Bridging the Gap between Response Time and Energy-Efficiency in Broadcast Schedule Design

Wai Gen Yee, Shamkant B. Navathe,
Edward Omiecinski, and Christopher Jermaine

College of Computing, Georgia Institute of Technology
Atlanta, GA 30332-0280
{waigen,sham,edwardo,jermaine}@cc.gatech.edu

Abstract. In this paper, we propose techniques for scheduling data broadcasts that are favorable in terms of both response and tuning time. In other words, these techniques ensure that a typical data request will be quickly satisfied and its reception will require a low client-side energy expenditure. By generating broadcast schedules based on Acharya et al.'s broadcast disk paradigm, we bridge the gap between these two mutually exclusive bodies of work–response time and energy expenditure. We prove the utility of our approach analytically and via experiments. Our analysis of optimal scheduling is presented under a variety of assumptions about size and popularity of data items, making our results generalizable to a range of applications.

1 Introduction

Data broadcast is a well-known way of scalably answering data requests for a large client population [1,2]. A single broadcast of a data item can satisfy an unbounded number of outstanding requests for that item. The relevance of scalable data dissemination techniques increases with the number of computer users, especially in light of emerging wireless technologies and applications requiring only asymmetric communications capabilities. Some sample applications include traffic information systems, wireless classrooms, financial data, and news services.

Research in wireless broadcast typically assumes a high-powered server that transmits data to mobile clients over a wide area. The consequences of this architecture relate to *application performance* and *energy conservation* because data rates are low and battery lifetimes are limited.

In order to alleviate the poor performance due to low bandwidth, servers can skew the average amount of bandwidth allocated to each data item so that more popular data are transmitted more frequently [3,1,4,5]. Clients can further improve application performance via client-side caching [6,7]. In order to conserve energy, on the other hand, servers can reduce the amount of work a client must do to find data by broadcasting data indices [8,9].

1.1 Shortcomings of the State of the Art

Much of the work on response time and energy expenditure has been mutually exclusive. A typical bandwidth allocation algorithm constructs a schedule bottom-up by deciding at each time interval the priorities of each data item, and broadcasts the one with the highest priority. Although the average time required for an average client to receive a given data item is near-minimal, such schedules preclude the generation of indices, and makes client-side caching more difficult, resulting in poor energy conservation and application performance. The main reason for these poor results is that clients cannot predict the next data item to be broadcast and therefore cannot doze, prefetch or pin data. Our simulations show that even with uniform access to all items in the database, bottom-up schedules result in significant variation in the interarrival time of consecutive instances of a given data item. This variation increases with the number of data items in the database. (See [10] for more details of these experiments.) Moreover, these bandwidth allocation algorithms lack the notion of a beginning and an end of a schedule, making techniques such as consistency control among data items more complicated [11].

In contrast, work focusing on issues such as consistency and energy conservation [12,9,11,13] typically assumes a *flat* schedule – no data item can be broadcast twice before all other data in the database is broadcast once. Flat scheduling has benefits – such schedules are short, and the worst-case wait for any data item is minimal. If there is skew in data access, however, the average time required to receive data with a flat schedule (about one-half the time it takes to transmit the entire database) is far from optimal.

1.2 Goals and Outline of the Paper

The work presented in this paper attempts to bridge the gap between broadcast research that optimizes bandwidth usage and that which designs predictable schedules. To this end, we propose a top-down design based on broadcast disks [6]. Broadcast disks are logical *partitions* of the database. The average frequency that a data item is broadcast is inversely proportional to the size of the partition to which it belongs. We can use this property to skew the amount of bandwidth toward more popular data items. Furthermore, using broadcast disks allows us to generate predictable schedules in a straightforward manner. See Figure 1.

We start by modeling our problem and formalizing our problem statement in Section 2. We then show how broadcast disks can be optimally generated in Section 3. Since the optimal solution is computationally expensive, we also propose a cheap approximation that yields near-optimal results. We then show how our schedule can be applied to handle the case of different-sized data items (Section 3.4) and indexing (Section 3.5). We discuss related work in Section 4, and conclude the paper and describe future work in Section 5.

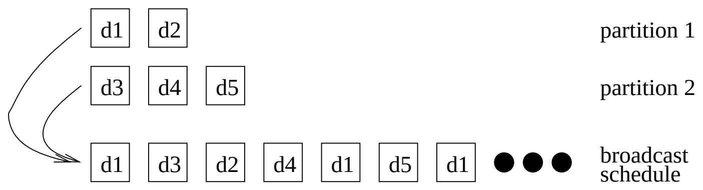

Fig. 1. An Example Broadcast Disk System. Clients listen for data items broadcast from each partition in a round-robin manner. In this example, the database consisting of 5 data items is partitioned into 2 broadcast disks, consisting of 2 and 3 data items, respectively.

2 Model and Problem Statement

In this paper, we assume that a server transmits the contents of a database containing N data items. Time is divided into units called *ticks*, and each unit of data requires a single tick to be transmitted. Clients request single data items. Each data item d_i $(1 \leq i \leq N)$ has a probability p_i of being requested. Requests are assumed to be exponentially distributed–the probability that a data item is requested at each time interval is fixed. A broadcast *schedule* is a sequence of L data items, where $L \geq N$–all data items must be transmitted at least once, and some may be transmitted multiple times during a schedule.

Our goal is to generate a broadcast schedule that minimizes the expected delay of a data request subject to the following constraints:

1. There is a fixed interarrival time between all successive transmissions of a given data item in a schedule.
2. There is a notion of a cycle – i.e., the schedule has a beginning and an end.

As in [1], we assume that all interarrival times w_i of a data item d_i that occurs multiple times in a schedule are equal. This assumption stems from a basic result in probability theory which states that fixed interarrival times minimize expected delay given exponentially distributed requests and a known arrival rate. For a given schedule, the average expected delay over all data items is therefore

$$\text{average expected delay} = \frac{1}{2}\sum_{i=1}^{N} p_i w_i. \tag{1}$$

By minimizing Equation 1 subject to constraints 1 and 2, we satisfy the goal of this paper, which is to bridge the work done in bandwidth allocation with other work done in the area of broadcast.

3 Heuristic

In this section, we describe a top-down technique for generating broadcast schedules. Bottom-up formulations of the broadcast bandwidth allocation problem have typically yielded intractable optimization problems [1]. Although heuristics that are near-optimal in terms of average expected delay (Equation 1) are available [3,4,5], none satisfies Constraints 1 and 2. (See Section 2.) In contrast, we show how to compute an optimal top-down solution subject to Constraints 1 and 2, and offer an inexpensive but effective approximation.

We borrow the concept of broadcast disks from [6]. A broadcast server picks a broadcast disk in a round-robin manner and transmits the least recently broadcast unit of data from it. Given K broadcast disks, the *beginning* of the broadcast schedule is defined as the transmission of the first data item of the first disk given that it is followed by the first data item of each of the next $K-1$ disks. The *end* is typically defined as the end of the last data item of the last disk, given that it is preceeded by the last data item of each of the previous $K-1$ disks.

Clearly, broadcast disk scheduling satisfies Constraints 1 and 2. Our goal is therefore to generate a set of broadcast disks that minimize average expected delay (Equation 1). Consider an arbitrary partitioning of N data items over K disks. The number of data items in each disk C_i is N_i, $\sum_{i=1}^{K} N_i = N$. We can reframe Equation 1 in terms of broadcast disks:

$$\text{broadcast disk average expected delay} = \frac{K}{2}\left(\sum_{i=1}^{K} N_i \sum_{d_j \in C_i} p_j\right). \tag{2}$$

In minimizing Equation 2, we first try to minimize the summation on the right-hand side. In other words, given K, how do we minimize $\sum_{i=1}^{K} N_i \sum_{d_j \in C_i} p_j$? (We call this subproblem the *partitioning problem.*)

In [10], we show how to optimally solve the partitioning problem using dynamic programming (*DP*). *DP* has $O(KN^2)$ computational complexity and consumes $O(KN)$ space to keep track of partial solutions.

3.1 A Cheaper Approximation for Partitioning

Although *DP* yields an optimal partitioning, its time and space complexity may preclude it from practical use. For example, depending on the application, redesign may occur frequently, or, as in the *DATACYCLE* case, the size of the problem may be large [14], i.e., data sets may be orders of magnitude larger than those of wireless applications. We therefore offer an alternative, significantly cheaper algorithm.

Assuming that each data item is equally sized, define C_{ij} as the average expected delay of a request for a data item in a partition containing data items i through j [15]:

$$C_{ij} = \left(\frac{j-i+1}{2}\right)\sum_{q=i}^{j} p_q, \quad j \geq i, \quad 1 \leq i, j \leq N. \tag{3}$$

Our partitioning algorithm, called *GREEDY*, finds the *split point* among all the partitions that minimizes cost. Cost change by splitting a partition (i, j)

at point s is computed in constant time by:

$$C_{ij}^{s} = C_{is} + C_{s+1,j} - C_{ij}, \quad i \leq s < j \tag{4}$$

The split process performed a total of $K - 1$ times. This algorithm is similar to the ones used in [16,17], but is differs in its assumptions about the data and applications.

Algorithm 1

```
GREEDY
input: set of $N$ unit sized data items ordered by popularity, $K$ partitions
begin
  $numPartitions := 1$;
  while $numPartitions < K$
        do for each partition $k$ with data items $i$ through $j$
              do comment: Find the best point to split in partition $k$
              for $s := i$ to $j$
                  comment: Initialize the best split point for this partition
                           as the first data item. If we find a better one
                           subsequently, update the best split point.
                  do if $s := i \lor localChange > C_{ij}^{s}$
                        do
                           $localS = s$; $localChange = C_{ij}^{s}$;
                        od fi od
              comment: Initialize the best solution as the one for the first
                       partition. If we find a better one subsequently,
                       update the best solution.
              if $k := 1 \lor globalChange > localChange$
                 do
                    $globalChange := localChange$; $globalS := localS$;
                    $bestpart := k$;
                 od fi od
        split partition $bestpart$ at point $globalP$
        $numPartitions := numPartitions + 1$;
  od
end
```

Example 1. Consider the problem of allocating the set of $N = 6$ data items to $K = 3$ partitions. Using the *GREEDY* algorithm, the first split occurs between data items 2 and 3. This is the point at which C_{ij}^{s}, $(i = 1, j = 6, s = 2)$ is minimized. The second split occurs between data items 1 and 2 for the same reason. These two splits reduce the average expected delay from 3 ticks to 0.95 ticks. See Figure 2.

Theorem 1. *The complexity of GREEDY is $O((N + K)\log K)$.*

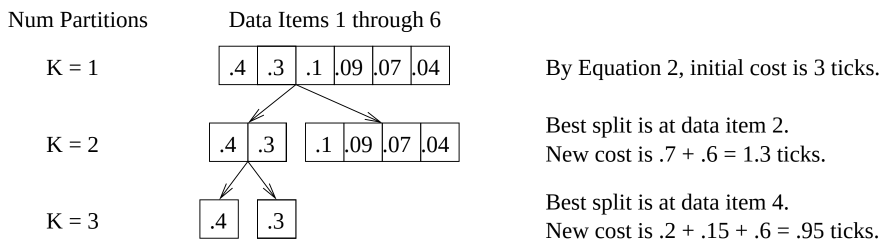
Num Partitions
Data Items 1 through 6
K = 1
.4
.3
.1
.09
.07
.04
By Equation 2, initial cost is 3 ticks.
K = 2
Best split is at data item 2.
New cost is .7 + .6 = 1.3 ticks.
K = 3
Best split is at data item 4.
New cost is .2 + .15 + .6 = .95 ticks.

Fig. 2. Example of application of $GREEDY$, to generate $K = 3$ partitions. The numbers in the boxes are the popularities of the data items.

Since sorting the data items generally has $O(N \log N)$ complexity and $N \geq K$, by Theorem 1 the asymptotic complexity of $GREEDY$ is minimal. The implementation details of $GREEDY$ and proof of Theorem 1 can be found in [10]. In that work, we also present experimental results that prove $GREEDY$'s near-optimal performance over a wide range of parameters.

3.2 Generating Broadcast Disk Schedules

In this section, we apply the partitioning results described above to a technique for generating broadcast schedules. We first formalize the means of mapping the broadcast disks onto a broadcast medium first mentioned at the beginning of this section.

Algorithm 2

```
MAPPING
input: $K$ disks of data, each containing $N_i$ data items
begin
  $x := 0$;
  while $TRUE$
       do
       for $i := 1$ to $K$
          do
          broadcast the $((x \bmod N_i) + 1)^{\mathrm{th}}$ unit of data from disk $i$
       od
       $x := x + 1$;
  od
end
```

One can see that the average expected delay of a schedule generated by Algorithm 2 is the cost given by Equation 2. Although this algorithm bears resemblance to one given in [6], it does not generate *holes*; all ticks contain data, thereby using bandwidth more efficiently.

Having solved the partitioning problem, our goal now is to find a value of K that minimizes the entire expression of Equation 2. We propose the simple

algorithm **f**or **o**ptimal **K** (abbreviated as *Safok*). This algorithm solves the partitioning problem for all values of K between 1 and N and returns the solution that is minimal when multiplied by K.

Algorithm 3

```
SAFOK
input: set of $N$ unit sized data items ordered by popularity
begin
  $initial\_cost := C_{1N}$
  $current\_cost := initial\_cost$
  for $i := 1$ to $N$
      do
      $cost_i = current\_cost;$
      $kCost_i = \frac{k}{2} * cost_i;$
      if $i = 1 \vee kCost_i < bestKCost$
        do
          $bestKCost = kCost_i;$
          $bestK = i;$
        od
        Search all partitions for the best split point, $s$, between partition
        boundaries $l$ and $r$;
        Make a split at $s$.
        $current\_cost = current\_cost - C^s_{lr}.$
      od
  end
```

Corollary 1. *Safok has $O(N \log N)$ computational complexity and requires $O(N)$ memory.*

Proof: The proof is similar to that of Theorem 1.

3.3 Experimental Results

In this section, we compare the performance of *flat* (mentioned in Section 1) against that of *Safok*. We also report the analytical lower bound[1] (denoted **opt**) derived in [1]:

$$\frac{1}{2}\left(\sum_{i=1}^{N}\sqrt{p_i}\right)^2 . \tag{5}$$

We conduct two sets of experiments. In the first set, we vary the skew of the access probabilities. Assuming a Zipfian distribution, skew varies from low ($\theta = 0$) to high ($\theta = 1$). In the second set, we vary the number of data items in the database from 10 to 1000. The control skew is an 80/20 distribution and the

[1] Not achievable in general.

control database size is 500 data items. These values are in the range of those typically used in broadcast studies [18,11].

As we can see, the performance of $Safok$ is at least as good as that of $flat$. When there is no skew ($\theta = 0$), there is no need to transmit any data item more frequently than others, and therefore $Safok$ does nothing. When skew increases, the performance difference becomes quite large (Figure 3). Moreover, the benefits of $Safok$ over $flat$ increase linearly with database size (Figure 4). Note that the average expected delay of the $Safok$ schedules are near-optimal.

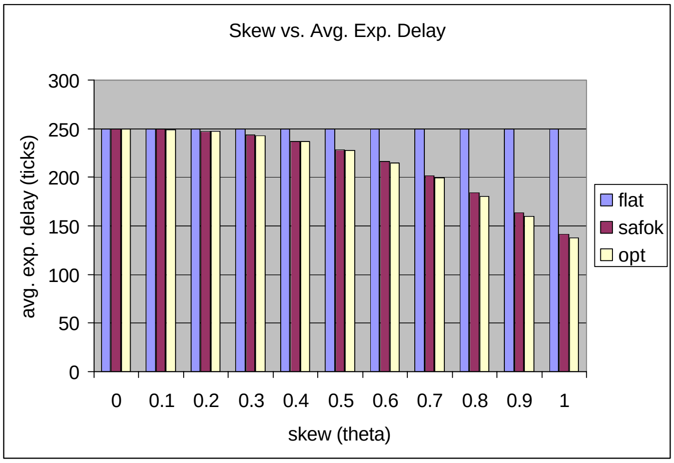

Fig. 3. *Safok* Performance. Varying Skew. Database Size = 500

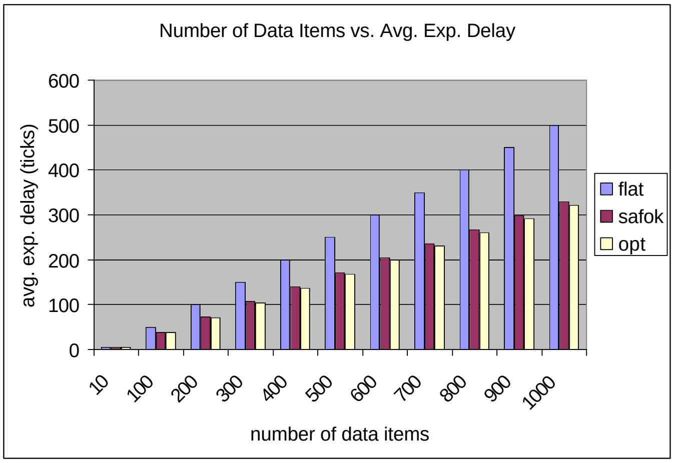

Fig. 4. *Safok* Performance. Varying Number of Data Items. Skew = 80/20

3.4 Different Sized Data Items

In many practical applications, data items span more than a single unit of data. For example, given multimedia content consisting of images and text, the data items can vary substantially in size. In this section, we consider the impact of size on $Safok$.

Given sizes $l_i \in Z^+$, $1 \leq i \leq N$, the cost function from Equation 1 becomes

$$\frac{1}{2}\sum_{i=1}^{K}\left[\sum_{d_k \in C_i} l_k \sum_{d_j \in C_i} p_j\right]. \tag{6}$$

Clearly, the problem of minimizing Equation 6 is a form of the NP-complete minimum sum of squares problem. We can, however, optimally solve Equation 6 exactly as we did in the unit-sized data item case (see Section 3). But again, because of the complexity of the optimal solution, we show how to adapt $GREEDY$ to the varying size case.

$GREEDY$ depends on ordering data items so that data that should be broadcast at similar frequencies are adjacent. For example, unit-sized data items are ordered by their access probabilities, $p_i, 1 \leq i \leq N$. To order data items of varying size, we borrow the *square-root rule* from [5]. This rule states that, in an optimal schedule, the number of ticks between consecutive instances of data item d_i is proportional to $\sqrt{\frac{l_i}{p_i}}$. The only modifications we make to Algorithms 2 and 3 are therefore:

- Data items are to be ordered by $\sqrt{\frac{l_i}{p_i}}, 1 \leq i \leq N$.
- Splits may not occur within data items d_i, where $l_i > 1, 1 \leq i \leq N$.
- The amount of data broadcast at each tick becomes $\gcd\{l_1, l_2, ..., l_N\}$ units.

The first modification is a way of clustering data items that should be broadcast at the same frequency. Because data items with similar $\sqrt{\frac{l_i}{p_i}}$ are contiguous, they can exist in the same partition and have similar broadcast frequencies.

The second and third modifications preserve the broadcast disk "semantics." In other words, the idea that multiple disks are simultaneously broadcasting their contents at different rates is preserved because the interarrival times between consecutive data items from a given disk are fixed. One practical implication of this is a simplified search. To find a data item of any size, a client only needs to listen to a single broadcast disk.

Experimental Results We compare the impact of different ways of ordering the data items on the average expected delay. We assume a database of 500 data items, and vary the Zipfian skew (θ) from 0 to 1. The ordering keys are

p_i	Access probability (denoted **p**), as done in previous experiments.
l_i	Size (denoted **l**).
$\frac{p_i}{l_i}$	Normalized access probability (denoted **p/l**).
$\sqrt{\frac{p_i}{l_i}}$	Square root of normalized access probability (denoted **root(p/l)**).

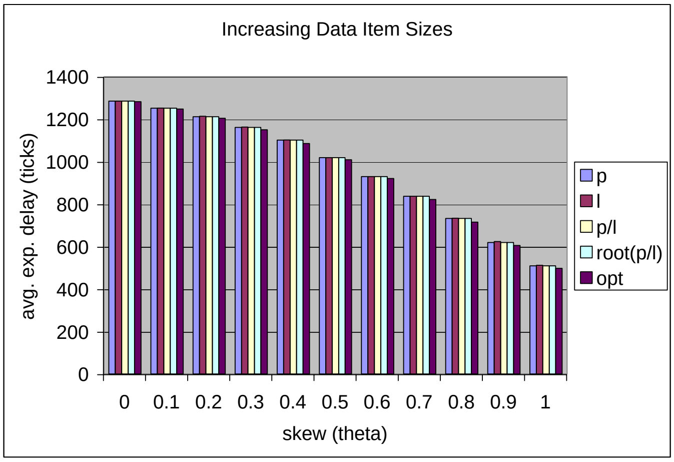

Fig. 5. *Safok* Performance With Varying Sized Data Items. p denotes ordering data by p_i. l denotes ordering data by l_i. p/l denotes ordering data by $\frac{p_i}{l_i}$. $root(p/l)$ denotes ordering data by $\sqrt{\frac{p_i}{l_i}}$. *opt* denotes the analytical lower bound. Number of data items = 500.

In the first set of experiments (Figure 5), we linearly increase the size of the data items with the following formula [5]:

$$l_i = l_{\min} + \text{round}\left(\frac{i \times (l_{\max} - l_{\min})}{N}\right), 1 \leq i \leq N, \tag{7}$$

where *round* is the round-off function.

In the second set of experiments (Figure 6), we uniformly distribute l_i between $l_{\min}$ and $l_{\max}$. Other experiments using different size-assignment functions yield similar results. As in [5], we set $l_{\min}$ and $l_{\max}$ to 1 and 10, respectively, for all experiments. All results are compared against the analytical lower bound of average expected delay[2] (denoted **opt**)[5]:

$$\frac{1}{2}\left(\sum_{i=1}^{N} \sqrt{p_i l_i}\right)^2. \tag{8}$$

Ordering by $\frac{p_i}{l_i}$ and $\sqrt{\frac{p_i}{l_i}}$ yielded identical results, which were better than those yielded by ordering by either p_i or l_i and are close to optimal. The performance improvements are only marginal in the increasing-sizes case (Figure 5), but are more noticeable in the random-sizes case (Figure 6). Both $\frac{p_i}{l_i}$ and $\sqrt{\frac{p_i}{l_i}}$ yielded near-optimal results.

In general, however, we expect the results of using $\frac{p_i}{l_i}$ to be worse than those of using $\sqrt{\frac{p_i}{l_i}}$. The key $\frac{p_i}{l_i}$ has been used in data partitioning algorithms for disk

[2] Not achievable in general.

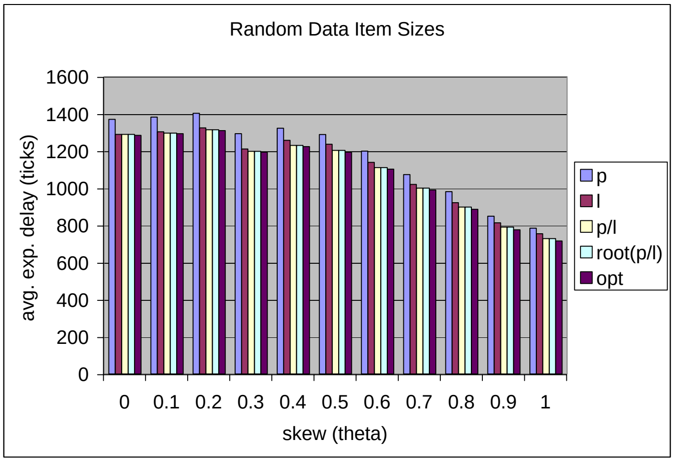

Fig. 6. *Safok* Performance With Random Sized Data Items.

arrays [19]. However, these disk arrays are assumed to allow simultaneous access to multiple disks. In the broadcast case, in contrast, the transmission of a data item necessarily excludes the transmission of other data. This tradeoff acts as the intuitive justification for using the square-root.

3.5 Indexing

Indexing and careful data organization are important to any broadcast scheme since they allow efficient client battery power management. If a client knows exactly when a desired data item is scheduled to appear, then it need only tune in when that data item is to be broadcast. Otherwise, it can stay in an energy-conserving *doze* mode. When indices are present, the delay in receiving desired data is broken up into *probe wait* – the average time it takes to read the index information – and *broadcast wait* – the average time it takes to read the data item once indexing information is known. Note that when indices are absent, probe wait is zero and broadcast wait is equal to our original formulation of expected delay. Another parameter is *tuning time*, which is the amount of time the client must *actively* listen to the broadcast channel (for data and/or index information). Tuning time is therefore one rough measure of client energy consumption. The goal of indexing is t o minimize tuning time with minimum impact on delay.

In this section, we argue that *Safok* schedules are amenable to existing indexing techniques and demonstrate how a well-known indexing scheme can be applied. We prove the usefulness of *Safok* by measuring the resulting average expected delay and tuning time.

In general, how might an existing indexing method be incorporated into our framework? We take advantage of the fact that while data items from different partitions may be transmitted at different frequencies in a *Safok* schedule, *within a partition, each data item is broadcast in a flat manner.* We can therefore apply existing indexing schemes that assumes flat schedules [12,8,9] to each partition, and index and organize them independently. These separate partitions are then interleaved into a single channel.

For example, we consider the simple $(1, m)$ indexing method proposed by Imielinski, et al [9]. In this method, an index for all of the data items is regularly broadcast m times over a flat schedule. Say we compute three partitions, having 1, 4, and 20 data items, respectively. To use the $(1, m)$ scheme in combination with our partitioning, all we need to do is to interleave a $(1, m)$ organization for each partition into our broadcast schedule, as is shown in Figure 7.

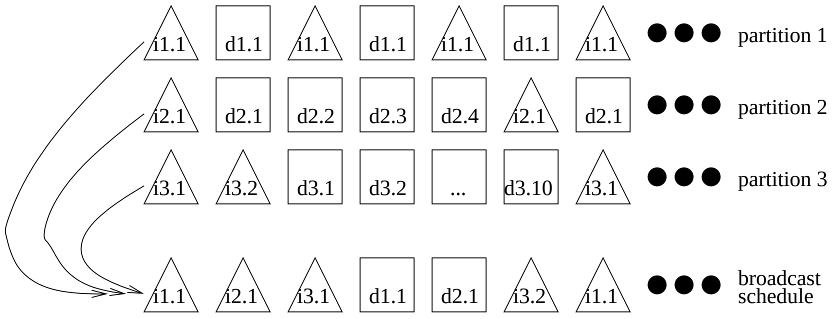

Fig. 7. Applying $(1, m)$ to a $Safok$ Broadcast Schedule. Triangles denote units of index information. Squares denote units of data. d$m.n$ and i$m.n$ denote the n^{th} data and index units from partition m, respectively. Each triangle or square requires a single tick to transmit.

The only additional modification we need to make to the $(1, m)$ organization is that each data item contains an offset to the next in.1 unit (that is, for *any* $1 \leq n \leq K$), not just to the next index unit within its own partition.

Using this modified $(1, m)$ scheme, whenever a client wishes to find a data item, it listens to the broadcast channel, encountering a random data item. The client uses offsets contained in the data item to determine when the start of the next index unit will be broadcast. It then sleeps until this index unit comes up, and follows the pointers in the index, looking for the desired data item. It also follows the offsets to index units in other partitions. This process continues until the desired data item (or a pointer to the item) is found.

Note that the searching of the different indices in each partition are done concurrently, since there is no reason to wait until one index has been totally searched before we begin searching the others. Likewise, we may stop searching all indices once a pointer to the desired data item has been found.

An important characteristic of this mapping to our partition-based framework are that many of the desirable performance characteristics of the original algorithm are preserved. In the example application of the $(1, m)$ organization to our partitioned model, the expected delay of a data request from partition i is

$$\overbrace{\frac{K}{2}\left(Index_i + \frac{N_i}{m_i}\right)}^{probe\_wait_i} + \overbrace{\frac{K}{2}\left(m_i \times Index_i + N_i\right)}^{broadcast\_wait_i}. \qquad (9)$$

In this expression, m_i refers to the m value (the number of times a complete set of index information appears in the partition) for partition i, and $Index_i$ is the number of units required to store the index for partition i. We point out how closely this analytical formula mirrors that of the original $(1, m)$ scheme [9].

In general, the tuning time is increased at most by a factor of K from the tuning time required by the original algorithm. However, the tuning time required will be much less than K for a data item from a smaller partition (which by definition will be a more popular data item). Why? If the item requested is from a smaller partition, then we will likely find a pointer to it in some index before we ever need to turn on and search other indices. Formally, for the $(1, m)$ algorithm, the expected tuning time to find an item located in partition i is

$$1 + \sum_{j=1}^{K} index\_lookups(j, probe\_wait_i), \qquad (10)$$

where $index\_lookups(j, t)$ is the number of index units from partition j that we expect to actively listen to over a duration of t ticks. In Equation 10, we limit t to $probe\_wait_i$ because that is the expected amount of time required before finding the final pointer in partition i to the desired data item. The 1 on the left-hand side of Equation 10 comes from the fact we must tune in once after the indices have been searched to actually read the data item.

How might we calculate *index_lookups*? The expected number of index units for partition j that are broadcast in t ticks is

$$\frac{t}{K}\left(\frac{Index_j}{Index_j + \frac{N_j}{m_j}}\right). \qquad (11)$$

Note that for a tree-based index, we only need to read one index entry at each level in the index. The final formula for *index_lookups* for partition j over t ticks is therefore[3]

$$index\_lookups(j, t) = 1 + \left\lceil \log_n\left(\min\left(Index_j, \frac{t}{K}\left(\frac{Index_j}{Index_j + \frac{N_j}{m_j}}\right)\right)\right)\right\rceil, \qquad (12)$$

where the 1 on the left-hand side is roughly the cost of finding an offset to the next index unit to be broadcast and n is the fanout (the number of (key, pointer) pairs an index unit may store) of each index unit.

[3] When the argument of the log function is at or below 1, we round the right-hand term.

The construction would be similar if an indexing scheme other than $(1, m)$ were used. For example, the *Distributed Indexing* scheme from [9] could also be used with a similar modification. In general, the tuning time is increased by at most a factor of K, but is significantly less for a more popular data item.

Experimental Results In this section, we describe some experimental results using $(1, m)$ indexing on a database consisting of 500 unit-sized data items. We assume that at each tick, 4K is transmitted. We assume that an index entry consists of a 16-byte (key, pointer) pair. The fanout, n, is therefore $4096/16 = 256$. Because of the tree-structure of each index, the size of the index information for each partition j is

$$Index_j = \sum_{i=1}^{\log_n N_j} \frac{N_j}{n^i}. \tag{13}$$

We also compute m_i^* (the optimal value of m_i, as described in [9]) for partition i as

$$m_i^* = \sqrt{\frac{N_i}{Index_i}}. \tag{14}$$

In the experiments we compare the average expected delay and tuning times of four scheduling algorithms.

flat	A typical flat schedule *without* indexing information.
flat_idx	A flat schedule augmented with $(1, m)$ indexing.
safok_idx	*Safok*, augmented with $(1, m)$ indexing.
opt	The analytically optimal solution *without* indexing information.

As expected, the time *safok_idx* takes to satisfy an average request decreases with increasing skew, but is always significantly worse than the optimal solution. In fact, in terms of average expected delay, *Safok_idx* does not beat *flat_idx* until medium skew ($\theta \approx .4$). The performances of both flat schedules is fixed and are outdone by both *safok_idx* and *opt* at high skew (Figure 8).

The tuning times of both indexed schedules are significantly lower than those of the non-indexed schedules, potentially saving lots of energy at the client. *safok_idx* is slightly worse than *flat_idx* at low skew, but is superior at high skew. At low skew, both algorithms should produce flat schedules, so should have similar index organizations. At high skew, using *safok_idx*, index information for very popular data items is easier to find and is smaller than it is with *flat_idx* (Figure 9).

4 Related Work

Su et al. studied the impact of various scheduling routines on client-side cache performance [7]. He uses some statistical information on client interests to partition data, but then uses a real-time priority-based scheme to pick each data item

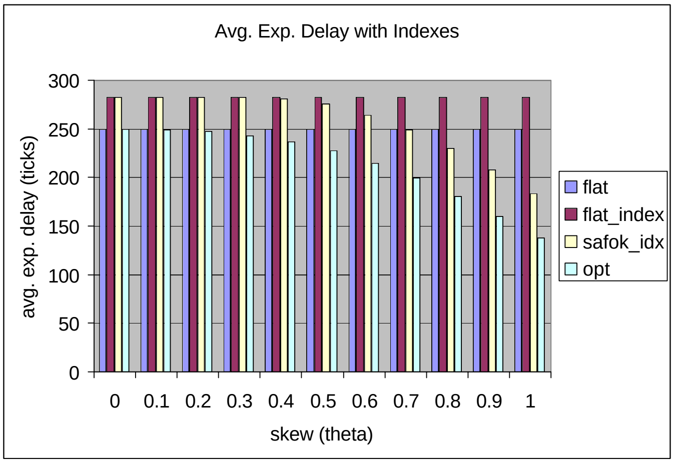

Fig. 8. Effect of Skew on Average Expected Delay using Indexed Schedules. Number of data items = 500.

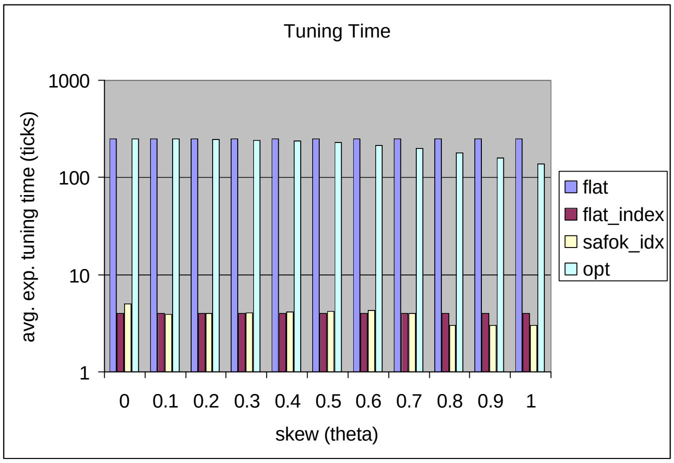

Fig. 9. Effect of Skew on Tuning Time using Indexed Schedules. Number of data items = 500. Note the log scale.

to transmit. So, although the resultant schedule has some fixed organization, it is basically bottom-up.

The partitioning problem from Section 3 is discussed in [16,15]. However, the algorithm in [15] is unnecessarily complex and has unreliable results over some range of parameters [10]. Furthermore [15] never considers mapping its results to a single channel and does not go into detail about the benefits of a predictible schedule. In [16], the partitioning problem was investigated for a query histogram application using stronger assumptions of the data.

5 Conclusions and Future Work

In this paper, we make improvements on bandwidth allocation algorithms while preserving the property of a predictable schedule. Our proposed algorithm, $Safok$, yields near-optimal average expected delays, and more importantly, $Safok$ requires fewer computational resources than typical scheduling algorithms. For example, the scheduling algorithm described in [1] requires $O(L)$ memory to store a schedule of length $L \geq N$, and has $O(N^3)$ computational complexity. The more modern *priority-based* schemes [3,4,5] require $O(N)$ computations *at each tick*. Furthermore, none of these algorithms, in contrast to the one proposed here, generates predictable fixed-length schedules, excluding them from enhancements such as indexing [9] or consistency control [11]. Our scheme requires $O(N \log N)$ computations and $O(N)$ memory regardless of the length of the schedule and results in near-optimal average expected delay given both unit-sized and varying-sized data items. A complete set of our experimental results is available upon request.

We also generate good tuning time results. Besides having superior average expected delay, the tuning time with $safok\_idx$ ($Safok$ using $(1, m)$ indexing) improves with increasing skew and beats $flat\_idx$ (flat using $(1, m)$ indexing) under high skew. We have therefore shown that a repetitive schedule can be easy to compute, deliver data in near-optimal time, and incorporate previous work on broadcast requiring flat schedules.

Due to space limitations, we do not discuss some other important results. One important consideration is related to schedule length and worst-case delay. Shorter schedules are desirable, because they allow schedules to change with user interests and workload [20] and consistency control is simpler [11]. Bounded worst-case delay is desirable for real-time applications [21]. Although other schedule-measuring schemes exist, the length of a $Safok$ schedule is typically $O(\text{lcm}\{N_i, i = 1, ..., K\}) \leq O(\Pi_{i=1}^{K} N_i)$. There are simple ways to control schedule-length and worst-case delay. See [1,10] for more details.

Our work is ongoing. We are keenly interested in generating broadcast schedules given requests consisting of *sets* of data items. Clearly, given two requests, or multiple non-overlapping requests, a shortest-job-first algorithm is best. In the general case, however, generating an optimal schedule is a combinatorial problem. Initial investigations of this problem have already been done in [22] with pre-emptible transmissions of long data items, and in [8] with multi-attribute indexing. But reflecting the divide in most of the literature, the former work focuses on bandwidth allocation while the latter focuses on power conservation. Our work would again bridge the gap.

We are also working on mapping our results into the multiple broadcast channel case [10]. The availability of multiple broadcast channels allows improved scalability and adaptability to changing environmental conditions and equipment configurations. Our partitioning results can be directly mapped to multiple channels, but we are also considering various indexing strategies as well. Indexing can have a serious impact as search over multiple channels can be more time and energy consuming than in a single channel case [23].

Acknowledgements

The authors would like to thank Professors Mostafa Ammar and Krithi Ramamritham for their technical comments and Mireille Jacobson for her editorial advice.

References

1. Ammar, M.H., Wong, J.W.: The design of teletext broadcast cycles. Performance Evaluation **5** (1985) 235–242
2. Herman, G., Gopal, G., Lee, K.C., Weinrib, A.: The datacycle architecture for very high throughput database systems. Proceedings of ACM SIGMOD International Conference on Management of Data (1987) 97–103
3. Aksoy, D., Franklin, M.: Scheduling for large-scale on-demand data broadcasting. Joint Conference Of The IEEE Computer and Communications Societies (1998)
4. Su, C.J., Tassiulas, L., Tsortras, V.J.: Broadcast scheduling for information distribution. Wireless Networks **5** (1999) 137–147
5. Vaidya, N.H., Hameed, S.: Scheduling data broadcast in asymmetric communication environments. Wireless Networks **5** (1999) 171–182
6. Acharya, S., Alonso, R., Franklin, M., Zdonik, S.: Broadcast disks: Data management for asymmetric communication environments. Proceedings of ACM SIGMOD International Conference on Management of Data (1995)
7. Su, C.J., Tassiulus, L.: Joint broadcast scheduling and user's cache management for efficient information delivery. Wireless Networks **6** (2000) 279–288
8. Hu, Q., Lee, W.C., Lee, D.L.: Power conservative multi-attribute queries on data broadcast. In: Proceedings of the IEEE International Conference on Data Engineering. (1998)
9. Imielinski, T., Viswanathan, S., Badrinath, B.: Energy efficient indexing on air. Proceedings of ACM SIGMOD International Conference on Management of Data (1994)
10. Yee, W.G., Omiecinski, E., Navathe, S.B.: Efficient data allocation for broadcast disk arrays. Technical report, College of Computing, Georgia Institute of Technology (2001)
11. Shanmugasundaram, J., Nithrakashyap, A., Sivasankaran, R., Ramamritham, K.: Efficient concurrency control for broadcast environments. Proceedings of ACM SIGMOD International Conference on Management of Data (1999) 85–96
12. Datta, A., VanderMeer, D., Celik, A., Kumar, V.: Adaptive broadcast protocols to support efficient and energy conserving retrieval from databases in mobile computing environments. ACM Transactions on Database Systems **24** (1999)
13. Stathatos, K., Roussopoulos, N., Baras, J.S.: Adaptive data broadcast in hybrid networks. Proceedings of the International Conference on Very Large Databases (1997)
14. Bowen, T.F., Gopal, G., Herman, G., Hickey, T., Lee, K.C., Mansfield, W.H., Raitz, J., Weinrib, A.: The datacycle architecture. Communications of the ACM **35** (1992)
15. Peng, W.C., Chen, M.S.: Dynamic generation of data broadcasting programs for broadcast disk array in a mobile computing environment. Conference On Information And Knowledge Management (2000) 38–45

16. Konig, A.C., Weikum, G.: Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. Proceedings of the International Conference on Very Large Databases (1999)
17. Navathe, S.B., Ceri, S., Wiederhold, G., Dou, J.: Vertical partitioning algorithms for database design. ACM Transactions on Database Systems **9** (1984)
18. Acharya, S., Franklin, M., Zdonik, S.: Disseminating updates on broadcast disks. Proceedings of the International Conference on Very Large Databases (1996)
19. Scheuermann, P., Weikum, G., Zabback, P.: Data partitioning and load balancing in parallel disk systems. The VLDB Journal **7** (1998) 48–66
20. Lee, W.C., Hu, Q., Lee, D.L.: A study on channel allocation for data dissemination in mobile computing environments. The Journal of Mobile Networks and Applications **4** (1999) 117–129
21. Fernandez, J., Ramamritham, K.: Adaptive dissemination of data in time-critical asymmetric communication environments. Proceedings of Euromicro Real-Time Systems Symposium (1999) 195–203
22. Acharya, S., Muthukrishnan, S.: Scheduling on-demand broadcasts: New metrics and algorithms. In: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking. (1998)
23. Prabhakara, K., Hua, K.A., Oh, J.: Multi-level multi-channel air cache designs for broadcasting in a mobile environment. Proceedings of the IEEE International Conference on Data Engineering (2000)