Data Dissemination and Broadcast Disks

R. K. Ghosh

Jan-Feb, 2002

4.1 Introduction

Push based data delivery can address three of the major issues related data retrieval in mobile computing environment, namely, mobility, scaling, and disconnections. By employing minor tweaking of conventional caching mechanisms at the client end, it is possible to improve response time when a client's applications need data. However, the specific issue concerning energy efficiency in push based retrieval is left unaddressed. Since a mobile operates with a limited power of a small battery, energy efficient operation of the clients is critical to the operations of mobile distributed systems. In this chapter, our aim is to examine indexing schemes through which a mobile can access data from broadcast channel in energy efficient manner.

The requirement for energy efficiency comes from following two fundamental limitations of mobile hosts.

1. *Need to conserve battery.*
2. *Need to utilize bandwidth.*

By conserving battery, a mobile computer can extend its battery life considerably. Of course, there are processors like Transmeta group's Crusoe and AT & T's Hobbit chips which are energy efficient. For example, Crusoe implementors claim that the processor can prolong the life of battery for mobile devices upto four fold. Typically, AT & T's Hobbit in full active mode requires about 250mW of power, but in doze mode it consumes about 50μW of power. So, the ratio of power consumptions between active to doze modes of a mobile device could be as much as 5000:1. Another important point is that if a battery drains out too often it will require frequent recharging. Although, the new generation of mobile batteries do not care how frequently they are recharged, a battery depending on its type, can last between 500-800 charge cycles. So, one of the important reason for prolonging the life of a battery is that disposal of used batteries is environmentally hazardous. Usually, wireless communication offers low bandwidth. So as indicated in the previous chapter, broadcast communication is far more effective than unicast communication. Broadcast allows the communication to scale up gracefully. All mobile devices can tune to a broadcast channel and retrieve the required data as and when they desire. In contrast unicast communication requires opening a separate channel of communication each time when the need arise. Simultaneous use of many separate channels in the same cell will partition the bandwidth, and hence, considerably reduce the utilization of bandwidth. Whereas the broadcast communication is independent of the number of listeners.

In wired network, an indexing mechanism is primarily reduces the access time of disk based files. The client is always online with an active network connection and waits until the relevant data is retrieve from the disk. If the data available in the local disk then the cost of retrieval is significantly low. Even when data is on a server disk, the distance of data from the client's memory is not significantly high, as a wired connection provides low latency with a high bandwidth.

In a broadcast oriented communication, the distance of data from a mobile client's memory is determined by the broadcast schedule of the relevant server. For example, if a client's request coincides with the appearance of a data item in the broadcast channel, then the data is available instantaneously. However, if the request is made after the data has just flown past the broadcast channel, then the client will be forced to wait for the data until a full broadcast cycle has elapsed.

Obviously, a mobile client can not remain in the active mode for a long time as it depends on the limited power of a battery. Therefore, push based data delivery should incorporate a method to organize the data that would enable a mobile client to anticipate the exact time when its data requirements can be met by the broadcast channel. Unfortunately, the broadcast channel is a strictly sequential medium. It forces a mobile client to turn into active mode from time to time to check and retrieve the requested data from the broadcast disks.

Listening to the downlink wireless channel for checking whether the specific data has arrived or not is referred to as a probe. Tuning time refers to the time that a mobile host spends in probing the downlink channel until it is able to download data from a server. Therefore, an efficient indexing scheme should minimize both the probe and the access times, where access time refers to the difference between the time the request is made to the time when the data gets fully downloaded.

4.2 Address Matching and the Directory

A mobile device operates between active and sleep modes. The sleep mode allows a mobile device to shut down its power hungry components including

CPU, display, and communication interface. Fortunately, power requirement of paging hardware is very little and it remains on. So, a mobile can be switched into active mode when needed. Ideally, a mobile client should switch on into *active* mode only when the required data appears on the broadcast channel. So, the broadcast should be organized in such a way that the pager can apply an address matching scheme to determine whether the data on downlink channel is relevant for the mobile terminal or not.

The other way to switch from sleep to active mode of operation is on detection of event triggers. Events triggers typically require tight synchronization and handled by system clocks. System clock is a very low energy circuit. It wakes up CPU and turns mobile into active mode at pre-determined timings. Thus a mobile terminal can turn into active mode to download the required data when it is available on the downlink channel by using one the following ways by address matching on the packetized data.

1. Multicast address matching:

2. Temporal address matching.

The pager matches the leading bits from the embedded address of an incoming packet and determines if the packet is for the multicast group to which the client has subscribed. If a match is found, the pager wakes up the CPU turning the mobile client to active mode. The client probes the downlink channel from time to time and determines the exact time when the relevant data is published on the broadcast channel. It switches itself to sleep mode for the duration of the time until the required data arrives on broadcast channel. Quite obviously, the technique of temporal address matching requires perfect or near perfect synchronization which is not necessary in the case of multicast address matching.

However, the basic problem in using any of the above methods is that a client must have the advanced knowledge about mappings between the data keys (primary/secondary) with addresses (multicast group/temporal). The mapping, known as a directory, must be either pre-cached by the clients or published by the server mulitplexing it along with the actual data on broadcast channel. The second approach is more appropriate in a broadcast environment, because it is oblivious to the dynamicity of data and the churn rate of clients in a mobile system. So, our goal in this chapter is to study this second approach for energy efficient indexing in mobile environment.

4.3 Preliminary Notions

We introduce some preliminary notions required in analysis of directory organization broadcast or published data.

Definition 1 (`Bucket`) *This the smallest logical unit (a page) of broadcast. For the simplicity, we assume that exactly one packet fits into a page of a bucket.*

The data in each bucket is identified by some attribute value or the `search key`. A client can retrieve the required data identified by `search key` by simple pattern matching.

Definition 2 (`Bcast`) *The broadcast data is divided into local segments called* `Bcast`*. Each* `Bcast` *consists of data interleaved with directory information or index.*

The directory information may include some replications. The replication makes search efficient when the directory information intermixed with data is pushed into broadcast channel. Each `Bcast` consists of only one instance of data.

4.4 Temporal Addressing Technique

The temporal address matching scheme requires that the directory of data files to be published along with the data. Alternatively, directory information can be cached by the clients. However, caching directory at the clients is impractical due to the following reasons.

- Firstly, the directory information across cells may not match. Therefore, cached directory information for one cell is unusable when a client moves to a new cell.
- Secondly, a client entering into system for the first time will not have any directory information for the data broadcast in its home cell.
- Thirdly the data and, hence, the directory information may change over time. It will require every client to recharge the cache with new directory information.

- Lastly, storing directory at client may require substantial local storage. So, pre-caching can only be applied to relatively less portable mobile clients.

Therefore, a simple approach will be to have the directory information published along with data on the broadcast channel.

4.5 Tuning Time and Access Latency

The parameters used for the measurement of energy efficiency and the access latency of the of the broadcast data are:

- **Tuning time**: The time spent by a client listening to the broadcast channel, in order to retrieve the required data. It provides a measure of the energy consumption at the client, because the energy spent will be proportional to the time client operates in active mode.

- **Access latency**: The time elapsed (on an average) from the moment a client requests for data to the point when the data actually downloaded by the client. Access latency affects the response time of the application running at the client. The request is deemed to have been issued when the client makes the initial probe.

Both the timing parameters are measured in terms of bucket numbers. The tuning time is proportional to access latency. The access latency will depend not only on the index but how the data is organized in the schedule. The number of probes will depend on the depth of the index tree to the appropriate leaf where data is located. A mobile client may have to probe a number times to climb down the index tree and determine the exact position of the requested data in broadcast schedule. Figure 4.1 illustrates the relationship between the tuning time and the access latency.

4.6 Indexing in Air

The key idea is to come up with an allocation method that multiplexes index with data in an efficient manner, such that

- It enables clients to conserving energy, and

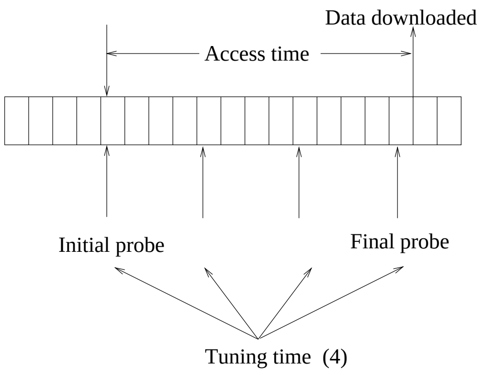


Figure 4.1: Illustrating timing parameters

- It enables server to utilize the broadcast bandwidth efficiently.

First of all, buckets should be self identifying, i.e., whether a bucket has index or data. Inclusion of certain header information in a bucket can provide the identification and allow clients to determine the position of the bucket in the broadcast schedule.

bucket_id: offset of the bucket from beginning of the `Bcast`.

bucket_type: whether bucket stores index or data.

bcast_pointer: the offset to beginning of the next `Bcast`.

index_pointer: the offset to beginning of the next index segment.

A pointer to a bucket can be found by *bucket_id* B. The offset to B from the current bucket B_{curr} is the difference $\textit{diff\_offset} = \textit{bucket\_id}(B)$ - $\textit{bucket\_id}(B_{curr})$. Therefore, the time to get B from B^{curr} is equal to $(\textit{diff\_offset-1}) \times t_B$, where t_B is the time to broadcast a single bucket. An index bucket consists of pairs of values $(\textit{attribute\_value}, \textit{offset})$. The *offset* is the *bucket_id* of the bucket that contains the data identified by the *attribute_value*. In general an index tree may have multiple levels.

4.6.1 (1, m) Indexing Scheme

A simple minded approach is to broadcast index m times within data buckets. This scheme is called (1, m) indexing wherein every $1/m$ fraction of data buckets is preceded by one index. The degenerate case of (1, m) indexing occurs when $m = 1$, index broadcast exactly once with data. So, the length of a `Bcast` is equals length of *Data* plus the length of *Index*, i.e., `Bcast` $=$ *Data* $+$ *Index*. The access algorithm using this (1, 1) indexing scheme is as follows.

- A Mobile Host (MH) tunes to the current bucket and determines the offset to the next nearest index on the broadcast channel.
- MH enters doze mode setting wake up time to the time when the next nearest index is broadcast.
- MH wakes up after doze mode expires and obtains the pointer to the data record with the required primary key.
- MH again enters doze mode setting wake up time to the time when the data record is available in the next `Bcast`.
- MH wakes up again after doze mode expires and downloads the relevant data.

Analysis of (1, 1) Indexing

A `Bcast` consists of *Data* interleaved with directory information. The access time has two parts: (i) *probe wait*, and (ii) *Bcast wait*.

Definition 3 (*Probe wait*) *It refers to the average delay to get to the next nearest index from the time of initial probe for accessing index is made.*

Definition 4 (*Bcast wait*) *It refers to the average delay to download the actual data from the time the index to the required data is known.*

In (1, 1) indexing, the index is broadcast once. Thus, the expected delay due to *probe wait* is equal to half the `Bcast` time. Also the expected delay for downloading data or the *Bcast wait* is half the `Bcast` time. It means the access time $=$ (*Data*+*Index*).

Tuning Time for (1,1) Indexing

Tuning time, the time spent on listening, depends on number of probes. It consists of (i) initial probe, (ii) the number of probes to get the correct index pointer for the required data, and (iii) one final probe to download the data.

The size of a fully balanced index tree T_{idx} is $Index = \sum_{i=0}^{k-1} n^i$, where T_{idx} has k levels. Suppose, each bucket has the space to accommodate n pairs of (`search_key`, `data_ptr`). Then, $k = \lceil \log_n Data \rceil$. Therefore, the tuning time $= 2 + \lceil \log_n Data \rceil$.

Analysis of (1,m) Indexing

The distance between two consecutive occurrences of index segments in broadcast schedule is

$$\frac{Data}{m} + Index.$$

Therefore, the probe wait is:

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

The Bcast wait is

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

Therefore, the access time is equal to:

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

As explained earlier, the tuning time is dependent on the number of probes. Thus the total tuning time is equal to

$$2 + \lceil \log_n Data \rceil.$$

However, the optimization of access time depends on the number of times index is replicated in a broadcast cycle. We can determine the minimum value of m by differentiating the expression for the access time. It gives the value of m as:

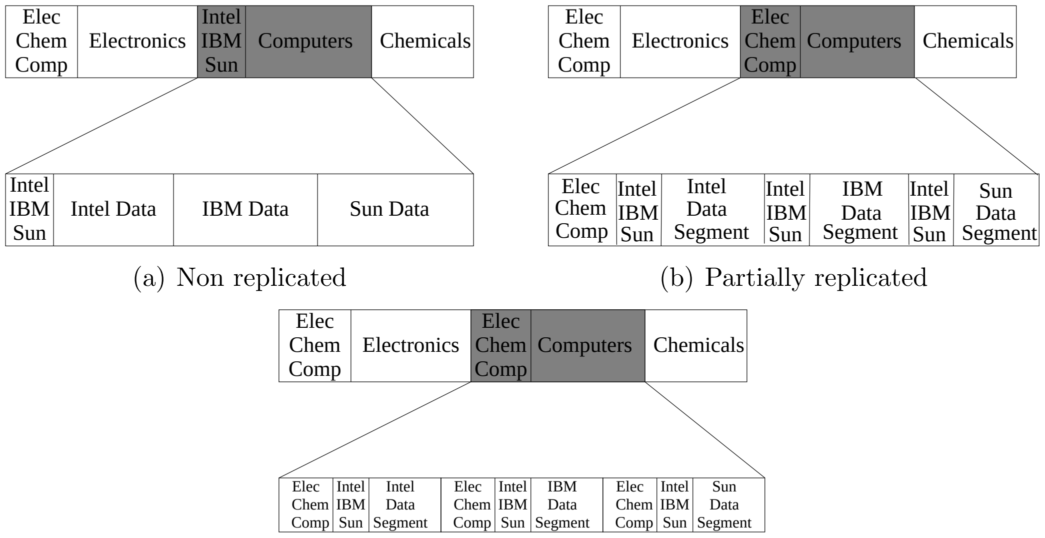
$$m = \sqrt{\frac{Data}{Index}}.$$

4.7 Distributed Indexing Scheme

Distributed indexing improves both access time and tuning time compared to (1, m) indexing by cutting down the replications on indexes. In the (1, m) indexing, every $1/m$th fraction of data is preceded by an index leading to m times replication of the index in a `Bcast`. In the distributed index, the replication of index is restricted. There are three distributed index methods differing in the degree of replication.

- Non replicated ditributed indexing.
- Fully replicated ditributed indexing.
- Partially replicated ditributed indexing.

In non-replicated distribution different index segments are disjoint. Figure 4.2(a) illustrates an example for this scheme. As the figure shows, the



(a) Non replicated

(b) Partially replicated

(c) Fully replicated

Figure 4.2: Distributed indexing

first level index refers to classification of organization based on area of operation such as chemicals, electronics or computers. The second level indexes precede a data segment, only the index relevant to the data segment appearing next are placed. Only the index segment associated with computer data segment appears before data segment for computers. The top level indexes for chemicals or for electronics are not replicated. In the case of partial replication, the path in an index tree from the root to the least common ancestor of index buckets B and B' is replicated just before occurrence of index bucket B'. As indicated in figure 4.2(b), the replicated part of the index prior to the occurrence of IBM data segment refers only to the root of the index subtree containing the data segment of computers which incidentally is the least common ancestor in the index tree path from root to Intel and IBM data segments. In fullly-replicated distribution the path from the root to an index bucket B is replicated just before the occurrence of a data bucket B For example, as figure 4.2(c) shows, the full index tree path from the root down to the subtree of computers is replicated just before the occurrence of the data buckets for computers.

4.7.1 Distributed Indexing with No Replication

Every data bucket maintains a pointer to the next nearest `Bcast`. The index file is distributed with the root of the index tree at the beginning of a `Bcast`, and the index buckets are interleaved with data buckets. The different index segments are disjoint. So, there is no replication of indexes. The data is organized level-wise. For example, a general index tree consisting of four levels and data file with 81 buckets is illustrated by figure 4.3 [1]. It indicates that top two levels of the index tree are replicated and bottom two levels are non-replicated. Each group of three lowest level branches leading to data buckets are merged into a single link. Extending the above four level index tree a general multi-level index tree having k levels can be described as follows. The top part of the tree including the root and upto a level, say r, constitutes the *replicated* part. The non replicated part of the index tree consists of a collection of subtrees each rooted at an index node belonging to the level $r + 1$.

The non replicated part appears only once in a `Bcast`. Whereas every node of the replicated part appears as many number of times as the number of children it has in the index tree. For example, consider the tree T_{idx} in figure 4.3, the root R of T_{idx} has three children and so does each of the nodes a_1, a_2 and a_3. The part of T_{idx} from the root to level 1 form the replicated part of the index. So, each of the nodes R, a_1, a_2, a_3 are replicated three times

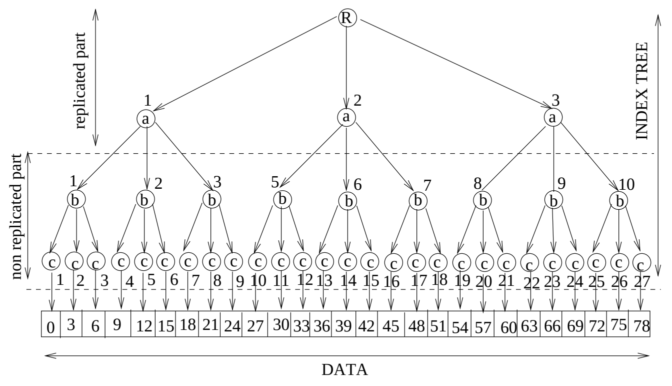
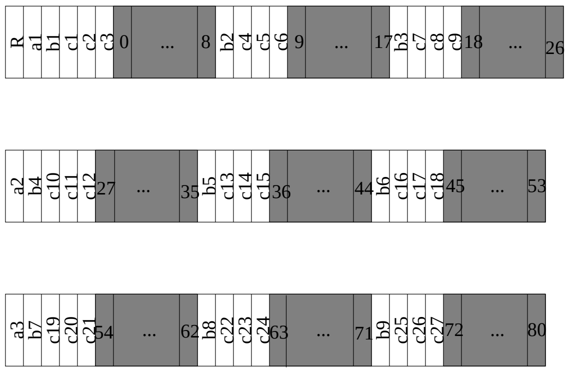


Figure 4.3: A three level index tree [1]



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

Figure 4.4: Distributed indexing with no replication

in a `Bcast` in distributed indexing with replication. The two extreme cases of distributed indexing (i) non replicated indexing, and (ii) fully replicated indexing, occur respectively when $r = 0$ and $r = k$.

Let us examine the organization of four-level T_{idx}, shown in figure 4.3, a bit more closely. Suppose, nine data buckets reachable from an index node at level 2 from the top are merged together in a single data segment on a broadcast schedule. When using distributed indexing with no replication a `Bcast` for dissemination of 81 data buckets appears as the one shown by figure 4.4 [1]. There is no replication of the index path in any part of the index segment. An index segments consists of either 4 or 6 buckets. Suppos a mobile host MH wants to access the data bucket 58, and MH makes it initially probe at bucket 2. Then sequence for probes, for downloading data bucket 58, occurs as follows.

- From bucket 2, MH gets the offset to the beginning of the next nearest `Bcast`. This is where the root of T_{idx} will be found.
- From the root R, the search is successively guided by the index buckets a_3, b_7, c_{19} and then finally to data bucket 58.

This means that, MH requires 4 probes to determine the position of the data bucket 58. So, including the initial probe, five probes will be needed to download data of bucket 58.

4.7.2 Replication Based Distributed Indexing

The other two distributed indexing based on replications increase the size of the `Bcast`. But the amount of replication is not as much as (1,m) indexing scheme. Therefore, the access time which is dependent on the size of `Bcast`, is less compared to that of (1,m) indexing scheme. But compared to the indexing with no replication the bandwidth utilization is poor. Because, the replicated indexes eat away some amount of bandwidth. However, if the index tree is not very big and the replication is not very high then it helps a mobile client to retrieve data in an energy efficient manner. In fact, the tuning time can be reduced considerably by optimizing the replication. An optimum value for r, the level up to which index tree should be replicated to minimize tuning time can be determined by a simple analysis provided later in the next section.

4.7.3 Full Path Replication Scheme

Let us examine the distributed index replication policies along with data organization for broadcast. For the sake of simplicity in discussion, let us assume the index tree T_{idx} has four levels as shown by the picture in the earlier figure. The full index path replication scheme is illustrated in figure 4.5. Each index segment represents the entire tree path for the data segment that occurs after it. Suppose a MH wishes to access data bucket with key 73, and the initial probe is made at any data bucket B_x. MH can retrieve the pointer to the next nearest occurence of the root of T_{idx} from B_x. MH then begins search on T_{idx} along the path from R to the required leaf storing a

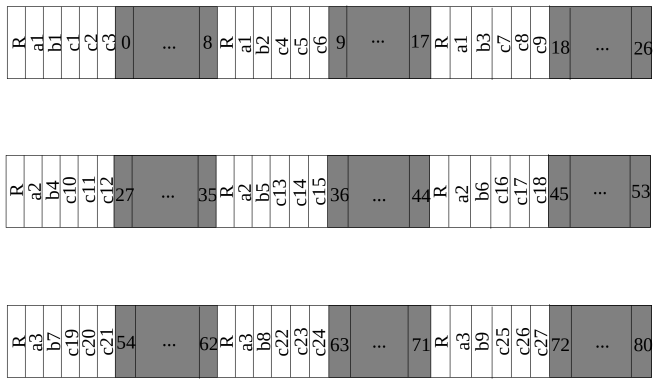


Figure 4.5: Distributed indexing with full path replication

pointer to record with the primary key 73. For the sake of concreteness, let us assume that $x = 15$. Then the index path traversed to reach the record with $key = 73$ is give by the sequence of index buckets:

$$\{R, \mathit{first\_a}_3, b_9, c_{25}\},$$

where $\mathit{first\_a}_3$ denotes the first copy of a_3 in the replicated part of T_{idx}. Since, R has the information about the child through which the search can be guided, the search takes us directly to *first_a3*. The from there to b_9, and c_{25} along the index tree path.

4.7.4 Partial Path Replication

The partial path replication improves on the full path replication by elimination of redundant replications. The basic idea, as explained earlier, is to partition the index tree into two parts, namely,

- Replicated part, and
- Non replicated part.

In our discussion we assume without loss of generality that the index buckets for Non Replicated Roots (NRRs) are ordered from left to right in consistent with the order they appear in the tree at level $r + 1$. The data buckets or the index buckets which are descendants of NRRs appear only once on each broadcast cycle. Before, we examine the organization further, it is convenient to introduce some notations.

1. R: the root of index tree.
2. NRR: denotes the set of index buckets at $r + 1$ level. These are the roots of the index subtrees in non-replicated part of the index tree.
3. B_i, $1 \leq i \leq |NRR|$: B_i is the ith index bucket in NRR.
4. *Path*(C, B_i): represent the sequence of buckets on the path in index tree from bucket C to B_i excluding B_i.
5. *Data*(B_i): the set of data buckets indexed by B_i.
6. *Ind*(B_i): the part of index tree below B_i including B_i.
7. $LCA(B_i, B_k)$: the least common ancestor of B_i and B_k in the index tree.
8. For a $B_i \in NRR$, *Rep*(B_i) represents a path in replicated part of index tree as defined below:

$$Rep(B_i) = \begin{cases} Path(R, B_i) \text{ for } i = 1, \\ Path(LCA(B_{i-1}, B_i), B_i) \text{ for } i = 2, \ldots, |NRR| \end{cases}$$

9. $Ind(B_i)$ denote non-replicated part of the path from R to B_i.

For the running example, $NRR = \{b1, b2, b3, b4, b5, b6, b7, b8, b9\}$. So, we have

	b1	b2	b3
Rep(B)	$\{R, a1\}$	$\{a1\}$	$\{a1\}$
$Ind(b)$	$\{b1, c1, c2, c3\}$	$\{b2, c4, c5, c6\}$	$\{b3, c7, c8, c9\}$
Data(B)	$\{0, \ldots, 8\}$	$\{9, \ldots, 17\}$	$\{18, \ldots, 26\}$

Each broadcast schedule is a sequence of tuples the form:

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

There can be $|NRR|$ tuples in a schedule.

Figure 4.6 illustrates the organization of the same 81 data buckets for the running example on a `Bcast` using partial path replication. Notice that there are exactly nine broadcast segments in the schedule one corresponding to each non replicated root.

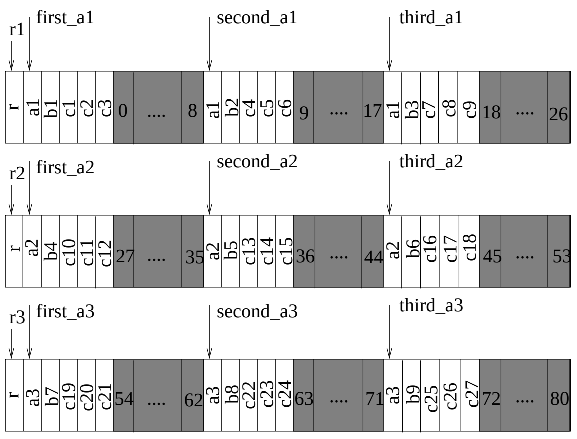

r1
first_a1
second_a1
third_a1
r a1 b1 c1 c2 c3 0 .... 8 a1 b2 c4 c5 c6 9 .... 17 a1 b3 c7 c8 c9 18 26
r2
first_a2
second_a2
third_a2
r a2 b4 c10 c11 c12 27 .... 35 a2 b5 c13 c14 c15 36 .... 44 a2 b6 c16 c17 c18 45 53
r3
first_a3
second_a3
third_a3
r a3 b7 c19 c20 c21 54 .... 62 a3 b8 c22 c23 c24 63 .... 71 a3 b9 c25 c26 c27 72 80

Figure 4.6: Partial path replication

Suppose the client wants to retrieve data bucket 66 and it makes the first probe at data bucket 3. The probe directs the client to *second_a1*. But as the entire tree is not replicated, the root can not be accessed. Furthermore, though the data bucket is yet appear in the broadcast schedule, the local index does not have a pointer which can access the data bucket 66. It necessitates that there should be some special provision to climb up index tree in order to guide the client to data bucket 66.

The simple trick that will provide the client enough information about climbing up the index tree is to have each replicated node store a small set of local control indexes. For example, the local *control index* at *second_a1* should be able to direct the client to $r2$ where the next replication of the root in the index tree. Once root is found, the client makes the sequence of probes: *first_a3*, $b8$, $c23$ and retrieves the data bucket 66. However, if instead of 66 the client want to find bucket 11, then the index *second_a2* should direct the client to probe $b2$, $c4$ and then 11. Thus, having a copy of the root before *second_a2* in the index tree would have been a waste of space.

The role of the control index is to facilitate the climbing up of the tree path to the root if the data bucket corresponding to the search key has gone past on the current `Bcast` and reach the correct index bucket from which pointer to the data bucket can be determined. So, the correctness of control indexes at each index bucket is critical to search. The control indexes for the example under discussion is provided in figure 4.7.

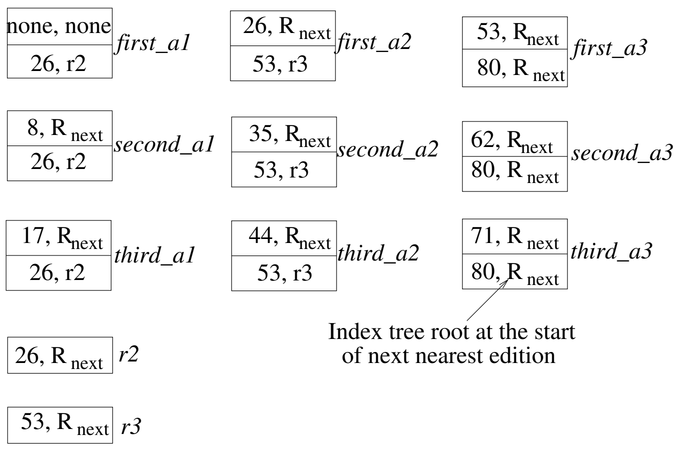


Figure 4.7: Control index

The control index is stored in each index buckets which is a part of replicated portion of the index tree. For understanding how control indexes are determined, let us consider a path $Path(R, B)$ where $B \in NRR$. Consider a sequence of buckets $\{B_1, B_2, \ldots, B_s\}$ in $Path(R, B)$. Each bucket B_i in the above path belongs to the replicated part of the index tree. Let $Last(B_i)$ represent the primary key value in the last record indexed by index bucket B_i, and $Next_B(i)$ denote the offset to the next-nearest occurrence of B_i.

The control index of a bucket B_i which belongs to $Rep(B)$ consist of i tuples of the form:

$$\begin{array}{l} [v, \textit{begin}] \\ [Last(B_2), Next_B(1)] \\ \qquad\qquad \vdots \\ [Last(B_i), Next_B(i-1)] \end{array}$$

where v is the primary key value of the last record in the broadcast segment prior to B. The number of control tuples in a replicated index bucket x depends on the level of x in the index tree. If the level of x is l, then x should have exactly $l+1$ control tuples. For example, the level of a_2 in index tree is 1. So $a2$ must have 2 control tuples for its control index.

For example, consider the replicated index path *Rep(R, b4)*=$\{$*R, a2*$\}$. In order to avoid the confusion about which instances of R and a being considered, let the replicated instances be referred to as $r2$ and *first_a2* respectively.

The value of the primary key in the last data record broadcast prior to $b4$ is same as the last data record accessible from $b3$, which is 26. So, the tuple defining control index in $r2$ should have $v = 26$. In fact, any record having a key value between 0 and 26 will no longer be available in the present broadcast. So, in order to retrieve a record having key value in the range [0, 26] one has to wait until the next broadcast cycle. The next new occurrence of R is the beginning of a new broadcast, which is represented here as R_{next}. So, putting the components of the tuple together, the control index $r2$ is $[26, R_{next}]$.

Next, consider how the control tuples for *first_a2* are determined. It will have two tuples since $level(a2) = 1$. The first tuple should be [26, R_{next}] because the last record accessible before $r2$ is 26. The second control tuple can be found by obtaining the key of the last record accessible from the current index (i.e., $first\_a2$) is $Last(a2) = 53$. The next nearest occurrence of $a2$ can be found only in the next instance of the index bucket R, which is $r3$ in figure 4.6. Therefore, the second control tuple for $first\_a2$ should be [53, $r3$]. Therefore, the control tuples in the replicated bucket *first_a2* are: $([26, begin], [53, r3])$ as indicated in figure 4.7. In fact, during the current broadcast cycle starting from $first\_a2$, it is possible to get any record whose key K lies in the range (26, 53].

As another example, index bucket $b5$. In order to find the control index for *second_a2*, we consider *Rep(R, b5)* = {*second_a2*}. The value of the last primary key broadcast before $b5$ is the last key accessible from $b4$ is 35. So, the record with the primary key 35 can not be retrieved during the current broadcast. Consequently the first tuple for the control index in *second_a2* should be $[35, R_{next}]$. The primary key of the last record reachable from *second_*$a2$ is 53 and if any data with primary key value greater than 53 is to be retrieved then the search must be made again from R. Since, the next instance of occurrence of R is $r3$, the second tuple in *second_a2* must be $[53, r3]$. Hence, the pair of tuples ($[35, begin]$, $[53, r3]$) constitute the control index for *second_a2*. From this control index, we conclude that any data bucket with primary key value $35 < k \leq 53$ can be found in the current broadcast.

The case of $first\_a1$ is bit different because none of data buckets has appeared yet in the current broadcast. All records have key in range [0,26] are accessible from $a1$. So the first control tuple should be $[none, none]$ where *none* indicate both values (key and offset) are meaningless or invalid. Whereas the offsets in the control tuples for any replica of $a3$ can only lead to the beginning of next broadcast cycle. Possibly, the second control tuples in each may be replaced by *none* to indicate that there is no record having a key greater than 80.

4.7.5 Access protocol

Let priamry key for the record to be accessed is K

Step 1. Tune to the current bucket of `Bcast`. Get the offset for the bucket containing the control index. Set the expiry time (till the control bucket appears) and turn into doze mode.

Step 2. Wakeup after expiry time, access the designated bucket, and get the control index I_{ctrl}.

Step 3.

```
if ($K \leq I_{ctrl}$) { // Data miss occurs in the current broadcast
    Get offset for the next new Bcast; // Offset from the first tuple
    Set expiry time to the next new Bcast and turn into doze mode;
    Proceed to Step 4 after expiry time;
}
else {
    Get the offset to appropriate higher level index bucket;
    Set expiry time for it, and go to doze mode;
    Wakeup after expiry time;
    Follow one the Next pointers and proceed as in Step 4.
}
```

Step 4. Probe the designated index bucket and follow the sequence of pointers (the client turn into doze and wakeup modes in between two successive probes) to find the offset to data bucket.

Set expiry time for data bucket having key K to appears on broadcast channel and go to doze mode.

Step 5. Wake up after expiry time, and download the bucket containing the record with key K.

Access latency

The access latency depends on *probe wait* and *Bcast* wait. Let r top levels of the index tree represent the replicated part and the tree is fully balance consisting of k levels. The replicated index buckets occur before $Ind(B)$ for each $B \in NRR$. The set of data buckets $Data(B)$ indexed by each $B \in NRR$ appear immediately after $Ind(B)$. Therefore, the maximum distance separating two consecutive occurrences of replicated index buckets is $Ind(B) + Data(B)$.

Since the control index is present in each replicated index bucket, the probe wait is determined by the average number of buckets in the segment consisting of $Ind(B)$ and $Data(B)$. The size of $Ind(B)$ is same as the number of nodes in a fully balanced tree of height $k-r-1$. This follows from the fact that $Ind(B)$ for a $B \in NRR$, is actually a subtree of the index tree rooted at B. Therefore, the number of nodes in $Ind(B)$ is given by the expression

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

The average size of $Data(B)$ is $\frac{Data}{n^r}$ as $|NRR| = n^r$, and $Data$ is the size of entire data. Hence, the probe wait is

$$\frac{1}{2}\left(\frac{n^{k-r} - 1}{n - 1} + \frac{Data}{n^r}\right).$$

The Bcast wait is the half of the total length of a `Bcast`. This is equal to *Index* + *Data* + *overhead*.

The overhead is introduced due to replication of the nodes in the top r levels of the index tree. We know that each bucket is replicated as many times as it has children. As illustrated in figure 4.8, all the replications put together can be viewed as a fully balanced tree tree of height $r+1$ having a single dummy node serving as the root. Excluding the dummy root, the total number of nodes in the tree shown above is

$$\sum_{0}^{r} n^i = \frac{n^{r+1} - 1}{n - 1} - 1.$$

The number of index buckets in replicated part equal to

$$\sum_{0}^{r-1} n^i = \frac{n^r - 1}{n - 1}.$$

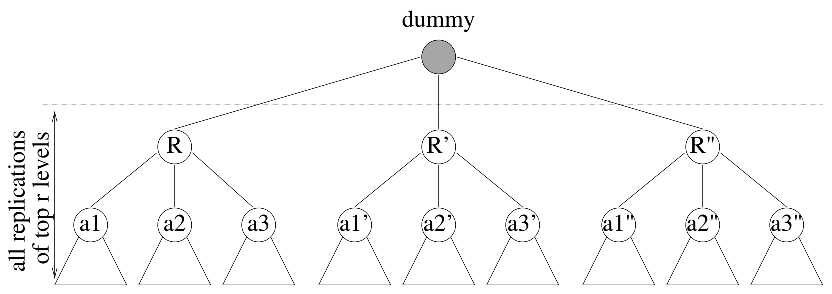


Figure 4.8: Number of index nodes for the replicated part.

So, the index overhead due to replication is

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

Adding probe wait and bcast wait, we get the access time as

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

Tuning time

The initial probe is for determining occurrence of a control index. The second probe is for accessing the control index. The control index may direct the client to the required higher level index bucket. In worst case, the client may make upto k = number of levels in index tree. Finally, the client downloads the data. Therefore, tuning time is $\lceil \log_n Data \rceil + 3$.

Optimizing number of replicated levels

The probe wait is a monotonically decreasing function reaching the maximum when $r = 0$ and the minimum when $r = k$. The Bcast wait is a monotonically increasing function reaching the minimum when $r = 0$ and the maximum when $r = k$. When r increases from 0 to 1, probe wait decreases by $\frac{1}{2}\left(\frac{Data*(n-1)}{n} + n^{k-1}\right)$ and the Bcast wait increases by $\frac{n^0*(n-1)}{2}$. In general as r increase to $r+1$

1. Probe wait decreases by $\frac{1}{2}\left(\frac{Data*(n-1)}{n^{r+1}} + n^{k-r-1}\right)$.

2. Bcast wait increases by $\frac{n^r*(n-1)}{2}$.

So increase r as long as

$$\begin{aligned}
\tfrac{1}{2}\left(\tfrac{Data*(n-1)}{n^{r+1}} + n^{k-r-1}\right) &> \tfrac{n^r*(n-1)}{2}, \text{ or}\\
Data*(n-1)+n^k &> n^{2r+1}*(n-1), \text{ or}\\
\tfrac{1}{2}\left(\log_n\left(\tfrac{Data*(n-1)+n^k}{n-1}\right)-1\right) &> r
\end{aligned}$$

Therefore the optimum value of r is

$$\left\lfloor \frac{1}{2}\left(\log_n\left(\frac{Data*(n-1)+n^k}{n-1}\right)-1\right)\right\rfloor + 1$$

4.8 Hashing

Hashing scheme is different from tree based indexing schemes [2]. It does not require broadcast of directory information with data. The information is embedded with data (contents). Each bucket has two parts, viz., (i) control part, and (ii) data part. The search for data bucket is guided by information available in control part. The control part of every bucket have following two information

1. *bucket_ID*: offset to the bucket from the beginning of the current `Bcast`.

2. Offset: offset of the bucket from the beginning of the next `Bcast`.

Furthermore, the control part of first N bucket will also carry two additional piece of information, viz.

3. Hash function: h.

4. Shift value: The pointer to a bucket B that contains the keys K such that $h(K) = address(B)$.

A client wishing to retrieve a record corresponding to a key K, start listening until it is able to read a full bucket B. From the control part of B, the client retrieves the hash function h. Then it computes $h(K)$ and determines if its initial probe has missed the index or not. Then it tunes in to physical bucket which matches with hash value $h(K)$. From the physical bucket it retrieves the shift value. The shift value s gives the offset to the logical bucket that

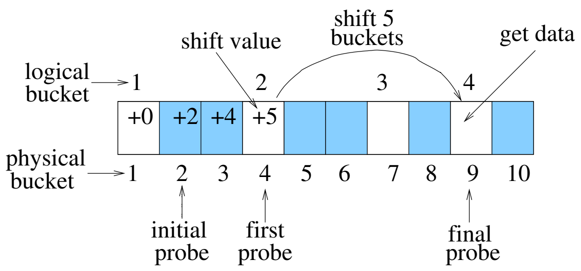


Figure 4.9: Retrieving data bucket with record key 15.

should have hash value $h(K)$. This bucket will be $s \times t_B$ units away from the current time. So, the client goes into doze mode for the duration of time. When the client wakes up after the expiry of doze mode, and starts listening. It either gets the data with key K or encounters a record with key L such that $h(K) \neq h(L)$. In the latter case, a failure is reported to the client's application. In summary, the access protocol for finding the record with K is as follows.

Step 1. Probe the current bucket and read the control data and compute hash h.

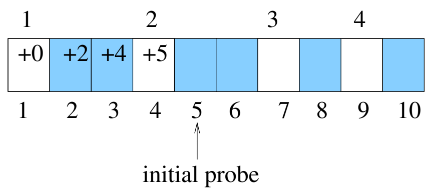
Step 2.

```
if $bucket\_ID < h(K)$
    go to doze mode listen again to the slot $h(K)$;
else
    wait till the next Bcast and repeat the protocol;
```

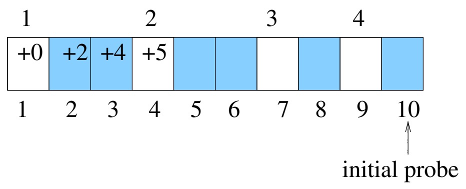
Step 3. At $h(K)$, get the shift value s, go to doze mode until shift number of buckets.

Step 4. Keep listening until either bucket containing record with key K is found, or a record with key L is found such that $h(L) \neq h(K)$.

Figure 4.10 provides a concrete example of the access protocol. The physical bucket IDs which represent overflow buckets do not carry any logical ID. Suppose we are interested in retrieving record with key $K = 15$. Let the



(a) Index miss scenario



(b) Data miss scenario

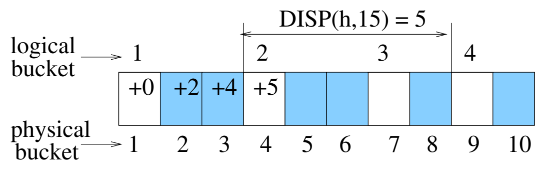
Figure 4.10: Scenarios depicting data miss and index miss.

initial probe be made at physical bucket 2. From this bucket hash function is retrieved and $h(15) = 4$ is computed. Let the hash function be

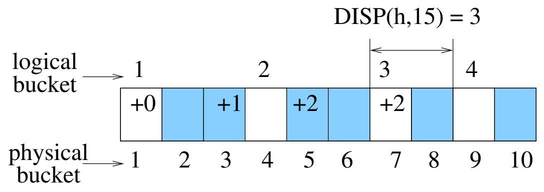
$$h(x) = x \mod 4 + 1$$

Since hash value does not match physical bucket $B = 2$, it gets offset $h(K)-B$ and sets expiry time for arrival of physical bucket 4. From physical bucket 4, shift value $s = 5$ is retrieved. Again the client goes into doze mode till 5 physical buckets pass by. The client wakes up and starts listening for data sequentially from there onwards. Assuming logical bucket 4 has the data, the record with key K gets downloaded. There are two scenarios where the client has to wait till next broadcast cycle to retrieve data. In the first scenario (figure 4.10(a)), the initial probe is made only after the data bucket containing the record has passed by in the current broadcast cycle. In the second scenario, the data is still to appear but the index (figure 4.10(b)) is missed. In this case the initial probe occurs at a bucket past the physical bucket $h(15) = 4$.

If a perfect hashing function is employed, then it minimizes the access time for conventional disk based files. In a broadcast based dissemination, the total number of buckets in a broadcast cycle has a direct impact on the access time. Although, no overflow occurs, it can not minimize the number of physical buckets required for a file. Since broadcast medium is sequential, the waiting time for the next version of the file becomes long, if the total number of buckets becomes large. On the other hand, when hash function is not perfect, many overflow buckets are needed to accommodate the data. But it leads to a better utilization of bucket space, because only the number of half empty buckets can be restricted in a broadcast cycle. In contrast, bucket utilization in perfect hashing may not be very good as there is a possibility of transmitting many half empty buckets (depends on the sizes of



(a) Original hashing



(b) Modified hashing

Figure 4.11: Displacement is reduced with modified hash function [2].

records). In summary, the hashing scheme exhibits random access behavior for tuning time (hash values and shifts are available as control parameters), and sequential behavior for the access time (the number of buckets matters).

4.8.1 Hash B

The initial hash function can be improved by modifying the hash function [2]. It works provided an estimate of minimum overflow d can be made. The revised hash function is defined as follows:

$$h'(K) = \begin{cases} h(K), \text{ if } h(K) = 1 \\ (h(K) - 1)(1 + d) + 1 \text{ if } h(K) > 1 \end{cases}$$

The access protocol itself remains unchanged. The suggested change in hash reduces the number of overflows. In fact, if the overflow chain for each bucket is the same, then the hash becomes a perfect hash wherein bucket size increased by $(1 + d)$ times. For concreteness let $d = 1$. Then 1, 3, 5, 7, etc., are the physical buckets to which the records are hashed for transmission of useful data. The space of 1 bucket is left out for overflow. So, the shift values should be computed for the physical buckets 1, 3, 5, 7, etc. Figure 4.11 shows the arrangement of broadcast data separately when hash fuctions $h(x)$ and $h'(x)$ are employed. The logical buckets 1, 2, 3, and 4 correspond to physical buckets 1, 4, 7, and 9 respectively with overflows as indicated to accommodate data. As shown in figure 4.11(b), when hash function $h'(x)$ is used, physical buckets 1, 3, 5, 7, 9, ..., may be used for storing consecutive data records and the remaining physical buckets 2, 4, 6, 8, ..., to serve as overflow buckets. Therefore, the shift values for physical buckets 1, 3, 5, 7, 9 are 0, 1, 2, 2 respectively. In contrast, as figure 4.11(b) depicts, when hash function $h(x)$ is employed the shift values for the physical buckets 1, 2, 3, 4 are 0, 2, 4, 5 respectively.

To calculate the access time, a displacement value $DISP(h, K)$ is introduced. It refers to the displacement needed to reach the physical bucket $B(K)$ containing the record with key K from the bucket at $h'(K)$, i.e., $B(K) - h'(K)$. The expected access time can be computed by finding the per key access time and then averaging it out. The per key expected access time computation depends on two instances.

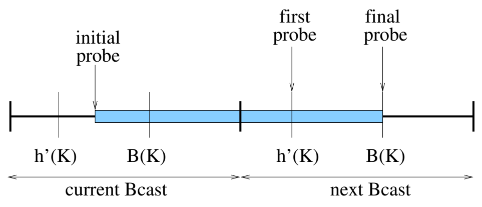
1. If initial probe is in the segment of the broadcast between $h'(K)$ and $B(K)$ then index miss occurs even though data has not appeared in the current broadcast. So, the client must wait until the next broadcast. The access time is then calculated as
$$\frac{DISP(h,K)}{Data(h)} \times \left(Data(h) + \frac{DISP(h,K)}{2}\right),$$
where $Data(h)$ denotes the size of the `Bcast`.

2. If initial probe is outside the displacement area then on an average, the client must wait half the file size and the displacement. So the per key access time in this case will be:
$$\left(1 - \frac{DISP(K)}{Data(h)}\right) \times \left(\frac{Data(h) + DISP(h,k)}{2}\right)$$

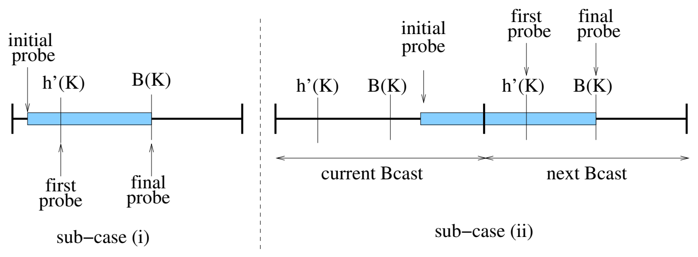
Figure 4.12 depicts the two possible scenarios and explains how the suggested expressions for access times can be derived. In the figure

1. $h'(K)$ denotes the physical bucket $h'(K)$.
2. $B(K)$ represents is the physical bucket where record corresponding key K is found.

According to figure 4.12(a), the part of the waiting time covers between the two consecutive occurrences of physical bucket $B(K)$ constitute a full broadcast cycle. Since, the initial probe can occur any where between $h'(K)$ to



(a) Case 1



(b) Case 2

Figure 4.12: Access time using modified hashing function.

$B(K)$, it adds in an average $DISP(h, K)/2$. The scenario depicted case 2 can be analyzed under two sub-cases: (i) when initial probe occurs before physical bucket $h'(K)$, and (ii) when initial probe occurs after physical bucket $B(K)$. These two sub-cases have been illustrated diagrammatically in figure 4.12(b). Since initial probe can be made any where between physical buckets in ranges $[1, h'(K) - 1]$ and $[B(K) + 1, end]$ the waiting time in an average is $(Data(h) + DISP(h, K))/2$. Once the per key access time is known, the average access time can be determined by summing it over all keys and dividing by the number of records in the broadcast channel. A good hash function is expected to keep the overflow under a finite small bound.

Bibliography

[1] Acharya, S., Franklin, M., and Zdonik, S. Dissemination-based data delivery using broadcast disks. *Personal Communications, IEEE 2*, 6 (2001), 50–60.

[2] Imielinski, T., Viswanathan, S., and Badrinath, B. Power efficient filtering of data on air. In *4th International Conference on Extended Database Technology, (EDBT '94)* (1994), vol. 779 of *LNCS*, pp. 245–258.