Efficient Data Access to Multi-channel Broadcast Programs

Wai Gen Yee
Department of Computer Science
Illinois Institute of Technology
10 West 31st Street
Chicago, IL 60616
yee@iit.edu

Shamkant B. Navathe
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
sham@cc.gatech.edu

ABSTRACT

This paper studies fast access to data that are broadcast on multiple channels. Broadcast is a useful data dissemination technique because of its scalability, but is lacking when it comes to response time. Increasing the number of available broadcast channels is a logical way of increasing throughput. Little work, however, has considered the access structures necessary for making effective use of the additional channels. We propose various indexing schemes for a multi-channel broadcast program. We demonstrate the effectiveness of our techniques in decreasing response time and tuning time via extensive experiments over a wide range of parameters.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms, Design, Performance

Keywords

broadcast, data dissemination, scheduling, indexing, query processing, access methods

1. INTRODUCTION

In this paper, we study fast access to data that are broadcast over *multiple channels*. Broadcast is a well-known way of scalably transmitting data to clients [11]. It is scalable because a single transmission of an item can satisfy all outstanding requests for it. On the other hand, broadcast suffers from high response times due to the sequential nature of data access. One way around this problem is to increase the available bandwidth by increasing the number of broadcast channels. Simply adding channels, however, does not improve response times. We also need structures and protocols that allow the effective use of the channels.

Broadcast is a useful medium for disseminating data in environments where there are many clients and communication capabilities are asymmetric–the server possesses much greater communications capabilities than do the clients. Such a condition precludes traditional client-server communications because there may be too many clients for the server to handle, either in terms of computing capacity or in terms of available bandwidth. Mobile clients may also benefit from a broadcast architecture in terms of energy expenditure. Such clients do not need to expend energy maintaining a two-way wireless connection. In terms of performance, broadcast's chief benefit is its scalability, as mentioned above.

Although scalable, the response times of requests made to broadcast data are generally high. Clients must typically wait for unwanted data to be transmitted before desired data are available. For example, if N items are sequentially (or *flatly*) broadcast, the client should expect $\frac{N}{2}$ of them to pass before its request is satisfied. One way around this problem is to allocate some bandwidth to on-demand requests: items not broadcast soon enough can be satisfied on channels reserved for explicit requests [8, 16]. Another way around this problem is to *skew* the broadcast program so that popular data are broadcast more frequently. For example, given multiple channels, some may be designated exclusively for those few frequently requested items, while the bulk of data, which are infrequently accessed are allocated together on other channels [1, 12, 19, 20].

Although this paper does not preclude support for on-demand request channels, it focuses on making those channels designated for broadcast perform better. Careful consideration of the broadcast environment makes clear the need for special structures to aid data access. If a client can only listen to a single channel at a time, and has no *a priori* knowledge of how data are allocated to them, then increasing the number of channels may not improve the response time for a request. In the worst case, a naive client may have to scan all the channels before finding the desired data. Searching for data in this way yields performance no better than listening to a *flat* broadcast program on a single channel.

We propose adding index information to a skewed broadcast to aid client search. Indexing broadcast schedules is traditionally a means of conserving energy [13]. At the expense of broadcasting more data (and thereby *increasing* response times of client requests), the index informs clients of item arrival times. Such information allows client optimizations

CIKM'03, November 3–8, 2003, New Orleans, Louisiana, USA.

such as energy saving via *dozing*, and improved cache management [2]. In addition, in contrast to the single-channel case, indexing a multi-channel broadcast program can actually reduce request response times as we will show.

1.1 Motivation for Multiple Channels

There are several reasons for dividing data over multiple channels instead of sending them over a single fast channel. Some are based on the physical constraints of the architecture. For example, clients may have heterogeneous communication capabilities, limiting the feasibility of single channel high-speed transmission [7].

There are also application-based uses of multiple channels. For example, channels can be combined or coordinated to offer variable quality of service to clients [10, 24]. They can also be used to ensure service quality in hybrid push-pull systems [16]. Bandwidth conservation is yet another use. Multiple multicast channels have been used to partition Web data in order to reduce total traffic [4, 18]. Finally, some applications (e.g., voice) typically require only a small fraction of the total available bandwidth, so leftover bandwidth can be used by other applications (e.g., other voice conversations).

Industry trends as well as government regulations may also play a role. Commercial digital radio satellites have already been deployed over the United States, and due to FCC mandates, commercial broadcast radio will soon also be digital [15]. In addition, the FCC has eased the ownership policy of commercial radio stations. One entity can now own multiple radio stations in a single market. Clear Channel Communications (`www.clearchannel.com`), for example, currently owns 9 radio stations in the Atlanta area. A new infrastructure that allows multi-channel data broadcasts is in development.

In this paper, we assume that clients are looking for individual items from a database that is broadcast over multiple channels. Consider a PDA programmed to request stock prices in this environment. What can the server do to aid the client's search? What protocols should the PDA follow? Previous work done in the single channel case [13] does not apply here, and little work has been done in the multi-channel case [20]. We address these issues in this paper.

We begin by formalizing our model and problem statement in Section 2. We discuss the salient features of the problem and discuss the shortcomings of previous approaches to solving it in Section 3. We then offer various index designs and describe techniques for applying it to a multi-channel broadcast program in Section 4. In Section 5, we present experimental results that demonstrate the performance benefits of our techniques over a wide range of parameters. Finally, we conclude the paper and discuss future work in Section 6.

2. MODEL

The database contains N equal-sized items, d_i, each identified by a key, and an access probability (or popularity), p_i, where $\sum_{i=1}^{N} p_i = 1$. These popularities are subject to change, depending on the server's perception of client interests. There are K equal bandwidth channels, C_i, each of which cyclically broadcasts N_i unique items ($\sum_{i=1}^{K} N_i = N$). The popularity of C_i is $P_i = \sum_{d_j \in C_i} p_j$, where $\sum_{i=1}^{K} P_i = 1$. Each item fits into a unit of transfer called a *bucket*, and each bucket requires a *tick* of time to transmit.

A client request consists of a single item. We assume that requests are exponentially distributed, so that at each tick, the probability of a given client requesting d_i is solely determined by p_i.

A client can do three things while trying to satisfy a request. It can *actively listen* to the broadcast program, *doze*, or *hop* to another channel. While actively listening, the client downloads the data currently broadcast on the channel. Active listening has been linked to client energy expenditure, and is measured in ticks by a metric called *tuning time* [13]. Dozing is an energy-saving feature common in mobile devices; while dozing, the client does not download any data, but its tuning time does not increase either. Finally, we assume that a client can only listen to a single channel at a time, so in order to read data from another channel, the client must hop to it. Hopping may take some non-zero time, depending on the broadcast medium, and when a client finishes a hop, it must *synchronize* itself to the broadcast. We informally define synchronization as the process of retracking the transmission signal of the channel. One thing a client may do during synchronization is read the current bucket being broadcast in the channel in order to gather metadata important for searching the channel.

The primary metric we consider is *average expected delay* (abbreviated *aed*). As the name implies, *aed* is the typical time required to satisfy a request. If we assume that the interarrival time of each d_i is fixed at r_i, *aed* can be expressed as the sum of the expected interarrival times of all items, weighted by their respective popularities:

$$\text{Average Expected Delay} = \frac{1}{2}\sum_{i=1}^{N} p_i r_i. \qquad (1)$$

This paper has two goals. The first goal is to design structures and protocols that allow clients to find requested data while minimizing both *aed* and *tuning time*. (*Hop count* is a secondary metric, because its importance depends on its impact on *aed* and *tuning time*.) The second goal is to incorporate these structures and protocols into a skewed multi-channel broadcast program.

3. THE SHORTCOMINGS OF PREVIOUS WORK

Previous work makes three assumptions that undermine its practicality. The first assumption is that hopping can be done instantaneously. Consider what happens when a typical AM radio tunes in to some frequency: the tuner adjusts to the correct frequency; the demodulator then decodes the signal; and finally, the receiver begins tracking the signal. In this case, hopping requires some variable non-zero amount of time, which is anecdotally on the order of milliseconds.

Models that assume instantaneous hopping allow clients to read any of the K items that will be broadcast next [17]. This assumption, in principle, allows a K-fold increase in throughput. In order to achieve this performance improvement, the server coordinates the allocation of data and index among the channels in a way that relies heavily on the instantaneous hop assumption. The resultant broadcast program performs very well if the assumption is true, but very poorly otherwise.

The second assumption is that clients can listen to multiple channels simultaneously [23]. In practice, however, clients may be physically incapable of listening to more than a single channel at a time. Radio devices typically have only one receiver, and unless channels are only *logical* (e.g., implemented using time-division multiplexing), the devices cannot listen to more than one at a time.

Furthermore, this assumption is strong because it somewhat reduces the problem to that of single-channel broadcast design, which has already been covered extensively in the literature [1, 3, 11, 13]. Single channel techniques are inapplicable to the multi-channel environment because their models fail to consider the issues mentioned in this section, such as hop cost. In addition, the work just cited considers minimizing either response time or tuning time exclusively, but we consider minimizing both in this paper.

The third assumption is that clients have *a priori* knowledge of the broadcast schedule [12, 19, 20]. In other words, the clients know the arrival time and channel of all broadcast data items. Indexing information becomes irrelevant, and the problem reduces to one of optimally allocating popular items to smaller channels, as in [27]. The *aed*, in this case, can be expressed as one-half the sum of the channel sizes, each weighted by its respective popularity:

$$aed_{apriori} = \frac{1}{2}\sum_{i=1}^{K} N_i P_i. \quad (2)$$

By assumption, the relative popularity of items can change, affecting the optimality of any broadcast program. Broadcast programs must adapt to these changes, rendering any a priori knowledge obsolete. The server must therefore use an index to convey changes made to the schedule.

4. DESIGNING EFFICIENT ACCESS BROADCAST PROGRAMS

Our solution consists of first determining a bandwidth efficient schedule, and then supplementing it with an index. In other words, we minimize *aed* before considering *tuning time*.

We divide the design process into two distinct parts because it is difficult to organize data by both popularity (which is done by scheduling algorithms) and by key (which is done by indexing techniques) simultaneously, as key and popularity are generally uncorrelated. Furthermore, organizing data primarily by key, as done in [13] to index a single channel broadcast database, generally forces a data organization that precludes the possibility of further manipulation. As a result, the lower bound for *aed* in key-based broadcast programs is $\frac{N}{2}$. In constrast, it is possible to index popularity-based broadcast program as we will show.

Note that the authors in [14] assume that keys can be assigned to data items based on their popularities, simplifying the broadcast program design problem. However, they fail to describe how key mappings can be efficiently conveyed to clients.

4.1 Scheduling the Data

In order to be efficiently indexable, a broadcast program must exhibit some regularity. That is, there must be a simple way to describe each item's arrival pattern. This condition precludes the use of the real-time scheduling algorithms described in previous work [3], or a similar monolithic algorithm described in [5]. The interarrival times of items in schedules generated by these algorithms exhibit a high degree of variability. Furthermore, these scheduling algorithms are designed exclusively for the single-channel environment.

Some existing multi-channel scheduling algorithms do generate regular schedules. Of these algorithms, we pick one called $GREEDY$ because of its demonstrated computational simplicity and performance [27]. $GREEDY$ works by successively splitting a sequence of items, ordered by p_i at the point that most reduces $aed_{apriori}$ (Equation 2) until there are K partitions. Each partition is allocated to a channel which cyclically broadcasts its contents. This technique has been shown to produce multi-channel schedules that are near-optimal in terms of *aed*. More to the point, the resultant schedule is clearly regular. Each item's location in the broadcast program can be described with two numbers: a channel number and a sequence number. The channel number indicates the channel to which the item is allocated. The item's sequence number describes the order in which the item is broadcast within the channel. An index can point to a particular item using this pair of numbers.

4.2 Indexing the Scheduled Data

In this section, we offer two alternatives for indexing a multi-channel broadcast program, assuming that the index is a balanced n-ary tree (e.g., a B+-tree). In the first alternative, which we call IAD, we treat the index as a single, large data item that is broadcast on a well-known channel. In the second alternative, which we call *stripe*, we sequentially stripe index levels across all K channels. Below, we discuss ways to design these indices as well as protocols for clients to use them.

4.2.1 The Index as an Item

With the index is an item, the client tunes in to the channel on which the index is located (C_{index}), traverses the index, finds the relevant pointer, and finally hops to the correct channel to complete the search. The major benefit of IAD is that the client has to perform at most one hop (excluding the one needed to initially tune in to the broadcast) to find an item. The downside is that traversing the index may take a long time, as the index item may be large. In this section, we describe ways of designing and incorporating the index item into the broadcast program.

We first consider when to broadcast the index item. Because we treat the index as an item, we can schedule it using $GREEDY$. $GREEDY$ schedules items defined by size and popularity. The size of the index item is a function of the capacity of a bucket. We assume that each bucket that stores the index contains as much data as possible. We define the index item's popularity to be 1, relecting the fact that, for each request, the index must be referenced.

We now consider ways of laying out the tree nodes within the buckets of the index item. As a tree, a *feasibility* constraint requires that ancestor nodes be broadcast before their descendants. There are two basic feasible tree layouts: breadth-first, where the index nodes are laid out level-by-level from root to leaves; and depth-first, where items are laid out according to a pre-order traversal. Breadth-first is simple to understand and analyze, however, pre-order traversal minimizes the average distance from the root to a leaf. With the latter method, therefore, clients can find

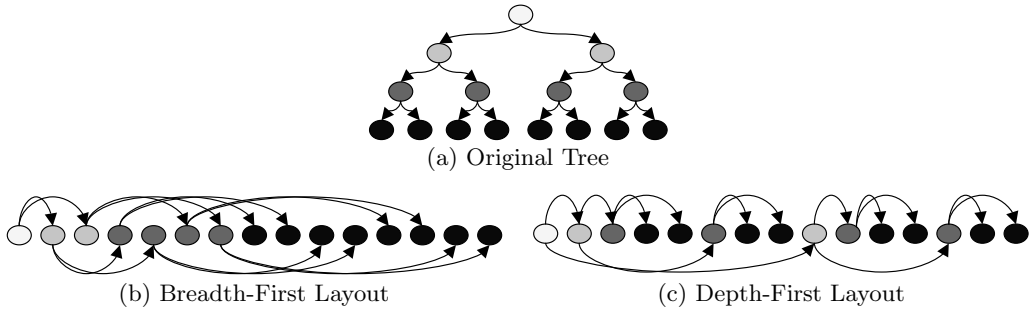


Figure 1: Canonical *IAD* Tree Layouts.

and follow a data pointer sooner. For this reason, we consider only depth-first index layouts in this paper. (Analyses and experiments using breadth-first indices can be found in [26].) Figure 1 graphically compares the two layouts, where arrows show possible tree traversal paths.

Client Search Protocol. In order to find data, the *traditional* client first tunes in to channel C_{index}, finds the root of the index, traverses the index, and then follows the pointer to data. If the first bucket a client reads does not contain the root, the client can use a *root pointer* contained in every bucket to find it.

Opportunistic search can improve *aed* over that of traditional search at the expense of tuning time [14]. Instead of waiting for the index's root, the client can start the search at the first subtree it finds. By adding the key range of the subtree rooted at each node, the client can determine if it should traverse that subtree. If the desired key is not in the node's key range, then it either checks the next subtree's key range or returns to the root. Opportunistic search works best when the index spans many buckets. More details of opportunistic search can be found in [26].

4.2.2 Striping the Index Over K Channels

An alterative to the *IAD* index organization is to stripe the index over the K channels. With striping, index levels are sequentially broadcast in substripes at a frequency f. Because clients can listen to only one channel at a time, and each request consists of a single item, the stripe design conserves bandwidth by transmitting desired and undesired index nodes simultaneously. The entire index can be broadcast and traversed on the order of K times more quickly. The downside of striping, however, is that clients may have to perform a hop to read each level of the index.

In designing an index stripe of width W, we first determine its broadcast frequency, f. Because requests arrivals are exponentially distributed (Section 2), fixing f minimizes average interarrival time as a consequence of the *inspection paradox* [21]. The choice of f must balance the amount of bandwidth allocated to index information and to data in order to minimize aed_{stripe}, the *aed* of requests that use stripe indices. Increasing the frequency of the index decreases the frequency of data, and vice versa. See Figure 2a. We now show how to approximate f^*, the frequency that strikes a near-optimal balance between the two.

The *stripe* organization requires a client to do two things in order to find requested data: find the root, traverse the index, and then follow the pointer to the requested item, so

$$aed_{stripe} = \text{find-root}_{stripe} + \text{find-data}_{stripe}, \tag{3}$$

where $\text{find-root}_{stripe}$, the expected number of ticks a client must wait for the beginning of the stripe to arrive, is

$$\text{find-root}_{stripe} = \frac{1}{2f}. \tag{4}$$

Once the client finds the index stripe, it traverses the index's levels to find the pointer, and then follows the pointer to the requested item. The time required to perform both of these steps is $aed_{apriori}$ (Equation 2), incremented by a factor $\frac{1}{1-Wf}$ that describes the increased interarrival times caused by the periodic broadcast of the index stripe. In addition, we compensate for the variability that stripes add to arrival times with an extra term $\frac{W}{2}$:

$$\text{find-data}_{stripe} = \left(\frac{1}{1-fW}\right)\frac{1}{2}\left(\sum_{i=1}^{K} N_i P_i\right) + \frac{W}{2}. \tag{5}$$

As the index is a balanced tree, the total width of the stripe containing an L level index is

$$W = \sum_{i=1}^{L} \left\lceil \frac{N}{Kn^i} \right\rceil. \tag{6}$$

Now, by relaxing the integral constraint of the stripe's period $\frac{1}{f}$, we see that aed_{stripe} is a continuously differentiable function of f. We now compute f^*, which minimizes aed_{stripe}.

$$f^* = \frac{1}{\sqrt{W}\left(\sqrt{W} + \sqrt{\sum_{i=1}^{K} N_i P_i}\right)}. \tag{7}$$

This result implies that we should broadcast the index stripes more frequently if it consumes little bandwidth (small W) or clients can follow pointers quickly (small $\sum_{i=1}^{K} N_i P_i$). In our experiments, we approximate $\frac{1}{f^*}$ with $\left\lceil \frac{1}{f^*} \right\rceil$.

We now consider the layout of the index nodes within a stripe. As mentioned before, a stripe is made of substripes. Each substripe contains a level of the index tree, and only one substripe is broadcast at a time. This design eliminates the possibility that a client can want two index nodes that are broadcast simultaneously while trying to satisfy a single request. The simplest way of mapping an index level to a substripe is by successively picking nodes, and placing them in sequence in the K channels. We then organize these substripes sequentially to form a stripe. See Figure 2b.

Alternatively, we can lay out the nodes among substripes so as to minimize the amount of hopping needed to traverse the index. The reason hopping is a concern is because of the potential synchronization costs it can incur (Section 2). To minimize the amount of hopping, we do two things. First, we attempt to place nodes in channels that contain their parents. In case of conflict, we select the node whose descendents are most likely to point to the respective channel. Compare the stripe organizations shown in Figures 2b and 2c. The organization shown in Figure 2c will likely result in fewer hops than the one shown in Figure 2b. We call such a stripe organization *ordered*.

The stripe design and model we just described assumes instantaneous hop cost for the sake of simplicity. To accomodate hop cost, we must first estimate the amount of hopping done, which is a function of the stripe layout. If H hops are expected, we simply augment Equation 3 with the term

$$\text{hop-cost} = \frac{H}{2f}, \tag{8}$$

and modify the other equations accordingly.

We also considered delaying the transmission of successive substripes to account for hop delay. We decided against this design for two reasons. First, without making strong assumptions about the time required to perform a hop, we cannot determine the optimal delay. Second, delaying successive substripes unnecessarily penalizes clients that do not need to hop while traversing the index and negates the utility of the ordered stripe design, described above.

4.2.3 Alternative Index Designs

There are clearly many other ways to design indexing schemes for broadcast data. One tree-based alternative is using unbalanced trees, where path lengths from root to leaf vary depending on the leaf's popularity [22]. Another alternative is to broadcast some tree nodes more frequently than others [13]. The benefit of these alternatives is not clear in our architecture. These schemes are typically computationally complex and add more metadata to the broadcast. These two factors reduce the available bandwidth for actual data, potentially nullifying their benefits. These schemes may also make strong assumptions about the schedule, e.g., that items are broadcast *flatly*.

One non-tree-based alternative is using a hash function to organize the data. A perfect hash function may minimize search costs, but works best when data are uniformly accessed [9]. Determining a hash function that reflects skewed access frequencies is a more difficult challenge.

One of the main reasons we choose to use a balanced n-ary tree index is because of its well-known structure and performance. For example, the work required to traverse such a tree is $O(\log N)$. We can use this knowledge as a basis of comparison when exploring other, more exotic techniques.

5. EXPERIMENTS

In this section, we present the experimental results that highlight some of the performance differences among our proposed search schemes. For comparison, we also present other results that are representative of alternative search techniques proposed in the literature. Due to space limitations, we only report a small subset of our results. More results can be found in [25].

5.1 The Simulator

Our simulator consists of a request manager and a data scheduler. At each tick, the request manager generates a set of new requests, and, for each pre-existing request, executes some action (e.g., listen to the broadcast, hop, doze). The main inputs to the request generator are the data access distribution, and the client search technique. At each tick, the data scheduler replaces the buckets being broadcast on each channel with new data according to a pre-computed schedule. The main inputs to the data scheduler are the data access distribution, the scheduling algorithm, and the indexing technique. We report the mean of 5 randomized trials of 50,000 randomized requests each for each parametric setting. Our parameters are described in Table 1 and are typical of those used in the literature.

5.2 The Search Techniques Considered

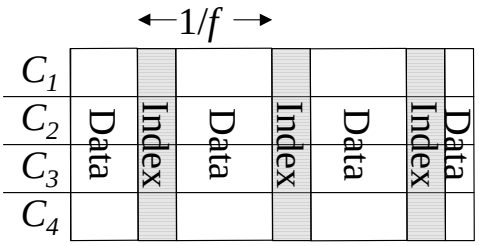
We test IAD with and without opportunistic client search (*depth*, *depth/opp*, respectively), and *striping* with and without index ordering (*stripe*, *stripe/ord*, respectively). We also report results on four "canonical" alternative techniques for searching over multiple broadcast channels. The first is called *apriori*, and refers the request performance when the client knows the exact channel and sequence number of each item precluding the need for an index. The performance of *apriori* serves as a lower bound for *aed*. The second is "smallest-channel-first" (SCF). With SCF, the client searches the channels in size-order for requested data; requests are most likely to be satisfied by the "hot" items contained in small channels. This is a arguably the best search technique to use if the client knows the channels' relative popularities, but has no other scheduling information. SCF also has better *aed* than any design that does not employ an global index on the broadcast database, e.g., the design described in [20]. Third, we report $ni/2k$, which is simply the database size, N, plus the index size, I, divided by the number of channels, K and by 2. This result is the analytical lower bound of *aed* of *any* indexed broadcast program that flatly broadcasts data over K channels, e.g., [17]. This performance is also comparable to that of a flat, indexed, single-channel program which transmits data K times faster than the channels we use. Finally, we consider the case where there is a dedicated index channel (*dedind*). All searches start from this channel and then hop to another channel containing data. We list the compared search techniques in Table 2.

5.3 A Note About Hopping

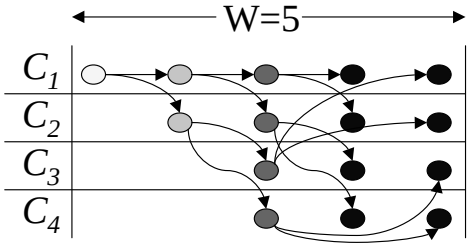
The *aed* for each organization should be treated as a lower bound for the actual *aed* because we assumed hopping to be instantaneous in the experiments. In practice, each hop can take on the order of milliseconds to complete, which should be added to the *aed* for each hop performed. The experiments, however, *do* consider the additional tuning time caused by post-hop synchronization.

5.4 Varying Database Size

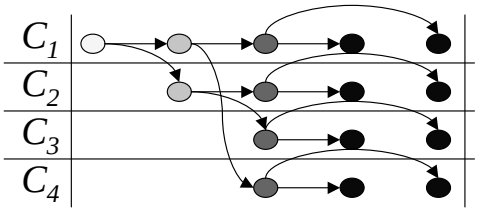
In our first set of experiments, we vary the number of items in the database. Search using *dedind* and SCF did the worst. SCF's *aed* increased at a fixed rate with database size. The rate of increase in *aed* actually increased for *dedind* with increasing databases size. This is due to the increasing relative cost of maintaining a dedicated index channel. The



(a) Striping the Broadcast.



(b) Simple Stripe Layout.



(c) Ordered Layout.

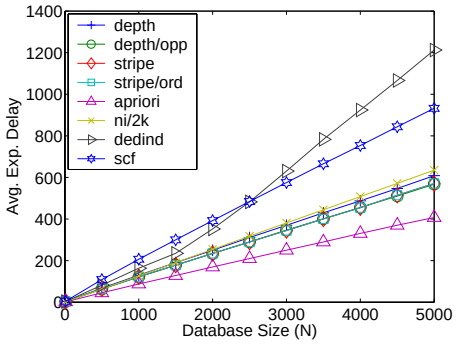
Figure 2: Striping. Interleaving index information in a broadcast program (a). Alternate layouts for the binary tree from Figure 1a over $K = 4$ channels (b, c).

Parameter	Limits	Step	Control
skew (Zipf, θ)	0, 1	.1	80/20
database size (N)	10, 5000	500	2500
fanout (n)	$2^1, 2^{11} = 2048$	$n = 2^i, i = 1, 2, 3, ...$	64
number of channels (K)	1, 64	$K = 2i + 1, i = 1, 2, 3, ...$	4

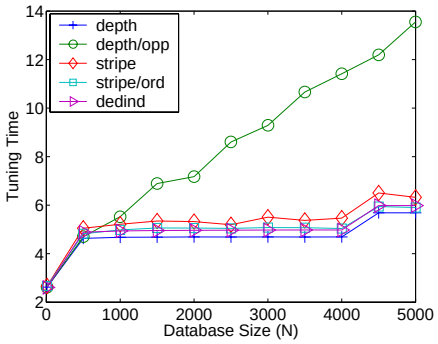
Table 1: Experimental Parameters for Indexing Experiments. Limits refers to the minimum and maximum parameter values, respectively. Step is the amount each parameter changes in each experiment. Control is the parameter value when it is not the independent variable.

Search Technique Notation	Description
depth	*IAD* indexing, where searches start at the root of the index.
depth/opp	*IAD* indexing, with an opportunistic client. (Section 4.2.1.)
stripe	*Stripe* indexing, without ordered nodes.
stripe/ord	*Stripe* indexing, with ordered nodes. (Section 4.2.2.)
apriori	No index. Client knows schedule. Demonstrates best case *aed*.
SCF	No index. Client searches the smallest channel first.
ni/2k	Data and index buckets are evenly allocated to all K channels. Analytical result.
dedind	*IAD* indexing, where the index is on a dedicated channel.

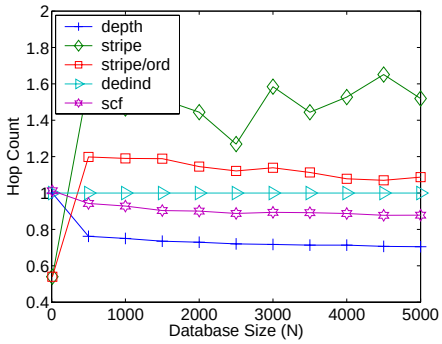
Table 2: Table of Considered Search Techniques



(a) DB Size vs. Avg. Exp. Delay



(b) DB Size vs. Tune Time



(c) DB Size vs. Hop Count

Figure 3: Comparing the Effect of Database Size on Average Expected Delay, Tuning Time, and Hop Count.

database grows at a quicker rate than does its index.

Depth and *stripe* beat *dedind* and SCF by at least 30%, are consistently better than $ni/2k$, and are near *apriori*. *Depth/opp* outperforms *depth* by about 5 to 10%, and has performance equivalent to that of *stripe*. *Depth* generally does worse than *stripe* for two reasons. First, because an IAD index is entirely on one channel, its frequency is usually lower, increasing the time needed to traverse it. Second, traditional search requires clients read the index's root to start all searches. We designed *depth/opp* to compensate for the latter problem, yielding results equivalent to those of stripe. None of these techniques are as fast as *apriori* because all of them require two searches to satisfy a request: one for the index, and then one for the item. *Apriori* requires only one search. See Figure 3a.

The tuning times for most search techniques are consistent with the $O(\log N)$ path lengths of tree indices. *Depth* did the best and *stripe* did the worst among the non-opportunistic techniques. In the case of *depth/opp*, the client incurs a higher tuning time as a consequence of its attempt to find the subtree without traversing the index root. We will explain this phenomenon in more detail in the fanout experiments below. See Figure 3b.

The hop counts for the IAD organizations are upper-bounded by 1. As more data are placed on the same channel as the index (as the database size increases), the average hop count naturally decreases. The hop count for *stripe* is relatively high, averaging about 1.5 in our tests. Hop count is bounded (as we shall see below) by the height of the index tree, which is no more than 3 in these experiments case. By ordering the index buckets so that children are in the same channels as their parents, we can reduce the average hop count to less than 1.2, a significant improvement. Reducing the number of hops also decreases tuning time by reducing the number of synchronizations the client must perform. See Figures 3b and 3c.

5.5 Varying Fanout

In our second set of experiments, we varied the fanout of the index. Low fanouts increase *aed*, tuning time, and hop count (if indices are striped over multiple channels); the index is larger, consumes more bandwidth, and takes longer to traverse. In terms of *aed*, SCF is superior until fanout is 8 for *stripe* and *depth/opp* and 16 for *depth* (Figure 4a). $Ni/2k$ is superior until fanout is 32 for *stripe* and *depth/opp* and 64 for *depth*. *Depth* does the worst by far with low fanouts. Because of opportunistic search, however, the performance of *depth/opp* is equivalent to that of *stripe*.

Increasing fanout generally decreases tuning time, as the index is shorter and smaller. Unexpectedly, the tuning time while using *depth/opp* actually increases when fanout increases from 16 to 32. This behavior can be explained by considering the client's opportunism. Assume that the client initially tunes in to an index leaf with a key range below its search key. In this case, it sequentially scans all the sibling leaves until it either finds the correct leaf, or it runs out of siblings and is directed to the root of the next subtree, based on the index's preorder layout. As the tree gets smaller (with increasing fanout), the probability of landing on a leaf decreases, decreasing the average tuning time (Figure 4b). In the worst case, the client must tune in to $O(n + \log N)$ buckets. The opportunistic client can easily be made more conservative in its search, but at the expense of *aed*.

Hop count sharply increases for *stripe* with a low fanout, but ordering the index nodes greatly alleviates this problem (Figure 4c). Naively allocating index buckets to the broadcast program can lead to a hop for every step in the traversal of the index. Ordering the index nodes reduces the hop count of *stripe/ord* to less than 2. The number of hops is constrained by the low number of channels (4). For example, if there were only one channel, hop count must be 0. Notice also the correlation between hop count and tuning time in the *stripe* experiments. Again, the hop counts for the IAD organizations are upper-bounded by 1.

6. CONCLUSION

In this paper, we address the often overlooked problem of accessing data in a multi-channel broadcast environment. Our main contribution is devising a way of applying indexing information to a skewed multi-channel schedule. Previous work on indexing assumed that the data schedule is flat, whereas previous work on scheduling did not consider indexing. In the multi-channel environment, both are effective in improving request response time by either broadcasting popular data more frequently, or reducing the search space.

Our solution consists of creating a skewed multi-channel schedule using a well-known algorithm $GREEDY$ [27], and then augmenting it with indexing information. Our two alternatives are IAD, which schedules the index as if it were a data item (Section 4.2.1), and *stripe*, which involves broadcasting index over all K channels simultaneously at an optimal frequency (Section 4.2.2). IAD, however, suffers from high latencies caused by the need to wait through all index nodes in order to find a pointer. We alleviate this problem using opportunistic search at the expense of tuning time. *Stripe* suffers from high hop count. We alleviate this problem by using *ordered* index node allocation to the stripe.

Tests show that IAD and *stripe* perform well compared with the alternatives. They generally outperform any technique that involves non-skewed schedules ($ni/2k$), and far outperform (on the order of 30%) any technique that does not use a global index (SCF) in terms of both *aed* and tuning time.

Note that indices are only useful at reducing *aed* when they do not consume too many resources. In this case, if fanout is very low (less than 16) or there are very few channels (less than 3, not shown in this paper), then SCF actually becomes a viable search technique.

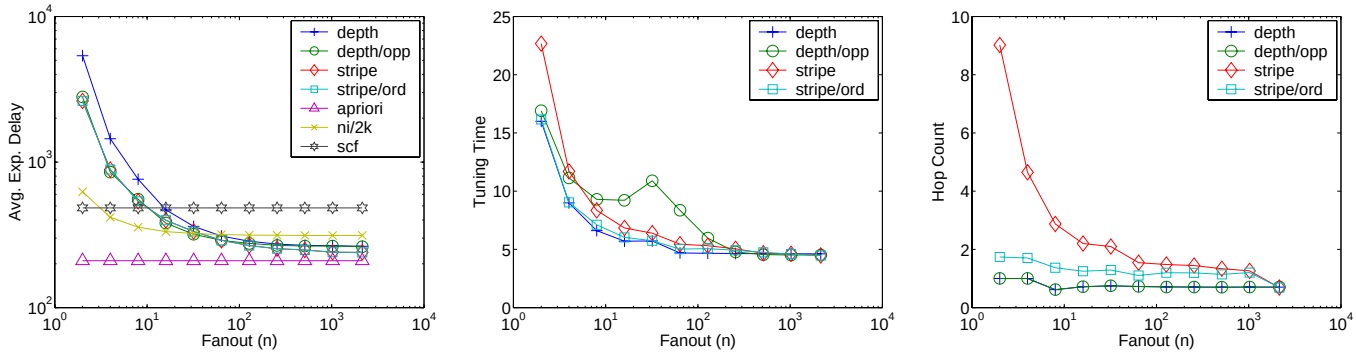
We are currently working on caching strategies to improve response time and availability, scheduling and access techniques for multiple item requests. We are also considering applying our work to recent streaming applications mentioned in [6].

7. REFERENCES

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

[2] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proc. ACM SIGMOD*, May 1997.

[3] D. Aksoy and M. J. Franklin. Rxw: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Trans. Networking*, 7(6):846–860, December 1999.



(a) Fanout vs. Avg. Exp. Delay (b) Fanout vs. Tune Time (c) Fanout vs. Hop Count

Figure 4: Comparing the Effect of Fanout on Average Expected Delay, Tuning Time, and Hop Count.

[4] K. C. Almeroth, M. H. Ammar, and Z. Fei. Scalable delivery of web pages using cyclic best-effort (udp) multicast. In *Proc. IEEE INFOCOM*, March 1998.

[5] M. H. Ammar and J. W. Wong. The design of teletext broadcast cycles. *Performance Evaluation*, 5(4):235–242, December 1985.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. ACM SIGMOD*, June 2002.

[7] M. J. Donahoo, M. H. Ammar, and E. W. Zegura. Multiple-channel multicast scheduling for scalable bulk-data transport. In *Proc. IEEE INFOCOM*, March 1999.

[8] J. Fernandez and K. Ramamritham. Adaptive dissemination of data in time critical asymmetric communication environments. In *Proceedings of Euromicro Real-Time Systems Symposium*, pages 195–203, 1999.

[9] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with o(1) worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[10] GSM Association. Gsm world. Web Document, 2002. www.gsmworld.com.

[11] G. Herman, G. Gopal, K. C. Lee, and A. Weinrib. The datacycle architecture for very high throughput database systems. In *Proc. ACM SIGMOD*, pages 97–103, 1987.

[12] C. H. Hsu, G. Lee, and A. L. P. Chen. A near optimal algorithm for generating broadcast programs on multiple channels. In *Proc. ACM Conf. on Information and Knowledge Mgt. (CIKM)*, pages 303–309, November 2001.

[13] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In *Proc. ACM SIGMOD*, May 1994.

[14] S. Khanna and S. Zhou. On indexed data broadcast. In *ACM Symp. on Theory of Computing*, pages 463–472, 1998.

[15] D. H. Layer. Digital radio takes to the road. *IEEE Spectrum*, July 2001.

[16] W. C. Lee, Q. Hu, and D. L. Lee. A study on channel allocation for data dissemination in mobile computing environments. *Journal Mobile Networks and Applications*, 4:117–129, 1999.

[17] S. C. Lo and A. L. P. Chen. Optimal index and data allocation in multiple broadcast channels. In *Proc. IEEE Intl. Conf. on Data Eng. (ICDE)*, February 2000.

[18] Marimba, Inc. Marimba, inc. web site. Web Document, 2002. www.marimba.com.

[19] W. C. Peng and M. S. Chen. Dynamic generation of data broadcasting programs for broadcast disk arrays in a mobile computing environment. In *Proc. ACM Conf. on Information and Knowledge Mgt. (CIKM)*, pages 35–45, November 2000.

[20] K. Prabhakara, K. A. Hua, and J. Oh. Multi-level multi-channel air cache designs for broadcasting in a mobile environment. In *Proc. IEEE Intl. Conf. on Data Eng. (ICDE)*, 2000.

[21] S. M. Ross. *Introduction to Probability Models*. Academic Press, 6th edition, 1997.

[22] N. Shivakumar and S. Venkatasubramanian. Efficient indexing for broadcast based wireless systems. *Journal Mobile Networks and Applications*, 1(4):433–446, December 1996.

[23] N. H. Vaidya and S. Hameed. Scheduling data broadcast in asymmetric communication environments. *Journal Mobile Networks and Applications*, 5:171–182, 1999.

[24] S. Vishwanathan and T. Imielinski. Pyramid broadcasting for video on demand service. In *IEEE Multimedia Computing and Networks Conference*, February 1995.

[25] W. G. Yee and S. B. Navathe. Efficient generation of broadcast schedules. In *Submitted for Publication*, 2002.

[26] W. G. Yee and S. B. Navathe. Fast data access on multiple broadcast streams. Technical report, Georgia Institute of Technology, May 2003.

[27] W. G. Yee, S. B. Navathe, E. Omiecinski, and C. Jermaine. Efficient data allocation over multiple channels at broadcast servers. *IEEE Trans. Computers, Special Issue on Mobility and Databases*, 51(10):1231–1236, October 2002.