ENERGY EFFICIENT INDEXING ON AIR

T. Imielinski S. Viswanathan B. R. Badrinath
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

We consider wireless broadcasting of data as a way of disseminating information to a massive number of users. Organizing and accessing information on wireless communication channels is different from the problem of organizing and accessing data on the disk. We describe two methods, $(1, m)$ *Indexing* and *Distributed Indexing*, for organizing and accessing broadcast data. We demonstrate that the proposed algorithms lead to significant improvement of battery life, while retaining a low access time.

1 Introduction

The physical requirements of wireless communication channels, make the problem of organizing wireless broadcast data different from data organization on the disk. Index based organization of data transmitted over wireless channels, is very important from the power conservation point of view and can result in significant improvement in battery utilization. New technology can utilize and build upon some well known techniques for file organization and access. These traditional solutions cannot be applied directly and need substantial modification because, of different physical limitations of the wireless communication channels. New solutions require merging interdisciplinary expertise ranging from new communication protocols to file system and database design.

In this paper, we consider wireless data broadcasting as a way of disseminating information to a massive number of clients equipped with battery powered palmtops. Palmtops are not connected to any direct power source and run on small batteries (such as AA) and communicate on narrow bandwidth wireless channels. These physical requirements call for energy and bandwidth efficient solutions both on hardware and software levels.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

This paper and [IVB94a], provide different organizing techniques for broadcasting data and also for accessing that data. In [IVB94a], we concentrate on hashing methods with special emphasis on flexibility, in terms of the tradeoff between power consumption and access time. In this paper, we analyze index based schemes with special emphasis on minimizing the power consumption and also the time to access the data. [IVB94b] describes algorithms for organizing and accessing data based on clustering and non-clustering indices. We concentrate on the wireless communication medium, although most of the presented work can also be applied to a fixed wired network.

We distinguish between two fundamental modes of providing users with information:

Data Broadcasting: Periodic broadcasting of data on a communication channel. Accessing broadcasted data does not require uplink transmission and is "listen only". Querying involves simple *filtering* of the incoming data *stream* according to a user specified "filter".

Interactive/On-Demand: The client requests a piece of data on the uplink channel and the server responds by sending this data to the client.

In practice, a mixture of the above two modes will be used. The most frequently demanded items, the so called *hot spots* (stock quotes, airline schedules etc) will be broadcasted creating a sort of "storage on the air". Since the cost of broadcasting does not depend on the number of users, this method will scale up with no penalty when the number of users (hence, the requests) grows. For example, if stock information is broadcasted every minute, then it doesn't matter whether 10 users or 10,000 users are listening, the average waiting time will be 30 seconds. This would not be the case if stock information was provided on demand. The "on-demand" mode will have to be used for the less often requested items. Broadcasting them periodically would be a waste of bandwidth. However, even in pure "on-demand" mode it makes sense to batch requests for the same data and send the data once rather than cater individually to each request. Periodic data broadcasting

is the topic of this paper.

1.1 Motivation

The lifetime of a battery is expected to increase only 20% over the next 10 years [She92]. A typical AA cell is rated to give 800 mA-Hr at 1.2 V (0.96 W-Hr). The constraint of limited available energy is expected to drive all solutions to mobile computing on palmtops. There is a growing pressure on hardware vendors to come up with the energy efficient processors and memories. The Hobbit chip from AT&T is one such processor which consumes only 250 mW in the full operation mode (*active mode*). The power consumption in *doze mode* is only 50 μW (the ratio of power consumption in active mode to doze mode is 5000). When the palmtop is listening to the channel, CPU must be in active mode for examining data packets (finding, if they match the predefined data). CPU is a much more significant energy consumer than the receiver itself and since it has to be active to examine the incoming packets it may lead to waste of energy, especially if on an average only a very few data packets are of interest to the particular unit.

Therefore, it is definitely beneficial if palmtops can slip into *doze mode* most of the time and come into *active mode* only when the data of interest is expected to arrive. This requires the ability of selective *tuning*, which is discussed in detail, in this paper. Later in the paper, we provide an example of realistic energy savings due to the ability of selective tuning.
Energy efficient solutions are important due to the following reasons :

- Energy efficient solutions make it possible to use smaller and less powerful batteries to run the same set of applications for the same time. Smaller batteries are important from the portability point of view since, palmtops can be more compact and weigh less.
- With the same batteries, the unit can run for a very long time without the problem of changing the batteries ever so often. This can result in substantial monetary savings and avoids recharging. Recharging can be cumbersome especially if the client is on the move. With energy efficient solutions batteries may have to be recharged only every few days, rather than every few hours.
- For environmental consideration. Every battery that is disposed is an environmental hazard.

The bandwidth of a wireless channel can vary from 1.2 $Kbps$ for slow paging channels, through 19.2 $Kbps$ (characteristic of cellular proposals like CDPD - Cellular Digital Packet Data) to about 2 $Mbps$ of the wireless LAN. The methods developed in this paper are independent of the bandwidth of the underlying network and are motivated mainly by the battery power constraints and using the available bandwidth efficiently.

Wireless data broadcasting can be viewed as *storage on the air* - an extension of the server's memory. On an average, the access time for such a storage is equal to half the time necessary to broadcast the whole data. Thus, in case of the low bandwidth channel the access time can be quite substantial. On the other hand, the access time is independent of the number of users who are listening. Hence, this "public" storage scales well with an increase in the number of users.

Broadcasting over a fast, fixed network has been investigated as an information dissemination mechanism in the past. In the *Datacycle* project at Bellcore [Bow92, Her87], database circulates on a high bandwidth network (140 $Mbps$) and users query this data by filtering relevant information using a special massively parallel transceiver, capable of filtering up to 200 million predicates a second.
The main differences between wireless broadcasting considered in this paper and broadcasting considered in the Datacycle architecture can be summarized as follows:

- Power conservation is of no concern in Datacycle architecture while it is a major physical requirement for the wireless broadcast.
- The wireless bandwidth is much lower than the bandwidth assumed in DataCycle architecture.

Gifford in [Giff85] describes a system where newspapers are broadcasted over the FM band and downloaded by PCs equipped with radio receivers. There is a single communication channel and power conservation plays little role since, the PCs are connected to a continuous power supply.

In section 2, we discuss data organization basics for broadcasting. In section 3 and section 4, we discuss two indexing algorithms, $(1, m)$ *indexing* and *distributed indexing* (respectively), for organizing and accessing data. Section 5 compares the performance of the two algorithms with the performance of optimal algorithms. Subsection 5.1 will interpret our results in terms of the improvement in power consumption and access time, with an example of an application broadcasted over cellular data link. We will demonstrate that our methods result in significant savings in terms of power consumption and access time, and is very close to that of optimal ways of broadcasting. In section 6, we present conclusions and discuss future work.

2 Data Organization for Broadcasting

Consider a file consisting of a number of records which are identified by their primary key. The file

is not static and can be updated frequently, so its content and its size can grow and shrink often. The server broadcasts this file periodically to a number of clients, on a communication channel which is assumed to have broadcasting capability. Henceforth, the communication channel will be referred to as broadcast channel. Clients will only receive the broadcasted data and fetch individual records (identified by a key) from the broadcast channel. However, updates to the file are reflected *only* between successive broadcasts. Hence, the content of the current version of the broadcast is completely determined before the start of broadcast of that version.

In our model, filtering is by simple pattern matching of the primary key. Clients will remain in doze mode most of the time and tune in periodically to the broadcast channel, in order to download the required data. Selective tuning will require that the server, in addition to broadcasting the data, also broadcast a directory that indicates the point of time in the broadcast channel when particular records are broadcasted. One idea is to let all the clients cache a copy of this directory. However, this solution has the following disadvantages:

- In a mobile environment, when a client leaves its *cell* and enters a new *cell*, it will need the directory of the data being broadcasted in that cell. The directory it had cached in its previous cell may not be valid in the new cell.
- New clients who have no prior knowledge of the broadcast data organization, will have to access the directory from the air. Palmtops that are turned off and switched on again, can be thought of being classified in this category.
- Broadcast data can change its content and grow or shrink any time between successive broadcasts. In this case, the client has to refresh it's cache. This may generate excessive traffic between clients and the server. In fact the directory will become a *hot spot*. Since we assume that the broadcasted data is very frequently accessed and thus it is broadcasted, the same argument holds even more strongly for the directory. Therefore, the directory is broadcasted as well.
- If many different files are broadcasted on different channels, then clients need excessive storage for the directories of all the files being broadcasted and palmtops have limited storage.

Due to the above reasons, we broadcast the directory of the file in the form of an index in the broadcast channel. The index we consider is a multi-leveled index.

Let us first justify the use of index for the broadcasted data. If data is broadcasted without any form of index, then the client in order to filter a data record, will have to tune to the channel on an average, half of the time it takes to broadcast the file. This is unacceptable as it requires the client to be active (be in *active mode*) for a long time, thereby consuming scarce battery resource. We would rather provide a selective tuning ability, enabling the client to become active only when data of interest is being broadcasted. The broadcast channel is the source of *all* information to the client including data as well as index. We consider a *single channel* since multiple channels are equivalent to a single channel with capacity (bit rate/bandwidth) equivalent to the combined capacity of the corresponding channels. There is no point in having separate channels for index and data, because index is also a form of data. Moreover, minimizing the data broadcasted on a single channel will encompass the problem of having different channels and optimizing data broadcast in each of them.

Each version of the file along with all the (interleaved) index information will constitute a *bcast*. Pointers to specific buckets within the *bcast* will be provided by specifying an $offset$ from the bucket which holds the pointer, to the bucket to which the pointer points to. The actual time of broadcast for such a bucket (from the current bucket) is the product of $(offset - 1)$ and the time necessary to broadcast a bucket.
For a file being broadcasted on a channel, the following two parameters are of concern:

- *Access Time*: The average time elapsed from the moment a client wants a record identified by a primary key, to the point when the required record is downloaded by the client.
- *Tuning Time*: The amount of time spent by a client listening to the channel. This will determine the power consumed by the client to retrieve the required data.

Listening to the broadcast channel requires the client to be in the *active mode*. Hence, the *tuning time* for accessing data is determined by the amount of time spent being in the *active mode* (*plus* a small amount for being in *doze mode*).
The *access time* for a broadcast is determined by the following two parameters:

- *Probe Wait:* When an initial probe is made into the broadcast channel, the client gets the information about the occurrence of the next-nearest index information relevant to the required data. The average duration for getting to this nearest index information is called the *probe wait*. This wait is equal to half the distance between two consecutive index information.
- *Bcast Wait:* The average duration from the point the index information relevant to the required data

is encountered, to the point when the required record is downloaded is called the *bcast wait*.

The *access time* is the sum of *probe wait and bcast wait*. These two factors work against each other. If we try to minimize one of them the other will increase. For example, in order to minimize *bcast wait*, we can broadcast the index once (at the beginning of each *bcast*). In this case *probe wait* will be large, since the client will always have to wait for the index (till the starting of the next *bcast*) missing the required data in the current *bcast*. On the other extreme, for minimizing the *probe wait*, index can precede each data bucket in the broadcast. This would minimize *probe wait* but would increase *bcast wait* (due to an increase in overall length of the broadcast).

In periodic wireless broadcasting, air behaves like a *storage medium* requiring new data organization and access methods. The broadcast *tuning time* roughly corresponds to the *access time* for the disk based files. There is no parameter in the disk that directly corresponds to the *access time* on the broadcast channel. The file's storage occupancy may seem the closest but it only accounts for one of the components of the access time - the *bcast wait* and does not capture the *probe wait* factor.

The main difference between the organization of broadcasted data (data on air) versus data on disk can be summarized as follows:

- Data on Air is characterized by two parameters: access time and tuning time, contrary to just one parameter (access time) for data on disk.

The goal of this paper is to provide methods for *allocating* index together with data on the broadcast channel. We do not provide new *types* of indices but rather, new index allocation methods which would multiplex index and data in an efficient way with respect to the two basic parameters: access time and tuning time. Our methods will allocate index and data for any type of index.

The smallest logical unit of the broadcast will be called a *bucket*. All buckets are of the same size. Bucket sizes are equal only for convenience and uniformity. The argument is the same as the reason for having blocks of equal size for indexing in conventional memory media. Bucket size will be equal to some multiple of the *packet size* (the basic unit of message transfer in packet switched networks). Both access time and tuning time will be measured in terms of number of buckets. We will discuss the organization and the access of the broadcasted data with the assumption that setup time and setup power consumption for tuning into a channel or going into *doze mode* are negligible. [IVB94b] discusses as to how the algorithms have to be modified

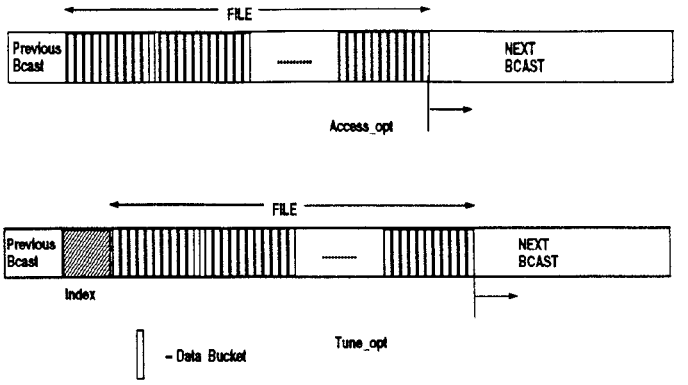


Figure 1: Optimal Methods

if we eliminate this assumption. In the algorithms that we describe, index will be interleaved with data. Index will provide a sequence of pointers which eventually lead to the required data.

In general, data organization algorithms which seek optimum in two dimensional space of access and tuning time are of importance. Below, we present two algorithms which are optimal in the one dimensional space of access time and tuning time. These will serve as benchmarks for comparisons to our algorithms.

Access_opt:

This algorithm provides the best access time with a very large tuning time. The best access time is obtained when no index is broadcasted along with the file. The size of the entire broadcast is minimal in this way. Clients simply tune into the broadcast channel and filter all the data till the required records are downloaded. For a file of size *Data* buckets, on an average it takes $\frac{Data}{2}$ time to get to the record with the required primary key. Thus, the access time for *access_opt* algorithm is $\frac{Data}{2}$. The average tuning time is the worst and is equal to $\frac{Data}{2}$. This is because the client has to to be in *active mode* throughout the period of access. This method is illustrated in Figure 1.

Tune_opt:

This algorithm provides the best tuning time with a large access time. The server broadcasts the index at the beginning of each *bcast*. The client which needs the record with primary key K, tunes into the broadcast channel at the beginning of the next *bcast* to get the index[1]. It then follows the index pointers to the record with the required primary key. The tuning time is equal to the number of levels in the multi-leveled index tree *plus* one (for the final probe to download the record). This method has got the worst access time because, clients have to wait till the beginning of the next

[1] It may not be possible for the client to have a knowledge about the beginning of the next *bcast*. This is because files could vary in size dynamically and also, in a mobile environment different *cells* might have different file sizes

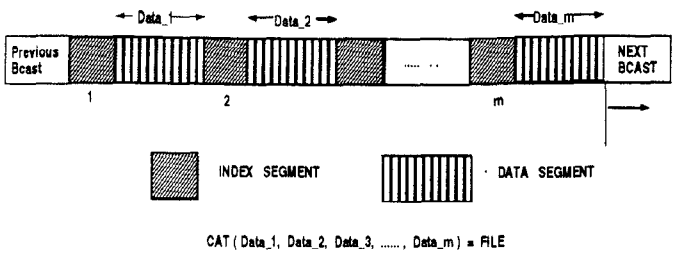


Figure 2: (1,m) indexing

broadcast even if the required data is just in front of them. With *Index* denoting the size of index of the file, *probe wait* is $(\frac{Data+Index}{2})$, *bcast wait* is $(\frac{Data+Index}{2})$ and access time is $(Data + Index)$. This method is also illustrated in Figure 1.

If tuning time is of no concern, then one can use *access_opt* for getting the least access time. If access time is of no concern, then one can use *tune_opt* for getting the least tuning time. Usually, both tuning time and access time are of interest, hence the above two algorithms have only theoretical significance. We use them as benchmarks for more sophisticated algorithms developed later in the paper.

In the next section, we present a simple indexed data organization called $(1, m)$ *indexing* and later, we present a more sophisticated indexed data organization called *distributed indexing*.

3 (1,m) Indexing

We distinguish between *index buckets* holding the index and *data buckets* holding the data. An *index segment* refers to the set of contiguous index buckets and a *data segment* refers to the set of data buckets broadcasted between successive index segments.

$(1, m)$ indexing is an index allocation method in which the index is broadcasted m times during the broadcast of one version of the file. The whole index is broadcasted following every fraction $(\frac{1}{m})$ of the file. Figure 2 illustrates this algorithm.

All buckets have an $offset$ to the beginning of the next index segment. The first bucket of each index segment has a tuple, with the first field as the primary key of the record that was broadcasted last and the second field as the offset to the beginning of the next *bcast*. This is to guide the clients that have missed the required record in the current *bcast* and have to tune to the next *bcast*.

The access protocol for record with key K is as follows:

- Tune into the current bucket on the broadcast channel.
- Read the $offset$ to determine the address of the next nearest index segment.
- Go into *doze mode* and tune in at the broadcast of the index segment.
- From the index segment determine when the data bucket containing the record with primary key K is broadcasted. This is accomplished by successive probes, by following the pointers in the multilevel index (the client might go into doze mode between two successive probes).
- Tune in again when the bucket containing the record with primary key K is broadcasted and download the record.

3.1 Analysis

In the following analysis, the probability distribution of the initial probe of the clients is assumed to be uniform within the *bcast*. Let *Data* denote the average size of the file (the file could change in size between two successive *bcasts*). Let n denote the capacity of the bucket i.e., the number of *(search-key plus pointer)*s a bucket can hold.

Let k denote the number of levels in the index tree and finally, let *Index* denote the number of buckets in the index tree.

When the index tree is fully balanced:

$k = \lceil \log_n(Data) \rceil$

$Index = 1 + n + n^2 + \ldots + n^{k-1}$

Access Time:

The *probe wait* is $\frac{1}{2} * (Index + \frac{Data}{m})$

and the *bcast wait* is $\frac{1}{2} * ((m * Index) + Data)$.

Hence, the access time is:

$\frac{1}{2} * (Index + \frac{Data}{m}) + \frac{1}{2} * ((m * Index) + Data)$ *i.e.,*

$$\frac{1}{2} * ((m+1) * Index + (\frac{1}{m} + 1) * Data)$$

Tuning Time:

The first probe is the initial probe that gets a pointer to the next nearest index bucket. Then, k probes are required for following the pointers in the index[2]. Finally, one last probe is required for downloading the required record.

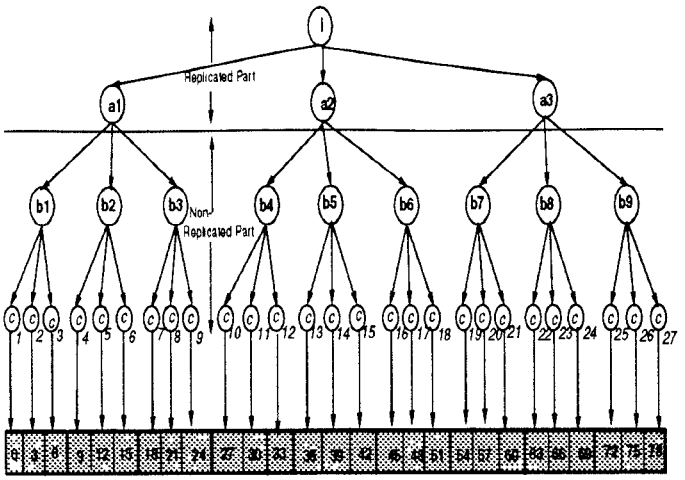
Thus, the tuning time is:

$$2 + \lceil \log_n(Data) \rceil$$

Optimum m:

Now, we present a formula to compute the optimal m so as to minimize the access time for the (1,m) indexing. For finding the minimal access time, we differentiate the above formula (for access time) with respect to

[2] in case the required record has been missed, which occurs with a probability of 0.5, then we need another probe at the beginning of the next *bcast*. This adds 0.5 to the average tuning time, but this is ignored in the formulae. This is true for both the algorithms that we describe in this section



NRR = { b1, b2, b3, b4, b5, b6, b7, b8, b9 }

Rep(b1) = { I, a1 }	Ind(b1) = { b1, c1, c2, c3 }	Data(b1) = { 0,.....,8 }
Rep(b2) = { a1 }	Ind(b2) = { b2, c4, c5, c6 }	Data(b2) = { 9,.....,17 }
Rep(b3) = { a1 }	Ind(b3) = { b3, c7, c8, c9 }	Data(b3) = { 18,... ,26 }
Rep(b4) = { I, a2 }	Ind(b4) = { b4, c10, c11, c12 }	Data(b4) = { 27,..... ,35 }
Rep(b5) = { a2 }	Ind(b5) = { b5, c13, c14, c15 }	Data(b5) = { 36,.....,44 }
Rep(b6) = { a2 }	Ind(b6) = { b6, c16, c17, c18 }	Data(b6) = { 45,.... ,53 }
Rep(b7) = { I, a3 }	Ind(b7) = { b7, c19, c20, c21 }	Data(b7) = { 54,.....,62 }
Rep(b8) = { a3 }	Ind(b8) = { b8, c22, c23, c24 }	Data(b8) = { 63, . ..,71 }
Rep(b9) = { a3 }	Ind(b9) = { b9, c25, c26, c27 }	Data(b9) = { 72,..... ,80 }

Figure 3: File in the running example

m, equate it to zero and solve for m, m^* denotes the optimum m.

$$m^* = \sqrt{\frac{Data}{Index}}$$

Hence, we divide the file into m^* equal parts (data segments). In the broadcast channel, each data segment is preceded by the index.

4 Distributed Indexing

We can improve upon (1,m) indexing algorithm by cutting down on the replication of index. Distributed indexing is a technique in which index is partially replicated. This method is based on the observation that there is no need to replicate the entire index between successive data segments - it is sufficient to have only the portion of index which indexes the data segment which follows it. Although the index is interleaved with data in both, (1,m) and distributed indexing, in the distributed indexing only *relevant* index is interleaved as opposed to interleaving the whole index as in (1,m) indexing.

Since the distributed indexing method is fairly involved, we will start with a subsection which motivates the method. The algorithm is formally described later.

4.1 Motivation

We will proceed by describing different variants of index distribution for the specific example of the file shown in figure 3. Figure 3 shows a data file consisting of 81 data buckets. The lower most level consisting of square boxes

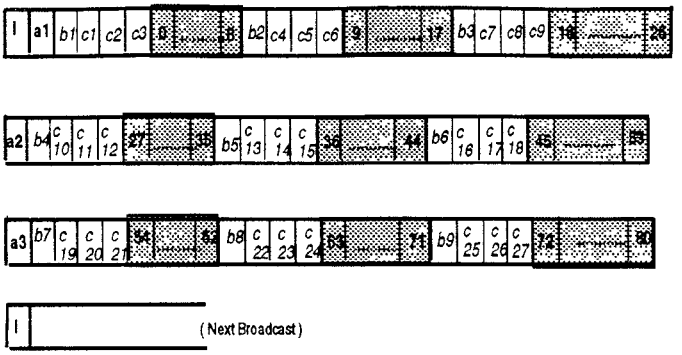


Figure 4: Non-replicated Distribution

represent a collection of three data buckets. The index tree is shown above the data buckets. Each index bucket has three pointers (the three pointers of each index bucket in the lower most index tree level is represented by just one arrow). The terms displayed below the picture of the index will be explained later.

We will consider three different index distribution methods with the last one being distributed indexing. In all three methods the index is interleaved with data and the index segment describes only data in the data segment which *immediately* follows it. The methods differ in the degree of the replication of index information:

- Non-replicated Distribution

  Different index segments are disjoint. Hence, there is no replication.

- Entire Path Replication

  The path from root of index to an index bucket B is replicated just before the occurrence of B.

- Partial Path Replication (Distributed Indexing)

  Consider two index buckets B and B'. It is enough to replicate just the path from the least common ancestor of B and B', just before the occurrence of B', provided we add some additional index information for navigation.

The non-replicated distribution and the entire path replication are two extremes. Distributed indexing aims at getting the best of both the schemes. Below, we demonstrate index distribution for the three schemes and show how data will be accessed in each of them. Figure 4, Figure 5 and Figure 6 will illustrate the three schemes. In these figures, the current *bcast* is represented in three levels for easy of illustration. The broadcast channel is organized by taking the first level followed by the second, which in turn is followed by the third level. The shaded portion denotes data buckets.

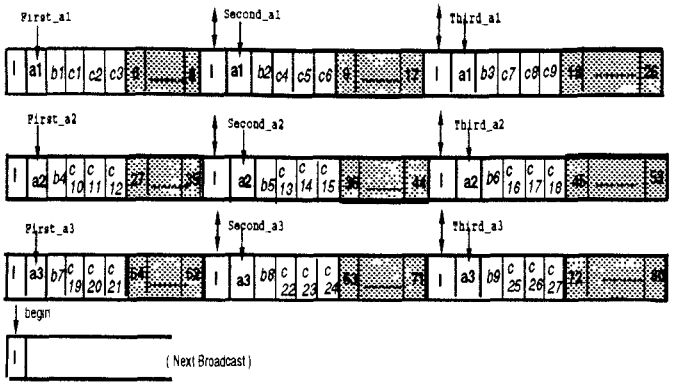


Figure 5: Entire Path Replication

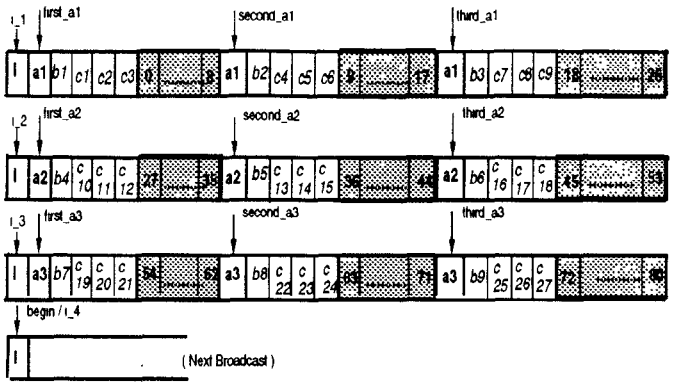


Figure 6: Distributed Indexing

In the running example to describe the three schemes, assume that the client requires a record in bucket 66 and makes the initial probe at data bucket 3.

Non-replicated Distribution

Figure 4 illustrates the layout for this scheme. The *offset* at bucket 3 points to the beginning of the next *bcast*. The client will make the following successive probes (in the next *bcast*) I, $a3$, $b8$, $c23$ and bucket 66. Since the root of index is broadcasted only once for each *bcast*, the initial probe will result in obtaining the offset to the beginning of the next *bcast*. In order to determine the occurrence of the required record, we have to get to the root of the index. Hence, the *probe wait* for this scheme will be quite significant and will offset savings in *bcast wait* due to the lack of replication[3].

Entire Path Replication

Figure 5 illustrates the index distribution when the entire path from root of the index tree to each index buckets b_i is replicated. The replication is just before the occurrence of bi. The *offset* at data bucket 3 will direct the client to the index bucket 'I' that precedes *second_a*1. Then, the client makes the following successive probes: *first_a*3, $b8$, $c23$ and bucket 66. The access time suffers from the replication of index information. In this example, the root was unnecessarily replicated six times, as demonstrated below.

Partial Path Replication - Distributed Indexing

Figure 6, shows that we can further improve on the access time provided by entire path replication. Instead of replicating the entire path we will replicate only a *part* of it.

Notice that root I is no longer replicated many times. The *offset* at the data bucket 3 will direct the client to *second_a*1. However, to make up for the lack of root preceding *second_a*1, there is a small index called the *control index* within *second_a*1. If the local index (in *second_a*1) does not have a branch that would lead to the required record then, the control index is used to direct the client to a proper branch in the index tree. The control index in *second_a*1, directs the client to i_2. At i_2 the root is available and the client makes the following probes: *first_a*3, $b8$, $c23$ and bucket 66. In case, a record in bucket 11 was being searched by the client, reading the bucket *second_a*1 would provide the client with the required information to successively tune in at $b2$, $c4$ and bucket 11. In this case, having the root just before *second_a*1 would have been a waste of space (this is true if the search key was in any data bucket 9 through 26). The additional space which is necessary to store the control index is a small overhead compared to the savings resulting from partial index path replication.[4]

Figure 7 shows the control index for index buckets that are part of the index tree described in Figure 3 and whose layout is shown in Figure 6. The first part of each control index element denotes the key to be compared with during the data access protocol. The second part denotes the pointer to be followed in case the comparison turns out to be positive. For example, if a record in bucket ≤ 8, is being searched for, then the control index at *second_a*1, directs the client to the beginning of the next *bcast*. However, if a record in bucket > 26, is being searched for, then the search is directed to i_2. Otherwise the search in the control index fails and the rest of *second_a*1 is searched.

[3] the *offset* at data bucket 3 could have directed the client to index bucket $b2$. In which case, the client could make the following successive probes: $b2$, $b3$, $a2$, $a3$, $b8$, $c23$ and bucket 66. The client need not have made an extra probe at $b3$. There are three pointers per index bucket, four levels in the index tree, and one extra probe was made. The number of extra probes grows linearly with the increase in the number of pointers in the index bucket and it also grows linearly with the number of levels in the index tree. This becomes substantial as the number of data buckets and the capacity of the index bucket increases. The tuning time is no longer logarithmic in the number of data buckets

[4] replicated buckets can be modified by removing the entries for records that have been broadcasted before the occurrence of that bucket. This will result in savings of some space, which when amortized over all the replicated buckets in the *bcast*, will make up for the space taken up by the control index

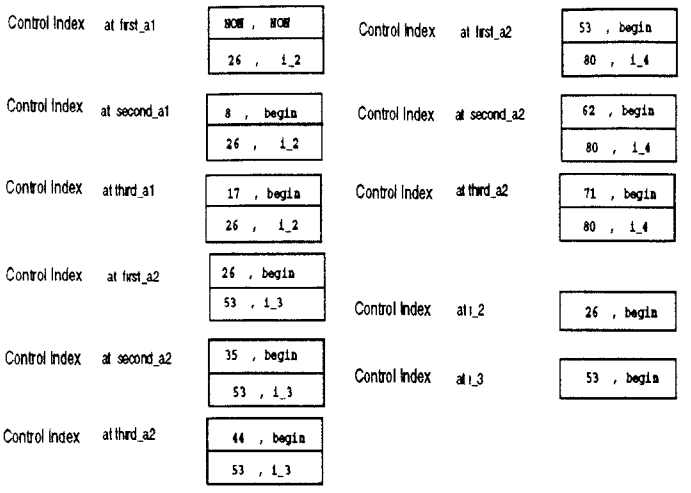


Figure 7: Control Index

In the following subsection we present a formal description of the distributed indexing algorithm.

4.2 Distributed Indexing Algorithm

Given an index tree, this algorithm provides a method to multiplex it together with the corresponding data file on the broadcast channel. Thus, the distributed indexing method is not a new method of index construction but a method of allocation of a file and its index on the broadcast channel.

The distributed indexing algorithm takes an index tree and multiplexes it with data by subdividing it into two parts:

- The replicated part
- The non-replicated part

The replicated part constitutes the top r levels of the index tree, while the non-replicated part consists of the bottom $(k - r)$ levels. The index buckets of the $(r + 1)$ th level are called *non-replicated roots* and they are collectively denoted by NRR. The index buckets in NRR are ordered left to right, consistent with their occurrence in the $(r + 1)$ th level.

Each index subtree rooted in a non-replicated root in NRR will appear only once in the whole broadcast just in front of the set of data segments it indexes (points to). Hence, each descendant node of a non-replicated root of the index will appear *only once* in the given version of the broadcast. On the other hand, each node of the index tree which appears above a non-replicated root is replicated the number of times *equal to a number of children it has.*

Definitions

- I: Denotes the root of the index tree.
- B: Denotes an index bucket belonging to NRR.
- B_i: Denotes the i th index bucket in NRR.
- $Path(C, B)$: The sequence of buckets along the path from index bucket C to B (excluding B).
- $Data(B)$: The set of data buckets indexed by B.
- $Ind(B)$: The part of the index tree below B (including B).
- $LCA(B_i, B_k)$: the least common ancestor of B_i and B_k in the index tree.

Let $NRR = \{B_1, B_2, \ldots, B_t\}$
$Rep(B_1) = Path(I, B_1)$, B_1 is the first bucket in NRR.
$Rep(B_i) = Path(LCA(B_{i-1}, B_i), B_i)$ for $i = 2, \ldots, t$.
Thus, $Rep(B)$ will refer to the *replicated* part of the path from the root of the index tree to index segment B. $Ind(B)$, on the other hand will refer to the *non-replicated* portion of the index. Figure 3 shows the values of $Data$, Rep and Ind for each of the index buckets in NRR.

Each version of the broadcast will be a sequence of triples:

$< Rep(B),\ Ind(B),\ Data(B) >$ $\forall B \in$NRR, *in the left to right order*

Let $P_1, P_2, \ldots, P_r$ denote the sequence of buckets in $Path(I, B)$.
Control index is stored in each of the P_i index buckets.
Let $Last(P_i)$ denote the value of the primary key in the last record that is indexed by bucket P_i.
Let $NEXT_B(i)$ denote the offset to the next-nearest occurrence of P_i (which in turn is the index bucket at level i in $Path(I, B)$). Let l be the value of the primary key in the last record broadcasted prior to B and let *begin* be the offset to the beginning of the next *bcast*.
Control index in P_i, which belongs to $Rep(B)$ (i.e., it precedes $Ind(B)$ and $Data(B)$) will have the following i tuples:

$[\ l,\ begin\ ]$
$[\ Last(P_2),\ NEXT_B(1)\ ]$
$[\ Last(P_3),\ NEXT_B(2)\ ]$
.....
$[\ Last(P_i),\ NEXT_B(i-1)\ ]$

The control index in bucket P_i, is used as follows: Let K be the value of the primary key of the required record. If ($K < l$) is true, then the search is directed the beginning of the next *bcast* (i.e., the *begin* pointer is followed). If the result of the comparison is false, then ($K > Last(P_j)$) is checked for smallest such j to be true. If $(j \leq i)$, $NEXT_B(j-1)$ is followed, else the rest of the index in bucket P_i is searched as in conventional indexing.
The access protocol for a record with primary key K is as follows:

- (i) Tune to the current bucket of the *bcast*. Get the offset for the bucket containing control index.
- (ii) Tune again to the beginning of the designated bucket with control index. Determine, on the basis of the value of the search key K and the control index, whether to:
  - a) Wait until the beginning of the next *bcast* (the first tuple). In this case tune to the beginning of the next *bcast* and proceed as in (iii).
  - b) Tune in again for the appropriate higher level index bucket. i.e., follow one of the "NEXT" pointers and proceed as in (iii).
- (iii) Probe the designated index bucket and follow a sequence of pointers (the client might go into doze mode between two successive probes) to determine when the data bucket containing the record with key K is going to be broadcasted.
- (iv) Tune in again when the bucket containing the record with key K is broadcasted and download the record.

4.3 Analysis

In the following subsection, we derive formulas for access time, tuning time and the optimum number of replicated levels. We assume that the initial probe by the clients are uniformly distributed over the period of the whole *bcast*. Let r denote the number of replicated (top) levels. Let n denote the capacity of the bucket and finally, let k denote the number of levels in the index tree.

We consider index trees which are balanced (all leaves are on the same level) and assume that each node has the same number of children. Needless to say, in reality, index trees may be radically different (unbalanced, varying fanout) and our algorithm works for arbitrary index trees. The formula for the general case is given in [IVB94b].

Access Time:

The access time is a sum of *probe wait* and *bcast wait*. Let us estimate the *probe wait* first.

The control information is present in every replicated index bucket. The replicated index buckets are present in front of each $Ind(B)$, which in turn is in front of $Data(B)$. Thus, the maximum distance separating two replicated buckets is: $Ind(B) + Data(B)$

The average size of $Ind(B)$ can be calculated as:

$$1 + n + \ldots + n^{k-r-1} = \frac{n^{k-r}-1}{n-1}$$

The average size of $Data(B)$ is : $\frac{Data}{n^r}$

Hence, the *probe time* will be equal to

$$\frac{n^{k-r}-1}{n-1} + \frac{Data}{n^r}$$

Bcast wait is equal to half the total length of the broadcast, which includes the data, the index, and the additional overhead due to the index replication.

In general, as the index overhead increases (i.e., when r is increased) the bcast wait increases, but the probe wait decreases. Intuitively, the index overhead reflects the degree of index replication and is an "investment" towards reducing the probe time. This is because the more replicated index we have, the less time it will take to get to the control index bucket after initial tuning.

Each bucket b of the replicated part of the index tree (the top r levels) will be replicated as many times as the number of children that b has. Therefore, the total number of replicated index buckets will be equal to the total number of nodes in the upper $(r+1)$ levels of the index tree minus one. The replicated index buckets, include part of the index tree (top r levels) plus the index overhead. The index overhead is obtained by subtracting the number of buckets in the top r levels of the index tree, from the total replicated buckets. Thus, the index overhead is equal to the number of buckets in the $(r+1)$ th level *minus* one i.e., $(n^r - 1)$

Thus, the access time is

$$\frac{1}{2} * (\frac{n^{k-r}-1}{n-1} + \frac{Data}{n^r} + (n^r - 1) + Index + Data)$$

Tuning Time:

Tuning time primarily depends on the number of levels of the index tree. The initial probe of the client is for determining the occurrence of control index. The second is for the first access to control index. Control index directs the client to the required higher level index bucket. Next, the number of probes by the client is equal to at most k, the number of levels in the index tree. Finally, the client has to download the required record.

Thus, the tuning time using distributed indexing is bound by:

$$\lceil \log_n(Data) \rceil + 3$$

Optimizing the number of replicated levels:

Optimizing the number of replicated levels, has no impact on the tuning time. It affects only the access time. Thus optimizing the number of replicated levels r, corresponds to minimizing the access time. For decreasing *probe wait*, the length of $(Ind(B)+Data(B))$ has to be reduced. This can be done by increasing the number of replicated levels. However, this would amount to an increase in *bcast wait* as the length of the entire broadcast would increase (due to increased replication). Hence, there is a trade off between *probe wait* and *bcast wait*.

Probe wait is a monotonically decreasing function, with a maximum when $r = 0$ and a minimum when $r = k$. *Bcast wait* is a monotonically increasing function, with

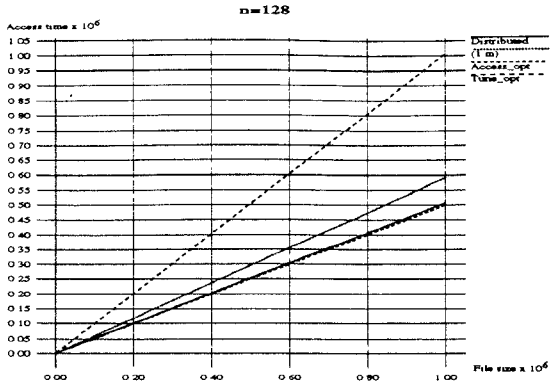


Figure 8: Comparison for Large Bucket Capacity

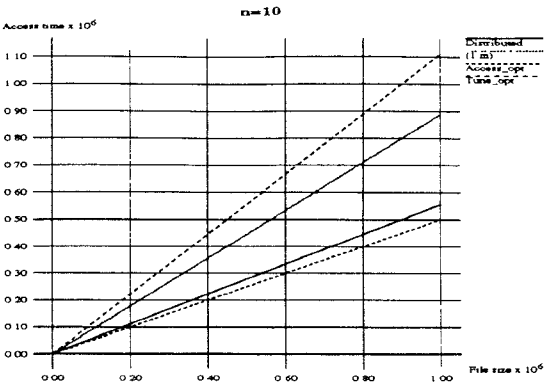


Figure 9: Comparison for Small Bucket Capacity

a maximum when r=k and a minimum when $r = 0$. Consider the case when r increases from 0 to 1:
probe wait decreases by $\frac{1}{2} * (\frac{Data*(n-1)}{n} + n^k)$
and *bcast wait* increases by $(\frac{n^0*(n-1)}{2})$.
In general, when r increases by one to $(r + 1)$:
probe wait decreases by $\frac{1}{2} * (\frac{Data*(n-1)}{n^{r+1}} + n^{k-r})$
and *bcast wait* increases by $(\frac{n^r*(n-1)}{2})$.
This is because for each increment of r, the total number of replicated buckets grows to $(n^{r+1} - 1)$ from $(n^r - 1)$, thus a factor of $(n^{r+1} - 1 - (n^r - 1)) = n^r * (n - 1)$ is added for the number of replicated index buckets.
Thus, for achieving the minimum access time, we should keep increasing r as long as:

$$\frac{1}{2} * (\frac{Data * (n-1)}{n^{r+1}} + n^{k-r}) \quad > \quad (\frac{n^r * (n-1)}{2})$$

$$Data * (n-1) + n^{k+1} \quad > \quad n^{2*r+1} * (n-1)$$

$$\frac{1}{2} * (\ \log_n(\frac{Data * (n-1) + n^{k+1}}{n-1}) - 1\ ) \quad > \quad r$$

Thus the optimum r, r' is:

$$r' \quad = \quad \lfloor \frac{1}{2} * (\ \log_n(\frac{Data * (n-1) + n^{k+1}}{n-1}) - 1\ ) \rfloor + 1$$

5 Comparisons

In the following, we compare (1,m) indexing and distributed indexing algorithms with the two optimal algorithms, *access_opt* and *tune_opt*. We consider files of various sizes, for two different bucket capacities (n =10 and n = 128). Figure 8 shows the access time obtained using *distributed indexing*, *(1,m) indexing*, *access_opt* and *tune_opt*, when n is 128. Figure 9 shows the access time obtained for the algorithms for n equal to 10. X-axis denotes the number of buckets in the file. Y-axis denotes the access time (in terms of the number of buckets). These two graphs illustrate that distributed indexing performs much better (in terms of the access time) than (1,m) indexing (except in case of the number of levels being one, as discussed below). Both (1,m) indexing and distributed indexing always perform better than tune_opt. The more the capacity of the buckets (n) the better the performance of the suggested algorithms.

File	Index	T_opt	(1,m)	(D I)	A_opt
1000	9	3	4	5	500
5000	41	3	4	5	2500
10000	80	3	4	5	5000
10^5	790	4	5	6	50000

Table 1: Tuning time comparison for n=128

Table 1 illustrates the tuning time required by the four algorithms, for different file sizes, for n=128. Table 2 shows the same for n=10. The numbers in the two tables are in terms of buckets. The first and the second column refer to the file size and the index for that file. The third, forth, fifth and sixth columns denote the tuning time of tune_opt, (1,m) indexing, distributed indexing and access_opt algorithms respectively.

As the tables illustrate (and so do the formulae) the tuning time due to tune_opt and (1,m) indexing is almost the same and is optimal. The tuning time of distributed indexing is also almost equal to that of tune_opt (difference of just two buckets – always). The tuning time of access_opt is very large and is very much higher than the other three.

When the index size is equal to one bucket, then distributed indexing is not advantageous. This is

File	Index	T_opt	(1,m)	(D I)	A_opt
1000	111	4	5	6	500
5000	556	5	6	7	2500
10000	1111	5	6	7	5000
10^5	11111	6	7	8	50000

Table 2: Tuning time comparison for n=10

because we cannot distribute the index in this case (as a bucket is a unit, which is indivisible). In this case (1,m) indexing scheme is better. In a file with a multi-leveled index tree, distributed indexing always performs better in terms of access time and the tuning time is one more than in (1,m) indexing.
From the above, we can conclude the following:

- (1,m) indexing achieves a tuning time that is almost equal to the optimal tuning time (tune_opt). The tuning time of (1,m) indexing is substantially better than that of access_opt.
- The access time achieved by (1, m) indexing is between the optimal access time (access_opt) and the access time due to tune_opt.
- In terms of access time, distributed indexing is always better than (1,m) indexing, except in the case when the index fits into a single bucket. In terms of tuning time, both are almost equivalent, tuning time with distributed indexing is just one more than that of (1,m) indexing.
- Distributed indexing achieves a tuning time very near the optimal (tune_opt) and it also results in an access time very close to the optimal access time (access_opt) for large capacities (n) of the bucket. Distributed indexing is almost as good as the optimal algorithm for access time and the optimal algorithm for tuning time.
- Our algorithms are significantly better than access_opt and almost as good as tune_opt, in terms of the power consumption. The proposed algorithms achieve an access time almost equal to access_opt and the access time achieved is much better than that of tune_opt.

5.1 Practical Implications

Consider a broadcasting system that is similar to the *quotrex system*[5] where, a stock market information of size $16*10^4$ *Bytes*, is being broadcasted. The broadcast channel has a bandwidth of 10 *Kbps*. Let the bucket length be 128 *bytes*. Thus, there are 1250 buckets of data. Let n, the number of *(search-key plus pointer)*'s that can fit in a bucket, be 25. The index size is 53 buckets. It takes around 0.1 seconds to broadcast a single bucket and 125 seconds to broadcast the whole file(with no index). Let the clients be equipped with the Hobbit Chip (AT&T). The power consumption of the chip in *doze mode* is 50 μW and the consumption in *active mode* is 250 mW. For simplicity we will neglect other components which use energy during data filtering and assume that 250 mW constitutes the total energy consumption.
With access_opt method, access time is 625 buckets (half of *bcast* duration) i.e., 62.5 seconds. Tuning time is also 625 buckets i.e., the power consumption is 62.5 $sec*250$ $mW = 15.625$ *Joules*.
With tune_opt method, access time is 1303 (1250+53) buckets i.e., 130.3 seconds. Tuning time is minimum at 4 buckets i.e., the power consumption is 0.1 $sec * (4 * 250 + 1299 * 50 * 10^{-3})$ $mW = 0.106$ *Joules*.

If we want a low access time and also low power consumption then we can use (1,m) indexing or distributed indexing.

- $(1, m)$ *indexing*: Optimum m can be computed to be 5. The access time is 909 buckets i.e., 90.9 seconds. The tuning time is 5 buckets. Hence, the power consumption is 0.1 $sec * (5 * 250 + 904 * 50 * 10^{-3})$ mW i.e., 0.130 *Joules*.
  Therefore, in this case, the energy consumed *per query request* is 120 times smaller than that of access_opt. This is achieved by compromising on the access time which increases by 45.4%.
  Comparing with tune_opt, the power consumption is almost the same. However, the access time improves to 69.8% of the access time in tune_opt.
- *Distributed indexing*: The optimum number of replicated levels in distributed indexing is 2. The access time is 689 buckets i.e., 68.9 seconds. The tuning time is 6 buckets. Hence, the power consumption is 0.1 $sec * (6 * 250 + 683 * 50 * 10^{-3})$ mW i.e., 0.153 *Joules*.
  Therefore, in this case, the energy consumed *per query request* is 100 times smaller than that of access_opt. This is achieved by compromising on the access time which increases by just 10%.
  Comparing with tune_opt, the power consumption is almost the same. However, the access time improves to 52.9% of the access time in tune_opt.

5.2 How much energy do we save ?

The overall increase in the battery life due to indexing on air heavily depends on the overall structure of the set of applications that run on the palmtop. Assume that two AA batteries, each rated at around 1.0 W Hr, are used in the receiver, one being used for filtering and the second for other applications. One such battery is good for 4 hours of continuous data filtering (without indexing). Using distributed indexing, we can serve the same number of filtering requests with 100 times less energy. The saved energy can be used for extra queries or for other additional work. For example, in our case, four hours of data filtering will take approximately only 1% of the energy of one AA battery for the same number of requests. This savings can be used to almost double

[5] Quotrex system broadcasts stock quotes continuously over FM band

the overall working time for other applications. In case, data filtering was the only application being run, then using distributed indexing, we can serve 100 times as many requests.

6 Conclusion

We have explored data organization and access for digital broadcast data, taking a stand that periodic broadcasting is a form of storage. Data is *stored on the air* with the latency of access proportional to the duration of the *bcast*. Broadcasted data can be reorganized "on the fly" and refreshed (reflecting updates) between two successive *bcasts*. The main difference with the disk based files is that we need to minimize two parameters (access time and tuning time) contrary to just one (access time) for the disk based files. We investigated two data organization methods namely *(1,m) indexing* and *distributed indexing* and demonstrated the advantages of these methods, by comparing them with the optimal methods of broadcasting - for access time and tuning time. Optimum solution for access time has a large tuning time and the optimum solution for tuning time has large access time. A solution that has good performance, both in terms of tuning time and access time is required. We have attempted to solve this problem.

In [IVB94a] which is orthogonal to this paper, we explore data organizing methods based on hashing. We have demonstrated that contrary to the disk based files, perfect hashing does not provide the minimum access time. Distributed indexing is better than any Hashing scheme for small key sizes. In [IVB94b], algorithms for organizing and accessing data based on any clustering index is described. Algorithms for non-clustering indices and indexing based on multiple keys, for static files, is also described in [IVB94b].
There are a number of research problems which have to be investigated:

- If data records are of different size and requested with different frequencies then more sophisticated data organization techniques are needed.
- Sophisticated synchronization mechanisms are necessary when broadcasting multi-media information involving text, audio and video.
- There are a number of communication issues which have to be looked at in more detail. How to achieve reliability of the broadcast in the error prone environments such as wireless cellular? Since the clients are only listening there is no (or very limited) possibility of the acknowledgment. Multiple Access protocols which guarantee timely delivery of information are necessary for the broadcasting (especially the directory) to work correctly.
- In practice, a mixed technique, where only the most frequently accessed items are broadcasted and the rest of the items are provided on demand. Efficient techniques towards this end are necessary.

References

[Alo92] Rafael Alonso and Hank Korth, "Database issues in nomadic computing," MITL Technical Report, December 1992.

[Bow92] T. F. Bowen et. al., "The DATACYCLE Architecture," Comm. of the ACM, Vol 35, No. 12, December 1992, pp. 71 - 81.

[Cher92] David Cheriton, "Dissemination-Oriented Communication Systems," Stanford University, Tech. Rept. 1992.

[Giff85] David Gifford et. al., "The application of digital broadcast communication to large scale information systems," IEEE Journal on selected areas in communications, Vol 3, No. 3, May 1985, pp.457–467.

[Good91] David J. Goodman, "Trends in Cellular and Cordless Communications," IEEE Communications Magazine, June 1991.

[Her87] G. Herman et al., "The Datacycle architecture for very large high throughput database systems," in Proc ACM SIGMOD Conf., 1987, pp 97-103.

[IB93] T. Imielinski and B. R. Badrinath, "Wireless mobile computing: Challenges in data management," To appear in Comm. of the ACM.

[IVB94a] T. Imielinski, S. Viswanathan and B. R. Badrinath, "Power Efficient Filtering of Data on Air," Proc of 4 th Intl Conference on Extending Database Technology, Cambridge - March 94.

[IVB94b] T. Imielinski, S. Viswanathan and B. R. Badrinath, "Data on Air : Organization and Access," Submitted for publication.

[Ray93] Bob Ryan, "Communications get personal," BYTE, February 1993, pp. 169–176.

[She92] Samuel Sheng, Ananth Chandrasekaran, and R. W. Broderson, "A portable multimedia terminal for personal communications," IEEE Communications Magazine, December 1992, pp. 64–75.

[Terr92] D. B. Terry, D. Goldberg, D. A. Nichols, and B. M. Oki, "Continuous queries over append-only databases," Proc of the ACM SIGMOD, June 1992, pp. 321–330.