Data on Air: Organization and Access

T. Imielinski, S. Viswanathan, and B.R. Badrinath, *Member*, *IEEE*

Abstract—Organizing massive amount of data on wireless communication networks in order to provide fast and low power access to users equipped with palmtops, is a new challenge to the data management and telecommunication communities. Solutions must take under consideration the physical restrictions of low network bandwidth and limited battery life of palmtops. This paper proposes algorithms for multiplexing clustering and nonclustering indexes along with data on wireless networks. The power consumption and the latency for obtaining the required data are considered as the two basic performance criteria for all algorithms. First, this paper describes two algorithms namely, (1, *m*) *Indexing* and *Distributed Indexing*, for multiplexing data and its clustering index. Second, an algorithm called *Nonclustered Indexing* is described for allocating static data and its corresponding nonclustered index. Then, the *Nonclustered indexing* algorithm is generalized to the case of multiple indexes. Finally, the proposed algorithms are analytically demonstrated to lead to significant improvement of battery life while retaining a low latency.

Index Terms—Broadcasting, clustered index, indexing, latency, mobile computing, power conservation, wireless network.

1 Introduction

Wireless technology is rapidly gaining popularity. Many predict an emerging gigantic market where millions of mobile users will carry small, battery powered palmtops with wireless connections. Mobile users will be in constant need of stock information, traffic directions, local directory, weather information, etc. Wireless medium will be used as the "first mile" of the information highway that will disseminate massive amounts of information across the country. Organizing and accessing data on wireless communication channels[1] are new challenges to the database and telecommunication communities.

We show how the physical restrictions of wireless communication channels make the problem of organizing wireless broadcast data different from data organization on disks. We demonstrate that providing index based organization and access to data transmitted over wireless channels is very important from a power conservation point of view and can result in significant improvement in battery utilization. It turns out that wireless technology can utilize and build upon some well known techniques for file organization and access. However, these traditional techniques cannot be applied directly and need substantial modifications because of different physical limitations of wireless channels. New solutions require merging interdisciplinary expertise ranging from new communication protocols to file system design and database design.

We consider wireless data broadcasting as a way of disseminating information to a massive number of users equipped with battery powered palmtops. Fig. 1 shows the structure of a hypothetical system that provides mobile users with information services. The wireless information servers called Mobile Service Stations (*MSS*) will be equipped with wireless transmitters (denoted by tall antennas) capable of reaching thousands of users (denoted by small dark rectangles) residing in macrocells.[2] Users equipped with small palmtops with wireless connections will move freely between cells and query information provided by wireless information servers. Palmtops are not connected to any direct power source and run on small batteries (such as AA) and communicate on low bandwidth wireless channels. These physical restrictions call for power and bandwidth efficient solutions both at the hardware and the software levels [4].

We consider an asymmetric wireless infrastructure[3] where the downlink channel and the uplink channel are asymmetric in terms of bandwidth. The downlink channel is of much higher bandwidth than the uplink channel bandwidth. Each wireless cell will have a choice of the following two basic forms of information dissemination:

- *Broadcasting Mode*. Data is periodically broadcast on the downlink channel. Accessing broadcast 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" [3].
- *On-Demand Mode*. The client requests a piece of data on the uplink channel and the server responds by sending this data to the client on the downlink channel.

While the on-demand mode of interaction is the traditional client-server approach, the broadcasting mode is

1. Channel is an abstraction for a communication medium and it is used as a generic term to refer to a computer network, satellite channel, TV channel, radio channel or a virtual circuit over a cellular channel. The meaning is context dependent. Channel is used as a logical term to hide the physical details of the underlying network.

2. As opposed to the microcells that are expected to facilitate applications such as e-mail, macrocells of a bigger size will be used for wide area information services.

3. Each channel has a capacity to carry information which is called its *bandwidth*.
- *Downlink channel* refers to the bandwidth reserved for the information flow from the server to the clients (a client is a machine and a user is the person operating that machine.).
- *Uplink channel* refers to the bandwidth reserved for the information flow from the clients to the server.

- *T. Imielinski and B.R. Badrinath are with the Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08902.*
- *S. Viswanathan is with the Database Research Group, Bell Communications Research, Morristown, New Jersey. E-mail: vish@ares.sbi.com.*

Manuscript received May 11, 1994; revised Mar. 8, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96102.

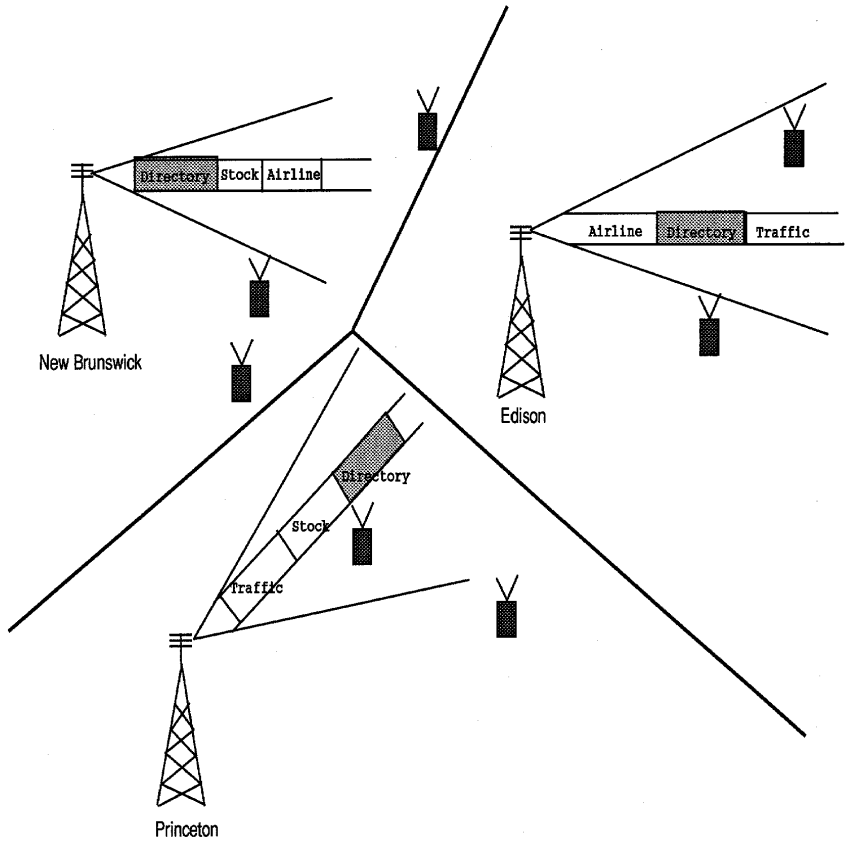


Fig. 1. Wireless information servers.

different [4]; here the server periodically broadcasts information (alternatively, the server just sends it on a specific channel, say a multicast address as discussed in [7]). This is analogous to the way information is disseminated via newspapers and regular TV broadcasting. There are many benefits of the broadcasting mode. Broadcasting the most frequently accessed data items saves bandwidth, as it cuts down the number of separate but identical responses to requests. It also saves power at the client end by avoiding power consuming uplink transmissions. Providing a directory along with data on a wireless channel helps clients to selectively tune only to relevant information. This strategy saves considerable amount of power as demonstrated in [8], [9].

In practice, a mixture of the above two modes will be used. The most frequently accessed pieces of data, the so called *hot spots* (stock quotes, airline schedules, etc.) will be broadcast. Since the cost of broadcasting does not depend on the number of users, this method scales with the number of users. For example, if stock information is broadcast every minute, then it doesn't matter whether 10 users or 10,000 users are listening, the average waiting time will be 30 sec. This will not be the case if stock information is provided on demand. The on-demand mode will be used for the less often requested data. Broadcasting these periodically would be a waste of bandwidth. However even in the 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. An optimal method to decide which data items to broadcast and which ones to provide on-demand is described in [7]. Furthermore, it proposes an algorithm for optimal allocation of the network bandwidth between the broadcast data and the on-demand data. In addition, [7] addresses policies for interleaving on-demand data with broadcast data. Periodic data broadcasting is the topic of this paper.

The (downlink) channel is divided into two (logical) parts: the *broadcast channel* (the bandwidth allocated for broadcast data) and the *on-demand channel* (the bandwidth allocated for data provided on demand).

In applications which involve dissemination of information to a large number of users, the communication network can be treated as a *storage* medium, where frequently accessed data is periodically transmitted [3], [6], [8], [9]. Our proposal can be treated as data organization and access methods for a new, rather nontraditional memory medium —the telecommunication network. In fact, we believe that in the near future the distinction between communication network and remote storage will gradually disappear and users will treat the network as one gigantic database. Thus, wireless data broadcasting can be viewed as *storage on the air*—an extension of the server's memory. The latency for accessing data in such a storage on the average is equal to half the time necessary to broadcast the whole data. Thus, in case of a low bandwidth channel, the latency can be quite substantial. On the other hand, the latency is independent of the number of users who are listening. Hence, this "public" storage scales well with an increase in the number of users.

1.1 Motivation

The lifetime of a battery is expected to increase only 20% over the next 10 years [11]. A typical AA cell is rated to give 800 mA/hr at 1.2 V (0.96 W/hr). The constraint of limited available power is expected to drive all solutions to mobile computing on palmtops. Assume that the power source of a palmtop with a CD-ROM and a display, to be 10 AA cells. The constant power dissipation in a CD-ROM (for disk spinning itself) will be about 1 W and the power dissipation for display will be around 2.5 W. Thus, the assumed power source will last for 2.7 hr. To increase the longevity of the batteries, the CD-ROM and the display may have to be powered off most of the time.

Apart from CD-ROM and display, the CPU and the memory of palmtops also consume power. There is a growing pressure on hardware vendors to come up with power efficient processors and memories. The Hobbit chip from AT&T is one such processor that operates in two modes: the full operational mode called the *active mode*, and the power conserving mode called the *doze mode*. The power consumption in the *active mode* is 250 mW and the power consumption in the *doze mode* is 50 μW. The ratio of power consumption in the active mode to the doze mode is 5,000. When the palmtop is listening to the channel, the CPU must be in the active mode for examining data buckets (determining whether they match the predefined data). The CPU consumes more power than some receivers, especially if it has to be active to examine all incoming buckets. This is true, if on the average, only a few data buckets are of interest to the clients. Therefore, it will be beneficial if the CPU can slip into the *doze mode* most of the time and come into the *active mode* only when the data of interest arrives on the broadcast channel. This requires the ability of selective *tuning*.

Transmitting and receiving consumes power as well. A number of factors like the terrain, landscape, the height and kind of trees, foliage, season, rain, etc., play an important role in determining the power required by a client in transmitting to a server. This power grows as the fourth power of the distance between the client and the server [15]. As the distance increases, the power required becomes very high. For large cells the energy required for transmission could reach tens of watts. This constitutes a major drain of the power at the client. For example, a Wavelan card consumes 1.7 W with the receiver "on" and 3.4 W with the transmitter "on." The ability to selectively switch off the receiver and avoid transmitting as much as possible will be very important to conserve battery power.[4]

Later in the paper, we discuss in detail the ability of selective tuning and provide an example of realistic power savings due to selective tuning.

Power efficient solutions are important because:

- Power efficient solutions make it possible to use smaller and less powerful batteries to run the same set of applications. Smaller batteries are important from the portability point of view since palmtops can be more compact and weigh less. This is especially true in palmtops, since batteries constitute the largest source of weight.
- With the same batteries, a client can run for a very long time without the problem of changing the batteries often. This avoids frequent recharging and results in substantial monetary savings. Recharging can be cumbersome especially if the client is on the move. With power efficient solutions, batteries may have to be recharged only every few days rather than every few hours. It also avoids the frequent "memory effect"[5] problem that is prevalent in most rechargeable batteries (especially the Nickel Cadmium batteries).
- Every improperly disposed battery is an environmental hazard.

The effective bandwidth of wireless networks is only a fraction of the bandwidth that is available in wired networks. The current ATM (Asynchronous Transfer Mode) standards are designed to yield a bandwidth of up to 622 Mbps. This bandwidth is projected to go up to gigabits [10]. Even the widely used Ethernet has a bandwidth of 10 Mbps. Compared to the wired bandwidth, the bandwidth of a wireless channel is very limited. It varies from 1.2 Kbps for slow paging channels, through 19.2 Kbps (characteristic of cellular proposals like Cellular Digital Packet Data (CDPD)) to about 2 Mbps of the wireless LAN. Therefore, bandwidth utilization is of vital importance in a wireless network.

The focus of this work is on the wireless communication medium, although most of the presented work can also be applied to wired networks. Our work is important for wireless networks because the wireless bandwidth will continue to remain a major bottleneck [1].

Mobile clients are typically lightweight portable units that are restricted in their power source and these clients use wireless channels that are restricted in their bandwidth. Hence, in a wireless environment, utilizing the bandwidth efficiently and saving power consumed by mobile clients are of paramount importance.

1.2 Related Work

Broadcasting over a fast wired network has been investigated as an information dissemination mechanism in the past. In the *Datacycle* project at Bellcore [3], [6], a 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 2,000,000 predicates a second.

The main differences between wireless broadcasting considered in this paper and broadcasting considered in the Datacycle architecture are as follows :

- Power conservation is of no concern in the Datacycle architecture, while it is a major physical requirement for wireless broadcasting.

4. Cordless phones listen to the base stations only for a small fraction of time and they keep their receivers switched off most of the time. A slight delay in receiving a call is not noticeable by humans. This idea of switching off the receiver most of the time increases the battery life of cordless phones from hours to a week. Similar ideas can be used for palmtops and data transmission in general.

5. The memory effect occurs when certain batteries are charged without being fully discharged first. This results in the batteries not being able to be recharged fully ever again.

- The wireless bandwidth is much lower than the bandwidth assumed in the Datacycle architecture. Hence, bandwidth utilization is a major concern in our architecture.

Gifford in [5] describes a system called Boston Community Information System where newspapers are broadcast over a FM channel (using FM Subsidiary Communications Authority channel) and downloaded by PCs equipped with radio receivers. There is a single communication channel and power conservation plays little role since the PCs (equipped with wireless modems) are connected to a continuous power supply.

A comparative study of broadcasting techniques and on-demand scheduling algorithms can be found in [16]. This paper also provides a comprehensive study that addresses the concept of broadcasting some data and providing the rest on-demand. In this work, the emphasis has again been only on the efficient use of network bandwidth, while power consumption at the clients has been of no concern.

Different organizing techniques for broadcasting and accessing data, based on a primary key are provided in [8], [9]. In [8], index based schemes are analyzed with special emphasis on minimizing the latency while keeping the power consumption at (almost) the minimum. In [9], hashing based algorithms are proposed with special emphasis on flexibility in terms of the tradeoff between the tuning time and the latency. In this paper, organizing techniques for broadcast data have been generalized to access data based on secondary indexes and indexes based on multiple attributes. Recent papers [7], [12] provide broadcast strategies for data items with different access frequencies.

In Section 2, we discuss data organization basics for broadcasting. In Section 3, two techniques are described for organizing and accessing data based on any clustered index. Section 3.1 describes (1, *m*) *indexing* and Section 3.2 discusses *distributed indexing*. In Section 4, an algorithm called *nonclustered indexing* algorithm, for organizing and accessing static files based on a nonclustering index is described. Section 5 presents a generalization of the nonclustered indexing algorithm to provide multiple indexes on the broadcast channel. Section 6 will interpret our results in terms of the improvement in power consumption and latency, with an example of an application broadcast over cellular data link. The practical impact of a number of limitations like unreliability, nonzero setup time, dynamic organization, etc., of wireless channels are addressed in Section 7. We present the conclusions in Section 8.

2 Data Organization for Broadcasting

Consider a file consisting of a number of records. Each record is a tuple of attribute values. Attribute values are used as search keys to identify a single record or a set of records.

The server broadcasts this file periodically on the broadcast channel. Clients receive the broadcast data and filter the required records (identified by the attribute value) from the broadcast channel by simple pattern matching. In this paper, we deal with general purpose information services and not private files (databases). Writes to a file are sent to the server and the server updates the file. Updates to the file are completed only between successive broadcasts of the file. Hence, the content of a version of the broadcast is completely determined before the start of broadcast of that version and remains unchanged throughout that broadcast.

Let us first justify the use of a directory for broadcast data. If data is broadcast without any form of directory, then the client will have to be tuned to the channel continuously until all the required records are downloaded. On the average, the client has to be tuned to the channel for half the duration of the broadcast (of the current version of the file). This is unacceptable as it requires the client to be in the *active mode* for a long time, thereby consuming scarce battery resource. We would rather provide a selective tuning ability, enabling the client to come into the active mode only when data of interest is being broadcast. Selective tuning will require that the server in addition to broadcasting the data, also broadcasts a directory that indicates the point of time when particular records are broadcast on the broadcast channel. Clients will remain in the *doze mode* most of the time and tune in periodically to the broadcast channel.

One idea is to let all 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 broadcast 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 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 its 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 broadcast data is very frequently accessed and thus it is broadcast, the same argument holds even more strongly for the directory. Therefore, the directory is broadcast as well.
- If many different files are broadcast on different channels, then clients need excessive storage for the directories of all the files being broadcast and in palmtops storage is a scarce resource.

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 multilevel index.

The smallest logical unit of a broadcast will be called a **bucket**. Each bucket is a unit of information that is sent on the broadcast channel (a bucket is made up of a fixed number of *packets*—the basic unit of message transfer in networks). 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.

We distinguish between **index buckets** which hold the index and **data buckets** which hold the data. The **index segment** refers to a set of contiguous index buckets; the

data segment refers to a set of data buckets broadcast between successive index segments.

Each version of the file (all data segments) interleaved with the index information (all index segments) will constitute a *bcast*. Thus, each *bcast* is made up of a number of *buckets*, some data buckets and some index buckets. Each *bcast* is periodically broadcast on the wireless channel.

In order to make all buckets self-identifying, each bucket has the following header information:

- *bucket_id:* the offset of the bucket from the beginning of the *bcast*
- *bcast_pointer:* the offset to the beginning of the next bcast
- *index_pointer:* the offset to the beginning of the next index segment
- *bucket_type:* data bucket or index bucket

Pointer to a specific bucket *P* within a *bcast* can be provided by specifying the *offset* of bucket *P*. The *offset* of bucket *P* from the current bucket is computed as the difference between the *bucket_id* of *P* and the *bucket_id* of the current bucket. The actual time of broadcast for bucket *P* from the current bucket is the product of (*offset* – 1) and the time necessary to broadcast a single bucket.

An index bucket is arranged as a sequence of *(attribute_value, offset)*, where *offset* is a pointer to the bucket that contains the record identified by *attribute_value*. The multilevel index tree will provide a sequence of pointers which eventually lead to the required records. A data bucket is arranged as a sequence of data records.

Organizing the file on the broadcast channel involves determining how the data buckets and index buckets will be interleaved to constitute a *bcast*. There are a number of ways of interleaving data buckets and index buckets. We explore different techniques of interleaving data along with its corresponding clustering index, nonclustering index and multiple indexes. 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 indexes but rather new index allocation methods that multiplex index and data in an efficient way with respect to conserving the power of clients and utilizing the wireless bandwidth efficiently. Our methods allocate index and data for any type of index. The broadcast channel is the source of both data and 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 broadcast on a single channel encompasses the problem of having different channels and optimizing data broadcast in each of them.[6]

The general access protocol for retrieving data involves the following steps.

- The initial probe, in which the client tunes into the broadcast channel and determines when the next index segment will be broadcast.
- Then, a sequence of pointers (in the index segment) is accessed to find out when to tune into the broadcast channel to get the required data.
- Finally, the client tunes to the channel when buckets containing the required data arrive, and downloads all the required records.

If a client moves into another cell before completing an access protocol, then it starts the access protocol from scratch in the new cell. This is necessary because the index information obtained in one cell may not correspond to the index information of the broadcast data in the new cell. Since each cell's channel will be completely self-explanatory (i.e., each bucket will carry enough control information) the client will be able to "get in sync" just by tuning into the bit stream in the new cell.

Notice that although the discussed model deals only with one server, multiple servers which broadcasts their individual data streams can be accommodated easily. Indeed, each such data stream can be identified and this identification will be a part of the description of each bucket of such a stream. The directory information will include the stream (service) identification along with the identification of individual buckets.

Before we proceed further, let us present an overview of the communication issues underlying the data organization schemes presented in this paper.

2.1 Overview of Communication Issues

In a broadcasting environment a number of practical communication issues have to be addressed. In this section we touch upon these issues and discuss them in more detail later in the paper.

- **Self-explanatory Channels**. Mobile clients have to be able to interpret the incoming bit stream in each cell at any time, since they could be reconnecting after a long disconnection period or they could be moving into a new cell.[7] Thus, the communication channel has to be self-explanatory, by having each bucket carry sufficient information about the relative position of this bucket is the *bcast*. This is the reason why each bucket carries a pointer to the next index bucket. Alternatively, the client upon reconnecting to the MSS (after a disconnection or a move) could receive a *greeting message*[8] with a pointer to the index information. However, this requires uplink messages from the client after each reconnection, and hence the first solution of a self-explanatory channel is preferable. In this way, the client can interoperate in different cells.
- **Setup Time**. The setup process is informally defined as the process of tuning out of the broadcast channel or tuning back in. The setup time refers to the time taken by the CPU for changing from the doze mode to the active mode. It can also refer to the time necessary

6. This is because each physical channel can be time multiplexed into a set of logical channels, such that there is a one-to-one mapping from the physical channel to the set of logical channels and vice versa [14].

7. Periodically, a **beacon** signal (with a particular id) is broadcast by the MSS in its cell. Each MSS has a unique *beacon* id. All the clients in a cell keep polling the beacon signal periodically. On noting that two consecutive beacons have different ids, a client knows that it has entered a new cell.

8. When a mobile client enters a new cell or it reconnects after a period of disconnection, a set of messages called *greeting messages* are exchanged between the mobile client and the MSS.

to switch the receiver on. In this paper we assume that the setup time for tuning into the broadcast channel and going into the doze mode is negligible. In general, each tuning out (dozing off) and tuning in (waking up) operation will not occur instantaneously, but will take some small amount of time. In cases where the setup time cannot be ignored, our solutions have to be extended—this issue is discussed in detail in Section 7.1.

- **Reliability Issues**. The error rates in wireless networks are much higher than the error rates in wired networks. Thus, maintaining the reliability of wireless broadcast is an important issue. In the following, we first assume that no additional steps are taken to increase the reliability of the broadcast. Note that broadcasting is eventually reliable due to its periodic nature—a client can always wait for the next *bcast*. We discuss ways to enhance the reliability of the broadcast in Section 7.2.
- **Synchronization**. Since the addressing of buckets in a *bcast* is temporal, in order for the client to "wake up" at the right time, we need the channel to be *synchronized*. This is a reasonable assumption for exclusively used channels such as Motorola's Satellite Networks where information is guaranteed to come at a specific time, since there is only one server and the clients only listen. Synchronization is not difficult, especially with the current accurate quartz technology. In order to take care of the small discrepancies in the clock synchronization, the clients may tune in, *epsilon* (buckets) *ahead* of time (the required bucket is expected to arrive on the broadcast channel). That is, the tuning begins epsilon *prior* to the expected timestamp of broadcast and continues until a bucket received by the client has a higher id than that required by the client. Epsilon varies from client to client depending on the accuracy of its clock and the time after which it is tuning into the broadcast channel (the smaller the time period, the smaller the epsilon).

  Synchronization can, in general, be easily achieved if the MSS has higher priority than any client. This is not the case in Ethernet where the broadcasting traffic has to be interleaved with on-demand requests and it is very difficult to adequately predict the time of transmission. For such channels different techniques are required. One solution is to use multicast addresses, as described in [7]. Each data item is multicast on a different multicast address and clients join only multicast groups of data items they are interested in. In [7], we discuss how to use multicast addresses to achieve addressing which does not rely on any synchronization.

  Whenever a client comes into the active mode (soon after power-on or from the doze mode) or it moves into a new cell, it reads a stream of ones and zeros. Resynchronization will be necessary in order to parse the data stream into proper buckets (i.e., a mechanism should be present for identifying buckets). The following two assumptions are made in order to facilitate resynchronization:

  - The bit stream is divided into packets. The packets are consecutively numbered and have the usual error-detecting codes and synchronization bits built in [15].
  - A bucket exactly fits into one packet.

  Note that the second assumption is made just to simplify explanations. In reality, each bucket could occupy any number of packets. Each bucket will have synchronization bits of its own. These synchronization bits could be different from that of the synchronization bits of the packets. In case a bucket fits exactly into a packet, there is no need to have separate synchronization bits for packets and buckets. The synchronization bits (such as Barker sequence [15]) of packets can serve as bucket demarcators in addition to serving as packet demarcator.

2.2 Parameters of Concern

For a file being broadcast on a channel, the following two parameters are of concern:

- **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.
- **Latency**. The time elapsed (on the average) from the time a client requests data[9] to the point when all the required data is downloaded by the client.

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 the doze mode).

The *latency* for a broadcast is determined by the following two factors:

- **Probe Wait**. When an initial probe is made into the broadcast channel, the client gets a pointer to the next index segment. The average duration for getting to the next index segment is called the *probe wait*. This wait is equal to half the distance between two consecutive index segments.
- **Bcast Wait**. The average duration between the point the index segment is encountered and the point when all the required records are downloaded is called the *bcast wait*. *Bcast wait* consists of waiting for the first occurrence of a record with the required attribute value (on the average, this is equal to half the total length of the *bcast*) plus the time to download all the required records.

The *latency* is the sum of the *probe wait and* the *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 the *bcast wait*, we can broadcast the index once (at the beginning of each *bcast*). In this case, the *probe wait* will be large, since the client will always have to wait for the index (until the starting of the next *bcast*) miss-

9. Note that in this case, "requesting" for data does not mean an uplink request to the server. It only means that the user has specified that he/she is interested in some data and the palmtop (client) takes up the task of getting that data. The moment of requesting data refers to the point at which the client takes up the task of getting the data.

ing the required data in the current *bcast*. On the other extreme, for minimizing the *probe wait*, the index can precede each data bucket in the *bcast*. This would minimize the *probe wait*, but would increase the *bcast wait* (due to an increase in *bcast* length.)

Both the latency and the tuning time will be measured in terms of number of buckets.

Both the access time in disks and the broadcast tuning time, are affected by the presence of an index. If data is indexed then the access time in disks reduces because less number of disk accesses have to be made. This is also the case with the tuning time of broadcast files, since the amount of time that a client has to listen to a broadcast channel reduces if there is an index that is broadcast along with the data. In this sense, the broadcast tuning time roughly corresponds to the access time for disk based files. There is no parameter in disks that directly corresponds to the latency of broadcast data.

In periodic wireless broadcasting, air behaves like a *storage medium* requiring new data organization and access methods. The main difference between the organization of broadcast data (data on air) versus data on disk can be summarized as follows:

- *Data on Air is characterized by two parameters: the latency and the tuning time, contrary to the data on disks being characterized by just one parameter: the access time.*

In the next section, we develop the basic index allocation techniques. We first discuss the case of clustering indexes. We follow it up with the general case of nonclustering indexes and multiple indexing.

3 Clustering Index

A file is *clustered* on an attribute if all records with the same value of this attribute appear consecutively in the file. An index defined on the clustered attribute is called a *clustering index*. The *coarseness* '*C* ' of an attribute[10] is defined as the average number of buckets containing records with the same attribute value.

In general, data organization algorithms that seek optimum in two dimensional space of the latency and the tuning time are of importance. Below, we present two algorithms that are optimal in the one dimensional space of the latency and the tuning time, for a clustered index. These will serve as benchmarks for comparisons with our algorithms.

Latency_opt. This algorithm provides the best (lowest) latency with a very large tuning time. The best latency is obtained when no index is broadcast along with the file. The size of the entire *bcast* 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 the average it takes $(\frac{Data}{2})$ time to get to the first record with the required attribute value. After this, it takes a duration of *C*, to download all the required records. Thus, the latency for *latency_opt* algorithm is $(\frac{Data}{2} + C)$. The average tuning time is the worst and is equal to $(\frac{Data}{2} + C)$. This is because the client has to be in the *active mode* throughout the period of access. This method is illustrated in Fig. 2.

Tune_opt. This algorithm provides the best tuning time with a large latency. The server broadcasts the index at the beginning of each *bcast*. A client which needs all records with attribute value *K* tunes into the broadcast channel at the beginning of the next *bcast* to get the index.[11] It then follows the index pointers to the first record with the required attribute value. Then in order to download the required records, the client on the average tunes to *C* consecutive buckets. The tuning time is $(k + C)$, where *k* is the number of levels in the multileveled index tree. This method has the worst latency because clients have to wait until the beginning of the next *bcast* even if the required data is just in front of them. With *Index* denoting the size of index of the file, the *probe wait* is $(\frac{Data+Index}{2})$, the *bcast wait* is $(\frac{Data+Index}{2} + C)$ and the latency is $(Data + Index + C)$. This method is also illustrated in Fig. 2.

The proposed indexing schemes are not aimed at getting the required data item faster than constant listening; in fact no technique can achieve this. Listening constantly results in the minimum latency (as explained for *latency_opt*). If an index is provided to conserve power (improve the tuning time) then the latency shoots up. The proposed methods aim at reducing this increase in the latency.

Usually, both the tuning time and the latency 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 this section, we develop a method for efficient (in terms of the latency and the tuning time) multiplexing of a data file with its clustering index. This index allocation method, called *distributed indexing*, is generalized in the next section to nonclustering indexes as well. We start from a simple index allocation method called (1, *m*) *indexing*.

3.1 (1, *m*) Indexing

(1, *m*) *indexing* is an index allocation method in which the index is broadcast *m* times during the broadcast of one version of the file. The whole index is broadcast preceding every fraction $(\frac{1}{m})$ of the file. Fig. 3 illustrates this algorithm.

The first bucket of each index segment has a tuple whose first field is the attribute value of the record that was broadcast last and the second field is the offset to the beginning of the next *bcast*. This tuple guides clients who missed the required record in the current *bcast* and have to tune to the next *bcast*.

The access protocol for records with attribute value *K* is as follows:

- Tune into the current bucket on the broadcast channel.
- Get the pointer to the next index segment.
- Go into the *doze mode* and tune in at the broadcast of the index segment.

10. Note that coarseness is the inverse of the well known quantity in databases: *selectivity*.

11. 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.

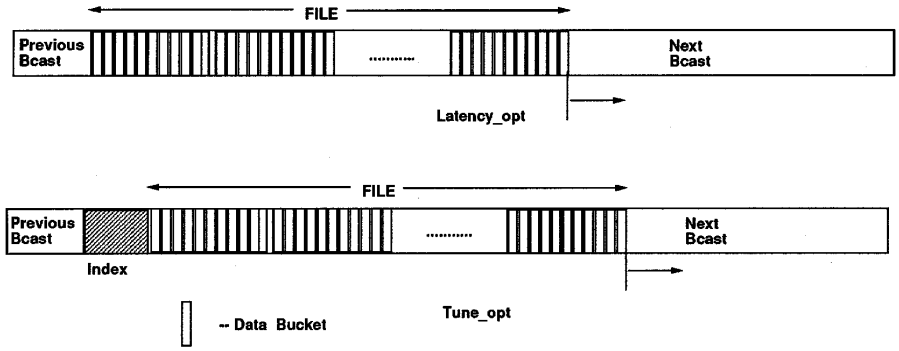


Fig. 2 . One dimensional optimal methods.

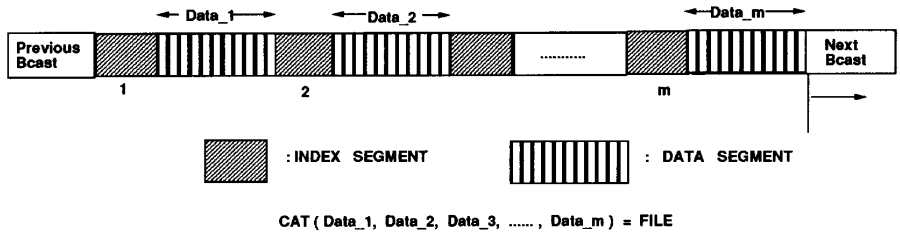


Fig. 3. (1, *m*) Indexing.

- From the index segment, determine when the data bucket containing the first record with attribute value *K* will be broadcast. This is accomplished by successive probes, by following the pointers in the multilevel index (the client might go into the doze mode between two successive probes).
- Tune in again when the bucket containing the first record with attribute value *K* is broadcast and download all the records with attribute value *K* (keep downloading records until a record with a value different than *K* is encountered for the attribute).

3.1.1 Analysis

In the following analysis, the probability distribution of the initial probe of clients is assumed to be uniform within a *bcast*. Let *Data* denote the average size of the file (note that the file could change in size between two successive *bcasts*). The coarseness of the (index) attribute is denoted by *C*. In order to avoid unnecessary repetitions of *(attribute_value, offset)*s in the index buckets, the index can have pointers only to the first occurrence of a record with the attribute value. Thus, the index tree can be constructed on $(\frac{Data}{C})$ data buckets. Let *n* denote the capacity of a bucket, i.e., the number of *(attribute_value, offset)*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:

$$k = \left\lceil \log_n \left(\frac{Data}{C} \right) \right\rceil$$

$$Index = \sum_{i=0}^{k-1} n^i$$

Latency. The *probe wait* is $\frac{1}{2} * (Index + \frac{Data}{m})$ and the *bcast wait* is $\frac{1}{2} * ((m * Index) + Data) + C$. Hence, the latency is

$$\frac{1}{2} * \left(Index + \frac{Data}{m} \right) + \frac{1}{2} * ((m * Index) + Data) + C,$$

i.e.,

$$\frac{1}{2} * \left((m+1) * Index + \left(\frac{1}{m} + 1 \right) * Data \right) + C.$$

Tuning Time. The first probe is the initial probe that gets a pointer to the next index bucket. Then, *k* probes are required for following the pointers in the index.[12] *C* more probes are required for tuning in for getting the required records. Thus, the tuning time is

$$1 + k + C$$

Optimum m. Now, we present a formula to compute the optimal *m* so as to minimize the latency for the (1, *m*) indexing. For finding the minimal latency, we differentiate the above formula (for latency) with respect to *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) and broadcast each data segment preceded by the index, on the broadcast channel.

3.2 Distributed Indexing

We can improve upon (1, *m*) indexing by cutting down on the replication of an index. Distributed indexing is a tech-

12. In case the required record is missed, which occurs with a probability of 0.5, then another probe is needed to 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.

nique in which an 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 that 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 the *relevant* index is interleaved as opposed to interleaving the whole index as in (1, *m*) indexing. Fig. 4 shows the full replication of the index in front of data segments as in (1, *m*) indexing. It also shows the replication of just the relevant index in front of the data segments as in the methods described below.

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

3.2.1 Motivation

We will proceed by describing different variants of index distribution for the file shown in Fig. 5. This figure shows a data file consisting of 81 data buckets. Each square box in the bottommost level represents 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 consider three different index distribution algorithms with the last one being distributed indexing. In all the three algorithms, the index is interleaved with data and the index segment describes only data in the data segment

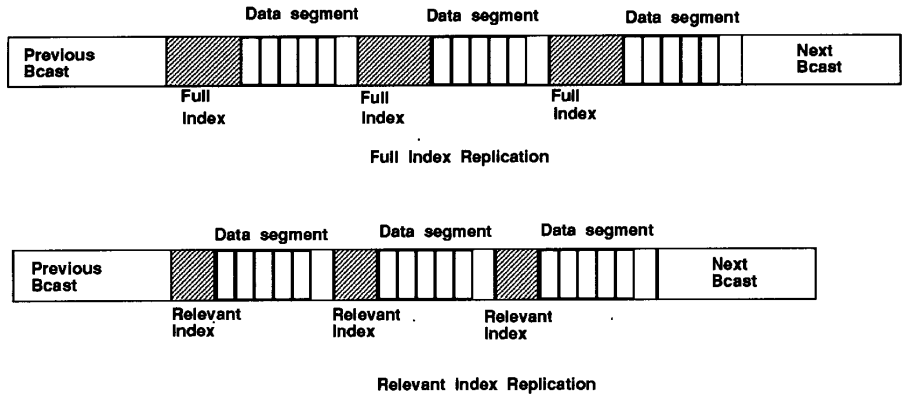


Fig. 4. Index distribution.

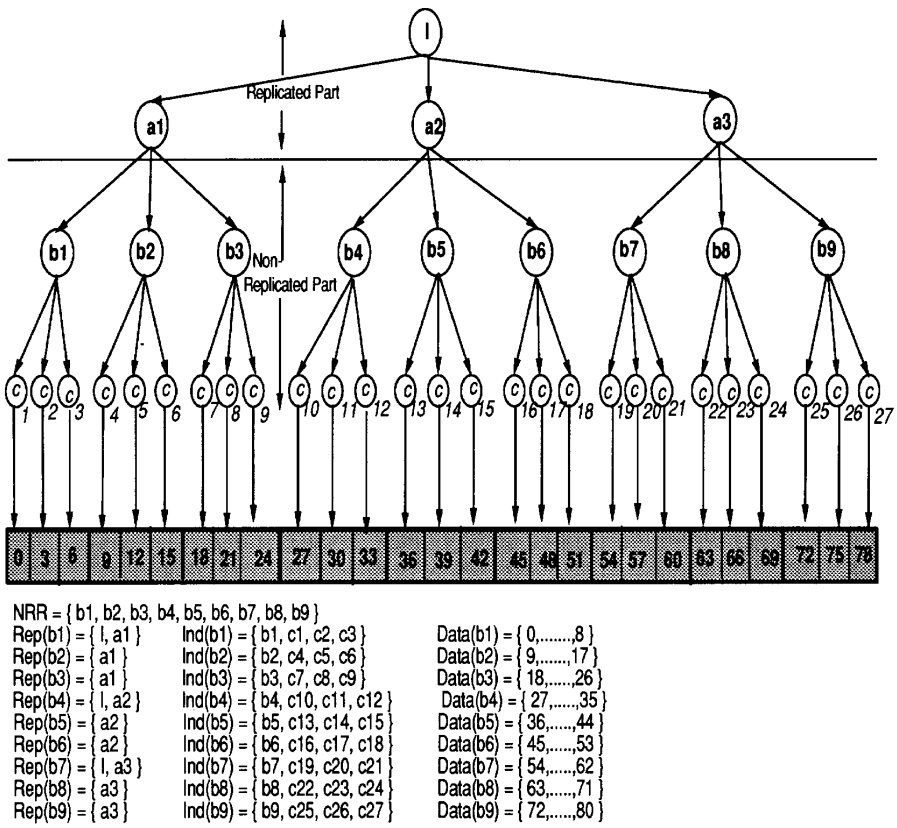


Fig. 5. File in the running example.

that *immediately* follows it. The algorithms differ in the degree of the replication of index information:

- *Nonreplicated Distribution*. Different index segments are disjoint. Hence, there is no replication.
- *Entire Path Replication*. The path from the root 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 nonreplicated 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 is accessed in each of them. Figs. 6, 7, and 8 illustrate the three schemes. In these figures, the current *bcast* is represented in three levels for ease 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. In the running example to describe the three schemes, consider a client that requires a record in bucket 66 and makes the initial probe at data bucket 3.

Nonreplicated Distribution. Fig. 6 illustrates the layout for this scheme. The *bcast_pointer* at bucket 3 will direct the client to the beginning of the next *bcast*. The client will make the following successive probes (in the next *bcast*) *I*, *a*3, *b*8, *c*23, and bucket 66. Since the root of index is broadcast 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, the client has to get to the root of the index. Hence, the *probe wait* for this scheme will be quite significant and will offset savings in the *bcast wait* due to the lack of replication.[13]

Entire Path Replication. Fig. 7 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 b_i. 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, *b*8, *c*23, and bucket 66. The latency 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. Fig. 8 shows that we can further improve on the latency 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, *b*8, *c*23, 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 *b*2, *c*4, 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 buckets 9 through 26). The additional space that is necessary to store the control index, is a small overhead compared to the savings resulting from partial index path replication. Replicated buckets can be modified by removing the entries for records that have been broadcast 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.

Fig. 9 shows the control index for index buckets that are part of the index tree described in Fig. 5 and whose layout is shown in Fig. 8. The first part of each control index element denotes the search 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.

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

3.2.2 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:

1) The replicated part
2) The nonreplicated part

The replicated part constitutes the top r levels of the index tree, while the nonreplicated part consists of the bottom $(k - r)$ levels. The index buckets of the $(r + 1)$th level are called *nonreplicated 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 nonreplicated root in *NRR* will appear only once in the whole *bcast* just in front of the set of data segments it indexes. Hence, each descendant node of a nonreplicated root of the index will appear *only once* in a given version of a *bcast*. On the other hand, the number of replications of each node of the index tree that

13. The *offset* at data bucket 3 could have directed the client to index bucket *b*2. In which case, the client could make the following successive probes: *b*2, *b*3, *a*2, *a*3, *b*8, *c*23, and bucket 66. The client need not have made an extra probe at *b*3. 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.

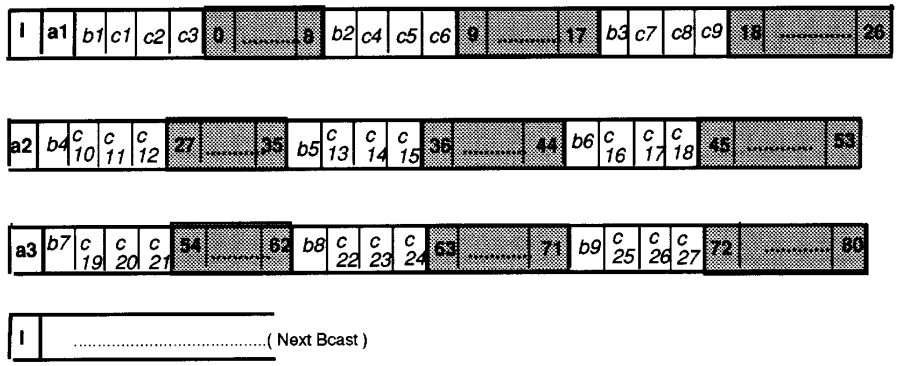


Fig. 6. Nonreplicated distribution.

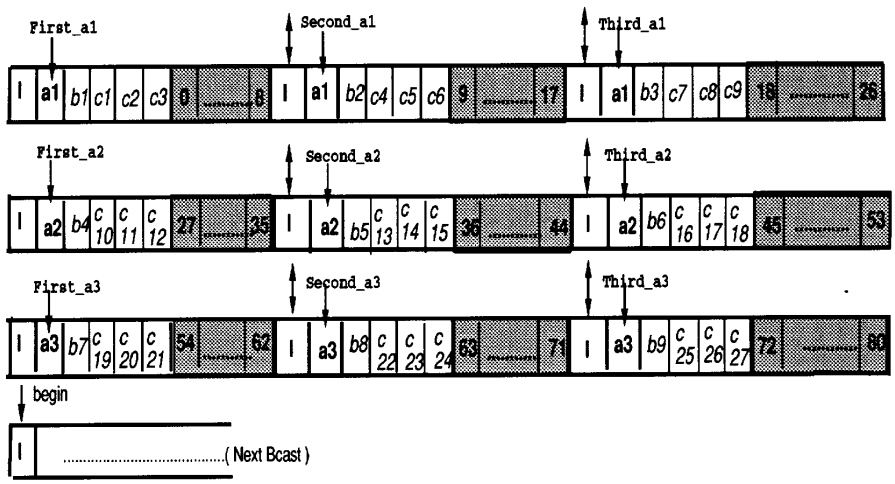


Fig. 7. Entire path replication.

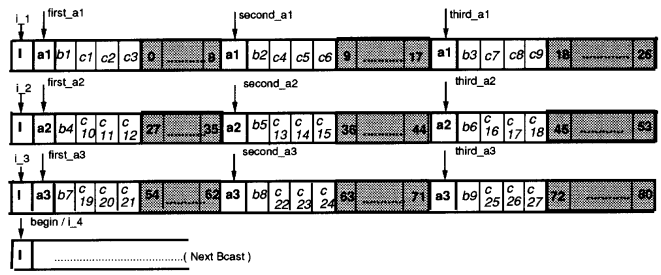


Fig. 8. Distributed Indexing.

appears above a nonreplicated root, is *equal to the number of children* it has

EFINITIONS.

I denotes the root of the index tree.
B denotes an index bucket belonging to NRR.
B_i denotes the ith index bucket in NRR.
Path(C, B) denotes the sequence of buckets along the path from index bucket C to B (excluding B).
Data(B) denotes the set of data buckets indexed by B.
Ind(B) denotes the part of the index tree below B (including B).
LCA(B_i, B_k) denotes 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*.

$$Rep(B_i) = Path(LCA(B_{i-1}, B_i), B_i) \text{ for } i = 2, \ldots, t.$$

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 *nonreplicated* portion of the index.

Fig. 5 shows the values of *Rep*, *Ind*, and *Data* for each of the index buckets in *NRR*. Each version of the broadcast will be a sequence of triples:

$$< Rep(B),\ Ind(B),\ Data(B) > \quad \forall B \in NRR,$$

in left to right order.

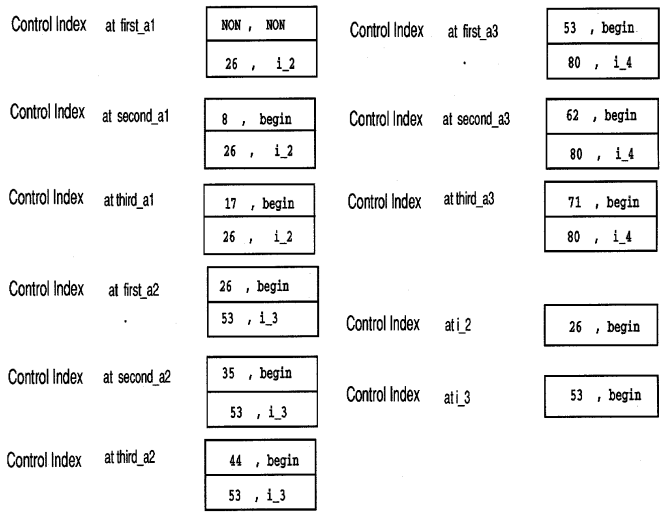


Fig. 9. Control Index.

Let P_1, P_2, …, 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 attribute in the last record that is indexed by bucket P_i.

Let $NEXT_B(i)$ denote the offset to the next 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 attribute in the last record broadcast prior to *B* and let *begin* be the offset to the beginning of the next *bcast*. Control index in P_i, that 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), $NEXT_B(i-)$]

The control index in bucket P_i is used as follows: Let *K* be the value of the attribute of the required records. If $K < l$, then the search is directed to 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 $j \le i$, then $NEXT_B(j-1)$ 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 attribute value *K* is as follows:

1) Tune to the current bucket of the *bcast*. Get the pointer to the next control index.
2) Tune again to the beginning of the designated bucket with control index. Determine, on the basis of the value of the attribute value *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 step 3).
   b) Tune in again for the appropriate higher level index bucket, i.e., follow one of the "NEXT" pointers and proceed as in step 3).
3) Probe the designated index bucket and follow a sequence of pointers (the client might go into the doze mode between two successive probes) to determine when the data bucket containing the first record with *K* as the value of the attribute is going to be broadcast.
4) Tune in again when the bucket containing the first record with *K* as the value of the attribute is broadcast and download all the records with *K* as the value of the attribute.

3.2.3 Analysis

Let *Index* denote the number of buckets in the index tree. Let *Level*[*r*] be the number of nodes on the *r*th level of the index tree and let *Index*[*r*] be the size of the top *r* levels of the index tree. Let $\Delta Index_r$ denote the additional index overhead due to the replication of the top *r* levels of the index tree. Finally, let *B* denote a nonreplicated root.

Latency. The latency is a sum of the *probe wait* and the *bcast wait*. The detailed description of how the *probe wait*, the *bcast wait*, and $\Delta Index_r$ are computed is described in [8]. In this section we just give the final formulae.

$$\Delta Index = Level[r+1] - 1$$

$$probe\ \ wait = \frac{1}{2} * \left(\frac{Index - Index[r]}{Level[r+1]} + \frac{Data}{Level[r+1]} \right)$$

$$bcast\ \ wait = \frac{1}{2} * \left(Data + Index + \Delta Index_r \right) + C$$

$$latency = \frac{1}{2} * \left(\frac{Index - Index[r]}{Level[r+1]} + \frac{Data}{Level[r+1]} + Data + Index + \Delta Index_r \right) + C \qquad (*)$$

Tuning Time. Tuning time primarily depends on the number of levels of the index tree and the coarseness *C* of the given attribute. The initial probe of a client is for determining the occurrence of control index. The second probe is for the first access to control index. The client is directed to the required higher level index bucket by the control index. Next, the number of probes by the client is equal to at most *k*, the number of levels in the index tree. The last pointer in the index tree points to the next occurrence of the required data bucket. Finally, the client has to download *C* buckets of the required records. Thus, the tuning time using distributed indexing is bound by

$$2 + k + C$$

Optimizing the Number of Replicated Levels. Optimizing the number of replicated levels has no impact on the tuning time. It only affects the latency. Thus optimizing the number of replicated levels *r*, corresponds to minimizing the latency. This will be accomplished by choosing *r* in such a way that, the following expression is minimal:

$$\Delta Index_r + \left(\frac{Index - Index[r]}{Level[r+1]} + \frac{Data}{Level[r+1]} \right) \qquad (**)$$

Given a particular index tree, the above expression (**) is evaluated for *r* ranging from 1 to *k* (the number of levels in the index tree). The value of *r*, that results in the minimal value for (**) is the optimal number of replicated levels.[14]

14. In principle, the replication "boundary" for the index tree does not have to be associated with a level of the index tree. It could go along nodes on different levels. For simplicity, we have not considered that here.

3.3 Comparison

In general, the distributed indexing algorithm has a much lower latency than the (1, *m*) indexing algorithm. Both (1, *m*) indexing and distributed indexing have a lower latency than *tune_opt*. Distributed indexing achieves almost the optimal latency (that of *latency_opt*).

The tuning time due to *tune_opt* and (1, *m*) indexing is almost the same. The tuning time of distributed indexing is also almost equal to that of the optimal (*tune_opt*), the difference is just two buckets—always. The tuning time of *latency_opt* is very large and is very much higher than the other three.

A detailed comparison between (1, *m*) indexing and distributed indexing is presented in [14].

4 Nonclustering Index

Usually at most one index is clustered. Other indexes are defined over nonclustered attributes. In this section, we discuss index allocation for nonclustered attributes. To this end, we generalize the distributed indexing algorithm developed in the previous section. The proposed generalization is based on the observation that even if a file is not clustered on a particular attribute, it can always be decomposed into smaller fragments that are clustered.

As in the case of clustered index, we first define two benchmark algorithms that are one dimensional optimal solutions: one which provides optimal latency but has a very poor tuning time, and another which provides optimal tuning time but has a poor latency.

Noncluster_latency_opt. This algorithm is similar to the *latency_opt* algorithm. The best latency is provided by this algorithm as no index is broadcast with the file. Clients tune into the broadcast channel and listen continuously, until all the required records are downloaded. For a file of size *Data*, the latency is *Data*. This is because the records are nonclustered and hence distributed across the file. The tuning time is the worst and is equal to *Data* (clients have to be in the *active mode* throughout).

Noncluster_tune_opt. This algorithm is similar to the *tune_opt* algorithm. The index of the file is broadcast in front of the file. Index entries point to the first occurrence (in the current *bcast*) of a record with a given attribute value. Additionally, all records that have the same attribute value are linked by pointers. The client that needs all records with attribute value *K* tunes into the broadcast channel at the beginning of the next *bcast* to get the index. It then follows the pointers from one index level to another in a successive manner, to the first occurrence of a record with attribute value *K*. It then downloads all consecutive records with that attribute value. Then it follows the pointer to the next occurrence of records with the same attribute value, this is done until the end of the current *bcast*. Assuming that the nonclustering attribute has a *coarseness* of *C* (buckets), the tuning time is equal to (*k* + *C*), where *k* is the number of levels in the index tree of the nonclustered attribute. This method has a poor latency because the client has to wait until the beginning of the next *bcast* even if the required data is broadcast immediately after the initial probe of the client. Let *Index* denote the size of the nonclustered index. The *probe wait* is equal to $(\frac{Data + Index}{2})$ and the *bcast wait* is (*Data* + *Index*). Hence, the latency is $\frac{3*(Data+Index)}{2}$.

The above one dimensional optimal algorithms will be used as benchmarks for comparison with the proposed algorithm. In the following, we describe an algorithm called *nonclustered Indexing*, for nonclustering index allocation.

4.1 Nonclustered Indexing Algorithm

The method of organizing the index for a nonclustering attribute is a generalization of distributed indexing algorithm. The structure of the file is as follows:

- The file is partitioned into a number of segments called *meta segments*. Each meta segment holds a sequence of records with nondecreasing values in the nonclustering attribute. Even though the entire file is not clustered on the index attribute, the meta segments are clustered based on the index attribute. The *scattering factor* of an attribute is defined as the number of *meta segments* in the file.
- Each meta segment is further divided into a number of data segments. Each data segment DS(K) is a collection of records with the value of nonclustering attribute equal to K.
- The last element of each data segment DS(K) has an offset to the next occurrence of the data segment DS(K). The last DS(K) of a *bcast* has an offset to the first occurrence of the data segment DS(K) in the next bcast.

There is an index for each meta segment, and it indexes all the values of the nonclustered attribute, rather than being restricted to indexing just the values that occur in the current meta segment. Index entries corresponding to attribute values that are present in the current meta segment will point to the appropriate data segments within the meta segment. For attribute values that do not occur in the current meta segment, the pointer in the index tree will point to the next occurrence of the data segment for that value. The nonclustered index is organized as in *distributed indexing* for each *meta segment* separately.

Each meta segment is partitioned by a set of *index segments*. Each index segment is a set of index buckets. Let *IS*(*K*) denote the index segment immediately following the data segment *DS*(*K*). The first element of *IS*(*K*) is an offset to the beginning of the next *meta segment*. Clients looking for records whose attribute value *P* is lesser than *K*, use this offset. The remaining elements of *IS*(*K*) are built as a directory with offsets to the next data segments *DS*($P\ K < P \leq L$, where *L* denotes the largest value of the attribute. The directory here corresponds to *Rep*(*B*) followed by *Ind*(*B*), where *B* is a bucket belonging to *NRR* as defined in distributed indexing. As in the case of distributed indexing, even here the index segment contains a control index. However, the first tuple in the control index is a new element. This new element indicates the total number of buckets $\#_B$ in the *bcast*. Each data bucket has an offset to the next index segment. The access protocol for following the pointers in *IS*(*K*) is exactly as in distributed indexing.

EXAMPLE. Fig. 10 shows a file that has a nonclustering attribute *b*. It shows various *meta segments* and *data segments*. The file is partitioned into data segments for attribute *b*. The data segments for *b* are *DS*(*b*1), *DS*(*b*2), and *DS*(*b*3). Fig. 10 also shows the index segment for *b*1 and *b*3. The index segment in this example appears after every data segment.

Each data bucket contains a pointer to the next occurrence of the index segment for *b*. Additionally, every data segment of attribute *b* contains a pointer to the next data segment that contains records with the same value of the attribute. The pointers for attribute value *b*1 is shown in the figure. Finally, each data segment is followed by an index segment which provides a directory to the nearest occurrence of data segments for other values of *b*.

Immediately following (the second) *DS*(*b*1) is a pointer to the next occurrence of *DS*(*b*1). This is followed by the index segment *IS*(*b*1). Consider the index segment *IS*(*b*1), the first element is *non* because there is no value for the attribute *b* that is less than *b*1. The second value indicates that *DS*(*b*2) occurs after eight buckets. Note that even though *b*2 does not occur in the current meta segment, there is an entry for it. The third element indicates that *DS*(*b*3) occurs in the next bucket. For search keys less than *b*3, the first element (only element) in *IS*(*b*3) directs the client to the next bucket.

The access protocol for records with the attribute value *K* is as follows:

- Probe the current bucket and get the pointer to the next index segment. Go into the *doze mode* until the next occurrence of the index segment.
- Upon tuning in at the next index segment, get the number of buckets *B* in the current *bcast*. Then follow the pointers to get to the offset for *DS*(*K*). The client might go into the *doze mode* between successive probes. Go into the *doze mode* until *DS*(*K*).
- Upon tuning in at *DS*(*K*), download all the consecutive records of the current data segment and get the *offset* to the next occurrence of *DS*(*K*). Assuming that *b* denotes the number of buckets passed from the point the first index segment was encountered. Repeat until ($b + offset < \#_B$); then exit.

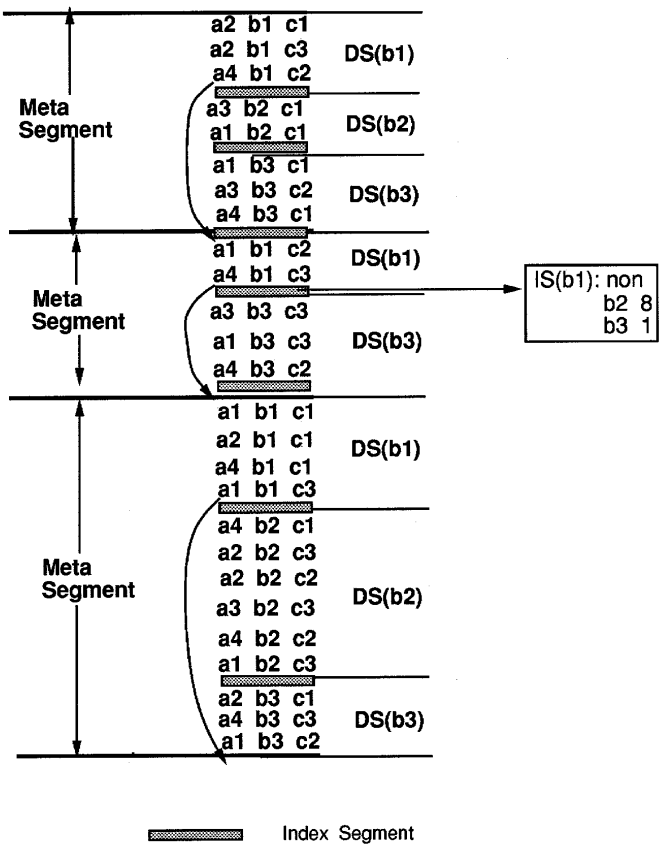


Fig. 10. . Example of nonclustered indexing.

4.2 Analysis

Let *C* denote the coarseness of the nonclustered attribute. Let *M* denote the *scattering factor*, i.e., the number of *meta segments* of the attribute. *Data* denotes the total number of data buckets and *Index* denotes the number of buckets occupied by the index tree.

Latency. Each meta segment has its own index tree that is distributed according to the distributed indexing technique. Let $\Delta Index$ denote the average index overhead *per* meta segment. The *bcast wait* is equal to the size of the whole broadcast, i.e.,

$$M * (Index + \Delta Index) + Data$$

Note that the *bcast wait* is slightly less than that given by the above formula. This is because on the average, only half of the last meta segment has to be accessed. This is also true for the *noncluster_latency_opt* and *noncluster_tune_opt* algorithms. This small difference is ignored for simplicity.

The *probe wait* is analogous to the probe wait in the case of distributed indexing with respect to a meta segment. Let *r* denote the average number of optimal replicated levels (within each meta segment), in all the meta segments. Hence, the latency is:

$$\frac{1}{2} * \left(\frac{Index - Index[r]}{Level[r+1]} + \frac{Data}{M * Level[r+1]} + M\right) * (Index + \Delta Index) + Data$$

Tuning Time. In the first probe, the client fetches a pointer to the next index segment. After getting to the index segment a number of probes have to be made to find the required data segment *DS*(*K*). The number of probes required for getting to *DS*(*K*) is at most (k + 1) (as in distributed indexing), where *k* is the number of levels in the index tree of the indexed attribute. Once the first data segment *DS*(*K*) is encountered, then *C* probes have to be made to get all the required records. Furthermore, in the worst case, *M* extra probes may have to be made. This is because the data segments may not occupy an exact multiple of number of buckets, in which case the last bucket of a given data segment might contain only a few records belonging to this data segment. Nevertheless this bucket has to be probed. This factor also needs to be added to the tuning time of *noncluster_tune_opt* for a general case. Therefore, the tuning time is bounded by

$$1 + (k + 1) + C + M$$

i.e.,

$$2 + k + C + M.$$

In the following section, *nonclustered indexing* is compared with *noncluster_latency_opt* and *noncluster_tune_opt* algorithms.

4.3 Comparison

The tuning time achieved by *nonclustered indexing* is always substantially better than the tuning time due to *noncluster_latency_opt*. The latency of *nonclustered indexing* depends on the scattering factor *M*. *Nonclustered indexing* achieves a tuning time close to that of the optimum (*noncluster_tune_opt* algorithm). *Nonclustered indexing* achieves a latency close to that of the optimum (*noncluster_latency_opt* algorithm), when the scattering factor is small.

If the latency is very important then *noncluster_latency_opt* is used. If the tuning time is important, then either *noncluster_tune_opt* or *nonclustered indexing* is used. *Nonclustered indexing* and *noncluster_tune_opt* have almost the same tuning time hence, when tuning time is of concern, the decision as to which of the two algorithms to use depends on the latency for the two algorithms.

We now compare *nonclustered indexing* with the benchmarks. For simplicity of the analysis, we assume that the index tree is 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.

Consider a file with 10,000 data buckets. Let there be 100 different values for the indexed attribute, thus the coarseness of the attribute is 100. Let a bucket be capable of holding 10 search key and pointer combinations (i.e., $n = 10$), the index will then occupy 11 buckets.

Fig. 11 illustrates the influence of the scattering factor on the latency. The X-axis denotes the scattering factor and the Y-axis denotes the latency. The bottom line denote the optimum latency (due to *noncluster_latency_opt*) and the top line denotes the latency due to *noncluster_tune_opt*. The latency due to *nonclustered indexing*, is shown by the third line. Note that the latency due to *noncluster_tune_opt* and *noncluster_latency_opt*, is independent of the scattering factor. As the scattering factor increases, so does the latency due to *nonclustered indexing*, as illustrated by the figure. The graph indicates that *nonclustered indexing* is better than *noncluster_tune_opt* in terms of the latency, for a scattering factor $M \leq 250$. Beyond that threshold *noncluster_tune_opt* has a better latency.

REMARKS. The *nonclustered indexing* method and the allocation method for indexing on multiple attributes (described in the next section) are applicable only for static files. By a static file we mean that the broadcast file does not change between successive *bcasts*. Under this assumption, the client that requests all data records with a particular attribute value may download them from two consecutive *bcasts* (i.e., part of the records will be downloaded from the current *bcast* and the rest from the next). Such a scheme will lead to inconsistent results, should the data change between two successive *bcasts*.

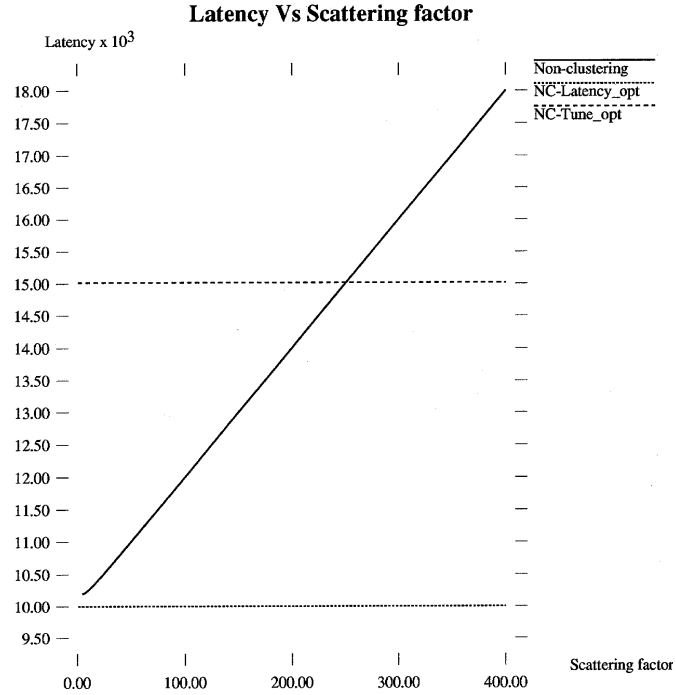


Fig. 11. Influence of scattering factor.

5 MULTIPLE INDEXES

This section discusses a method for organizing and accessing broadcast file indexed on multiple attributes. Since all indexes and data share the same broadcast channel, it is important to specify how indexes and data are going to be multiplexed.

Let each record of the file have *n* attributes. Assume that the attributes are ordered according to the frequencies of their access. Let the leftmost attribute be the most frequently accessed one, and the rightmost one be the least frequently requested. Let the sorted attributes be:

$$a_1, a_2, \ldots, a_n.$$

Partition the file into meta segments based on each attribute separately. The number of meta segments for the attributes increase with the value of the subscript. Usually only attribute a_1 is clustered and the rest are nonclustered. Let us assume that the indexes are available for the first l attributes. The total index overhead will depend on the subscript of the index attribute—in general, the lesser the subscript of the attribute the lesser the index overhead.

Let $I = \{a_1, \ldots, a_l\}$ be the indexed attributes. Partition the file separately for each attribute belonging to *I*. Allocate the file and the index for all the l attributes as in *nonclustered indexing* (except possibly the first attribute that is allocated as in distributed indexing). While allocating the index for a particular attribute, the indexes of the other $(l - 1)$ attributes are treated as part of the file. The index segment of each index attribute has to take into account the additional offset introduced by the presence of the index segment of other attributes. Each data bucket of the file stores a pointer to the next occurrence of the control index for all attributes in *I*.

If an attribute has a clustering index, then the access protocol for accessing records based on that attribute is

similar to that of accessing records in distributed indexing. For nonclustering attributes the access protocol is similar to that of *nonclustered indexing*.

Notice that contrary to disk based files, the latency for accessing records based on some attribute is dependent on the presence or absence of the index for other attributes. Each additional index increases the overall latency while decreasing the tuning time for the indexed attribute.

Consider Fig. 12, the records have three attributes: *a*, *b*, and *c*. The file is lexicographically sorted (with *a* as the most significant attribute, followed by *b* and then by *c*). Let *a* and *b* be the attributes over which the file is indexed, i.e., $I = \{a, b\}$. The file is partitioned into data segments for attributes *a* and *b*. The data segments for *a* are denoted by *DS*(*a*1), *DS*(*a*2), *DS*(*a*3, and *DS*(*a*4). The data segments for the attribute *b* are *DS*(*b*1), *DS*(*b*2), and *DS*(*b*3).

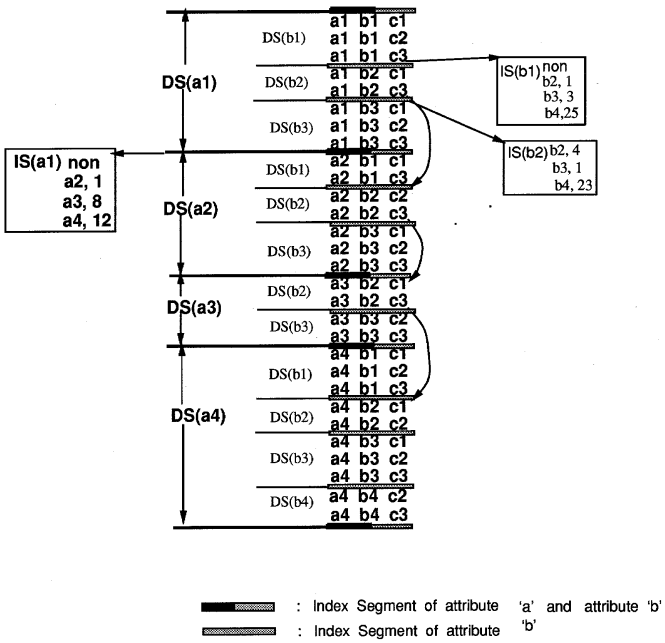


Fig. 12. Example of multiple attribute indexing.

A data segment corresponding to the attribute *a*(*b*), contains a pointer (following the last record of the data segment) to the next data segment, that contains records with the same value for the attribute *a*(*b*). These pointers are shown for attribute value *b*2. Each data segment corresponding to attribute *a*(*b*) is followed by an index segment, which provides a directory to nearest occurrences of data segments for other values of *a*(*b*). Additionally (not shown in the figure), each data bucket contains a pointer to the next occurrence of the control index for *a*, and the control index for *b*. In the figure, the index segment for attribute *b* occurs at every index segment for attribute *a*. This is not true in general.

Immediately following the (first) data segment *DS*(*b*2) is a pointer to the next occurrence of *DS*(*b*2). This is followed by the index segment *IS*(*b*2). Consider the index segment *IS*(*b*2). The first element indicates that if the attribute's value is less than or equal to *b*2 (i.e., *b*1 or *b*2) then the client has to probe the fourth bucket ahead of the current bucket. The second element indicates that if the requested attribute value is *b*3, then the client has to probe the first bucket following the current bucket. The third element indicates that if records with value of *b*4 (for attribute *b*) is being searched for, then the client has to tune after 23 buckets. Note that even though *b*4 does not exist in the current meta segment, there is still an index entry for it.

5.1 Analysis

DEFINITIONS. *We use the following notations:*

- C_i: *the coarseness of attribute* a_i *(in buckets).*
- M_i: *the scattering factor of the attribute* a_i.
- $Index_i$: *the size of the index tree of attribute* a_i.
- k_i: *the number of levels in the index tree corresponding to attribute* a_i.
- r_i: *the average number of optimum replicated levels in the index tree of attribute* a_i.
- $\Delta Index(i)$: *the average index overhead (per meta segment) due to attribute* a_i.

Latency. Each meta segment of attribute a_i has ($Index_i$ + $\Delta Index(i)$) number of extra buckets due to distributed indexing on the attribute. The total number of extra buckets E_i due to indexing the attribute a_i is

$$E_i = M_i * (Index_i + \Delta Index(i))$$

The *bcast wait* is

$$\sum_{i=1}^{l} E_i + Data$$

For an attribute a_i that is indexed, the *probe wait* is

$$\frac{1}{2} * \left(\frac{Index - Index[r_i]}{Level[r_i + 1]} + \frac{Data}{M * Level[r_i + 1]} \right)$$

Thus, the latency for an attribute a_i that is indexed is

$$\frac{1}{2} * \left(\frac{Index - Index[r_i]}{Level[r_i + 1]} + \frac{Data}{M * Level[r_i + 1]} \right) + \sum_{i=1}^{l} E_i + Data$$

For an attribute a_i that is not indexed, the *probe wait* is 0.

The *bcast wait* and the latency for such an attribute is

$$\sum_{i=1}^{l} E_i + Data$$

Tuning Time. The tuning time for indexed attribute a_i is calculated below. In the first probe, the client fetches a pointer to the next index segment for the attribute a_i. After getting to the index segment a number of probes have to be made in that index to get to the required data segment $DS(K_i)$. The number of probes required for getting to $DS(K_i)$ is $(k_i + 1)$, where k_i is the number of levels in the index tree for a_i. Once the first data segment $DS(K_i)$ is encountered, then C_i probes have to be made to get all the required records. Furthermore, M_i extra probes may have to be made to get all the required records.[15] The tuning time for the indexed attribute a_i is bound by

$$1(k_i + 1) + C_i + M_i$$

i.e.,

$$2 + k_i + C_i + M_i$$

15. This is because the data segments may not occupy an exact multiple of number of buckets in which case the last bucket might contain only a few records belonging to this data segment.

For a nonindexed attribute a_i the client looses the ability of selective tuning.

In the worst case, the whole length of the meta segment for attribute a_l has to be scanned to get to the next record with the requested value of a_i. This is because the last indexed attribute is a_l. After this, the number of probes required is $(C_i + M_i)$ as described above.

Thus, the tuning time for records on attribute a_i that is not indexed is bound by

$$\frac{Data}{M_l} + C_i + M_i$$

6 Practical Implications

Consider a broadcasting system that is similar to the *quotrex system*[16], where stock market information of size $16 * 10^4$ bytes is being broadcast. Let the broadcast channel have a bandwidth of 10 Kbps. Let the bucket length be 128 bytes. Thus, there are 1,250 buckets of data. Let the capacity of the bucket be 25, i.e., the number of tuples of the form *(attribute_value, offset)* that can fit in a bucket (n) is equal to 25. It takes around 0.1 sec to broadcast a single bucket and 125 sec 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 neglect other components that consume power during data filtering and assume that 250 mW constitutes the total power consumption.

6.1 Clustering Index

Let the coarseness of the attribute that has a clustering index be one. The index size is 53 buckets. With *latency_opt* algorithm, the latency is 625 buckets (half of *bcast* duration), i.e., 62.5 sec. The tuning time is also 625 buckets, i.e., the power consumption is 62.5 sec * 250 mW = 15.625 J.

With *tune_opt* algorithm, the latency is 1,303 (1,250 + 53) buckets i.e., 130.3 sec. The tuning time is minimum at 4 buckets, i.e., the power consumption is 0.1 sec * (4 * 250 + 1, 299 * 50 * 10^{-3}) mW = 0.106 J.

If we want a low latency 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 latency is 909 buckets, i.e., 90.9 sec. 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 J.

  Comparing with *tune_opt*, the power consumption is almost the same. However, the latency improves to 69.8% of the latency in *tune_op.t.*[17]

  The power consumed *per query request* due to *(1,m) indexing* is 120 times smaller than that of *latency_opt*. This is achieved by compromising on the latency which increases by 45.4%.
- *Distributed indexing*: The optimum number of replicated levels in distributed indexing is 2. The latency is 689 buckets, i.e., 68.9 sec. 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 J.

  Comparing with *tune_opt*, the power consumption is almost the same. However, the latency improves to 52.9% of the latency in *tune_opt.*[18]

  The power consumed *per query request* due to *distributed indexing* is 100 times smaller than that of *latency_opt*. This is achieved by compromising on the latency which increases by just 10%.

Distributed indexing is almost as good as the optimal algorithm for latency and the optimal algorithm for tuning time. Our algorithms are significantly better than *latency_opt* and almost as good as *tune_opt*, in terms of the power consumption. The proposed algorithms are much better than *tune_opt* and almost equivalent to *latency_opt*, in terms of the latency.

How much power 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 W/hr are used in the receiver, one being used for filtering and the second for other applications. Let one such battery be good for 4 hr of continuous data filtering (without indexing). Using distributed indexing, we can serve the same number of filtering requests with 100 times less power. The saved power 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 power of one AA battery for the same number of requests. This savings can be used to almost double 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.

6.2 Nonclustering Index

Let there be five *meta segments* for a nonclustered attribute that has to be indexed. On the average, there will be around 250 buckets in each meta segment. Let there be 63 different values for the indexed attribute. Hence, the coarseness of the indexed attribute is 20. Thus, the index will consist of four buckets (as each bucket can hold 25 pointers). The optimum number of replicated levels in the indexed tree is one. The total number of index buckets in each meta segment is six. The total number of index buckets in the file is 30.

With *noncluster_latency_opt* algorithm, the latency is 1,250 buckets, i.e., 125 sec. The tuning time is also 1,250 buckets, i.e., the power consumption is 125 sec * 250 mW = 31.250 J.

With *noncluster_tune_opt* algorithm, the latency is 1,875 buckets, i.e., 187.5 sec. The tuning time is minimum at 22 buckets, i.e., the power consumption is 0.1 sec * (22 * 250 + 1,853 * 50 * 10^{-3}) mW = 0.559 J.

- *Nonclustered indexing.* The latency due to *nonclustered indexing* is 1,324. The tuning time is 24 buckets. Hence, the power consumption is 0.1 sec * (24 * 250 + 1,300 * 50 *10^{-3}) mW, i.e., 0.607 J.

16. Quotrex system broadcasts stock quotes continuously over an FM channel.

17. For the optimal latency, the improvement is $\frac{625}{1,250} * 100 = 50\%$.

18. For the optimal latency, the improvement is $\frac{625}{1,875} * 100 = 66.6\%$.

- Compared to *noncluster_tune_opt*, the power consumption is almost the same. However, the latency improves to 70.61% of the latency in *noncluster_tune_opt*.[19]

  The power consumed *per query request* due to *nonclustered indexing* is 51 times smaller than that of *noncluster_latency_opt*. This is achieved by compromising the latency which increases by just 6%.

7 Communication Issues

7.1 Setup Time

In all the data organization algorithms described so far, we have ignored the setup time. Usually, the setup time is negligible compared to the time it takes to broadcast a bucket. For example in GSM, the setup time for a typical receiver is 5 msec compared to 120 msec to broadcast a bucket.[20] The setup time of a typical CPU for a palmtop (*Piranha chip from AT&T*), is less than 20 msec. In such an environment, the setup time can be ignored for the estimation of the power savings and all the access protocols remain the same. Note that the organization of buckets on the broadcast channel is not affected by the setup time.

Let T_s denote the setup time. If precise power consumption measurements for a data organization technique have to be made, then the tuning time is multiplied by a factor of $2 * T_s$, where T_s denotes the setup time and the factor of 2 is for the setup time for tuning in and for tuning out of the broadcast channel. The power consumption is computed as a product of the tuning time and the power it takes to download a bucket.

In case the setup time is much bigger than the time it takes to broadcast a bucket, then the setup process may be taken under consideration by assuming that tune in and tune out operations (together) take the time equivalent to read t buckets. The access protocols of the proposed data organization techniques, will change as follows:

- When a client needs to probe a particular bucket, it has to do so in advance, so as to take into account the setup time. It has to tune in roughly $\frac{t}{2}$ buckets in advance.
- Upon reading the offset and determining when the next relevant bucket will be broadcast, the client has to check if the following holds:

  $$((offset > t) = TRUE).$$

  If the answer is positive, then the client goes into the doze mode, else it remains in the active mode.

7.2 Reliability

In our discussion so far, we have assumed reliable delivery of data. However, broadcasting itself is inherently unreliable and methods are needed for recovery, especially in an error prone wireless environment. The conventional method of achieving reliability by means of retransmissions and acknowledgments is not suited for broadcast based techniques. A flood of negative acknowledgments will result in the acknowledgment implosion problem.

However, broadcasting being *periodic* in nature, data is retransmitted anyway in the next *bcast*. This mitigates the effect of lost buckets. Clients who miss buckets in a particular *bcast* can receive them in the next *bcast* with a very high probability. This results in an increase in latency for those clients who loose data.

The latency penalty can be reduced and at the same time reliability can be enhanced by using one of the following techniques.

Parity Across Buckets. Parity checks can be set up across buckets. A collection of, say five buckets, called a *parity group* may be associated with one parity bucket (which follows these buckets immediately).

If any bucket in a parity group is received erroneously or simply missed, then the missing bucket can be recomputed from the data in the other five. If this technique is used, then the client has to download all the buckets in the parity group of the bucket which is of interest. This increases the tuning time. The increase in the tuning time could be as much as six times (number of buckets in the parity group plus one). Since reducing the tuning time is one of the primary concerns, having parity across buckets is not attractive.

Redundant Buckets. Having redundant buckets increases the length of a *bcast*, but it increases the likelihood for the clients to receive a bucket in the same *bcast*. Unfortunately, if each data bucket has a redundant bucket then the length of the *bcast* doubles, thereby doubling the overall latency. This is unacceptable, since usually only a small percentage of the clients get erroneous buckets and it is simply better for those clients to tune to the next *bcast*.

However, having redundant *index* buckets can be useful. If an index bucket is lost, then effectively all the data buckets that are indexed by this index bucket are lost. Hence, redundant index buckets can be used to mitigate the effect of increased latency due to lost index buckets. It may be worthwhile (depending on the probability of loosing a bucket) to have redundant broadcast of index buckets to reduce the latency. The number of redundant replications of an index bucket per *bcast* will be determined by the level of the bucket in the index tree (the higher up the bucket, the more replications required) and the probability of loosing a bucket. Since the number of index buckets is significantly less than the number of data buckets, redundant index buckets do not contribute to significant increase in the length of a *bcast*.

7.3 Dynamic Organization

Reorganizing broadcast files (*bcast*), is substantially cheaper than reorganizing disk based files. Indeed, in case of broadcasting, each new *bcast* starts from "scratch," while the previous *bcast* is "gone and forgotten." The new *bcast* can substantially differ from the preceding one, both in terms of data ordering as well as index structure. The reorganization cost is limited to the server's memory reorganization. The reorganization of a *bcast* itself simply amounts to a different

19. For the optimal latency, the improvement is $\frac{625}{1,250} * 100 = 50\%$.

20. GSM (Global System for Mobile communication) stands for Pan-European Digital Cellular Mobile Radio System which is aimed at providing a wide range of services and might support Personal Communication Services (PCS) in the future. GSM provides data rates of up to 271 Kbps. The *packet* size for GSM is 156 bytes. Each bucket (called multipacket) is a multiple of packets. In GSM, a bucket consists of 26 packets, i.e., a bucket is 4,056 bytes long. It takes 120 msec for a bucket to be broadcast [13].

downloading order by the server. Hence, there is minimal need for dynamic data organization methods such as *B*-trees and dynamic hashing for broadcasting. In fact, dynamic indexing methods such as *B*-trees are generally inappropriate for broadcasting, since in order to ensure the "locality" of changes they require reservation of extra pointer space which would unnecessarily increase the overall *bcast* size and consequently the latency.

7.4 Bucket Size

In this paper we have assumed a fixed bucket size in a *bcast*. The bucket size has different effects on latency and on tuning time. The latency generally decreases with the bucket size, while the tuning time increases with the bucket size. Larger bucket size has a positive effect on latency, which in general, decreases with increasing bucket size. This is because the index size decreases with the bucket size and so does the overhead due to the number of references to the data buckets. In general, the bucket size for minimum latency depends on the method of organizing the data. Thus the quantitative impact of the bucket size on the tuning time and the latency depends on the particular indexing method used. A more detailed discussion on the effects of bucket size can be found in [14].

8 Conclusion

We have explored methods for allocating clustering and nonclustering indexes (along with the corresponding data) on wireless networks. We took a stand that the communication network can be treated as a *storage* medium, where frequently accessed data is periodically broadcast on the network to a large number of users. The problems encountered for data organization on the broadcast channel are different from the data organization on disks, due to different performance criteria. Data that is broadcast on a network is characterized by two parameters: the latency and the tuning time, contrary to the data on disks being characterized by just one parameter: the access time.

For clustering indexes, we have provided two basic algorithms: (1, *m*) indexing and distributed indexing. Nonclustering index case was treated by decomposing the data into sub parts for which the index has the clustering property. Finally, we described a method for multiplexing multiple indexes with data. Adding indexes increases the latency but provides radical improvement in terms of the tuning time and consequently improves battery utilization. Distributed indexing seems to be a powerful method for multiplexing any type of index together with broadcast data. The resulting latency and tuning time is close to the optimum as our analyses indicate.

Acknowledgment

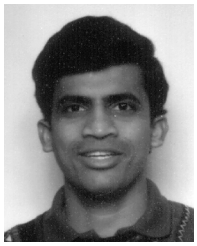
We would like to thank the anonymous referees for their helpful comments, which improved the presentation of the paper.

References

[1] R. Alonso and H. Korth, "Database Issues in Nomadic Computing," MITL Technical Report-36-92, Dec. 1992.
[2] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environment," *Proc. ACM-SIGMOD, Int'l Conf. Management of Data*, Minnesota, pp. 1-12, May 1994.
[3] T.F. Bowen et al., "The Datacycle Architecture," *Comm. ACM*, vol. 35, no. 12, pp. 71-81, Dec. 1992.
[4] D. Cheriton, "Dissemination-Oriented Communication Systems," Technical Report, Stanford Univ., 1992.
[5] D. Gifford et al., "The Application of Digital Broadcast Communication to Large Scale Information Systems," *IEEE J. Selected Areas in Comm.*, vol. 3, no. 3, pp. 457-467, May 1985.
[6] G. Herman et al., "The Datacycle Architecture for Very Large High Throughput Database Systems," *Proc. ACM SIGMOD Conf.*, San Francsco, pp. 97-103, May 1987.
[7] T. Imielinski and S. Viswanathan, "Adaptive Wireless Information Systems," *Proc. Special Interest Group on DataBase Systems (SIGDBS), Conf.*, pp. 19-41, Japan, Oct. 1994.
[8] T. Imielinski, S. Viswanathan and B.R. Badrinath, "Energy Efficient Indexing on Air," *Proc. ACM-SIGMOD, Int'l Conf. Management of Data*, Minnesota, pp. 25-36, May 1994.
[9] T. Imielinski, S. Viswanathan and B.R. Badrinath, "Power Efficient Filtering of Data on Air," *Proc. Fourth Int'l Conf. Extending DataBase Technology, (EDBT)*, Cambridge, UK, pp. 245-258, Mar. 1994.
[10] C. Partridge, *Gigabit Networking*. Addison-Wesley Professional Computing Series, Dec. 1993.
[11] S. Sheng, A. Chandrasekaran, and R.W. Broderson, "A Portable Multimedia Terminal for Personal Communication," *IEEE Comm.*, pp. 64-75, Dec. 1992.
[12] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments," *Proc. ACM-SIGMOD, Int'l Conf. Management of Data*, San Jose, pp. 199-210, June 1995.
[13] R. Steele, *Mobile Radio Comm.* London: Pentech Press, 1992.
[14] S. Viswanathan, "Publishing in Wireless and Wireline Environments," PhD dissertation at Rutgers, State Univ. of New Jersey, 1994.
[15] L., William., "Mobile Cellular TeleComm. Systems," New York: McGraw-Hill, 1989.
[16] J.W. Wong., "Broadcast Delivery," *Proc. IEEE*, vol. 76, no. 12, pp. 1,566-1,577, Dec. 1988.

T. Imielinski received his PhD from the Polish Academy of Science (Warsaw) in 1982. He is on the faculty of the Department of Computer Science at Rutgers University, New Brunswick, New Jersey. His initial work dealt with the issues of representation and querying of databases with incomplete information. He also formulated one of the first approaches to provide intensional answers to database queries. He has co-edited a book on this subject published recently by Oxford University Press.

His current interests include database mining and mobile wireless computing. The work in database mining (initiated jointly with Rakesh Agrawal) resulted in the early algorithm for massive rule discovery, which is currently used in a number of data mining tools including the DataMine system at Rutgers. Dr. Imielinski is also the principle investigator of the DataMan project at Rutgers University, involving data management issues in mobile computing. This work, supported by industry through Wireless Information Networks Lab (WINLAB) and governmental agencies (Advanced Research Projects Agency (ARPA) and the National Science Foundation (NSF)) includes location dependent information services for mobile users and the higher level communication protocol support for mobile and wireless environments. Dr. Imielinski is co-editor of the book *Mobile Computing* (Kluwer) describing the current state of research in this area.



S. Viswanathan received his MTech (1991) and undergraduate degree (1989) from the Indian Institute of Technology (Bombay) and BITS (Pilani), respectively. He received his PhD and masters degrees from Rutgers University (New Jersey) in 1994 and 1993, respectively. He is a research scientist at Bell Communications Research, Morristown, New Jersey. Dr. Viswanathan received the Gold Medal for the best graduate academic record at the Indian Institute of Technology. His research interests include networking and database issues in mobile computing, multimedia, and computational geometry.

B.R. Badrinath (S'82-M'83-S'84-M'87-S'88-M'88) received the BE degree in electronics from Bangalore University, the ME degree at the School of Automation from the Indian Institute of Science, and the PhD degree in computer science from the University of Massachusetts, Amherst, in 1989. He is currently an associate professor in the Computer Science Department at Rutgers University. He is a co-principal investigator of the DataMan project at Rutgers University. His research interests include wireless networking, distributed systems, and wireless mobile computing. As part of his research, he is working on developing communication protocols for wireless networks—protocols optimized for wireless and mobility that can support file and database access. He is also investigating operating systems support for mobile hosts. Dr. Badrinath is a member of the IEEE and the ACM.