Multiversion Data Broadcast Organizations

Oleg Shigiltchoff[1], Panos K. Chrysanthis[1] and Evaggelia Pitoura[2]

[1] Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{oleg,panos}@cs.pitt.edu

[2] Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
pitoura@cs.uoi.gr

Abstract. In recent years broadcasting attracted considerable attention as a promising technique of disseminating information to large number of clients in wireless environment as well as in the web. In this paper, we study different schemes of multiversion broadcast and show that the way broadcast is organized has an impact on performance, as different kind of clients needs different types of data. We identify two basic multiversion organizations, namely Vertical and Horizontal broadcasts, and propose an efficient compression scheme applicable to both. The compression can significantly reduce the size of the broadcast and consequently, the average access time, while it does not require costly decompression. Both organizations and the compression scheme were evaluated using simulation.

1 Introduction and Motivation

The recent advances in wireless and computer technologies create expectation that data will be "instantly" available according to client needs at any given situation. Modern client devices often are small and portable, therefore they are limited in power consumption. As a result, the significant problem arises: how to transfer data effectively taking into consideration this limitation.

One of the schemes which can solve this problem is *broadcast push* [1]. It exploits the asymmetry in wireless communication and the reduced energy consumption in the receiving mode. Servers have both much larger bandwidth available than client devices and more power to transmit large amounts of data.

In broadcast push the server repeatedly sends information to a client population without explicit client requests. Clients monitor the broadcast channel and retrieve the data items they need as they arrive on the broadcast channel. Such applications typically involve a small number of servers and a much larger number of clients with similar interests. Examples include stock trading, electronic commerce applications, such as auction and electronic tendering, and traffic control information systems. Any number of clients can monitor the

broadcast channel. If data is properly organized to cater to the needs of the client, such a scheme makes an effective use of the low wireless bandwidth. It is also ideal to achieve maximal scalability in regular web environment.

There exist different strategies which can lead to performance improvement of broadcast push [6, 8]. The data are not always homogeneous and clients sometime are more interested in particular data elements. Therefore some data, more frequently accessed, are called "hot" and the other data, less frequently accessed, are called "cold". To deal with this kind of data the idea of broadcast disks was introduced [3, 4, 2]. Here the broadcast organized as a set of disks with different speeds. "Hot" data are placed on the "hot" (or "fast") disk and the "cold" (or "slow") data are placed on the "cold" disk. Hence if most of the data that client needs are "hot" it reduces the response time.

Another strategy capable to reduce the access time is client caching. However when data are being changed, there arises a problem how to keep the data cached in a client consistent with the updated data on the server [10, 12, 5]. Clearly, any invalidation method is prone to starvation of queries by update transactions. This same problem also exists in the context of broadcast push, even without client caching. Broadcasting is a form of a cache "on the air." In our previous work, we effectively addressed this problem by maintaining multiple versions of data items on the broadcast as well as in the client cache [9]. With multiple versions, more read-only transactions are successfully processed and commit in a similar manner as in traditional multiversion schemes, where older copies of items are kept for concurrency control purposes (e.g., [7]). The time overhead created by the multiple versions is smaller than the overall time lost for aborts and subsequent recoveries.

The performance (determined by the access time and power consumption) of multiversion broadcast is directly related to the issue of the size of the broadcast. Towards this we try to find ways to keep the size of broadcast as small as possible. There is no need to assume that all data have to be changed every time interval such that data values of adjacent versions are always different. Hence, we can reduce the communication traffic by not explicitly sending unchanged part of the older versions [11]. Consequently the client can retrieve the needed version of data sooner if the data do not change very often, which reduces the time during which the client stays on. We exploit this idea in the compression scheme we are proposing in this paper.

The main contributions of this paper are:

1. Identification of two different broadcast organizations for multiversion broadcast, namely *Vertical* and *Horizontal*.
2. Development of a compression scheme along the lines of *Run Length Encoding* (RLE) [11], applicable to both of the proposed broadcast organizations and which incurs no decompression overhead at the client.
3. Evaluation of circumstances under which each of our proposed broadcast organizations performs better.

The rest of the paper is structured as follows. In Section 2, we present the system model. Section 3 and 4 describe server side broadcast organization and

client access behavior, respectively. Sections 5 presents our experimental platform whereas our experimental results are discussed in Section 6.

2 System Model

In a broadcast dissemination environment, a data server periodically broadcasts data items to a large client population. Each period of the broadcast is called a *broadcast cycle* or *bcycle*, while the content of the broadcast is called a *bcast*. Each client listens to the broadcast and fetches data as they arrive. In this way data can be accessed concurrently by any number of clients without any performance degradation (compared to "pull", on-demand approach). However, access to data is strictly sequential, since clients need to wait for the data of interest to appear on the channel. We assume that all updates are performed at the server and disseminated from there.

Without lose of generality, in this paper we consider the model in which the bcast disseminates a fixed number of data items. However, the data values (values of the data items) may or may not change between two consecutive bcycles. In our model, the server maintains multiple versions of each data item and constantly broadcasts a fixed number of versions for each data items. For each new cycle, the oldest version of the data is discarded and a new, the most recent, version is included. The number k of older versions that are retained can be seen as a property of the server. In this sense, a k-multiversion server, i.e., a server that broadcasts the previous k values, is one that guarantees the consistency of all transactions with span k or smaller. *Span* of a client transaction T, is defined to be the maximum number of different bcycles from which T reads data.

The client listens to the broadcast and searches for data elements based on the pair of values (data id and version number). Clients do not need to listen to the broadcast continuously. Instead, they tune-in to read specific items. Such selective tuning is important especially in the case of portable mobile computers, since they most often rely for their operation on the finite energy provided by batteries and listening to the broadcast consumes energy. Indexing has been used to support selective tuning and reduce power consumption, often at the cost of access time. In this paper, we focus only on broadcast organization and how to reduce its size without adopting any indexing scheme.

The logical unit of a broadcast is called *bucket*. Buckets are the analog to blocks for disks. Each bucket has a header that includes useful information. The exact content of the bucket header depends on the specific broadcast organization. Information in the header usually includes the position of the bucket in the bcast as an offset time step from the beginning of the broadcast as well as the offset to the beginning of the next broadcast.

The broadcast organization, that is where to place the data and the old versions, is an important problem in multiversion broadcast. In the next section, we elaborate on this issue, considering in addition broadcast compression as a method to reduce the size of broadcast.

3 Broadcast Organization

3.1 Basic Organization

The multiversion data can be represented as a two-dimension array, where indexes are version numbers (*Vno*) and data ids (*Did*), and the values of the array elements are the data values (*Dval*). That is *Dval*[*Did*=*i*, *Vno*=*k*]=*v* means that *k*-version of *i*-data item is equal to *v*. This data representation can be extended to any number of data items and versions.

The simple sequential scheme can broadcast data items in two different orders: *Horizontal* broadcast or *Vertical* broadcast. In the Horizontal broadcast, a server broadcasts all versions (with different *Vno*) of a data item with a particular *Did*, then all versions (with different *Vno*) of the next data item with the next *Did* and so on. This organization corresponds to the *clustering* approach in [9]. In the Vertical broadcast, a server broadcasts all data items (with different *Did*) having a particular *Vno*, then all data items (with different *Did*) having the next *Vno* and so on. Formally, the Horizontal broadcast transmits [*Did*[*Vno,Dval*]*]* sequences whereas the Vertical broadcast transmits [*Vno*[*Did,Dval*]*]* sequences. To make the idea more clear consider the following example. Let us assume we have a set of 4 data items, each having 4 versions:

	Vno=0	Vno=1	Vno=2	Vno=3
Did=0	Dval=1	Dval=1	Dval=1	Dval=1
Did=1	Dval=8	Dval=8	Dval=8	Dval=5
Did=2	Dval=6	Dval=1	Dval=1	Dval=2
Did=3	Dval=5	Dval=4	Dval=4	Dval=4

For the Horizontal broadcast the data values on the bcast are placed in the following order (The complete bcast will include also the data ids and version numbers as indicated above):

1 1 1 1 8 8 8 5 6 1 1 2 5 4 4 4

while for the Vertical broadcast the data values on the bcast are placed in the following order:

1 8 6 5 1 8 1 4 1 8 1 4 1 5 2 4

Clearly for each of the two organizations, the resulting bcast has the same size. The two organizations differ in the order in which they broadcast the data values.

3.2 Compressed Organization

In both cases, Horizontal and Vertical, the broadcast size and consequently the access time can be reduced by using some compression scheme. A good compression scheme should reduce the broadcast as much as possible with minimal, if no, impact on the client. That is it should not require additional processing at

the client, so it should not trade access time to processing time. The following is a simple compression scheme that exhibits the above properties.

The current compression scheme was inspired by the observation that the data values do not always change from one version to another. In other words, *Dval*[*Did=i, Vno=k*]= *Dval*[*Did=i, Vno=k+1*]= ...= *Dval*[*Did=i, Vno=k+N*]=*v*, where *N*-number of versions at which the value of *i*-data item (having *Did=i*) remains equal to *v*. Then, when broadcasting data, there is no reason to broadcast all versions of a data items if its *Dval* does not change. Instead, the compressed scheme broadcasts *Dval* only if it is different from *Dval* of the previous version. In order not to lose information (as well as to support selective tuning) it also broadcasts the number of versions having the same *Dval*. In formal form the Horizontal broadcast would produce [*Did*[*Vno(Repetition,Dval)*] *] *, and the Vertical broadcast would create [*Vno*[*Did(Repetitions,Dval)*] *] *. Obviously we do not include into the broadcast those versions, which already have been included "implicitly" with other versions.

The way the compression works for Horizontal broadcast is quite straightforward, because we can see the repetitive data values in the simple sequential (or uncompressed) bcast. 1 1 1 1 transforms to 1x3, 8 8 8 transforms to 8x2 and so on. The data values from the example above are broadcast in the following compressed format:

1x3 8x2 5 6 1x1 2 5 4x2

The compression for Vertical broadcast is slightly more complex. To explain the idea let us redraw the previous table in a way that captures the first step of our compression algorithm. The second step is the vertical linearization of the array.

Did=0	1 for Vno=0–3			
Did=1	8 for Vno=0,1,2			5 for Vno=3
Did=2	6 for Vno=0	1 for Vno=1,2		2 for Vno=3
Did=3	5 for Vno=0	4 for Vno=1,2,3		

In the second step, the compressed data values would be broadcast in the following order:

1x3 8x2 6 5 1x1 4x2 5 2

For the Vertical broadcast, 1x3 means that *Dval*[*Did=0, Vno=0*]= *1* and three other versions (*Vno=1, 2 and 3*) of this data item (*Did=0*) also have *Dval=1*. In such a way the server implicitly broadcasts 4 data elements at the same time. Similarly 8x2 means *Dval*[*Did=1, Vno=0*]= *8*, *Dval*[*Did=1, Vno=1*]= *8*, 6 means *Dval* [*Did=2, Vno=0*]= *6*, and 5 means *Dval*[*Did=3, Vno=0*]= *5*. This completes the broadcast of all data elements having *Vno=0*. Then, it broadcasts the elements having *Vno=1*. The first two elements of *Vno=1* with *Did = 0* and *Did=1* have already been broadcast implicitly (in 1x3 and 8x2), so we do not need to include them into the broadcast. Instead, we include 1x1 corresponding to *Did=2* and so we broadcast explicitly *Dval*[*Did=2, Vno=1*]= *1* and implicitly *Dval*[*Did=2, Vno=2*]= *1*. Next to be broadcast is 4x2, corresponding to *Did=3*

and so on. Note that we broadcast the same number of elements, which are now compressed, for both Horizontal and Vertical broadcasts but in different order.

In the case of the Vertical broadcast, it also makes sense to rearrange the sequence of broadcast data elements within a single-version sweep and make them dependent not on *Did* but on the number of implicitly broadcast elements. Applying this reordering to our example, the resulting vertical broadcast is:

1x3 8x2 6 5 4x2 1x1 5 2

We can see that 4x2 and 1x1 belonging to version 2 switch their positions, because we broadcast implicitly two 4s and only one 1. The idea is that we broadcast first as "dense" data as possible, because when a client begins to read the string it has higher chances to find the necessary data elements in "more dense" data. Of course it works under assumption that client access data uniformly, without distinguishing between "hot" and "cold" data.

In order to make our broadcast fully self-descriptive, we add all necessary information about version number and data items. One of our design principles has been to make the system flexible, allowing a client to understand the content of a broadcast without requiring the client explicitly to be told of the organization of the broadcast. For this purpose, we use four auxiliary symbols:

(*Did*), **V** (*Vno*), **=** (Assignment to *Dval*), ^ (Number of repetitions)

Using these symbols, the *sequential bcast* for Horizontal broadcast discussed above is fully encoded as

V0#0=1V1#0=1V2#0=1V3#0=1V0#1=8V1#1=8V2#1=8V3#1=5
V0#2=6V1#2=1V2#2=1V3#2=2V0#3=5V1#3=4V2#3=4V3#3=4

and for Vertical broadcast

V0#0=1#1=8#2=6#3=5V1#0=1#1=8#2=1#3=4V2#0=1#1=8#2
=1#3=4V3#0=1#1=5#2=2#3=4

V0,V1,V2 and V4 are the version numbers. They determine *Vno* of the data elements which follows it in the broadcast. #0=1 means the element having *Did*=0 of the corresponding version (broadcast before) is equal to 1. So, V0#0=1#1=8 means *Dval*[*Did=0, Vno=0*]*=1* and *Dval*[*Did=1, Vno=0*]*=8*. Note that for Vertical broadcast we do not need to include the version number in the broadcast before each data element, but for Horizontal broadcast we have to do this. Because of this need of some extra auxiliary symbols, a Horizontal broadcast is usually longer than its corresponding Vertical broadcast. However, given that the size of an auxiliary symbol is much smaller (which is typically the case) than the size of a data element, this difference in length becomes very small.

In the case of *Compressed bcast*, the symbol ^ is used to specify that the following versions of a data item have the same value. The other auxiliary symbols are also used to give a client the complete information about *Did*, *Vno*, and *Dval* in a uniform format for both the compressed and uncompressed multiversion

broadcast organizations. Returning to our example broadcasts, the compressed Horizontal broadcast is encoded as:

V0^3#0=1V1V2V3V0^2#1=8V1V2V3^0#1=5 V0^0#2=6V1^1#2=1V2
V3^0#2=2V0^0#3=5V1^2#3=4V2V3

whereas the compressed Vertical broadcast as:

V0^3#0=1^2#1=8^0#2=6#3=5V1^2#3=4^1#2=1V2V3^0#1=5#2=2

Considering the Vertical bcast as an example, let us clarify some details of the broadcast. It starts from the version 0. First, it broadcasts the data elements with the most repetitive versions. V0^3#0=1^2#1=8^0#2=6#3=5 means that versions 0,1,2,3 of data element 0 are 1, versions 0,1,2 of data element 1 are 8, version 0 of data element 2 is 6, version 0 of data element 3 is 5. V1^2#3=4^1#2=1 means that versions 1,2,3 of data element 3 are 4 and versions 1,2 of data element 2 are 1. We do not broadcast versions 1 of data elements 0 and 1 because we broadcast them together with versions 0.

3.3 Discussion

We can roughly estimate the reduction of the broadcast length (and, consequently, the broadcast time) due to our compression scheme. In order to represent the repetitiveness of data from one version to another in numerical form, we introduce the *Randomness Degree* parameter, which gives the probability that $Dval[Did{=}k][Vno{=}i]$ is not equal to $Dval[Did{=}k][Vno{=}i{+}1]$. For instance, *Randomness Degree=0* means that $Dval[Did{=}k][Vno{=}i]{=}Dval[Did{=}k][Vno{=}i{+}1]$ for any i.

Obviously, the smaller degree of randomness the higher is the gain of this scheme of broadcast. Hence we can expect that the broadcast of the data having many "static" elements (for example, a cartoon clip with one-color background or a stock index of infrequently traded companies, etc.) may improve "density" of broadcast data. Naturally such compression works only in case we do have the data elements which do not change every time interval. In other words, the compression works if *Randomness Degree* is less than 1.

As an example, consider broadcast of the data with *Randomness Degree=0.1*. Then in average out of 100 versions we have 10 versions with the values different from the values of the previous versions and 90 versions repeating their values. It means that instead of broadcasting 100 data values we broadcast only 10. We can roughly estimate that overhead created by the auxiliary symbols will not exceed 1 symbol per "saved" data item from the broadcast. Assuming, one data item consumes 16 bytes and one auxiliary symbol consumes 1 byte, the gain is 100*16/(10*16+90*1)=6.4, which corresponds to 84% reduction of the broadcast length. Similarly, the broadcast shrinks about 45%, for *Randomness Degree=0.5* and about 9%, for *Randomness Degree=0.9*. These numbers do not depend on whether broadcast is Vertical or Horizontal. However, the system behavior can in fact depend on it, because the performance depends on when the desired data

is read. In the example presented, if a client wants to find a data element with *Did=3* and *Vno=0*, the Vertical broadcast reads only 3 data before it hits, and the Horizontal broadcast reads 6 data elements. It is easy to find the opposite example, so a question arises: *Which organization is more preferable?*

We would expect that different strategies would be more appropriate for different applications. If users require different versions of a particular data (for example, the history of a stock index change), the horizontal broadcast is preferable. If users need the most recent data (for example, the current stock indexes), the vertical broadcast is supposed to be more efficient. In our experiment, we study the performance of these two broadcast strategies under different workload scenarios, that is client behaviors.

4 Client Access Behavior

Clients may have different tasks, and the way a client searches for data depends on the task. The first way, called the *Random Access*, is used when a client wants a randomly chosen data element. In this case the client requests pairs of random *Did*s and random *Vno*s. The second way, called the *Vertical Access*, is used if a client needs a specific version of some data elements. In this case, the client requests one specific *Vno* and a few *Did*s, so all required data belong to one version. The third way, called the *Horizontal Access*, is used if a client wishes different versions of a specific data item. Then the client requests one *Did* and a few corresponding *Vno*.

The client does not always know the data elements and their versions in advance and a particular choice of data may depend on the value of the previously found data. We call this type of client *dynamic search client* (in contrast, we call *static search client* a client whose all its data needs are known before first tuning into the broadcast). For dynamic search client, it is also possible to have three different access patterns: Vertical, Horizontal and Random. For Vertical one, the client requests a data item and its version. When found, it requests another element of the same version. For Horizontal access, the client requests another version of the same data item. For Random access, the client requests a new data item and a new version every time.

In all the cases, a dynamic search client may find the new data element either within the same broadcast as the previous data element or, with probability 50%, it will need to search for the new data element in the next broadcast. In general, in order to find n data elements, a dynamic search client needs to read roughly $2n/3$ broadcasts. In other words, the access time for all access patterns depends on number of broadcasts necessary for finding the elements. This is in contrast with static search client where the access time is determined by the order the data values are read within the same broadcast. As a result, all Random, Vertical and Horizontal access patterns have roughly the same access time for dynamic search client and so, the access pattern is not important anymore for the selection of the type of the broadcast organization. Therefore, in our experiments we consider only the behavior of static search clients with predetermined data needs.

5 Experimental Testbed

The simulation system consists of a broadcast server which broadcasts a specified number of versions of a set of data items, and a client which receives the data. The number of data items in the set is determined by the *Size* parameter and the number of versions by the *Versions* parameter. The communication is based on the client-server mechanism via sockets. For simplicity the data values are integer numbers from 0 to 9.

The simulator runs the server in two modes, corresponding to the two broadcast organizations, namely Vertical Broadcast and Horizontal Broadcast (determined by the *Bcast Type* parameter). The broadcast could be either Compressed or basic Sequential (determined by the *Compression* parameter). The server generates broadcast data with different degree of randomness (from 0 to 1), which is determined by the parameter *Randomness Degree* (the definition of Randomness Degree was given in Section 3). The client searches the data by using three different access types: Random, Vertical and Horizontal (determined by the *Access Type* parameter).

The client generates the data elements it needs to access (various versions of data items) before tuning into the broadcast. The parameter *Elements* determines the number of the data elements to be requested by the client. For the Random access, the data items and their versions are determined randomly to simulate the case when all versions of all data items are equally important for a client. For the Vertical and Horizontal accesses, the requested data elements are grouped into a number of strides (determined by *StrideN*), each containing l elements (determined by *StrideL*). (Clearly, *StrideL*StrideN* = *Elements*.) For example, if *StrideN=2* and *StrideL=5*, for Vertical access, the client searches for two versions (determined randomly with uniform distribution) of 5 consecutive data elements. For Horizontal access, the client tries to find 5 versions of 2 data items (determined randomly with uniform distribution).

The client may tune in at any point in the broadcast, but it starts its search for data elements at the beginning of the next broadcast. Thus, if a client does not tune in at the beginning of a broadcast, it sleeps to wake up at the beginning of the next broadcast which is determined by the next broadcast pointer in the header of each bucket. A client reads a broadcast until all the desired data elements are found. In this way, it is guaranteed that the desired data elements are found within a single broadcast. While the client is reading, it counts the number and type of characters it reads. This can be converted into *Access Time* – the time elapsed between the time the client starts its search and until it reads its last requested data element, given a specific data transmission rate. In our study, access time is the measure of performance for both response time and power consumption (recall we do not consider selective tuning in this paper, hence a client stays in active mode throughout its search). The smaller the access time, the higher the performance and the smaller the consumption of energy. We assume that the auxiliary characters (#, =, ^ , V, annotations) consume one time unit and the data elements may consume 4, 16, 64 etc. time units, depending on complexity of the data. The *Length* parameter is used to specify the size of

Parameter	Values
Compression	Basic Sequential broadcast Compressed broadcast
Bcast Number	Number of broadcasts
Bcast Type	Vertical broadcast Horizontal broadcast
Size	Number of data items
Versions	Number of versions
Randomness Degree	0–1, (0: all versions have the same value, 1: versions are completely independent)
Length	Size of a data element (size of an auxiliary symbol is 1)
Elements	Number of the requested data items
Access Type	Random access Vertical access Horizontal access
StrideN	Number of the strides for Vertical/Horizontal accesses
StrideL	Length of the strides for Vertical/Horizontal accesses
Tries	Number of the same experiments to reduce deviations

Table 1. Simulation Parameters

data element. In the experiments reported in this paper, we have chosen *Length* to be 16, which may correspond to 16 bytes.

In order to estimate confidence intervals we performed the measurements 80 times (parameter *Tries*). Then we calculate the average access time and the corresponding standard deviation which are shown in our graphs. The discussed parameters are summarized in Table 1.

6 Performance Results

In this section, we report on the results of our experiments that demonstrate the applicability of our proposed two broadcast organizations and the advantages of our compression technique.

The results presented in Figure 1 to Figure 3 are obtained for the Vertical Broadcast organization and Random Access of the client.

As mentioned before, effectiveness of the Compressed Broadcast may depend on size of the data elements on the broadcast (represented by *Length* parameter). Figure 1 (*Size=90, Elements=5, Tries=80, Randomness Degree=0.5, Vertical Broadcast, Random Access*) shows dependence of the access time on the size of the data item for the Compressed and the Sequential server broadcasts. It is quite obvious from the figure that the compression reduces the client's access time about 50% for any size of the data. (This can also be seen in Figure 2 for Randomness degree=0.5) The greatest gain in terms of absolute access time occurs for the largest data sizes.

The main contributor to the performance improvement of compressed broadcast over a simple sequential broadcast may be how often we can save time by

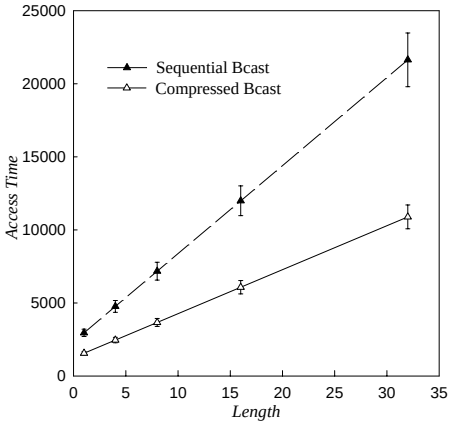

Fig. 1. Dependence of the access time on the size of the data item

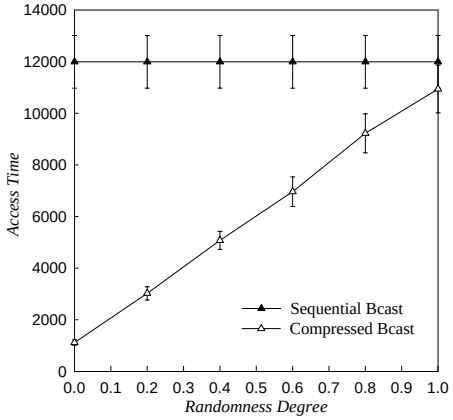

Fig. 2. Compression performance for different Randomness Degree

not broadcasting a data element of a certain version if it has the same value as the data element of the previous version. Intuitively and from simple estimation we may see that the smaller *Randomness Degree* is, the greater gains are. Figure 2 (*Size=90, Elements=5, Tries=80, Length=16, Vertical Broadcast, Random Access*) confirms our estimations and the performance dependence of the compression on the *Randomness Degree*. For *Randomness Degree=0.0* one can observe about 10 times improvement. When the versions become more different, the performance of the compressed broadcast worsens, getting close to that of the sequential broadcast as *Randomness Degree* approaches 1.

We should note that in the worst case (absolutely uncorrelated versions) we could expect that overhead from the auxiliary information would degrade the performance of our optimization. However, it has not been observed in any of our simulation experiments. This is because we used simple data type and even with *Randomness Degree=1* some data elements have the same values for adjacent versions. This happens because *Randomness Degree* determines only the probability that two version values are not correlated but does not guarantee that they are different. The experiment shows that the proposed compressed scheme works best if the data do not change from one version to another *every* time interval. However, even if they do change, the compressed broadcast just converts to a simple sequential broadcast. The auxiliary symbols overhead is so small that the fact that even at *Randomness Degree=1* there exist some data items for which the data values are the same for adjacent version numbers (and so, we still have some minimal compression) is enough to have some minor performance improvement. This is a situation to be expected in reality.

The dependence of the *Access Time* on the number of elements requested (given by *Elements* parameter) is shown in Figure 3 (*Size=90, Randomness Degree=0.5, Tries=80, Length=16, Vertical Broadcast, Random Access*). We can see that at the beginning the increase of the number of elements requested leads

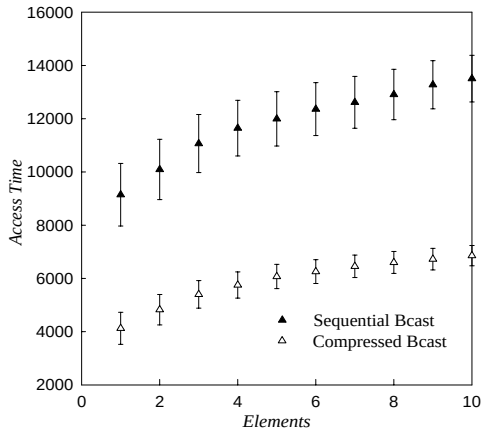

Fig. 3. Performance for different number of searched elements

to significant increase of the access time, but later (for *Elements* higher than 4) the access time increases more slowly. This behavior does not look very surprising if we consider the access time as the time needed to search from the beginning of a bcast to "the furthest data element". The other requested date elements are "in between" and are "picked up" on the way. As *Elements* increases, the place where the last searched element was "picked up" shift towards the end of the broadcast string, making the *Access Time* "saturated". The important feature is that the absolute difference between the access time for the compressed broadcast and the sequential broadcast is the biggest for *Elements* higher than 4. However, the relative difference stays approximately the same (about 2 times).

The results presented in Figure 1 to Figure 3 are obtained for the Vertical Broadcast and Random Access only, but qualitatively the tendencies mentioned are valid for all other broadcast organizations and access schemes. Figure 4 and Figure 5 show the differences between these schemes.

Figure 4 (*Size=10, Elements=20, Tries=80, Length=16, Vertical Broadcast*) shows the dependence of the *Access Time* on *Randomness Degree* when the server uses the Vertical Broadcast, whereas Figure 5 (*Size=10, Elements=20, Tries=80, Length=16, Horizontal Broadcast*) when the server uses the Horizontal Broadcast. The main conclusion from Figure 4 is that for the Vertical Broadcast the most efficient access scheme is the Vertical Access and the worst is the Horizontal Access (about 1.5 times worse than the Vertical Access). The Random Access is somewhere in between (about 1.4 times worse than the Vertical Access) closer to the Horizontal Access. Figure 5 shows the opposite results. The best scheme for the Horizontal Broadcast is the Horizontal Access, and the worst is the Vertical Access (about 1.4 times worse than the Horizontal Access).

These results are valid for both Compressed and Sequential Broadcasts. The interesting feature is that for small values of *Randomness Degree*, it is more important for the performance whether the broadcast is Compressed or Sequential

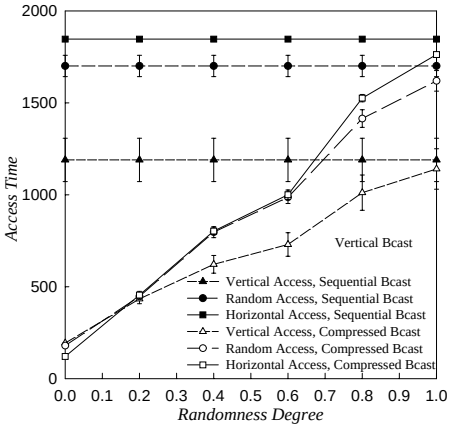

Fig. 4. Vertical broadcast at different Randomness Degree

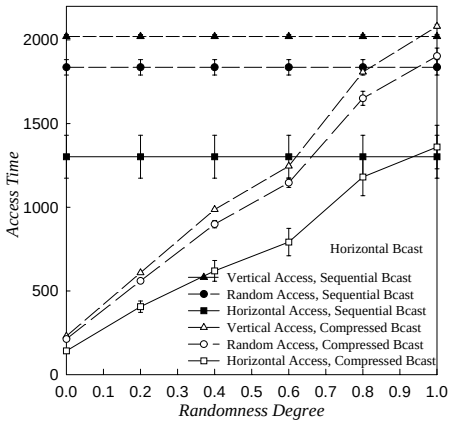

Fig. 5. Horizontal broadcast at different Randomness Degree

than whether the access scheme "corresponds" to the broadcast. We can see on the figures that for *Randomness Degree* less than 0.7 the *Access Time* for any access type is smaller in the case of the Compressed Broadcast. But for *Randomness Degree* higher than 0.7, there are cases when the Sequential Broadcast with "right" access scheme can beat the Compressed Broadcast with "wrong" access scheme. Hence, in order to have the best performance, the broadcast organization and access scheme should have "similar patterns", either Vertical Broadcast organization and Vertical Access, or Horizontal Broadcast organization and Horizontal Access.

7 Conclusion

In this paper we showed that besides the size of a broadcast, the organization of the broadcast has an impact on performance, as different kind of clients needs different types of data. We recognized three kind of clients applications based on their access behavior: "Historical" that access many versions of the same data, "snapshot" that access different data of the same version and "browsing" that access data and versions randomly. The performance of our proposed Compressed and basic Sequential, Horizontal and Vertical broadcast organizations was evaluated in terms of these three different kind of applications.

Specifically, if the primary interest of clients is "historical" applications, the best way to broadcast is the Horizontal Broadcast. If the primary interest of clients is "snapshot" applications, the best way to broadcast is the Vertical Broadcast. In case of mixed environment it is possible to create adaptive broadcast with no extra cost due to flexibility of the broadcast format.

The suggested compression technique does not require extra time for client side decompression and works for both Vertical and Horizontal broadcasts. The

auxiliary symbols overhead is small if the size of one data element significantly exceeds a few bits. The effectiveness of a compressed broadcast depends on the repetitiveness of the data. The less frequently data change, the better the gains are. But even in the worst case (completely random data), the Compressed broadcast does not exhibit worse performance than the Sequential broadcast.

Currently, we are evaluating the two broadcast schemes in the context of broadcast disks. Further, we are developing caching schemes that integrate with the different broadcast organizations.

Acknowledgments: This work was supported in part by the National Science Foundation award ANI-0123705 and in part by the European Union through grant IST-2001-32645.

References

[1] S. Acharya et al. Balancing Push and Pull for Data Broadcast. *ACM SIGMOD Conferences* (1997) 183–194

[2] S. Acharya, M. Franklin, S. Zdonik Disseminating Updates on Broadcast Disks. *22nd VLDB Conference* (1996) 354–365

[3] S. Acharya et al. Broadcast Disks : Data Management for Asymmetric Communication Environments. *ACM SIGMOD Conference* (1995) 199–210

[4] S. Acharya, M. Franklin, S. Zdonik Dissemination-based Data Delivery Using Broadcast Disks. *IEEE Personal Communications*, **2(6)** (1995) 50–61

[5] J. Jing, A. H. Elmargarmid, S. Helal, R. Alonso Bit-Sequences: An adaptive Cache Invalidation Method in Mobile Client/Server Environment. *ACM/Baltzer Mobile Networks and Applications*, **2(2)** (1997) 115–127

[6] T. Imielinski et al. Data on Air : Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, **9**, No. 3, (1997) 353–372

[7] C. Mohan, H. Pirahesh, and R. Lorie Efficient and Flexible Methods for Transient Versioning to Avoid Locking by Read-Only Transactions. *ACM SIGMOD Conference* (1992) 124–133

[8] E. Pitoura and P. K. Chrysanthis Scalable Processing of Read-Only Transactions in Broadcast Push. *19th IEEE Int'l Conf. on Distributed Computing Systems* (1999) 432–439

[9] E. Pitoura and P. K. Chrysanthis. Exploiting Versions for Handling Updates in Broadcast Disks. *25th VLDB Conference* (1999) 114–125

[10] J. Shanmugasundaram et al. Efficient Concurrency Control for Broadcast Environments. *ACM SIGMOD Conference* (1999) 85–96

[11] S.W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing (1997)

[12] K.-L. Wu, P. S. Yu, M.-S. Chen. Energy-Efficient Mobile Cache Invalidation. *Distributed and Parallel Databases*, **6** (1998) 351–372