5 Conclusions

We have studied organization and access for broadcast data, taking a stand that periodic broadcasting is a form of storage. Data is stored "on the air" with the latency of access proportional to the duration of the *bcast*. Broadcasted data can be reorganized "on the fly" and refreshed (reflecting updates) between two successive *bcasts*. The main difference with the disk based files is that we need to minimize two parameters (access time and tuning time) contrary to just one (access time) for the disk based files. While broadcast tuning time roughly corresponds to disk access time, the broadcast access time is an equivalent of the disk space taken by the file.

We investigated two data organization methods namely *Hashing* and *flexible indexing* and we have demonstrated the relative advantages of both the schemes. In [3] which is orthogonal to this paper we also study data organization methods based on different types of indexing called $(1, m)$ *indexing* and *distributed indexing*. Distributed Indexing is better than any Hashing scheme for small key sizes. We provided evidence in [4] that hashing performs better then distributed indexing in case the key sizes are large.

There are a number of research questions which have to be investigated. If filtering has to be done by complex predicate matching then more sophisticated data organizations techniques are needed. In particular we would like to investigate further techniques of secondary indexing. There are a number of communication issues which have to be looked at in detail. How to achieve reliability of the broadcast in error prone environments such as wireless cellular? Since the clients are only listening there is no (or very limited) possibility of the acknowledgment. Multiple Access protocols which guarantee timely delivery of information are necessary for the broadcasting (and especially the directory) to work correctly. Finally, we view data broadcasted on the channel as another level of storage hierarchy, where data is literally "stored on the channel". This view allows us to look at data in a uniform way regardless of whether it is stored in one location or multicasted on the network.

References

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

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

[3] T. Imielinski, S.Viswanathan and B.R. Badrinath, "Data on the Air - Organization and Access," Tech Report, DCS - Rutgers University – Oct 93.

[4] T. Imielinski, B. R. Badrinath and S.Viswanathan, "Data Dissemination in Wireless and Mobile Environment," Winlab – Rutgers University, Oct 93.

[5] David Gifford, John Lucassen, and Stephen Berlin, "The application of digital broadcast communication to large scale information systems," IEEE Journal on selected areas in communications, Vol 3, No. 3, May 1985, pp.457–467.

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

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

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

As the capacity of buckets (n) increases the access time due to flexible indexing increases very slowly and the access time due to hashing increases quite fast. The difference between the access times due to flexible indexing and due to hashing is quite big for large values of p. Another important parameter that affects the access time is the capacity of the buckets i.e., the number of tuples a bucket can hold. Below a threshold of the capacity of buckets the access time of flexible indexing doesn't perform very well. This threshold is when $n/2 < \lceil \log_2 p \rceil + m$, where m is the number of data subsegments per data segment.

The graph in figure 5(Right) illustrates the point we made above. The number of data buckets ($File$) considered is 1024 and p, the number of data segments, is 100 (i.e., the overflow size is 10). The x-axis represents the values of n, the capacity of buckets and the y-axis the access time. When the value of n is small $n/2 < \lceil \log_2 100 \rceil + m$ [15], then hashing performs better than flexible indexing. For large values of n flexible indexing is better in terms of the access time. In the figure $\forall n, 1 \leq n < 16$, $(16/2 \not< (7+1))$ the access time due to hashing is better and $\forall n, 16 \leq n < 25$, $(25/2 \not< (7+5))$ both the methods are comparable in terms of their access times and and $\forall n, 25 \leq n$ flexible indexing performs better.

PRACTICAL IMPLICATIONS

Consider a stock market data of size 128 $Kbytes$ that is being broadcasted in a channel of bandwidth 20 $Kbps$. Let the packet length be 128 $bytes$. It takes around 50 seconds to broadcast the whole file and 0.05 seconds to broadcast or tune into a single packet. 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.

The $tuning\ time$ as well as the $access\ time$, if no indexing is used is 25 $seconds$ (half of the $bcast$ time). With $perfect\ hashing$ we will have to tune into three buckets, resulting in a $tuning\ time$ of 0.15 $seconds$. Thus the battery life is increased by 165 folds. But then the penalty is that the $access\ time$ doubles to 50 seconds.

If we use $hashing\ B$ with 20 logical buckets, then the $tuning\ time$ is 53 buckets i.e., 2.65 $seconds$. Thus the battery life increases by almost 10 times at the cost of the $access\ time$ increasing by just 1%. On using $flexible\ indexing$, for the same $access\ time$ we can do by just tuning into just 6 ($\lceil log_2 20 \rceil + 1$) buckets. The $tuning\ time$ is 0.3 $seconds$, thus the battery life increases by more than 80 times.

RESULTS

Our results can be summarized as :

- If the access time is of importance (of the two parameters) then if $n/2 < (\lceil \log_2 p \rceil + m)$ then use Hashing. i.e., if the capacity of the buckets are large use Flexible indexing else use Hashing.
- If the tuning time is of importance and if $Data > 2 * p * \log_2 p$ then use flexible indexing i.e., if the overflow is small use hashing else use flexible indexing.
- Hashing schemes should be used when the tuning time requirements are not rigid and when the key size is relatively large compared to the record size [3].
- Indexing schemes should be used when tuning time is restricted and the key size is relatively small compared to the record size [3].

[15] as specified before, $m = min((n/2 - \log_2 p), (L\_data\_segment/2))$ when $(n/2 - \log_2 p) > 0$ else $m = 1$

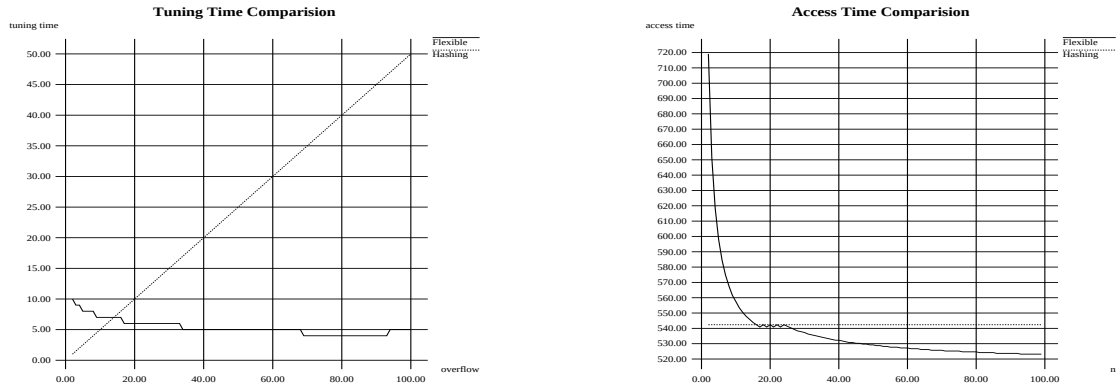

Figure 5: Comparison of Tuning Time & Access Time

4.2 Comparison of Hashing and Flexible Indexing

In the comparison we will consider a very good hashing function which will give us equal overflow (of length *over*) for all the values i.e., the minimum overflow and the average overflow are the same. Let the hashing function have p logical buckets i.e., the keys will be mapped into some number between 1 through p. To have a fair comparison we assume that *over* and the length of the data segment ($L\_data\_segment$) are the same. The hashing function divides the data buckets into p parts (each of length $L\_data\_segment$), as the overflow is the same for all the logical buckets.

The access time using hashing in this case is
$0.5 * (over) + 0.5 * (File + p/2)$ that is

$$0.5 * ((File + p/2) * (1 + p)/p)$$

and the average tuning time is $0.5 * (over)$ i.e.,

$$0.5 * (L\_data\_segment)$$

Let us first compare the tuning time using the two schemes. The tuning time due to flexible indexing is better when

$$(0.5 * (L\_data\_segment)) \ > \ (\log_2 p + Data/(2 * p * (m + 1))) \text{ i.e.,}$$

$$File \ > \ (2 * p * \log_2 p * m/(m + 1) - (\log_2 p + m) * p/n)$$

where the number of tuples in the *local index*, $m = min((n/2 - \log_2 p), (L\_data\_segment/2))$ when $(n/2 - \log_2 p) > 0$ else $m = 1$.

Figure 5(Left) illustrates a comparison of the tuning times using flexible indexing and hashing. The x-axis represents the length of *over* (overflow size which is equal to the size of the data segment) in terms of number of buckets. The number of data buckets in the file is 1024. The capacity of a bucket, n, was assumed to be 100. As the size of *over* increases, the number of data segments (p) decreases. In the hashing scheme the tuning time is half the data segment size and this grows linearly with the increase in the size of data segment. But in case of flexible indexing the tuning time grows very slowly and more over it is not a monotonically increasing function. Notice that when the size of data segment is less than 14 buckets (i.e., $Data < 2 * p * \log_2 p$) the tuning time using flexible indexing is larger than that of the hashing scheme. On the other hand, for for the sizes of data segment of more than 14, flexible indexing is a clear winner.

Let us now consider the access time comparison. Flexible indexing gives a better (lesser) access time when the following is true:
$0.5 * (File + p * (\lceil \log_2 p \rceil + m)/n) * (1 + p)/p \ < \ 0.5 * ((File + p/2) * (1 + p)/p)$

$$i.e., when \qquad (\lceil \log_2 p \rceil + m) \ < \ n/2$$

- Search through the *local index* to see if K is greater than or equal to the first field of each tuple, if the answer is positive follow the pointer of the first such tuple, tune in at the designated bucket and proceeds as in (iii)

- (iii) Search the next $(L\_data\_segment/(m+1))$ buckets, sequentially to locate K, where $L\_data\_segment$ denotes the length of a data segment.

Let us consider an example to illustrate the above protocol. Consider Figure 4 and let the key we are looking for be 54 and let the initial probe be made at bucket 20. The offset at bucket 20 will direct us to the beginning of the next data segment, in this case it is 25. The client tunes in at bucket 25. It checks to see if the query key (54) is lesser than 25, the answer is in negative. So the next tuple is checked. $54 > 49$ and hence the pointer 25 is selected. The client now tunes in after 25 data buckets to bucket 49. Searches through the control index in bucket 49, the comparison against the fourth tuple is successful ($54 > 53$) and hence the client tunes in at the fifth bucket. Then it searches sequentially through buckets 53 and 54 to find K.

4.1 Analysis

The tuning time[12] using the flexible indexing technique is : $\lceil \log_2 p \rceil + L\_data\_segment/(m+1)$ in the worst case. In general, the average tuning time is $\lceil \log_2 i \rceil + L\_data\_segment/2 * (m+1)$ where i is the number of data segments in front of this data segment (including this one). This is because we require (atmost) $\lceil \log_2 p \rceil$ tunings after the initial probe to get to the relevant data segment. Once we get to the data segment that has the search key, then on an average we have to search $Data/(2*(m+1)*p)$ buckets sequentially [13]. Thus on an average the tuning time is

$$(\sum \lceil \log_2 i \rceil + Data/(2*(m+1)))/p$$

$\forall i \; 1 \leq i \leq p$ and $Data$ being the size of the entire broadcast [14], thus upper bound on the average tuning time is

$$(\lceil \log_2 p \rceil + Data/(2*p*(m+1)))$$

Now, let us analyze the access time. Let the time required to get to the first bucket of the next data segment, on making an initial probe be called *probe wait*. On an average the probe wait is half the size of $L\_data\_segment$. After coming to the first bucket of the next data segment the client has to wait half the size of the *bcast* size ($Data$), on an average. This wait is called the *data wait*. The access time is the sum of the above two *waits*. Hence the access time is $0.5*(L\_data\_segment)+0.5*(Data)$ i.e.,

$$0.5 * Data * (1+p)/p$$

Let the size of the raw file (without the control index) be $File$. The total space(in terms of the number of buckets) occupied by the control index is
$(\sum \lceil \log_2 i \rceil + p*m)/n$, $\forall i \; 1 \leq i \leq p$
that is, $p*(\lceil \log_2 p \rceil + m)/n$ is an upper bound for the number of additional buckets due to the control index. Where n is the number of tuples that a bucket can hold.

The size of the final file (after the control index is added) is $Data$ and
$Data = File + p*(\lceil \log_2 p \rceil + m)/n$ Thus the access time using flexible indexing is :

$$0.5 * (File + p*(\lceil \log_2 p \rceil + m)/n) * (1+p)/p$$

[12] we will ignore the initial probe and another first probe (if the former resulted in going to the next *bcast*) in the following discussion to convey the formula without confusion

[13] The half of the size of the data segment and $L\_data\_segment = Data/p$

[14] Data is the collection of all the data buckets and all of the control index

instance, the index in the data bucket #25 specifies that for all key values which are smaller than 25 the client has to tune to the bucket that is 43 buckets away which is bucket # 1 of the next *bcast* (in this case the client simply missed the key and has to wait for the next broadcast). If the key is larger than 49 then the client has to tune again to the data bucket #49 (which the offset #= 25 indicates). This data bucket will provide an index to help the client further on. Similarly, if the searched key is larger than 33 (but not larger than 49) then the client should tune again 9 buckets ahead where he will use the index at the bucket 33. Not all data buckets contain the index though. Notice that if we go "further down" the index for the data bucket #25, we reach the tuple with (first field) 31. Here if, the key is between 31 and 33 then the client will not search the tuples that follow and will tune to the buckets between 31 and 33 in search of the key. Notice that the index information is *distributed* between different data buckets.

Formally, the control index can be divided into two parts: the *binary control* index and the *local index*. The *binary control index* and the *local index* together will be called *control index*. The control index consists of tuples. Each tuple has two fields. The first field is a key for a data record and the second field is a pointer to the data bucket containing that record. By a pointer we mean an $offset$ value, which denotes the relative position of the data bucket from this bucket. Each bucket has an $offset$ to the beginning of the next data segment.

The *binary control index* has $\lceil \log_2 i \rceil$ tuples. Where i is the number of data segments in front of this data segment (including this one).

- (a) The first tuple consists of the key of the first data record in the current data bucket and an offset to the beginning of the next broadcast
- (b) The k th tuple consists of the key of the first data record of the $\lfloor \log_2 i/2^{k-1} \rfloor + 1$ th data segment followed by an offset to the first data bucket of that data segment.

The *local index* consists of m tuples (m is a parameter which will depend on number of tuples a bucket can hold, the access time desired etc). The *local index* further partitions the data segment into $m+1$ data *subsegments* D_1, D_2,...,D_{m+1} and consists of the following m tuples :

- (a) The first tuple consists of the key of the first data record of the D_{m+1}. The specified offset points to the first data bucket of D_{m+1}.
- (b) The k th tuple consists of the key of the first data record of the D_{m+1-k} followed by an offset to the first data bucket of D_{m+1-k}.

The first bucket of each data segment stores control index as well as (if space allows) data records. The access protocol for a record with key k is as follows :

- (i) Make an initial probe and get the offset to the beginning of the next data segment and go into doze mode
- (ii) Tune in again to the beginning of the designated next data segment.
  - If the search key K is lesser than the first field of the first tuple in the *binary control index* then (the record has been missed) doze till the offset given by the second field and proceed as in step (ii) (else)
  - Search through the rest of the *binary control index* from the top to the bottom of the index to see if K is greater than or equal to the first field of each tuple, if the answer is positive follow the pointer of the first such tuple and proceeds as in step (ii) (else)

---

this one)

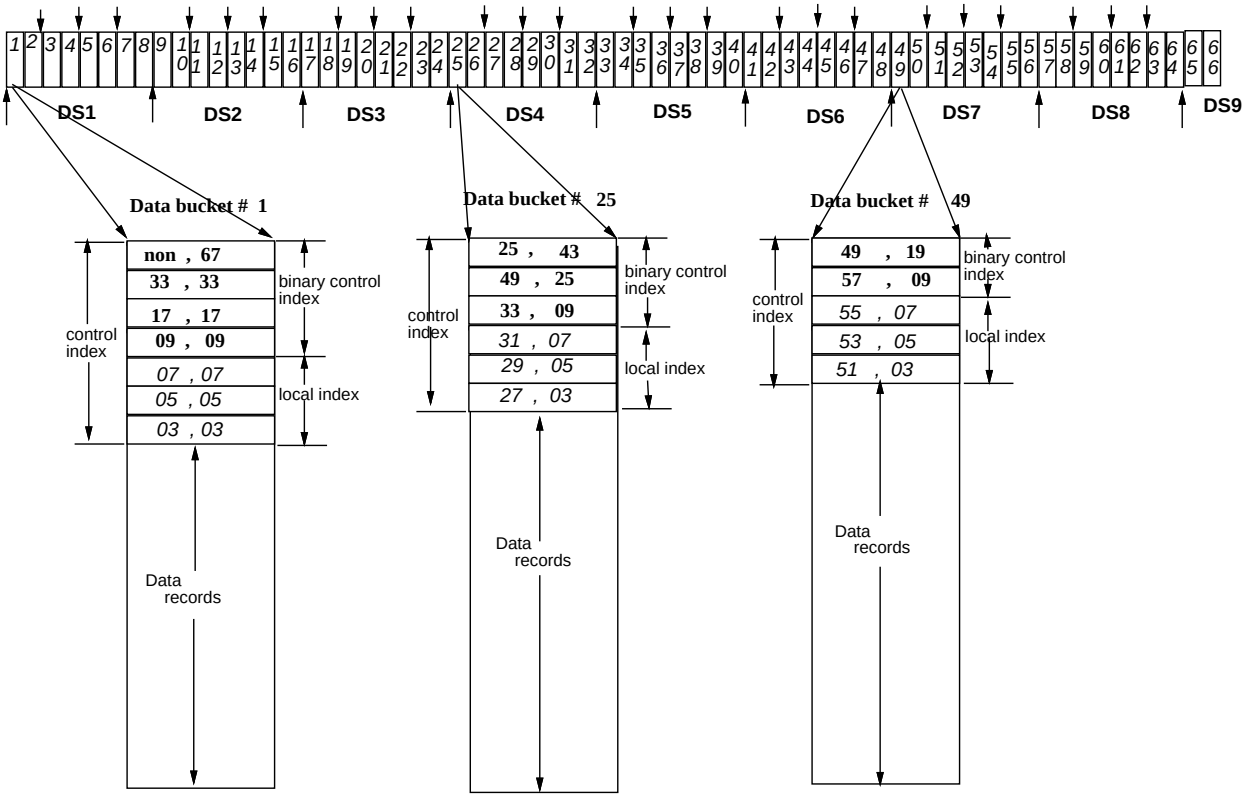

Figure 4: Example for Flexible Indexing

time.

4 Flexible Indexing

We are now going to explore the fact that the file is completely known to the server prior to the broadcast. Notice that hashing divides the file into p logical buckets of statistically varying size. The sizes of each individual logical bucket vary and depend on the hashing function itself. Rather than using hashing we can simply divide the file into p segments and provide some indexing to help the user "navigate" and reduce the tuning time. The parameter p will make the indexing method proposed in this section *flexible* since depending on its value we will either get very good tuning time or very good access time.

We will assume that the data records are sorted in ascending (or descending) order. We divide the set of data buckets into p parts [8]. The data segments are numbered 1 through p. The first bucket in each of the data segment will contain a control part consisting of the *control index*. The control index is a binary index [9] which, for a given key K helps to locate the data bucket which contains that key.

Let us begin describing the technique with an example. Figure 4 shows a set of 66 data buckets[10]. Let $p = 9$, with the length of all but the last of the data segments be 8 buckets. The first bucket of each data segment contains an index. Three such buckets are illustrated in Figure 4. Each index entry is a pair which consists of the key value and the *offset* [11] showing the client when to tune again to find the searched key. For

[8] all but with the possible exception of the last, of the these parts are equal in length

[9] There is no other specific reason beyond simplicity for keeping the index binary

[10] the bucket number is also the key of the first record in that bucket

[11] as explained earlier $offset$ denotes the relative distance of the bucket from this bucket (including

Buckets	Physical.Buckets	Av. Overflow	Access Time	Tuning time
2	1025	513	770	258
4	1026	257	642	130
16	1032	65	549	34
32	1040	33	537	18
64	1056	17	537	10
128	1088	9	549	6
512	1280	3	642	3
2048	2048	1	1025	2

Table 1: Access time for Hashing Scheme B

the perfect hashing function which gives 1024 logical buckets (which corresponds to 2048, physical buckets) gives the access time (1025) which is far from optimal, thought the tuning time (of 3) was the minimum for this case. In fact the optimal access time is provided by the hashing function with 32 or 64 logical buckets, but here we compromise on the tuning time which is 18 or 10 times as much (respectively). This is a consequence of the small number of physical buckets. Notice that the lower level hashing functions have even smaller number of buckets but then the data miss probability due to hitting the first time the right logical bucket is too high (if this happens we have to wait for the next broadcasted version). Thus the higher the number of logical buckets the lower the tuning time. For lower number of logical buckets the access time is also high as the data miss is high.

Hence, *perfect hashing function does not provide the minimal access time for the broadcasted files*

In fact the minimal access time is achieved by the hashing functions with significantly smaller number of logical buckets than the perfect hashing function. Notice that the tuning time goes asymptotically down to 3 data accesses when the number of logical buckets grows (the minimum is achieved for the perfect hashing function).

There are a number of possible other hashing schemes which can be used as well for the broadcasted files, however in our view the hashing scheme presented above provides us with the best access time/tuning time profile. For instance, grouping all overflow buckets at the end of the file with each logical bucket having a pointer to its overflow area leads to alternative hashing scheme. Its access time will be comparable to the first of our hashing schemes since the data miss behavior in this case will be equivalent to the first of our schemes. The second scheme will provide better access time due to the reduced data miss.

Notice that hashing is a *flexible* method of organizing broadcasted data. Indeed, if we have more leverage in terms of the tuning time we can improve the access time by simply reducing the number of logical buckets (thus by increasing the overflow).

However, the price to be paid in case we want to minimize the tuning time in terms of the access time increase is rather high for hashing methods considered here (perfect or near perfect hashing increases the size of the broadcast and consequently the access time). A natural question is whether can improve the tuning time of hashing method without paying a high access time penalty same level? One way of achieving this goal is to provide a hybrid scheme mixing hashing and indexing: use hashing to reach the right logical bucket and then use indexing within each logical bucket to reach the right physical bucket. Statistically, there is some empty space in the "last" physical, overflow bucket of each logical bucket so we could *reuse* this extra space to store this local index. Unfortunately, the amount of this empty space is a statistical variable and cannot be guaranteed - hence additional indexing would almost surely increase the overall size of the broadcast and add to the access time.

In the next section we show a method which is not based on hashing but which can maintain its good access time characteristics while significantly lowering the tuning

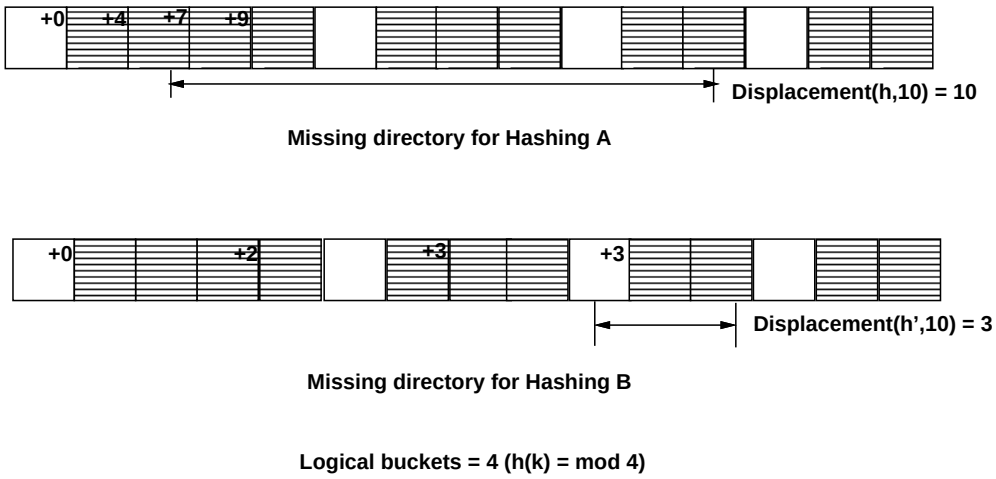

Figure 3: Displacement Comparison

The expected access time for the Hashing B can be calculated as follows. Let h' be the modified hashing function of h. By *Displacement*(h, K) we denote the difference between the address of the physical bucket where K resides (*Physical_bucket(k)*) and the designated bucket for k ($h'(k)$)computed as:

$Displacement(h, K) = Physical\_bucket(K) - h'(K)$

The expected access time is computed by calculating for each key K in the file, the access time "per key" and then averaging it out. The expected access time "per key" is the combination of two factors:

- If the initial probe is in the part of the broadcast between the designated bucket and the physical bucket of the key, then the data miss occurs despite of the fact that the key is still ahead in the current broadcast. Thus, the unit has to wait an extra revolution. This is calculated as[7]

  $(Displacement(h, K)/Data(h)) * (Data(h) + 1/2 * Displacement(h, K))$

- If the initial probe is outside of the displacement area then:

  $(1 - (Displacement(h, K)/Data(h))) * (Data(h) + Displacement(h, K))/2$

  Since in this case the unit has to wait, on average between the *Displacement*(h,K) and the file size.

The expected access time is computed as a sum of expected access times per key divided by the total number of keys in the file (again assuming that all query requests refer to the records in the file)

Figure 3 illustrates the source of the basic advantage of the hashing scheme B over the hashing scheme A. Given the same file and the same key K (K=10), the probability of the directory miss for the *Hashing B* is much smaller than that for *Hashing A*. This probability is proportional to the value of $Displacement(h, K)$ which is shown for both schemes and is demonstrated to be much smaller for the Hashing B. In both the Hashing schemes, getting a good hashing function depends on the distribution of the keys.

Table 1 illustrates how the expected access time and the tuning time depend on the hashing function used. The table shows the average access time and the average tuning time for the situation when there are 17 records per bucket (each record with 30 bytes of length). The size of "control part" is 24 bytes(per bucket). The size of the data file is 1024 buckets hence the total number of data records is 17408. Notice that

[7] *Data(h)* is the size of bcast for the given data file, for a given hash function h

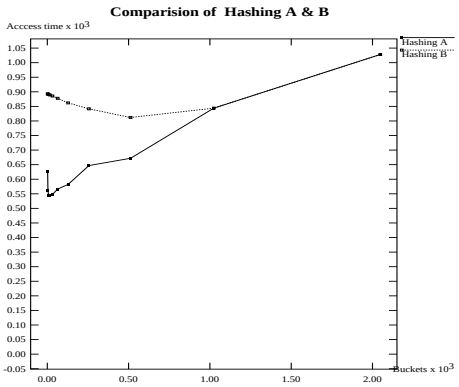

Figure 2: Comparison of Hashing A & Hashing B

Given the file, what is the hashing function minimizing the access time? For the disk based files the best function is the perfect hashing function (the one with no overflow). For the broadcasted files this is not the case. This critical difference comes from the fact that the total number of broadcasted buckets has immediate impact on the access time (the more one has to wait for the next version of the file). The perfect hashing function does not minimize the number of physical buckets necessary for the file bcast. On the contrary, the more overflow buckets are used, the smaller the total number of broadcasted buckets is. Indeed, the more overflow buckets the file has, the lesser "half-empty" buckets are broadcasted and this consequently results in better bucket utilization. The smaller the overflow area the lower is the tuning time which reaches minimum for the perfect hashing function. This further shows the basic differences between the file organization for broadcast and file organization for the disk storage. Hashing based scheme for the broadcasted file displays the *random access* behavior for the tuning time and the *sequential access* behavior for the access time (when the size of the file matters). Access schemes for the disk based files are only characterized by one parameter - the access time. Two parameters: tuning time and the access time are needed for the broadcasted files and the behavior of the access time for broadcasted files is drastically different from the access time for the disk based ones. For example, the perfect hashing function is not always "perfect" for the broadcasted files.

3.2 Hashing B

Hashing A can be further improved if we notice that the directory miss phenomenon may be significantly reduced with a minor modification to the hashing function. Assume that d is the size of the minimum overflow chain. We can now modify the hashing function h to

$$h'(K) \begin{cases} h(K) & \text{if } h(K) = 1 \\ (h(K) - 1)(1 + min\_overflow) + 1 & \text{if } h(K) > 1 \end{cases}$$

and leave the rest of the scheme unchanged. In this way, the new hashing function h' takes under consideration the shift introduced by the overflow (taking under consideration the value of the minimum overflow).

Figure 2 illustrates the improvement, which is because the probability of a data miss is significantly reduced in case of *Hashing B*. If the sizes of the overflow chains per logical bucket do not differ much, then the reduction is substantial [6]. In the extreme case, when all buckets have the same size of the overflow chain, h' turns into the perfect hashing function for the file whose bucket size is increased by $(1 + d)$ times.

[6] the hashing function over here is a uniform hashing function

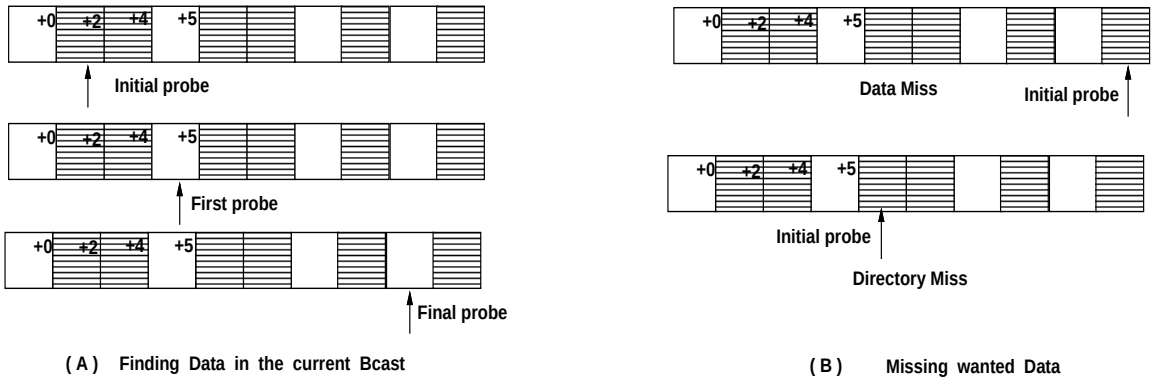

Figure 1: Hashing A

We will now present two hashing protocols: *Hashing A* and *Hashing B*. *Hashing B* will be an improvement of *Hashing A*. *Hashing B* will require maintaining some additional information.

3.1 Hashing A

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

- Probe the current bucket, read the control data from it and calculate $h(K)$.
- IF *Current bucket #* $< h(K)$
  —THEN Go into *doze mode* and listen again to slot $h(K)$
  —ELSE wait till the beginning of the next *bcast*, repeat protocol again
- At $h(K)$, get the *Shift*, go into *doze mode* to wake up after *Shift* number of buckets
- This time listen until either the record with key K is found (successful) or with key L *($h(L) \neq h(K)$)* is found (failure)

Figure 1(A)[5] illustrates a simple scenario of locating a key K = 15 in the file, hashed Modulo 4. Assuming that the initial probe takes place on the second physical bucket, the client reads the hashing function from the control part of the that bucket and proceeds with the first probe to the bucket number of four which, in case there is no overflow would contain the key K = 15, if it exists in the file. Since there is overflow, the keys are shifted and the client reads the value of the shift (5) and goes into the doze mode, and tunes in, in the final probe to the beginning of the "logical bucket" number four (to conclude whether the key K =15 is present it must look at all the overflow buckets of that bucket).

Figure 1(B) illustrates the cases when the client has to wait until the next broadcast in order to locate the given key K. This may occur either due to the *data miss* or due to the *directory miss*. In the data miss scenario the client's initial probe comes after the bucket containing the key which he wants. In the directory miss case, the client's initial probe comes before the bucket containing his key but *after* the bucket which contains a proper pointer (in this case the physical bucket number four). In such a case there is no way for the client to find out that the key which he is looking for is still to be broadcasted and the client has to wait until the next bcast.

[5]In all the figures, the buckets which are fully white denote the logical buckets (by a *logical bucket* of k, we mean the $h'(k)$ bucket) and the ones with bars denote the overflow buckets (of the preceding white bucket). The numbers in the right hand side top corners of some buckets denote the shift value in the control part of that bucket.

the bucket to which the pointer points to. The actual time of the broadcast for such a bucket will be calculated by multiplying ($offset$ - 1) by the time necessary to broadcast a single bucket (which depends on the bit rate of the channel and the size of the bucket).

A naive method of providing a directory would be to broadcast the index before each broadcast of the file. This, however, leads to an unacceptably large average access time (having to wait first for directory and then for the data). Hence, there is a need to interleave the directory with the data more often to reduce the initial waiting time. Then however, we increase the overall length of the broadcast.

We could also provide hashing based schemes to improve the direct access properties and reduce the tuning time. However, even if the hashing function is perfect we may end up with many buckets which are "half empty" - increasing the size of the broadcast and consequently the access time.

Notice that in the traditional I/O terms the tuning time roughly corresponds to the disk access time in terms of number of disk blocks being accessed. However, the tuning time is fixed per broadcasted bucket, while the access time to the disk block varies depending on the position of the read/write head. The broadcast access time, on the other hand, corresponds to *space* requirement of the data on the disk. The larger, the file size the longer broadcast access time. Therefore, adding a data directory (index or hash) improves the tuning time while it *increases* the access time[3].

In general, different types of users may need different tradeoffs between tuning time and access time. Some may value lower access time and may have more leverage in terms of the tuning time (larger, laptop machines which may have more powerful batteries), some others will prefer lower tuning time and will be ready to pay for it in terms of the access time. Thus we need *flexible* data organization methods capable of accommodating different classes of users[4]. Distributed indexing method provided in [3] is not flexible in this sense, since it does not benefit from a more lenient tuning time requirements. In this paper we introduce two data organization methods which can be used for different priorities in terms of tuning and access time. The hashing method makes it possible to trade tuning time for the access time by changing the size of the overflow area. The *flexible* indexing method is also parameterized in such a way that depending on the value of the parameter we may change the ratio of the access to tuning time.

3 HASHING

Hashing based schemes do not require a separate directory to be broadcasted together with the data. The hashing parameters are simply included in the data buckets. Each bucket has two parts : the *Data* part and the *Control part*. The control part is the "investment" which helps guide searches to minimize the access and listening times. Control part for the first N buckets (B) includes:

- Hash Function: h
- Shift: The pointer (i.e. the actual bucket number) to a bucket which contains keys K such that h(K) = address(B)

The shift function is necessary since most often the hashing function will not be perfect. In such a case there will be collisions and the colliding records will be stored immediately following the bucket assigned to them by the hashing function. This will create an offset for other buckets (pushing the rest of the file "down"). The control part of the rest of the buckets have an offset to the beginning of the next broadcast.

[3]Sounding really strange from the traditional file access point of view. It is obvious however, since broadcasting index increases the total size of the broadcast

[4]Notice this whole discussion arises because we have two basic performance parameters instead of just one as in the case of disk based files

communication channel and tuning time plays little role since the PCs are connected to a continuous power supply.

In section 2, we discuss data organization methods suitable for broadcasting. In section 3 and in section 4 we discuss two indexing schemes for organizing and broadcasting data. In section 3 we discuss the *hashing* scheme and in section 4 *flexible indexing* is discussed; we also compare the performance of the two schemes in this section. In Section 5 we present conclusions and discuss future work.

2 Data Organization Basics of Broadcasting

Consider a file consisting of a number of records which are identified by keys. The file is not static and can be updated frequently so its size can grow and shrink at any time. Suppose that the server broadcasts this file periodically to a number of clients. The clients will only receive the broadcasted data and are interested in fetching individual records (identified by a key) from the file. Therefore fetching individual records from the broadcasted file will be performed *without* transmitting an uplink request but by *filtering* the incoming broadcasting *stream* for the given data item. Hence, queries will be answered only by *listening* to the channel. Data filtering is done by direct key matching.

Example *Consider the stock ticker tape broadcasted continuously. Each "edition" of the broadcast may include different groups of stocks; those which are currently "on the move". Stocks are identified by their symbols and the clients may be interested in monitoring pre-specified stocks or just in obtaining an isolated stock quotation.*

We would like to organize the broadcasted data in such a way that the following two parameters are minimized:

- *Access Time*: Time elapsed from the moment a client issues a query to the moment the answer is received by the client.
- *Tuning Time*: Amount of time spent by the client, listening to the channel.

Having these two parameters makes the data organization on the broadcast a problem which is different from the organization of the disk based files, where only one parameter - access time is taken into consideration. Here, we will have to optimize with respect to two parameters which work against each other.

If data is broadcasted without any form of directory, then the client in order to filter a data item, will have to tune to the channel on an average half of the time it takes to for the broadcast. As we will see, this is unacceptable as it requires the client's CPU to be active for a long time, consuming scarce battery resources. We would rather provide a selective tuning enabling the client to "wake up" only when data of interest is being broadcasted. We assume that the communication channel is the source of *all* information to the client including data as well as directories. We assume a *single channel* since multiple channels are really equivalent to a single channel with capacity (bit rate/bandwidth) equivalent to the combined capacity of the corresponding channels.

Selective tuning will require that the server broadcasts directory together with the data. The directory may be eventually cached by the client but new clients who have *no prior knowledge* of the broadcasted data organization will have to access it from the air. Besides, we will assume that the file will be changed by the server and can grow and shrink any time between successive broadcasts. The smallest logical unit of the broadcast will be called a *bucket*. The size of the bucket is a multiple of the size of a packet. Both access time and the tuning time will be measured in terms of number of buckets.

Broadcasts will contain successive versions of the file which will constitute (together with the directory) successive *bcasts* [2]. Each bucket of the current *bcast* will have a number called the *address* of the bucket - the sequence number of this bucket within the current *bcast*. Pointers to the specific buckets within the *bcast* will be provided by specifying the $offset$ between the bucket which holds the pointer and

[2] a version of the broadcasting of the file along with any index or other control information

Querying is viewed as *filtering* of the incoming data *stream* according to the user specified "filter".

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

In practice, a mixture of the above two modes will be used. The most frequently demanded items (weather, stock, traffic) will be broadcasted. Since the cost of broadcast does not depend on the number of users who "listen" this method will scale up with no penalty when the number of requests grows. For example, if the weather information is broadcasted every minute, then it doesn't matter whether 10 or 10000 users are listening, the average access time will be 30 seconds. This would not be the case if the weather was provided on demand. The "on-demand" mode will have to be used for the less often requested items. Broadcasting them periodically would be a waste of bandwidth. However, even in the pure "on-demand" mode- it makes sense to batch similar requests together (multicast as opposed to unicast) and send the answer once rather than cater individually to each request. Periodic data broadcasting is the main topic of this paper.

MOTIVATION

Power conservation is a key issue for small palmtop units which typically run on small AA batteries [2]. The lifetime of a battery is expected to increase only 20% over the next 10 years [7]. A typical AA cell is rated to give 800 mA-Hr at 1.2 V (.96 W-Hr). The constraint of limited available energy is expected to drive all solutions to mobile computing on palmtops. Assuming that the power source of the palmtop to be 10 AA cells with a CD-ROM and a display, the constant power dissipation in a CD-ROM (for disk spinning) will be about 1 W. The power dissipation for display will be around 2.5 W. Thus the assumed power source will last for 2.7 Hrs. Thus 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 the palmtops also consume power. There is a growing pressure on hardware vendors to come up with the energy efficient processors and memories. The Hobbit chip from AT&T is such a processor which consumes only 250 mW in the full operation mode. The power in "doze" mode is only 50 μW (the ratio of power consumption in normal operating mode to doze mode is 5000). When the palmtop is listening to the channel, the CPU must be in active mode for examining data packets (finding, if they match the predefined data). This is a waste of energy since on an average only a very few data packets are of interest to the particular unit. It is definitely beneficial if the palmtop can slip into doze mode most of the time and "wake up" only when the data of interest is expected to arrive. This requires the ability of selective *tuning* which is discussed in detail in this paper. In our model we explore filtering of data from the data broadcast using selective tuning features. The mobile clients will remain in a doze mode most of the time and tune in periodically to get information which is broadcasted on the communication channel.

Wireless data broadcasting can be viewed as *storage on the air* - an extension of the client's memory. Because of its periodic nature, latency of broadcast (the period between two successive broadcasts) will serve as the access time of such a memory and will not depend on the number of users accessing it. Such "public" storage will actually outperform any traditional storage media, for a sufficiently large number of users (this is shown in [4]).

Broadcasting over a fast, fixed network has been investigated as an information dissemination mechanism in the past. In the Datacycle project [6] at Bellcore the database circulates on a high bandwidth network (140 Mb/s) and individual users query this data by filtering the relevant information using a special massively parallel transceiver capable of filtering up to 200 million predicates a second. The main difference between the broadcasting considered in this paper and the broadcasting model used in the Datacycle architecture is that battery life was of no concern at all in the Datacycle architecture and this precisely our concern.

Gifford in [5] describes a system where newspapers are broadcasted over the FM band and down loaded by a PC equipped with radio receivers. There is a single

Power Efficient Filtering of Data on Air

T. Imielinski[1], S. Viswanathan and B. R. Badrinath[1]

Dept of Computer Science, Rutgers University
New Brunswick, NJ 08903

Abstract

Organizing massive amount of information on communication channels is a new challenge to the data management and telecommunication communities. In this paper, we consider wireless data broadcasting as a way of disseminating information to a massive number of battery powered palmtops. We show that different physical requirements of the wireless digital medium make the problem of organizing wireless broadcast data different from data organization on the disk. We demonstrate that providing index or hashing based access to the data transmitted over wireless is very important for extending battery life and can result in very significant improvement in battery utilization. We describe two methods (*Hashing* and *Flexible Indexing*) for organizing and accessing broadcast data in such a way that two basic parameters: tuning time, which affects battery life, and access time (waiting time for data) are minimized.

1 Introduction

In this paper, we consider wireless data broadcasting as a way of disseminating information to a massive number of battery powered palmtops. In this scenario the clients, equipped with palmtops will filter the incoming stream of information in order to match the pre-specified requests. Filtering will not involve transmitting any requests to the server - it will be a *receive only* activity targeted at monitoring and ad hoc querying of the data stream.

We show that different physical requirements of the wireless digital medium make the problem of organizing wireless broadcast data different from data organization on the disk. We demonstrate that providing index or hashing based access to the data transmitted over wireless is very important from the battery life point of view and can result in significant improvement in battery utilization, possibly of orders of magnitude. New technology can utilize and build upon some well known techniques (file organization and access). These traditional solutions cannot be applied directly though and need substantial modification because of the different physical limitations. New solutions require merging interdisciplinary expertise ranging from new communication protocols to file system and database design.

This paper and [3] provide different organization and access methods for the wireless broadcast data. In [3] we concentrate on indexing methods, with special emphasis on the access time, while in this paper we analyze a hashing scheme and an index based scheme with special emphasis on minimizing the tuning time. The schemes presented in this paper are flexible in the sense that, we can sacrifice access time for a gain in tuning time and vice versa. We will concentrate here on the wireless communication medium, although most of the presented work will also apply to the fixed network.

We will distinguish between two fundamental ways of providing users with information:

- *Data Broadcasting*: Periodic broadcasting of data on the channel. Accessing broadcasted data does not require uplink transmission and is "listen only."

[1]Work supported in part by NSF (SGER) award IRI-9307165