level 0 with $N.\mathcal{L} = []$.
For $h = 1$ to $|Q|$
    For each node N at level $h-1$ with $N.\mathcal{L} = [M_1, M_2, ...]$
        Create children N_0, N_1, ..., $N_{|N.\mathcal{L}|}$ with $N_i.\mathcal{L} = N.\mathcal{L}$
        Append $\{q_h\}$ to $N_0.\mathcal{L}$
        For each child N_i, $i \geq 1$
            Add q_h to the ith element of $N_i.\mathcal{L}$
For each leaf node N evaluate the costs of $N.\mathcal{L}$
The leaf with the minimum cost is the optimal solution.

Figure 23: Partition Algorithm

Appendix III: Exact Distance Metric for the Clustering Algorithm

Given a $\mathcal{M}$, where two queries q_1 and q_2 are not in the same set, we want to obtain an exact distance metric. We can find such an expression by computing the maximum possible benefit that we might obtain if those two queries are merged. If that benefit is negative, then we put the queries in different clusters.

To find the maximum possible benefit, we will analyze each component of the cost model in turn. The factors multiplying the constants K_m and K_t can be positive when merging queries, while K_u is always negative (or zero). To find the maximum possible benefit, we need to find the maximum possible benefits for the factors of K_m and K_t and the minimum cost for the factor of K_u.

- Factor K_m: The best case is when the merging of two queries allows us to merge all queries. Therefore the maximum possible benefit is $K_m(|Q| - 1)$

- Factor K_t: The $\mathcal{M}$ set before merging q_1 and q_2 has the form $\{\{q_1, ...\}, \{q_2, ...\}, ...\}$. Unfortunately, the maximum possible cost of this set is very high, as potentially, we could send all the database to each client (and in that case the high bound will be $|Q| \cdot size(database)$). However, it is safe to assume that even the most inefficient algorithm, will only send the portion of the database in which somebody is interested, reducing the maximum cost to $|Q| \cdot size(mrg(\{q_1, q_2, ..., q_n\}))$. Finally, as our cost model has the single-allocation property, we know that we will not include q_i in more than one merged set, therefore, the maximum cost will be $size(mrg(\{q_1, q_3, ..., q_n\})) + size(mrg(\{q_2, ..., q_n\})) + (|Q| - 2) \cdot size(mrg(\{q_3, ..., q_n\}))$

  The $\mathcal{M}$ set after merging q_1 and q_2 will have the form $\{\{q_1, q_2, ...\}, ...\}$, this set has a minimum cost of $size(mrg(\{q_1, q_2\}))$.

Therefore, the maximum possible benefit would be: $K_t \cdot ((size(mrg(\{q_1, q_3, ..., q_n\})) + size(mrg(\{q_2, ..., q_n\})))$ $+(|Q|-2) \cdot size(mrg(\{q_3, ..., q_n\})) + size(mrg(\{q_1, q_2\}))$

- Factor K_u: The minimum possible cost is $K_u \cdot (2size(mrgq_1, q_2) - size(q_1) - size(q_2))$

Therefore, merging 2 queries, q_1 and q_2, will only be beneficial when:

$$K_m \cdot (|Q|-1) + K_t \cdot ((size(mrg(\{q_1, q_3, ..., q_n\})) + size(mrg(\{q_2, ..., q_n\}))) + (|Q|-2) \cdot size(mrg(\{q_3, ..., q_n\}))$$
$$-size(mrg(\{q_1, q_2\}))) - K_u \cdot (2 \cdot size(mrg\{q_1, q_2\}) - size(\{q_1\}) - size(\{q_2\})) > 0$$