DATA CACHING ISSUES IN AN INFORMATION RETRIEVAL SYSTEM

Rafael Alonso
Daniel Barbara
Hector Garcia-Molina

CS-TR-101-87

Revised June 1989

DATA CACHING ISSUES IN AN INFORMATION RETRIEVAL SYSTEM

Rafael Alonso
Daniel Barbará
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Currently, a variety of information retrieval systems are available to potential users. These services are provided by commercial enterprises (such as Dow Jones [Du84] and The Source [EdDa83]), while others are research efforts (the Boston Community Information System [Gi85]). While in many cases these systems are accessed from personal computers, typically no advantage is taken of the computing resources of those machines (such as local processing and storage). In this paper we explore the possibility of using the user's local storage capabilities to cache data at the user's site. This would improve the response time of user queries albeit at the cost of incurring the overhead required in maintaining multiple copies. In order to reduce this overhead it may be appropriate to allow copies to diverge in a controlled fashion. This would not only make caching less costly, but would also make it possible to propagate updates to the copies more efficiently, e.g., when the system is lightly loaded, when communication tariffs are lower, or by batching together updates. Just as importantly, it also makes it possible to access the copies even when the communication lines or the central site are down. Thus, we introduce the notion of *quasi-copies* which embodies the ideas sketched above. We also define the types of deviations that seem useful, and discuss the available implementation strategies.

Index Terms: Distributed data management, distributed systems, information retrieval systems, caching, cache coherency, data sharing, data replication.

June 30, 1989

DATA CACHING ISSUES IN AN INFORMATION RETRIEVAL SYSTEM

Rafael Alonso
Daniel Barbará
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

1. INTRODUCTION

In many of today's information retrieval systems (IRS's) all the stored data (e.g., the abstracts of journal articles, the airline schedules) resides at a central node. This central site can be reached by a large number of remote terminals connected via relatively slow communication lines. Users at these terminals do no local processing; they simply send their queries to the central machine and wait for their replies. Data can be added or deleted at the central site, but in many cases it cannot be updated.

A number of developments are slowly changing this IRS model. First, the number of users is growing rapidly. In our ''information society'' it is becoming increasingly important to have access to timely information. At the same time, the number of personal computers, at home and in the workplace, has grown tremendously, giving more people the hardware necessary to access the IRS's.

The first development implies that the increased IRS services and requirements will tax both the processing and communication capacity of the central site. There are a number of potential solutions to this problem, but the one we will focus on in this paper is *data caching*. This solution is becoming feasible precisely because of the second development, i.e., that the IRS is frequently accessed from personal or mini computers with substantial processing and storage capacity (for example, in 1984, Dow Jones estimated that about 125,000 of its 165,000 customers used personal computers [Du84]).

In principle, caching can improve system performance in two ways. First, it can eliminate multiple requests for the same data. For example, consider an automobile manufacturing plant where a number of people are interested in news wire stories on trade and protectionism. In this case, it makes sense to cache the relevant articles at the company's local computer, eliminating redundant requests to the central IRS site. A second way in which caching can improve performance is by off-loading work to the remote sites. For instance, if a user is interested in chemical companies he may store the latest stock prices of those companies at his own computer. There he can run his own analysis programs on the data, without using any more central cycles.

However, caching has an associated cost. Every time a cached value is updated at the central site, the new value must be propagated to the copies. Furthermore, the propagation must be done immediately if cache consistency (or *coherency*) is to be preserved. (A cached value for an object is consistent if it equals the value of the object at the central site.) This propagation cost can be significant.

Caching has been successfully used in other environments, but there are some important differences in this case. In a computer hardware cache, [Sm82] it is not expensive to keep the cached and main memory data consistent. This is because updates are small (e.g., a byte is modified), the communication delays are short, and the number of copies is small (e.g., in snooping cache architectures typically there are less than 10 caches connected to a memory system). In an IRS, on the other hand, the communication costs can be much higher. For instance, users typically communicate over telephone lines. Also, the number of caches may be quite large. Finally, the updates can be extensive (e.g., the abstract of an article or the article itself can be added to a file). The idea of caching data in workstations has been also used in [Gl86].

In light of these difficulties, it is important to explore strategies for making update propagation less costly while still retaining the inherent advantages of caching. In this paper we study two such strategies. Both involve taking advantage of the application semantics. The first idea is to let the user explicitly define the information that is of interest and to cache only it. This obviously reduces the need to refresh data that is not going to be used.

The second idea is to allow, whenever possible, a weaker type of consistency between the central data and its copies. For instance, the user interested in the stock prices of chemical companies may be satisfied if the prices at his computer are within five percent of the true prices. This makes it unnecessary to update the cached copy every single time a change occurs. When the deviation exceeds five percent, then a single update can bring the cached copy up-to-date. At a manufacturing company, users may tolerate a delay of one day in receiving the articles of interest. If the system takes advantage of this, it can transmit all the articles during the night when communication tariffs are lower. If a communication or central node failure occurs and its duration is less than 24 hours, then users can continue to access information that is correct by their standards.

We call a cached value that is allowed to deviate from the central value in a controlled way a *quasi-copy*. The management of quasi-copies is called *quasi-caching*, to differentiate from conventional caching where no data deviations are allowed. Quasi-copies have the potential for reducing update propagation overhead and giving the system flexibility for scheduling the propagation at convenient times. Note that the information flow in an IRS with quasi-copies is similar to the flow in many real organizations. The manager of a company is not told every time an employee is hired or leaves. The information is filtered so that he only is informed periodically of personnel changes, or if an exceptional condition occurs (e.g., a mass exodus of employees). Hence, the manager's view of the company (the cached data) deviates from the true state (the central value). Similarly, when a person desires news, he subscribes to magazines and newspapers. The news arrives periodically and there is again a discrepancy between the local and ''central'' data. In human organizations, people have little control over this process, e.g., Time magazine arrives every week and the New York Times every day, and there is no way to change this. In a computerized IRS, however, we can let users precisely define the limits of divergence of quasi-copies, and the system can take advantage of this to improve performance.

We should point out that quasi-copies are not free either. The reductions in transmission costs are paid for by increases in processing time for bookkeeping, both in the central processor and in the workstations. Hopefully, trading off transmission time for processing time will pay off as workstations become more powerful while transmission costs remain fixed. We will quantify these tradeoffs further in Section 6.

The concept of quasi-copies is somewhat related to the idea of *materialized views*. A materialized view is a stored relation whose data is derived from the base relations by evaluating an expression constructed from any combination of project, select and join operations. As base relations are modified, the derived relations may have to be refreshed. That can be done by reevaluating the expression after every transaction that modifies the base relations. However, the cost associated to this strategy would be unacceptable. In [Bl86] techniques are presented for detecting when an update to a base relation cannot affect a materialized view, and for detecting when the derived relation can be updated using its own data only. This last technique is particularly useful when the derived relation is stored in a remote site. Both strategies are based on an screening algorithm to test each modification, insertion and deletion made to the base relations. The test, based in boolean satisfiability, detects whether the new tuple may cause the materialized view to change. A variant to this algorithm in which the updating of materialized data is deferred until just before it is used, is presented in [Ha87]. The idea is also proposed in [RoKa86] as a method of materializing copies of views in remote workstations. Another technique of view maintenance is called *snapshot refreshing* [Li86]. A snapshot is a read-only table whose contents are extracted from the base tables. The snapshots are periodically refreshed to reflect the current state of the database. Snapshots were developed as a cost effective substitute for replicated data in distributed applications. Effective algorithms for refreshing snapshots are presented in [Li86].

The main difference between these techniques and quasi-copies is that our concept allows the user to establish the degree of coherency of the cached copy. By establishing how much the cached data can deviate from the central copy, the user has control over the currency of the data used rather than having to comply with a given degree of coherency (which for instance, in the snapshot technique is given by the frequency of the application of the refreshing algorithm.) In this way, the coherency can be adjusted to the needs of the application that is to be run in the remote workstation. The degree of coherency can vary from a perfect up-to-date copy to a simple ''hint'' of the data. We feel that this concept is a powerful tool that encompasses a wide spectrum of choices. (A related notion of letting users specify an "age threshold" for hint information has been studied in [Te87].)

In this paper we will assume that all information is controlled at a single central site. This site executes all updates and hence has the most up-to-date version of all data. Usually remote users only read data. If they want to modify something, they may submit an update transaction to the central site. If the modifications are based on data read from the IRS, the reads must occur at the central site at the time the transaction runs, not at the remote node. To illustrate, let us consider the following example. Suppose that a stock market information IRS also allows users to purchase stock. If a user observes at his terminal that the price of a certain stock price is favorable, he can submit a transaction to the central site to purchase some amount of stock. However, the ''real'' price, i.e., the price at the central site, can differ from the value observed by the user when he made his decision. Hence, the user must either be willing to buy stock at a ''slightly'' different price or must include in his update transaction code to read the price once again and abort the operation if the price is no longer acceptable. (Note that this is the way stocks are usually purchased in reality.) This decision is similar to that faced by users of database browsers based on the idea of *portals* [StRo82].

Quasi-caching could easily be extended to a system with several central sites, each controlling a fragment of the data, as long as modifications to a particular datum could only take place at one computer. Fragmenting the database like this is another way of reducing the workload at the central site, but for simplicity, we will continue to assume that there is a *single* central site.

However, allowing updates to a datum to originate at multiple sites (e.g., at a user remote machine and at the central site) does complicate quasi-caching substantially and will not be considered here. Central data control is essential to our approach since it simplifies the types of inconsistencies that can occur in a distributed system with replicated data. For a survey of distributed control strategies for replicated data see [Da85].

Even though quasi-copies seem to be crucial for effective caching in an IRS, very little is known about them. Hence, the objective of this paper is to study quasi-caching and to attempt to answer some of the basic questions. What types of quasi-copies are most useful? How can they be defined? How can conventional data consistency constraints (e.g., a manager's salary must be greater than his/her employee's salary) be enforced at the cached copies when the individual values can fluctuate? Quasi-copies can be implemented in a variety of ways. For instance, values that diverge too much can be invalidated or refreshed. Data sent to the caches can include an automatic expiration time and date. The quasi-copy requirements can be enforced at the central or at the remote sites. In this paper we will survey the various implementation strategies and their tradeoffs.

In the following section we define quasi-caching more precisely and introduce some terminology. There are two types of conditions that can be specified for quasi-data: selection and coherency. They are discussed in Sections 3 and 4. The impact of transmission delays and failures, as well as other implementation issues, is covered in Section 5. Performance issues are discussed in Section 6. Section 7 provides two simple examples. The final section offers our conclusions.

2. QUASI-CACHING

We start by defining more precisely quasi-caching and introducing notation that will be used in the rest of the paper. The database is stored at the central node, C, and consists of a set of objects O. Each object $x \in O$ can have a number of values (or fields) associated with it (e.g., object John has name, address, salary values), but for simplicity we assume there is just one value. As is customary, we use the same symbol x to represent both the object and its value. All updates take place at the central site. As an object is modified, new versions are created. We represent the latest version of object x by $v(x)$. It will sometimes be necessary to refer to the value of an object x at a time t. We represent this by $x(t)$. (Incidentally, we assume that all sites have accurate and synchronized clocks [La78].)

A set of nodes (or workstations) N $(C \notin N)$ may contain quasi-copies of the objects. (Several of our nodes may run on a single physical computer as separate processes.) The quasi-copy of object $x \in O$ at node $j \in N$ is x^j and is called an image of x. When the identity of node j is not important, we represent the image as x'. The set of objects that have quasi-copies at node j are the objects cached at j. Note that the quasi-caches at different nodes can have different objects, and objects can be cached at 0,1,2,... or all nodes. In the rest of the paper we drop the prefix "quasi" whenever it is clear we are referring to quasi-caching and quasi-copies.

We do not specify the granularity of objects. Conceptually, objects can be small (e.g., fields of records), or large (e.g., files). There are performance implications to granularity, but these will be covered in Sections 5 and 6.

Users at a node define how copies are managed by giving two types of conditions: selection and coherency. The selection conditions specify which object images will be cached at the user's site. The coherency conditions define the allowable deviations between an object and its images. In our stock market example, the user issues a selection condition to indicate that he wants copies of the stock prices of chemical companies. His coherency condition would then state that a five percent variation between the central and his site is acceptable. We discuss these types of conditions in the next two sections.

All users or application programs running at a node share the quasi-cache. They refer to the objects by the same names they have at the central site. An access to object x by a user or program will return the local image x' if it exists. If not, an access to the central site will be made. Users or programs must be capable of coping with data that deviates from the central data, as specified by the coherency conditions. If this is not the case, the user or program should not be running at a node where a quasi-cache has been defined. (Another option might be to allow each individual read to specify if the local quasi-cache can be used.)

3. SELECTION CONDITIONS

In a computer hardware cache, the decision as to *what* to hold in the cache is made automatically by the system. For example, the system might store every word that is fetched. To make room for the word, it may purge the least-recently-used (LRU) word from the cache. In an IRS, a better strategy might be to let the user specify what data is to be cached. Selection conditions let the user do this.

A selection condition always contains an *identification clause* that specifies the objects to be cached (or dropped from the cache). In addition, there may be one or more *modifiers* that determine how the selection is going to operate.

The identification clause can explicitly list the objects involved in the selection or can give an expression that evaluates to a set of objects. For example, if a relational language [Da75] is used for the expression, then the condition.

```
SELECT NAME, PRICE
FROM STOCKS
WHERE TYPE = ''Chemical Company''
```

can be used. It selects the NAME and PRICE attributes (or fields) of tuples (or records) that represent chemical companies. In our terminology, each NAME or PRICE value selected is an object that must be cached. There are many other languages and models for selecting data or information [Da75], but since they are well known, we will not cover them here. In this example objects are small, i.e., fields of records. As discussed earlier, caching and tracking small objects may be more expensive, so the language may be restricted to deal with larger objects such as files or relations. On the other hand, the techniques of Section 5 may make it feasible to manage the small objects of our example.

The modifiers in a selection condition may include the following items:

(1) *Add/Drop*. This item specifies whether the selected objects are to be added to the cache or removed from it. If ''Drop'' is specified, then the images at the subscriber nodes are removed from the caches (if they existed).

(2) *Enforcement*. A selection condition can be of two types: compulsory or advisory. If it is compulsory, then the system must guarantee that the selected objects are cached as requested. If it is advisory, then the caching is viewed exclusively as a performance enhancement. In this case, the selection condition is taken as a ''hint,'' and may or may

not be followed by the system.

A query optimizer can take advantage of the knowledge that a selection is compulsory. To illustrate, let us return to the STOCKS expression given earlier. Suppose that it is compulsory and that the user searches for the stock price for "Chemical" company "AJAX" at his computer. If the stock is not found locally, then AJAX is not a chemical company. No other action is necessary since the user is only interested in chemical companies. Similarly, a query to evaluate the average stock price for chemical companies can be executed locally. If the selection was advisory, then for both queries a check would have to be made at the central site to see if there were additional companies satisfying the query.

The advantage of advisory selection is that it gives the system greater flexibility. If the central site is overloaded, the caching of objects can be delayed or eliminated. Similarly, if storage space is limited at the remote site, data can be purged.

In practice, a judicious combination of compulsory and advisory selections may be best. For example, consider a legal IRS that contains summaries of court cases. The system also has an inverted list index that is used to locate summaries given a set of key words. In this case it may be advantageous to cache all objects that make up the index in a compulsory fashion, and the most relevant summaries in an advisory way. This way, queries can be processed locally yielding a list of summary identifiers. Requests would only be make to the central site to fetch summaries not found locally.

(3) *Static/Dynamic*. If the selection is static, then the objects are selected once when the condition is issued by a user. If it is dynamic, then changes in the data will continuously trigger a reevaluation of the identification clause, and objects will be added or dropped dynamically. For example, if the sample identification clause given earlier is static, no new stocks will be cached at the remote site. If it is dynamic, then every time that a new stock is added at the central site, a check will be made. If the stock is of a chemical company, then a copy will be made. When a company changes its classification from chemical to something else, its copy will be purged. Note that a dynamic selection will usually be of type "Add," and its enforcement of type "compulsory."

Both statically and dynamically selected objects can have coherency conditions specified. If an object was selected statically, its identification clause will not be reevaluated. However, if it has coherency conditions, they will be checked dynamically.

(4) *Triggering Delay*. When a dynamic selection is made, a change to the central database may cause a new object to be added to (or dropped from) the selection list. In some cases, it may be desirable to delay the addition (or deletion) of the object. For example, if while using an abstract service IRS a user selects abstracts on "compilers" and gives a 24 hour delay, then new abstracts on the topic can be batched together and sent more efficiently. In the case of a stock market IRS, if a user selects stocks with prices less than 100 dollars, then a delay of one hour can eliminate repeated additions and deletions of a stock whose price is fluctuating close to 100 dollars.

When a user wants to give a triggering delay, he states the maximum allowable delay Δ. The system is then free to add (or delete) an object to the selected set any time between the time the triggering occurs and Δ seconds later.

Note that a triggering delay can be used with either compulsory or advisory selections. If the selection is compulsory, then the cached data can be used for query optimization, but the results may not include the latest information. Let us return to the examples that were used to illustrate compulsory selections. Say a user has selected stocks for chemical companies. He has specified a compulsory selection and a triggering delay of 1 hour. Say he searches for the stock price of company "AJAX" and it is not found in the cache. In this case it is *not* necessary to look for company AJAX at the central site, even though it might have been created there within the last hour. The user has indicated that he can tolerate a delay of up to one hour for hearing about new chemical companies. Hence, the search for company AJAX need not involve recently selected objects.

Incidentally, setting the triggering delay Δ too low might make it hard to implement a compulsory selection. For instance, if sending a message from the central to a remote site takes T_D seconds, then the system cannot guarantee that the selected data will be at the remote site in less than T_D seconds. We will return to this issue in a later section.

4. COHERENCY CONDITIONS

Once an object has been selected for replication, the coherency condition(s) specify the allowable deviations of the image. The coherency conditions are enforced only when an image exists.

Every image has a default condition which defines the allowable values that it may contain, even if no other conditions are given. This default condition is enforced by the system.

Default Coherency Condition: An image x' must have a value previously held by the object. That is,

$$\forall \text{times } t \geq 0 \, \exists t_0 \text{ such that } 0 \leq t_0 \leq t \text{ and } x'(t) = x(t_0)$$

Users may specify additional constraints. Actually, any constraint on the values of the objects and images could be defined; however, our goal here is to identify and understand the more useful ones. Four useful constrain types are:

(1) *Delay Condition.* This is similar to the selection triggering delay. It states how much time an image may lag behind its object. For object x, and allowable delay of α is given by the condition

$$\forall \text{times } t \geq 0 \, \exists k \text{ such that } 0 \leq k \leq \alpha \text{ and } x'(t) = x(t-k)$$

Since this defines a window of acceptable value, we use the notation $W(x) = \alpha$ to represent this condition.

Note that a delay condition is different from the triggering delay in a selection condition. The triggering delay indicates the allowable time between issuing the selection condition and having the object appear in the cache. During this time, a delay condition is not applicable. Once the object appears in the cache, the delay condition specifies how far behind that image can fall.

(2) *Version Condition.* A user may want to specify a window of allowable values, not in terms of time, but of versions. For example, if an object represents a VLSI circuit, it may be useful to require a copy that is at most 2 versions old. We represent this condition as $V(x) = \beta$, where x is the object and β the maximum version difference. That is , $V(x) = \beta$ is the condition

$$\forall \text{times } t \geq 0 \ \exists k, t_0 \text{ such that } 0 \leq k \leq \beta$$
$$\text{and } 0 \leq t_0 \leq t$$
$$\text{and } v(x(t)) = v(x(t_0)) + k$$
$$\text{and } x'(t) = x(t_0)$$

(3) *Periodic Condition.* With a periodic condition a user indicates that the image must be refreshed periodically. For instance, a user may desire the stock prices every day when the market closes. The condition $P(x) = \alpha, \beta$ states that the image of x must match the object at time α, and must be refreshed every β seconds thereafter. In other words, $P(x) = \alpha, \beta$ is the condition

$$\forall \text{times } t \geq 0 \ \exists n \text{ such that } n \geq 0$$
$$\text{and } \alpha + n\beta \leq t < \alpha + (n+1)\beta$$
$$\text{and } x'(t) = x(\alpha + n\beta)$$

(4) *Arithmetic Condition.* If the value of an object is numeric, the deviations can be limited by the difference between the values of the object and it's image. That is, we may state that

$$\forall \text{times } t \geq 0 \quad |x'(t) - x(t)| < \varepsilon$$

or that

$$\forall \text{times } t \geq 0 \quad \frac{x'(t) - x(t)}{x(t)} 100 \quad < \varepsilon\,\%$$

We represent the first condition by $A(x) = \varepsilon$; the second one by $A(x) = \varepsilon\,\%$.

Yet more conditions can be built out of the three elementary ones we have listed by connecting them with logical "OR", "AND", and "NOT" operators. For example, the condition

$$W(x) = 1 \text{ hour AND } V(x) = 2$$

specifies that x' can lag at most one hour behind x, unless x has been modified more than two times within this hour. The condition

$$W(x) = 1 \text{ hour OR } V(x) = 2$$

means that x' can always lag behind x by an hour. It can even lag longer if the image is still within 2 versions of x.

It may also be useful to allow the parameters of the coherency conditions to vary over time. For example, if a user is planning an important financial operation in 30 days, he may want his data to have a smaller window as the day of the transaction approaches. Thus, he may define $W(x) = (k - 30)$ minutes, where k is the day and x is an object of interest.

As a final point, we should mention that so far we have only discussed constraints on a single object and its image. However, there can also be constraints among objects (usually called consistency constraints). For example, if $x_1, x_2, \ldots x_n$ are stock prices in an IRS, and $\bar{x}$ is their average, then we have the constraint $\bar{x} = \text{average}(x_1, x_2, \ldots x_n)$. A user that reads the images of the stock prices and their average cached at his workstation would like to see the condition hold.

To illustrate some of the difficulties involved in maintaining consistency constraints consider two objects x, y and the constraint $x + y \leq 10$. Say that x's image has the coherency condition $A(x) = 3$ and y has $A(y) = 1$ (see item 4 above). Initially, we have the following situation:

$$x = 7 \qquad x' = 7$$

$$y = 3 \qquad y' = 3$$

An update transaction decreases x by 2 at the central site. Since $A(x) = 3$, the image does not have to be updated:

$$x = 5 \qquad x' = 7$$

$$y = 3 \qquad y' = 3$$

Note that the multi-object constraint holds at both sites: $x + y \leq 10$ and $x' + y' \leq 10$. Next, a second update increases y by 2. Since $A(y) = 1$, this change does have to be propagated. The situation is now:

$$x = 5 \qquad x' = 7$$

$$y = 5 \qquad y' = 5$$

Although the multi-object constraint holds at the central site, it does not hold at the copy site.

We give the user that desires multi-object consistency two choices: the first is to explicitly give the system the constraints that must be satisfied. When a constraint is violated, then the missing updates must be propagated. A second option is to state that a group of objects $x_1, \ldots x_n$ have constraints but not give them explicitly. This option is useful when the users cannot list all their constraints or when it is too expensive to check them. In this case the system must ensure that updates to $x_1, \ldots, x_n$ are applied in order at the copies. That is, let T_0 be the last update transaction whose modifications were applied to one or more of $x'_1, \ldots, x'_n$. Let $T_1, \ldots, T_m$ be other updates to one or more of $x_1, \ldots, x_n$ that were not propagated because they did not violate any single object coherency conditions. Finally, let T_{m+1} be an update that does modify one or more of $x_1, \ldots x_n$ and does have to be propagated. Then all the updates of $T_1, \ldots, T_m, T_{m+1}$ must be made on the images $x'_1, \ldots x'_n$, to avoid violating any constraints at the remote site.

5. IMPLEMENTATION

In this section we consider a number of issues that arise in the implementation of quasi-copies.

5.1. Transmission Delays and Failures

We now address two complications that may make it difficult to enforce the conditions given by a user: transmission delays and failures. To illustrate, consider the condition $A(x) = \varepsilon$ and assume that an increment of more than 2ε is about to occur at the central site. The condition indicates that the difference between x and x' should "never" be larger than ε, and hence the update to the image must be performed "at the same time" as the object is changed. Strictly speaking, this is not possible. The problem due to failures is similar. For example, if the central site fails just after the 2ε increment is made but before the change is propagated to x', then the condition $A(x) = \varepsilon$ will be violated.

Although a 2-phase commit protocol (or similar strategy) could be used, the update overhead and temporary inaccessibility of data are potential drawbacks. Thus, we propose the following solution. The central site, when operational, will make sure that each remote site receives a message at least every δ seconds. This means that if the central site notices that no message has gone out to a site in $\delta - T_D$ seconds, where T_D is the maximum message delay, then it will send a "null" message. Null messages are numbered just like regular messages, and all messages must be received in order. When a site j notices that δ seconds go by without a message from the central site, it declares the central site failed, and sets a local variable $C\_FAILED(j)$ to true.

Then we interpret every condition $C(x)$ on object x set by a user at node j as

$$C(x) \quad \vee \quad W(x) = \delta \quad \vee \quad C\_FAILED(j)$$

(Condition W is defined in Section 4.) This means that all conditions have an implicit delay window of δ and do not have to be enforced if the central site is down. With this approach, the user can display *C_FAILED* at the same time he displays his data, and interpret it accordingly.

5.2. What to Propagate

When the central site wishes to inform remote sites of an update, it can send the following types of messages:

(a) *Data Message.* A data message contains the new values. These values should overwrite those found in caches.

(b) *Invalidation Message.* An invalidation message identifies the objects that have changed, but does not contain the new values. An invalidation message usually causes the remote node to purge from its cache the referenced images.

(c) *Version Number Message.* This message identifies the objects and provides their new version numbers. The new data values are not included. (The time of update could be included instead of, or in addition to the version number. However, this information can sometimes be inferred from the message arrival time.) The remote node uses the version number to decide if it should purge an image.

(d) *Implicit Invalidation.* In this last case, the central node sends *no* message. Instead, images are automatically invalidated after a certain time (i.e., they are *aged*). That is, when a copy is made, it is sent with a time limit. The remote node then guarantees that it will purge the image at the latest when the limit expires.

In some applications, propagating updates is a reasonable approach. However, in many cases, some of the other approaches may be useful. Implicit invalidation incurs the least overhead and is especially attractive if objects are large or communications faulty. For example, train schedules can be issued, as they are in reality, with an expiration date. When the schedule expires, a new copy must be requested explicitly if there is still interest.

Invalidation and version number messages also have reduced communication overhead, but more than implicit invalidation. They are especially attractive for broadcast environments, where the central node can inform everyone of changes but only those that actually need the new data request it. One application that already uses this strategy is catalogs for department stores. When a new catalog appears, all customers are mailed a postcard informing them. Interested customers must then pick up their copy at the store.

Version number messages are desirable when version coherency conditions have been specified. If version number messages are broadcast, the work of checking versions can be off loaded to the remote site. Each remote site can check its own conditions and decide what data must be purged.

5.3. When to Propagate

When an update arrives at the central site, it does not have to be propagated immediately. As a matter of fact, there are several choices:

(a) *Last Minute.* The updates can be delayed up to the point where a coherency or selection condition is about to be violated.

(b) *Immediately.* Updates could be propagated as soon as they occur.

(c) *Early.* Updates can also be propagated at any other time, after they arrive but before a condition is violated.

(d) *Delayed Update.* A last choice is to delay the installation of an update at the central site. If the update is not installed, the conditions cannot be violated, and the propagation can be delayed.

For example, consider the condition $A(x) = 5$, and the original value of x to be 10, with the cached copy $x' = 10$. Assume that updates begin to increase the value of x to, say 11,12,13,14,15,16. Under last minute propagation, the central site would broadcast the new value of x as soon as the $x = 16$ update comes in.†

Under immediate propagation, every new value of x results in a broadcast. Under early propagation we could schedule a transmission when $x = 13$, for example. And finally, for delayed update, $x = 16$ will not be installed until it is convenient.

Immediate propagation should only be used when the selection and coherency conditions require it, or when evaluating the conditions is too expensive. Last minute propagation has the greatest potential for reducing communication costs since it allows as many as possible updates to be ''batched'' together. However, sometimes early propagation can take advantage of lower communication tariffs or processor idle time. For example, telephone calls are usually more expensive from 8:00 AM to 5:00 PM. Suppose that a delay condition $W(x) =$ 5 hours has been defined, and that updates to x have taken place at 7:00 AM. If the telephone is going to be used, it is clearly better not to use last minute propagation (at 12:00 Noon), and to transmit the new data just before 8:00 AM.

Delayed update gives us even greater flexibility and potential for batching together even more updates. However, delaying updates at the central site is an inconvenience since the database there is made to diverge from the ''real world.'' Hence, late propagation is only an option when the applications can tolerate such divergence.

Conceptually, delaying updates at the central site is akin to increasing the delay window $W(x)$ of the objects. However, the advantage of delayed updates is that it can make implicit invalidation work more efficiently. When a remote node requests a copy of object x, the central site can respond with an expiration time t_e in the future. Now, if updates can be put off until time t_e, the copy will not have to be explicitly invalidated.

† Recall that, as it was explained in Section 4, there is an implicit window of δ that makes the condition true while the new value is being transmitted.

5.4. Collapsing Conditions

One of the major drawbacks of using quasi-copies may be the cost of checking coherency conditions at the central node. There are two aspects to this cost: processing and storage. If an object has n coherency conditions, then updating it will require checking all conditions, an $O(n)$ effort. To check each condition, the central site may need to know the value of the image at each workstation. For instance, consider an arithmetic condition $A(x) = 2$, when x is currently 5. An update $x = 6$ arrives. To know if the condition is violated, the central node must know the value of x'. If $x' = 5$, then the condition still holds; if $x' = 3$ it does not. Thus, if the database holds m objects, each of which with n coherency conditions, the central site may have to store $O(mn)$ values.

Note incidentally that checking coherency conditions is similar to the problem of checking whether an update violates an integrity constraint [HaSa78, BuCl79]. However, for coherency constraints we expect the actual checking to be relatively simple since each constraint is simple. It is the number of constraints that may be a problem, and this is the issue we address here.

The number of constraints that have to be checked at the central node can be reduced in several ways. One obvious one is to only allow conditions on large objects like relations or files (this reduces m). Another one is to limit the number of conditions (control n). For instance, quasi-copies may only be allowed at a few computers such as the major corporate clients of the IRS. If these clients contribute a large fraction of the queries, then quasi-copies can still be beneficial. The corporate computers can in turn institute a quasi-copy mechanism with their local workstations. In addition to these obvious solutions, we can try to automatically collapse constraints or to shed off some of the load to the workstations. The first of these ideas is discussed in the rest of this section; the other is presented in the next section.

In many cases it is possible to collapse several coherency constraints (from different workstations) on the same object into one. The collapsed constraint is chosen so that it triggers whenever any of the individual constraints would have triggered. The collapsed constraint may not be as effective as the original ones, but it will probably be better than having no constraints at all.

We illustrate this idea with a simple arithmetic condition, although it can be used with the other types of coherency conditions we have presented. Say workstation W_1 defines $A(x) = 5$. Later on, a second workstation W_2 requests a quasi-copy with $A(x) = 4$. At this point, the central node sends the value of X to W_2, and also to W_1 to ensure that both workstations have the same state. Both conditions are collapsed into $A(x) = 4$. If x changes by more than 4, both workstations are refreshed, even though W_1 may not require the update yet.

It is also possible to automatically collapse constraints on different objects. To illustrate consider two records x and y in some file F. Suppose that x has a delay constraint of 1 hour from workstation W_1, and y of 2 hours form W_2. The two constraints are collapsed into a delay constraint on file F of 1 hour (the minimum of the two individual delays). Say 1 hour has gone by since F was modified last and the copies have not been refreshed. Then both records are propagated, x to W_1 and y to W_2. The central site only checks the constraint on F, not on the individual records. The price to pay is that extra messages are sent, (such as the one to W_1) that could have been postponed.

5.5. Load Balancing

Another alternative for reducing the amount of work at the central site is to partially off-load the enforcement of coherency constraints to the workstations. Some coherency conditions are easy to check at the remote nodes, without imposing any processing or storage overhead at the central processor. An example is a delay condition in which the value x' is only valid for a window of time. After a period of time, the workstation simply purges the value from its cache, forcing the next request to be directed to the central node. Another type of check that is especially well suited for the remote nodes is the multi-object one. In this case, a constraint such as ''number of reservations is less than or equal to the number of available seats'' must be enforced. As updates arrive at a remote site, these checks can be made. Missing updates can be requested (or data can be purged) if the constraints are violated.

Although for most other types of conditions some work must be done centrally, there are ways the workstations may help. Consider the following scheme for load balancing while collapsing arithmetic conditions. Each time the i^{th} workstation receives a new (and obviously current) value for an item x from the central site, it computes and returns to the central site a pair of values, x^i_{low}, x^i_{high}. These values represent the lowest and highest values that x can reach in the central node, without the need to send an update to the i^{th} workstation (note that these may be computed locally by examining the individual coherency conditions as well as the current value of x). We define $\Delta^i_x = [x^i_{low}, x^i_{high}]$ as the *tolerance interval* for x at workstation i. And now, the central station need only store the global tolerance interval $\Delta_x = [x_{highest\_low}, x_{lowest\_high}]$, defined as the intersection of the individual tolerance intervals for the workstations. Whenever an update of x sets the value of x outside of Δ_x, the central site will broadcast the new value (or an invalidation message), to all the workstations holding a copy of x in their cache. The central station updates Δ_x upon receipt of any pair x^i_{low}, x^i_{high}, by computing

$$\Delta_x = \Delta^i_x \cap \Delta_x$$

This method only requires storing one pair of values per item cached (at a cost of $O(m)$ storage locations) and eliminates the need of storing the predicates in the central site thereby off-loading the associated predicate computation to the workstations. Notice that, as an added advantage, the users may change the predicates as frequently as they wish, without having to communicate the changes to the central site. Furthermore, the predicate may be as complex as the user desires; the complexity of evaluating it does not affect the overall performance of the central database.

Moreover, it should be pointed out that the technique just described may be applied to versions also. Instead of sending a tolerance version interval, the workstation simply sends the maximum version number that the central site can reach without updating the cache. However, this load balancing technique cannot be applied to delay and periodic conditions; in these cases, it is best to use collapsing alone, as discussed in the previous section.

The drawback of this approach is that the frequency of update propagation might be increased by taking the intersection of the intervals. This may happen for one of two reasons. If a particular user sets a very narrow interval, it can force the update propagation to every workstation holding that item in its cache for almost every update in the central site. (This problem also arises with constraint collapsing.) This problem might be corrected by clustering together workstations with similar coherency demands. A second cause of unnecessary updates occurs when individual workstations start to cache items at different times. For example, assume that most workstations cache the value of x when

$x = 50$, with replication condition $A(x) = 10$. Then $\Delta_x = [41,59]$ is the global tolerance interval. Suppose now that the value of x becomes 59, and a new workstation begins to cache x (with the same replication condition $A(x) = 10$). Now the new global tolerance interval is the intersection of $\Delta_x^i = [50,68]$ and the old global tolerance interval, giving $\Delta_x = [50,59]$, much tighter (in effect, a window of size 5) than most users need. This tighter constraint may quickly cause future updates to generate a broadcast. However, the problem is self-correcting, since as soon as that broadcast is done, all the workstations will compute their local tolerance intervals centered around the same value.

Lastly, a few comments about static and dynamic selection conditions. As we explained in Section 3, users select the items to be cached at their workstation. If static caching is chosen, the identification clause is only checked once. However, if dynamic caching is specified, the central site must store the identification clause and any addition of an object that satisfies the clause should be transmitted to the cache. Similarly, a deletion of an object cached at the workstation should cause the invalidation of the cache entry. Clearly, dynamic selection conditions should be used sparingly, since they cause added storage overhead, as well as force the central site to check every object insertion and deletion to determine whether all dynamic selection predicates are satisfied.

6. A PERFORMANCE MODEL

An IRS with data caching and quasi-copies is a complex system. The performance improvements provided by caching quasi-copies, if any, will be determined by many factors, including processor capacities, transmission speeds, query reference patterns, and update reference patterns.

In an attempt to illustrate the tradeoffs involved, we will present a *simple* performance model and some results. We emphasize that our goal is not to predict the performance of some actual system; rather, it is to exemplify under what circumstances gains can be achieved and what their magnitude might be.

In our model, the central site contains the main database copy and receives all updates to it. Update arrivals form a Poisson process with average arrival rate λ_u. We assume that a fraction h of the database is cached at each workstation. Thus, with probability h an arriving update refers to data replicated at a particular workstation and should be forwarded. However, quasi-copies may reduce the number of updates that are actually forwarded. We let q represent the fraction of the updates that are actually sent out. That is, an update will be sent to a workstation with probability hq.

Queries arrive at each of the n workstations at an average rate λ_q. If the data required by a query were random, the probability that a query could be processed at a workstation would be h. However, we assume that cached data is more likely to be accessed by a query. We use the parameter f to denote how much more likely is the access of "hot spots" in the database. That is, a query can be processed locally with probability hf; with probability $1-hf$ it is forwarded to the central site. Note that $0 \le f \le 1/h$.

Table I contains a summary list of all the parameters, including some we will define shortly. In addition, the table gives the *base parameter settings* used in our evaluation. For each parameter there is a wide range of reasonable values; the base settings represent a particular set of choices within the reasonable ranges that best illustrate the effect of quasi-copies. (We will study later on the effect of varying these base settings.)

We model the central and workstation nodes as M/G/1 servers. The average service times for the various types of requests (all exponentially distributed) can be determined from the following parameters:

t_q^w: the processing time for a query executed at a workstation.

t_u^w: the time to install an update at a workstation.

t_q^c: query processing time at central node (20 times faster than a workstation).

t_u^c: update installation time at central node.

t_p^c: overhead time at central site to propagate an update to a single workstation. If the update is sent to x workstations, the time is xt_p^c. (The base setting for t_p^c, 0.0007 seconds, is chosen such that nt_p^c is the same order of magnitude as the other times at the central site.)

Since in our model we only have a single server per node, these processing times can be considered CPU times. For example, t_q^c is the CPU time used in processing a query at the central site. In reality there may be additional I/O time. However, in this simple model we have ignored I/O time for simplicity. (Since any realistic central server will have lots of memory to serve as a cache, we would expect the amount of actual disk I/O to be relatively small.) In each M/G/1 server, all requests are processed with the same priority on a first-come first-served basis. For the time being we are assuming that the cost of checking coherency conditions is negligible compared to the cost of performing an update. We will return to this issue shortly.

Parameter	Description	Value
λ_u	arrival rate	20 updates /second
h	fraction of DB cached	ranges from 0 to 0.2
q	coherency index	ranges from 0 to 1
λ_q	query arrival rate	1 query per sec. per station
f	hot spot factor	5
n	number of workstations	150
t_q^w	query processing time at workstation	0.1 second
t_u^w	update processing time at workstation	0.2 second
t_q^c	query processing time at central site	0.005 second
t_u^c	update processing time at central site	0.01 second
t_p^c	propagation overhead	0.0007 second
t_r	transmission time	0.3 second
n_x	number of concentrators	10 concentrators
t_x	concentrator service time	0.02 second

Table I

Initially, let us assume that the network transmission time is a constant t_r (we are assuming that workstations will communicate with the central database over phone lines; the 0.3 second setting we have chosen for this parameter is approximately the time required to transmit a 40 character message over a 1200 baud line). Since we have no queuing delays at the network, we are assuming, for the time being, that the processors, and not the network, are the potential bottlenecks. (Later on we consider a queueing model for the network.)

The average query response time is given by the expression

$$R = hf\,[w_w + t_q^w] + (1-hf)[2t_r + w_c + t_q^c]$$

where w_w and w_c are the average queue wait times at a workstation and at the central site. The first term in the equation covers the case where the query is processed at the workstation (probability hf); the second covers processing the query at the central site (probability $1-hf$). We now describe how the wait times can be computed.

At a workstation there are two types of requests. Queries arrive at a rate of $hf\lambda_q$ and require t_q^w service time; updates have an arrival rate of $hq\lambda_u$ and require t_u^w seconds. Let us call the two arrival rates λ_1 and λ_2; the two average service times θ_1 and θ_2. The combined flow of requests also forms a Poisson process with arrival rate $\lambda = \lambda_1 + \lambda_2$. The service time of the combined flow, X, is no longer exponentially distributed, but its mean and second moment are

$$E[X] = (\lambda_1/\lambda)\theta_1 + (\lambda_2/\lambda)\theta_2$$
$$E[X^2] = (\lambda_1/\lambda)2\theta_1^2 + (\lambda_2/\lambda)2\theta_2^2$$

(Recall that the second moment of a simple exponential distribution with mean θ is $2\theta^2$.) The wait time at a workstation can now be computed with the Pollaczek-Khinchin formula [Kl75]

$$w_w = \frac{\lambda E[X^2]}{2(1-\lambda E[X])}$$

The wait time at the central node is computed in a similar fashion, except that there are three types of requests. Updates are installed at a rate of λ_u and have an average service time of t_u^c. Propagations to the workstations occur at a rate of $hq\lambda_u$ and each has a combined load (over all workstations) of nt_g^c. Queries arrive at a rate of $n(1-hf)\lambda_q$ and take t_q^c each.

Figure 1 shows the response time R as a function of h, the fraction of the database at each workstation, for several values of q, the "coherency" index. (Recall that when h is 0, the workstations have no data and all queries are processed at the central site. When h is 0.2, each workstation holds 20 percent of the database. Given our f value of 5, this means that all queries are processed locally.) When $q = 1$ all copies must track the central site closely. In this case caching is not very helpful (for the parameters we selected). As data is cached and h increases from 0 to 0.2, query processing work is offloaded to the workstations. Unfortunately, the savings at the central node are offset by the increased effort of keeping the copies up to date. As h approaches 0.2, the workstations also become saturated with work since they not only must process the queries but must also execute the updates forwarded to them.

As q decreases, the situation changes dramatically. Copies can diverge from the central value, so not every update must be propagated. The effort in update propagation is reduced and all computers can process the queries more quickly. When q is 0.75, a fourth of the updates are not propagated. In this case, the best response time occurs when about 14 percent of the database is cached. The central site has shed enough load to make it operate more efficiently, but not enough to overload the workstations. With values of q of 0.5 or less, the workstations never become overloaded, so it becomes feasible to perform all queries locally ($h = 0.2$) and save the transmission costs.

The total expected number of messages transmitted per second is given by the expression

$$M = 2n(1-fh)\lambda_q + nhq\lambda_u$$

The first term corresponds to query messages; the second term to update traffic. Figure 2 displays the message rate M as a function of h and q. Here again, caching can increase the network load. Quasi-copies (q less than 1) can reduce the network traffic and can even make it less than in the no caching case.

Our model so far assumes that there are no queueing delays in the network. What would happen if there were? To answer this question, we extended the model to include a limited network capacity. Suppose that the lines from the workstations arrive at concentrators or communication controllers. Say there are n_x concentrators, each handling n/n_x lines. We model each concentrator as an M/M/1 server with service time of t_x. The arrival rate at each concentrator is M/n_x, assuming that messages in either direction are equivalent. The expected total transmission delay for one message is given by

$$D = t_r + w_x + t_x$$

where w_x is the expected wait time at a concentrator. (Each queue is independent.) Note that t_r is now the transmission time from workstation to concentrator; we still assume this component is a constant.

If we substitute the above expression for t_r appearing in the R equation given earlier, we obtain the query response time for the new model. This response time is shown in Figure 3 for a value of t_r of 0.2 seconds. Comparing this figure to Figure 1, we observe that only the case for q=1 is substantially different. When q=1 the network traffic is sufficiently high to cause the network to saturate at about h = 0.13. As this limit is approached, the query response time grows without bound. For the other values of q, there is little change. The reason is as follows. When h is 0, the network is handling 300 messages a second yielding a delay D of 0.25 seconds (compared to 0.3 seconds in Figure 1). As h increases, the number of messages decreases and the delay shrinks to D = 0.2. However, as h increases, the impact of D on the average response time decreases (fewer queries must be transmitted), so that the impact of the transmission time savings is not noticeable.

Up to now we have assumed that the cost of checking coherency conditions is negligible compared to the cost of installing an update, t_u^c. This may be true in some cases (see Section 5), but what happens in those cases where it is not? Let t_{u+ck}^c be the combined cost of installing an update and checking any coherency conditions involved. Our performance model and equations are identical, except that t_{u+ck}^c replaces t_u^c. The value of t_{u+ck}^c will be larger than t_u^c, but estimating how much larger is difficult. It depends on the type of coherency conditions and their number.

Instead, let us compute how much larger t_{u+ck}^c *would have to be* so that the gains obtained from quasi-caching are lost. In other words, let R_0 be the response time when no caching, conventional or otherwise, is used. (In Figure 3, R_0 is 0.62 seconds.) With quasi-caching, the response time will be R, a function of t_{u+ck}^c and the rest of the parameters. We can numerically search for the value of t_{u+ck}^c that makes R equal to R_0.

This value of t_{u+ck}^c is shown in Figure 4 as a function of q, for h = 0.12 (using the extended network model). The area under the curve represents the values of t_{u+ck}^c where quasi-caching performs better than no caching. For example, when q = 0.75, t_{u+ck}^c must be more than twice as large as t_u^c so that both response times are equal. (Recall that the time to install an update, t_u^c, is 0.01.) This means that if the overhead of checking coherency conditions is less than the cost of actually installing the update, then quasi-copies will have a better response time. This is very encouraging since it is much more likely that the cost of checking coherency will be a *fraction* of the installation cost, as opposed to a multiple of it. And even if the cost of checking equaled that of installing, there will be no gain in response time but we would still be sending many fewer messages.

For values of q less than or equal to 0.85 it is clear that we can afford to spend a substantial amount of time checking coherency constraints without endangering our gains. However, as q gets close to 1, t^c_{u+ck} drops to zero. This is to be expected: quasi-copies are not buying us anything, so we cannot afford to spend anytime checking coherency conditions. Although we do not show it here, we have the same problem if h, the fraction of the database that is cached at a workstation, approaches zero.

Finally, we have also studied the sensitivity of the results to our choice of base parameter settings. Before showing some of our results, let us define

$$\delta = R\,(q{=}1,\, h{=}0.16) - R\,(q{=}0.5,\, h{=}0.16)$$

The value δ gives the saving in response time when we go to quasi-caching (with q = 0.5), as compared to a system with conventional caching, for the case when we have 16 percent of the database cached at each workstation. From Figure 1 we see that with our base settings δ is roughly 0.31. In other words, quasi-copies ''buy us'' a savings of 0.31 seconds in query response time. Of course, these are the savings in just one case, but let us study what happens to these savings if we change the base parameter values.

Figure 5 shows the value of δ as some of our parameters are individually changed. (For simplicity we once again ignore network queuing delays and the overhead of checking coherency conditions.) On the horizontal axis we plot the percentage deviation from a base setting. For example, if the value of f were 25 percent smaller than the value given in Table I (5*0.75), quasi-copies would only reduce the query response time by about 0.27 seconds. On the other hand, if f were 25 percent larger (5*1.25), the savings would be 0.40 seconds. The parameters *not* shown in Figure 5 behave very similar to t^c_u. These parameters do affect response time, but variations in their value have minimal impact on the difference between R at q = 1 and at q = 0.5.

What is interesting to note in Figure 5 is that with the exception of t^w_u, decreases in δ are very gradual, while the increases can be abrupt. This is good news for quasi-copies, we think. In other words, suppose that we have analyzed a particular system and have estimated that quasi-copies are beneficial. If our estimate were slightly pessimistic, then quasi-copies could be much more useful than what we had anticipated. However, if our estimates were optimistic, quasi-copies will be slightly less advantageous but will not be drastically counterproductive (unless of course our parameter estimates were way off). When t^w_u decreases, the drop in δ is more significant, but even with a 50 percent change, δ is still positive, i.e., quasi-copies still are advantageous.

In summary, the objective of conventional caching is to distribute the query processing load. However, conventional caching may be counterproductive because of updates: updates to cached data generate more messages and processing load. Fortunately, quasi-caching reduces the cost of update propagation and may make caching feasible and very worthwhile. Our performance model has helped us identify these situations.

We have studied other extensions to our basic model but we do not discuss them here. In essence, a more detailed model may cause saturation points and values to change, but we believe it does not change the basic tradeoffs we have shown here.

7. TWO EXAMPLES

In order to illustrate the use of the various concepts and mechanisms we have discussed, we now present two more extensive examples. The first one is that of the Boston

Community Information System implemented at M.I.T. [Gi85], which can be studied in the context of quasi-copies. This system consists of a central site with various databases, with users located throughout the Boston area. The central site is constantly transmitting (via FM radio) the entire databases. Users receive and process the data at their personal computers (PC's). Each user selects the type of data that he is interested in. That is, the PC acts as a filter and stores locally the parts of the database that are of interest.

In this system caching is clearly a must. If no local data were stored, a user query could only be answered by waiting for the answer to be transmitted. As of 1985, the time to transmit the database was 4 hours. This means that on the average a query would take 2 hours! Selection conditions are also important, for without them the entire database would have to be stored locally. The selection conditions are of type add, dynamic, and compulsory (see Section 3). The triggering delay is set by the system and is equal to the time it takes to transmit the full database. Objects are dropped from the cache manually by each user.

Multi-object conditions, if they are required, pose an interesting problem. Say for example that there is a consistency constraint that involves objects x and y. At the end of a database transmission the images will satisfy the constraint. Let $x'(a)$ and $y'(b)$ be the images at this time. During the next database transmission, there can be a time when one image has been updated but not the other. Since the values $x'(a+1)$ and $y'(b)$ may be inconsistent, they cannot be read by the same query. The only reasonable choice is to save the old values until the end of a database transmission, at which time they can all be installed atomically. Since it would be costly to save all the old values, it would be desirable to let users define their multi-object constraints explicitly. This way, the system would only save old versions involved in these types of conditions.

In addition to broadcasting the full database periodically, the system also transmits new items with a higher frequency. This in essence defines two types of triggering delays. When a selection condition is first installed, it may take longer to cache data that is old. However, once the initial database transmission delay takes place, new objects that satisfy the condition will be cached more promptly. The ability to broadcast some data at a higher frequency opens the door for a number of additional improvements. For example, the database could be divided into fragments and each fragment could have a different triggering delay defined. Updates do not have to be transmitted as part of the database; they can be broadcast any time after they take place. This shortens the window for the coherency conditions, and again the system can give some fragments shorter windows. All of these improvements will become more important as the size of the database and its full transmission time increases.

The system designers also plan to add a dial-in capability to the system. This can let users define selection and coherency conditions that are stricter than the default ones, and can include non-delay ones like version number or arithmetic conditions. The system would use these conditions to schedule the transmission of data.

As our second example, consider a calendar service that is to be provided to a hypothetical large corporation. The calendar is to be accessed from a large number of minicomputers located at the various departments and branch offices. Each calendar is a sequence of "days" 1, 2, 3, ... Each day object contains a list of events and descriptions, but the internals are not relevant here. There can actually be a set of different calendars, each one identified by a "type" (e.g., for the New York location, for the managers, etc.). The most common operation, as one might expect, is to look up the events for a given calendar and day. Of these, the lookups for the current day are the most frequent. Entries in a calendar can be added, deleted or modified. Such modifications typically must ensure that the obvious consistency constraints are satisfied (e.g., two events should not conflict).

There are a number of ways to implement the calendars. A full centralization approach is simple but leads to long delays for queries and poor availability. If the calendars are replicated but tight consistency is required, then all updates must be propagated immediately, regardless of how congested the systems and the communication network is. If control of the copies is distributed, then concurrent updates must be synchronized. This can be very difficult if network partitions occur. Furthermore, since updates originate at various places, it is not possible to batch them together for broadcast efficiency. On the other hand, a centralized solution with remote quasi-copies, while not perfect, does have some nice features. It can yield a good query response time and availability, while at the same time limiting the communication traffic.

To design the calendar system with quasi-copies we must chose the selection and coherency conditions that will be enforced. Here we will illustrate by choosing a set of reasonable ones, but clearly other choices are possible. We dynamically select to cache the next 30 days, starting at the current day c, for those calendars of interest to the local community. When a new calendar type is added to a selection, we would like to get it as soon as possible, so we define a small triggering delay. We make the selections compulsory so that all queries regarding the next month can be executed locally.

We use time varying coherency conditions. If a day in a calendar x represents c or $c+1$ (the current or the next day), then we define $W(x) = 15$ minutes and $V(x) = 3$. This ensures that all copies will be at most 15 minutes or three versions out of date for events occurring today or tomorrow. For days $c+2$ through $c+29$, we define a periodic condition $P(x)=6$ *am day* 0, 24 *hours* to guarantee that the calendars are refreshed at 6 am every day. In addition we define $V(x) = 3$ to ensure bursts of update activity are propagated quickly. Finally, we define each calendar to be a group of objects with multi-object constraints. This way, if a transaction schedules two events in the same calendar, say one for day $c+1$ and the other for $c+2$, both will appear in the cached copies, even if the second one did not violate the simple conditions.

To cope with failures, it does seem convenient to have a $C\_FAILED(j)$ accessible to users at site j, as discussed in Section 5. The interval for sending ''I am alive messages,'' δ, should be set at 15 minutes to be compatible with the coherency conditions for the current day.

As we know, there are many ways to implement the selection and coherency conditions we have given. In any case, since the conditions are simple and uniform for all calendars and sites, the overhead of managing the quasi-copies should be low. To propagate an update, sending the new value seems most appropriate here. However, implicit invalidation can be used to remove the current day from the cache at the end of the day. That is, all data broadcast for a day x includes an implicit expiration time of $x+1$, meaning that at day $x+1$ the data is no longer valid and should be purged. With respect to when to propagate, last minute propagation appears to be the most appropriate for this application. The division of labor is also clear: the central node executes updates and checks the conditions; the local nodes execute most of the queries.

8. CONCLUSIONS

In practice, quasi-copies are already in use for all types of information and data. However, they are mostly used in an ad-hoc fashion, outside of computer systems, and without any validity guarantees. In this paper we have suggested that quasi-caching can be useful in a computerized IRS, reducing substantially the overhead of managing replicated data and making data available during failure periods. We have defined the notion of quasi-copy, presented the types of conditions it can satisfy, and discussed the mechanisms with which a system can take advantage of the added flexibility.

As we have shown in Section 6, quasi-caching can potentially improve performance and availability. However, there are also potential problems. This may be so: (1) if a large fraction of the updates have to be propagated to the user workstations, i.e., if q is close to 1.0; (2) if the selection and coherency conditions are complex, the overhead of the bookkeeping (large t^{c}_{u+ck}) may outweigh the savings. This must be kept in mind when considering quasi-caching as an alternative.

Quasi-caching also raise a number of challenging questions: How much data should be cached? Which of the mechanisms we have outlined is better suited for a particular application? How does the choice of when to propagate updates affect the performance of the system? What are the breakpoints that make caching and quasi-copies worthwhile? At the user end, there are also open questions: What language is used to define conditions? How does a user determine the delays or deviations that are best suited for him? We are currently implementing a testbed to empirically answer these and other questions.

Acknowledgements: This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, by the National Science Foundation under Cooperative Agreement No. DCR-8420948, and New Jersey Governor's Commission Award No. 85-990660-6, and grants from IBM and SRI's Sarnoff Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

9. REFERENCES

[BuCl79] Buneman, P. O. and Clemons, E. K., ''Efficiently Monitoring Relational Databases,'' *ACM Transactions on Database Systems*, Vol. 4, No. 3, September 1979, pp. 368-382.

[Bl86] Blakeley, J.A., N. Coburn, and P.A. Larson, ''Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates,'' Proceedings of the 12th International Conference on Very Large Databases, pp. 457-466, Kyoto, Japan, August 1986.

[Da75] Date, C.J., ''An Introduction to Database Systems,'' *Addison-Wesley,* 1975.

[Da85] Davidson, Susan, Hector Garcia-Molina, and Dale Skeen, ''Consistency in Partitioned Networks,'' *ACM Computing Surveys,* vol.17, no. 3, September 1985.

[Du84] Dunn, Bill, ''Bill Dunn of Dow Jones: The Data Merchant,'' *Personal Computing,* pp. 162-176, December 1984.

[EdDa83] Edelhart, Mike and Owen Davies, ''OMNI Online Database Dictionary,'' Collier Mac-Millan Publishers, 1983.

[Gi85] Gifford, David, John M. Lucassen, and Stephen T. Berlin, ''The Application of Digital Broadcast Communication to Large Scale Information Systems,'' *IEEE Journal on Selected Areas in Communication,* May 1985.

[Gl86] Gladney, Henry, ''A Model for Distributed Information Networks,'' Research Report RJ5220, IBM Research Laboratory, San Jose, California, July 1986.

[HaSa78] Hammer, M. and Sarin, S.K., ''Efficient Monitoring of Database Assertions,'' *Supplement Proceedings ACM SIGMOD International Conference on Management of Data*, 1978.

[Ha87] Hanson, E., ''A Performance Analysis of View Materialization Strategies,'' Proceedings of the International Conference on Management of Data, pp. 440-453, 1987.

[Kl75] Kleinrock, Leonard, ''Queueing Systems, Vol. 1: Theory,'' *Wiley-Interscience,* 1975.

[La78] Lamport, Leslie, ''Time, Clocks, and the Ordering of Events in a Distributed System,'' *Communications of the ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[Li86] Lindsay, B. , L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, ''A Snapshot Differential Refresh Algorithm,'' Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 53-60, Washington D.C., May 1986.

[RoKa86] Roussopoulos, N., and H. Kang, ''Principles and Techniques in the Design of ADMS+/-,'' *Computer*, December 1986.

[Sm82] Smith, Alan, ''Cache Memories,'' *Computing Surveys,* vol. 14, no. 3, September 1982.

[StRo82] Michael Stonebraker and Lawrence T. Rowe, ''Database Portals: A New Application Program Interface,'' Electronics Research Laboratory Report UCB/ERL M82/80, U.C. Berkeley, November 2, 1982.

[Te87] Douglas B. Terry, ''Caching Hints in Distributed Systems,'' *IEEE Transactions on Software Engineering, January 1987.*

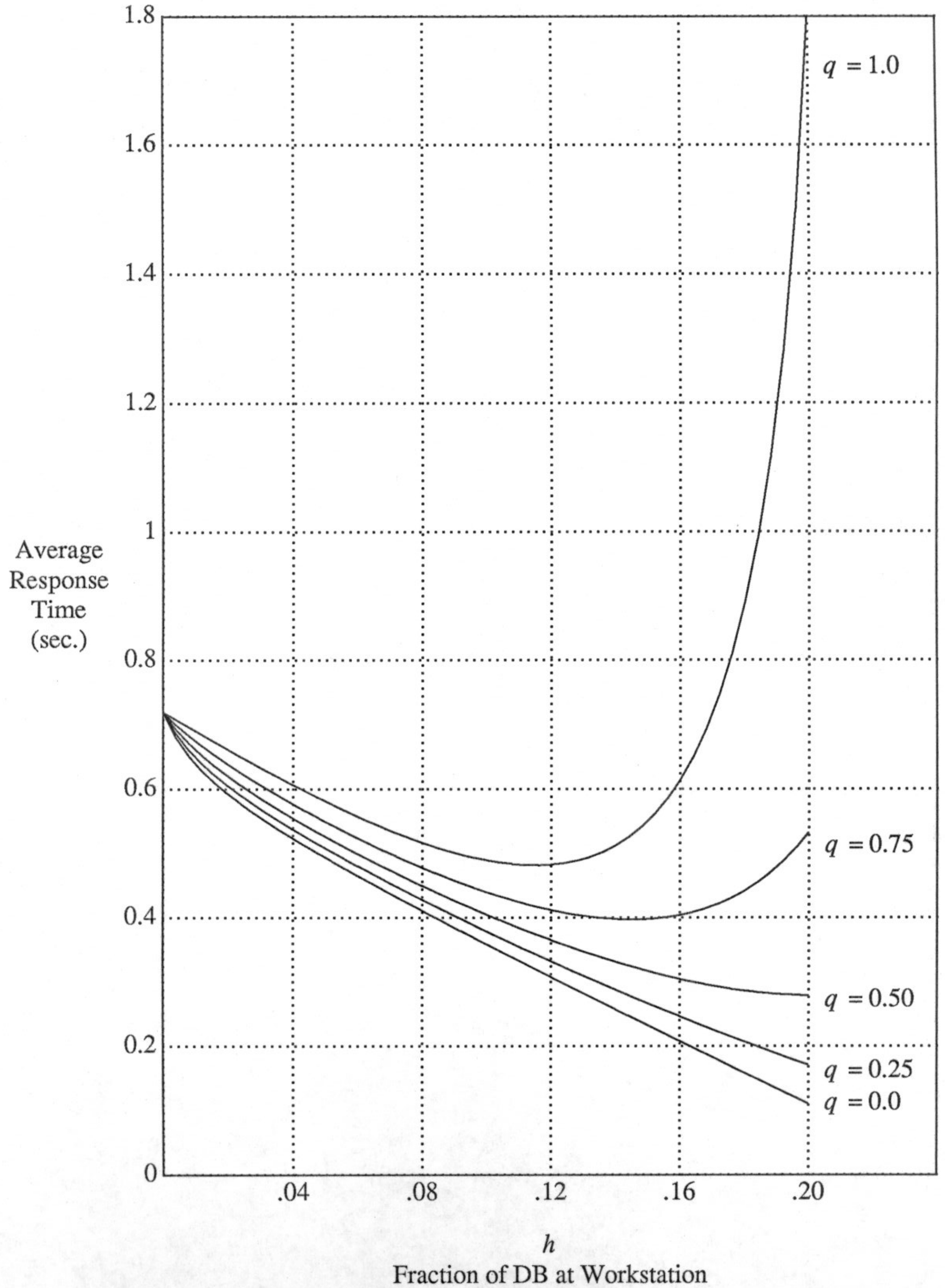


Figure 1
Basic Model

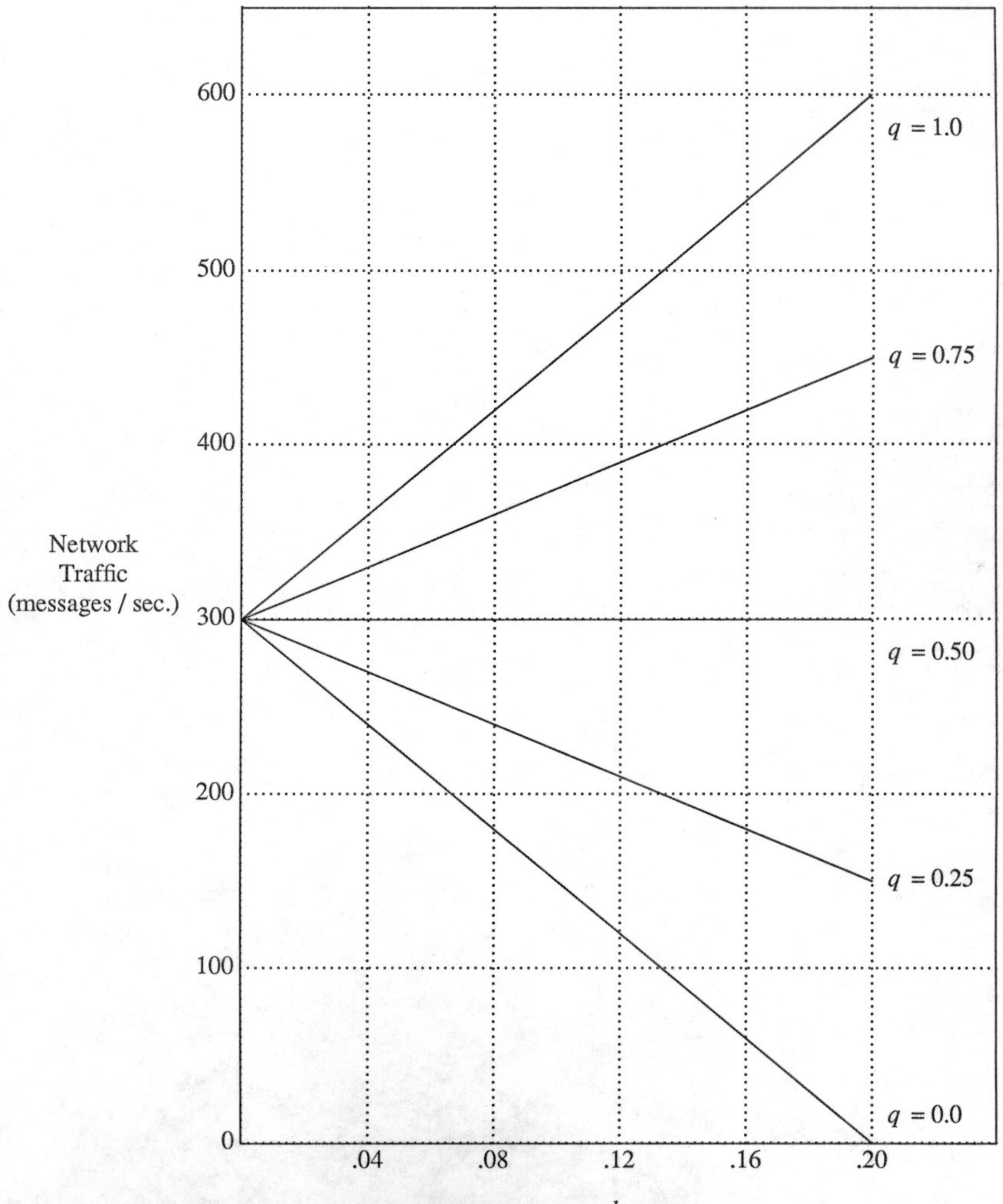


Figure 2
Basic Model

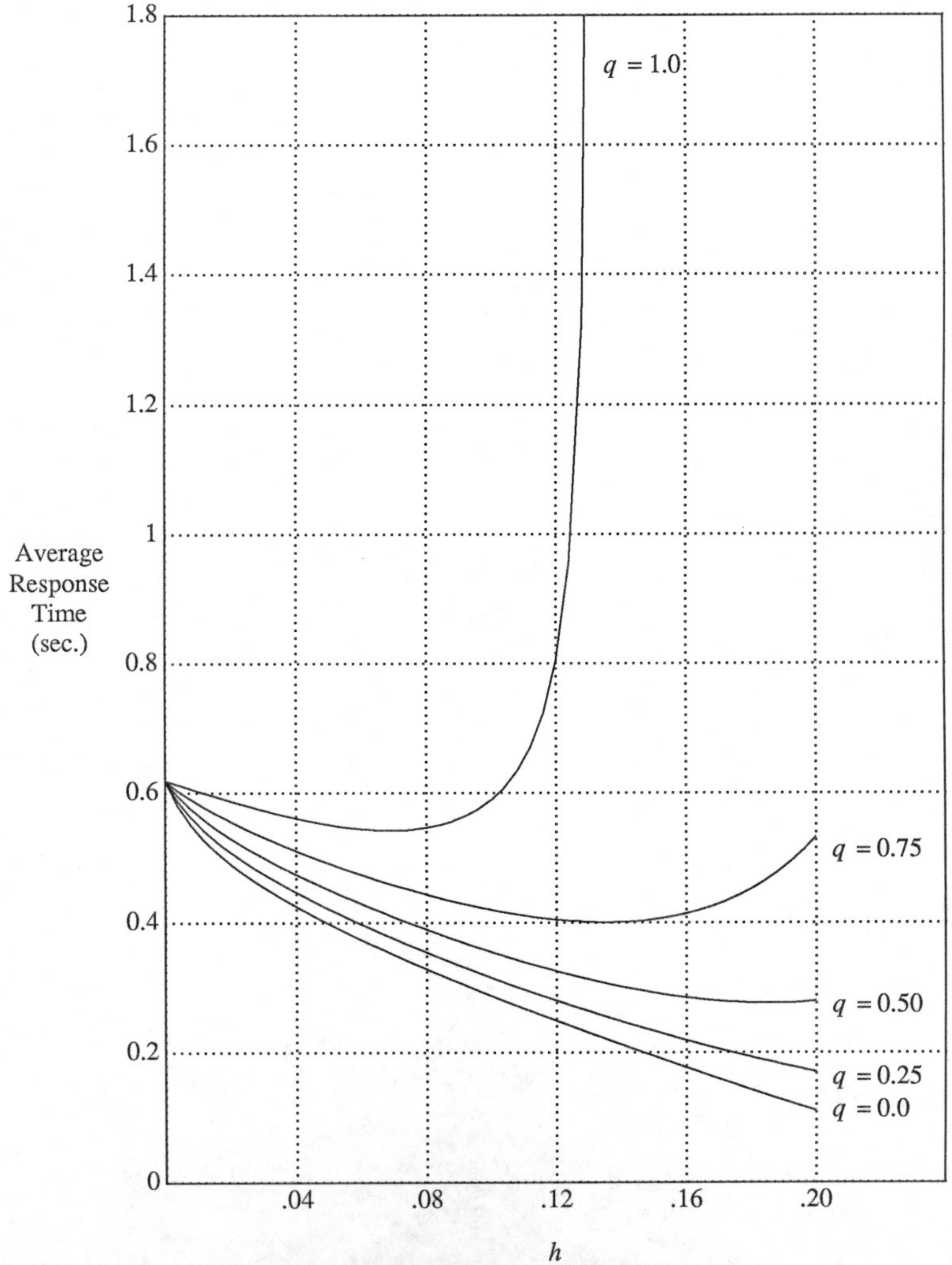


Figure 3
Extended Network Model

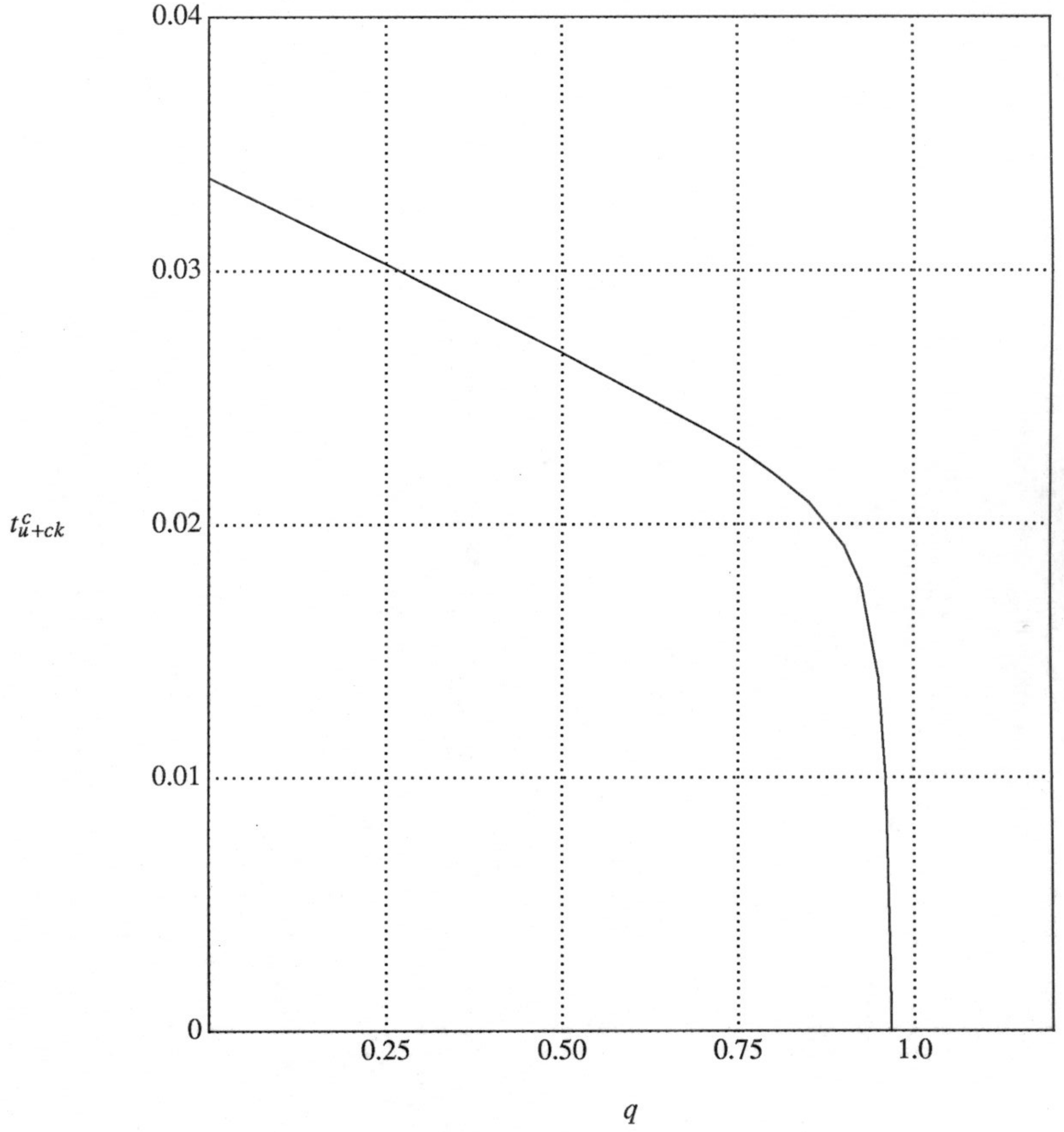
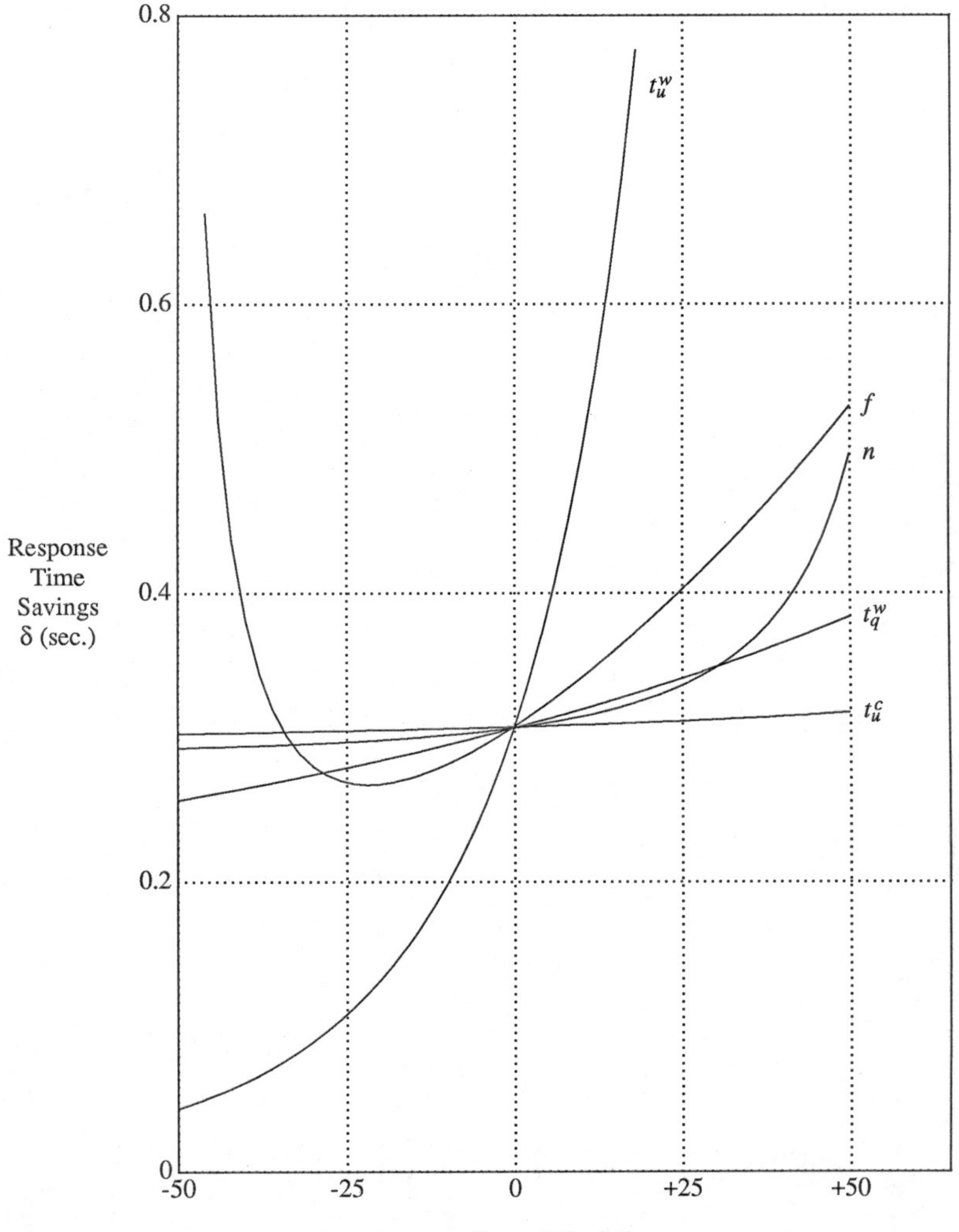


Figure 4
Basic Model



0.8
0.6
0.4
0.2
0
-50
-25
0
+25
+50
Response Time Savings δ (sec.)
Percent Deviation From Base Setting
t_u^w
f
n
t_q^w
t_u^c

Figure 5