Exploiting Versions for Handling Updates in Broadcast Disks

Evaggelia Pitoura
Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
email: pitoura@cs.uoi.gr

Panos K. Chrysanthis*
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, U.S.A.
email: panos@cs.pitt.edu

Abstract

Recently, broadcasting has attracted considerable attention as a means of disseminating information to large client populations in both wired and wireless settings. In this paper, we exploit versions to increase the concurrency of client transactions in the presence of updates. We consider three alternative mediums for storing versions: (a) the air: older versions are broadcast along with current data, (b) the client's local cache: older versions are maintained in cache, and (c) a local database or warehouse at the client: part of the server's database is maintained at the client in the form of a multiversion materialized view. The proposed techniques are scalable in that they provide consistency without any direct communication from clients to the server. Performance results show that the overhead of maintaining versions can be kept low, while providing a considerable increase in concurrency.

1 Introduction

While traditionally data are delivered from servers to clients on demand, a wide range of emerging database applications benefit from a broadcast mode for data dissemination. In such applications, the server repetitively broadcasts data to a client population without a specific request. Clients monitor the broadcast channel and retrieve the data items they need as they arrive on the broadcast channel. Such applications typically involve a small number of servers and a much larger number of clients with similar interests. Examples include stock trading, electronic commerce applications, such as auction and electronic tendering, and traffic control information systems.

The concept of broadcast delivery is not new. Early work has been conducted in the area of Teletext and Videotext systems [3, 23]. Previous work also includes the Datacycle project [10] at Bellcore and the Boston Community Information System (BCIS) [13]. In Datacycle, a database circulates on a high bandwidth network (140 Mbps). Users query the database by filtering relevant information via a special massively parallel transceiver. BCIS broadcasts news and information over an FM channel to clients with personal computers equipped with radio receivers. Recently, data dissemination by broadcast has attracted considerable attention ([12], [19]), due to the physical support for broadcast provided by an increasingly important class of networked environments such as by most wireless computing infrastructures, including cellular architectures and satellite networks. The explosion of data intensive applications and the resulting need for scalable means for providing information to large client populations are also motivated by the dramatic improvements in global connectivity and the popularity of the Internet [9, 24].

As such systems continue to evolve, more and more sophisticated client applications will require reading current and consistent data despite updates at the server. In most current research, updates have been mainly treated in the context of caching (e.g., [6], [2], [11], and [16]). In this case, updates are considered in terms of local cache consistency; there are no transactional semantics. Transactions and broadcast were

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

*This work is supported in part by National Science Foundation under grant IRI-9502091 and IIS-9812532.

first discussed in the Datacycle project [10] where special hardware was used to detect changes of values read and thus ensure consistency. The Datacycle architecture was extended in [4] for the case of a distributed database where each database site broadcasts the contents of the database fragment residing at that site. More recent work involves the development of new correctness criteria for transactions in broadcast environments [22] as well as the deployment of the broadcast medium for transmitting concurrency control related information to the clients so that part of transaction management can be undertaken by the clients [5].

In our previous work [18], we proposed and comparatively studied a suite of techniques for ensuring the consistency of client read-only transactions in broadcast environments. In this paper, we propose maintaining multiple versions of items to increase the concurrency of client transactions. Versions are combined with invalidation reports to inform clients of updates and thus ensure the currency of their reads. We assume that updates are performed at the server and disseminated from there. The currency and consistency of the values read by clients is preserved without requiring clients contacting the servers.

We consider three alternative means for storing older versions. One potential storage medium is the air, in which case, older versions are broadcast along with current values. We introduce protocols for interleaving current and previous versions as well as for determining the frequency of broadcasting old versions. A second proposal is maintaining older versions in the client's cache. In this case, garbage collection of old versions is possible since there is local information about active client transactions and their access requirements. Lastly, we exploit the scenario of maintaining part of the server's database at the client in the form of a multiversion materialized view. The novel aspect is that the base relations are on air. Hybrid approaches where older versions are on air, in cache, and in client's main memory or disk are also possible.

Performance results show that the overhead of maintaining older versions can be kept low, while providing a considerable increase in concurrency. For instance, when about 10% of the broadcast items are updated per broadcast, maintaining 5 versions per item increases the number of consistent read-only transactions that successfully complete their operation from 10% (when no versions are maintained) to 80% – 90%. The increase of the broadcast size is around 10% to 15% of the original size depending on the broadcast organization used. For less update-intensive environments, the overhead is considerably smaller.

The remainder of this paper is organized as follows. Section 2 introduces the problem, defines currency and presents two basic approaches for maintaining correctness. Section 3 describes the multiversioning scheme and related issues. Section 4, 5 and 6 discuss keeping old versions on air, in cache and in a warehouse respectively. Section 7 discusses disconnections and updates, while Section 8 presents our performance model and experimental results. Section 9 concludes the paper.

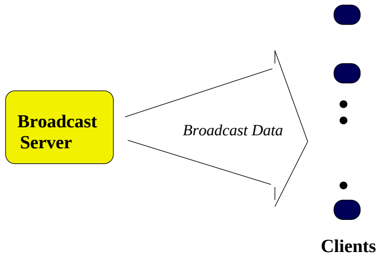

Figure 1: Broadcast Architecture

2 Broadcast and Updates

In a broadcast dissemination environment, a data server periodically broadcasts data items to a large client population (Figure 1). Each period of the broadcast is called a *broadcast cycle* or *bcycle*, while the content of the broadcast is called a *bcast*. Each client listens to the broadcast and fetches data as they arrive This way data can be accessed concurrently by any number of clients without any performance degradation. However, access to data is strictly sequential, since clients need to wait for the data of interest to appear on the channel. We assume that all updates are performed at the server and disseminated from there.

2.1 The Broadcast Model

Clients do not need to listen to the broadcast continuously. Instead, they tune-in to read specific items. Such selective tuning is important especially in the case of portable mobile computers, since they most often rely for their operation on the finite energy provided by batteries and listening to the broadcast consumes energy. However, for selective tuning, clients must have some prior knowledge of the structure of the broadcast that they can utilize to determine when the item of interest appears on the channel. Alternatively, the broadcast can be self-descriptive, in that, some form of directory information is broadcast along with the data. In this case, the client first gets this information from the broadcast and uses it in subsequent reads. Techniques for broadcasting index information along with data are given for example in [15, 11].

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

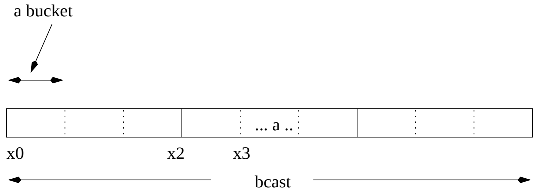

Figure 2: Currency of Updates

of the next bcast when the size of the broadcast is not fixed. Data items correspond to database records (tuples). We assume that users access data by specifying the value of one attribute of the record, the search key. Each bucket contains several items.

2.2 Updates and Consistency

During each bcycle, the server broadcasts items from a database. A database consists of a finite set of data items. A database state is typically defined as a mapping of every data item to a value of its domain. Thus, a databases state, denoted DS, can be defined as a set of ordered pairs of data items in D and their values. In a database, data are related by a number of integrity constraints that express relationships of values of data that a database state must satisfy. A database state is consistent if it does not violate the integrity constraints [8].

While data items are being broadcast, transactions at the server may update their values. There are a number of reasonable choices, regarding the currency of data on the broadcast. For example, the values on the broadcast may correspond to current values at the server, that is to the values produced by all transactions so far committed at the server. Alternatively, updates at the server may not be reflected in the bcast immediately but at pre-specified intervals, such as at each bcast or at fractions of the bcast. We call such intervals *currency intervals*. In particular, we assume that, when an item is to appear on the broadcast, the value that will be broadcast is that produced by all transactions committed at the server by the beginning of the current currency interval (which may not be its current value at the server). For uniformity of presentation, when updates are immediately reflected in the bcast, we say that the currency interval is that of an item.

Figure 2 depicts possible currency intervals. The value of data item a depends on which definition of the currency interval is adopted. For instance, the value of a is the value produced by all transactions committed by x_0, x_2, x_3, or just prior to the broadcast of a, if we assume that the currency interval is the whole bcast, three buckets, a bucket, or an item correspondingly.

A client transaction may read data items from different currency intervals. We define the *span* of a client transaction T, $span(T)$, to be the maximum number of different currency intervals from which T reads data. We define the *readset* of a transaction T, denoted $Read\_Set(T)$, to be the set of items it reads. In particular, $Read\_Set(T)$ is a set of ordered pairs of data items and their values that T read. Our correctness criterion for read-only transactions is that each transaction reads consistent data. Specifically, the readset of each read-only transaction must form a subset of a consistent database state [21].

We make no assumptions about transaction management at the server. Our only assumption is that the values broadcast for each item are those produced by *committed* transactions. Since the set of values broadcast during a single currency interval correspond to the same database state, this set is a subset of a consistent database state. Thus, if for some transaction T, $span(T) = 1$, T is correct. However, since, in general, client transactions read data values from different currency intervals, there is no guarantee that the values they read are consistent.

When information about the readset of a transaction is available, query optimization can be employed to reduce the transaction span. One approach is to re-order reads based on the order by which items appear on the broadcast. Another query optimization technique would be to introduce additional reads. Additional reads may be used to execute reads in all control branches of a query; such an approach is cheap in a broadcast environment, since the data are on air anyway. Such query optimization techniques can effectively reduce the span of a transaction but can not guarantee that all values in the readset would belong to the same currency interval, especially when currency intervals are short.

2.3 Invalidation Techniques

A way to ensure the correctness of read-only transactions is to invalidate, e.g., abort, transactions that read data values that correspond to different database states. To achieve this, a timestamp or version number is broadcast along with the value of each data item. This version number corresponds to the currency interval at the beginning of which the item had the corresponding value. Let v_0 be the currency interval at which a transaction performs its first read. For each subsequent read, we test that the items read have versions $v \leq v_0$. If an item has a larger version, the transaction is aborted. We call this method the *versioning* method. Since the values read by each transaction correspond to the database broadcasted at v_0, the versioning method produces correct read-only transactions.

Another way is to broadcast an invalidation report at pre-specified points during the bcast. The invalidation report includes a list with the data items that have been updated since the previous invalidation report was broadcast. Let us assume that an invalidation report is broadcast at the beginning of each currency interval. In addition, at each client, a set $RS(R)$

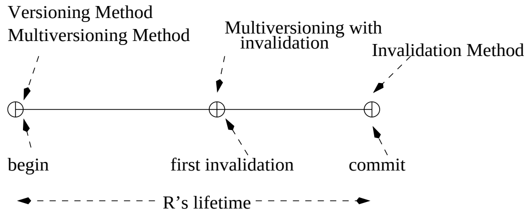

Figure 3: Currency of Reads

is maintained for each active transaction R, that includes all data items that R has read so far. The client tunes in at the pre-specified points to read the invalidation reports. A transaction R is aborted if an item $x \in RS(R)$ appears in the invalidation report, i.e., if x is updated[1]. We call this method the *invalidation* method.

Theorem 1 *The invalidation method produces correct read-only transactions.*

Proof. Let c_c be the currency interval during which a committed transaction R performed its last read operation and DS^{c_c} be the database state that corresponds to the currency interval c_c, i.e., the database state at the beginning of c_c. We claim that the values read by R correspond to the database state DS^{c_c}. For the purposes of contradiction, assume that the value of a data item x read by R is different then the value of x at DS^{c_c}, then an invalidation report should have been transmitted for x at the beginning of c_c and thus R should have been aborted. □

With the versioning method, transaction R reads values that correspond to the database state at the beginning of the currency interval at which R performs its first read operation. With the invalidation method, R reads the most current values as of the beginning of the currency interval at which it commits (Figure 3).

There is no need to transmit invalidation reports at the beginning of each currency interval. Instead, invalidation reports may be broadcast more or less frequently. In the latter case, there is an additional requirement, that before committing, each transaction R must read the next invalidation report that will appear in the broadcast. The proof is similar to the proof above. Reading an additional invalidation report is necessary because in this case, items broadcast between invalidation reports do not necessarily correspond to a single database state. With this variation of the invalidation method, a read-only transaction R reads the most current values as of the time of its commitment.

3 Multiversion Schemes

Invalidation methods are prone to starvation of queries by update transactions. To increase the number of read-only transactions that are successfully processed, we propose maintaining multiple versions of data items. Multiversion schemes, where older copies of items are kept for concurrency control purposes, have been successfully used to speed-up processing of on-line read-only transactions in traditional pull-based systems (e.g., [17]).

3.1 The Basic Multiversion Schemes

The basic idea underlying multiversioning is to temporarily retain older versions of data items, so that the number of aborted read-only transactions is reduced. Versions correspond to different values at the beginning of each currency interval and version numbers to the corresponding currency interval. Thus, there is a trade-off between the length of the currency interval and the number of versions: the shorter the currency interval, the greater the number of versions that are created.

Let S_{max} be the maximum transaction span among all read-only transactions. Let v_0 be the currency interval at which R performs its first read operation. During v_0, R reads the most current versions, that is the versions with the largest version number. In subsequent intervals, for each data item, R reads the version with the largest version number v_c smaller than or equal to v_0. If such a version exists, R proceeds, else R is aborted. In the extreme case, in which, all S_{max} most current values for each data item are available, all read-only transactions proceed safely. We call this scheme *multiversioning*.

Theorem 2 *The multiversioning method produces correct read-only transactions.*

Proof. Let R be a read-only transaction, v_0 the currency interval at which R performs its first read operation and DS^{v_0} be the database state broadcast at this interval. We will show that the values read by R correspond to the database state DS^{v_0} which is consistent and thus R is correct. For any data item $x \in RS(R)$, R reads the version with the largest version number v_c of x, such that $v_c \leq v_0$. This value is the most recent value of x produced by the beginning of the currency interval v_0, that is the value that the item had at DS^{v_0}. □.

In terms of currency, R reads the database state that corresponds to the currency point at v_0 as in the versioning scheme. If invalidation reports are available, we get the following variation of the multiversion method that we call *multiversioning with invalidation* method. Initially, R reads the most current version of each item. Let v_i be the currency interval at which R is invalidated for the first time, i.e., a value that R has read is updated. After v_i, R reads the version with the largest version number v_c such that $v_c < v_i$. If such a version exists, R proceeds, else R is aborted.

[1] A possible optimization is to just mark R as *invalid* if one of its $x \in RS(R)$ appears in an invalidation report and abort R only if it tries to read another data item.

Theorem 3 *The multiversioning with invalidation method produces correct read-only transactions.*

Proof. Let R be a read-only transaction, v_i be the first bcycle during which an item read by R is updated for the first time and DS^{v_i} the database state broadcast at interval v_i. We will show that the values read by R correspond to the database state DS^{v_i-1} which is consistent and thus R is correct. The items that are read before bcycle v_i were not updated prior to v_i thus their values correspond to the database state DS^{v_i-1}. In subsequent bcyles, R reads the version with the largest version number v_c, such that $v_c < v_i$. This value is the most recent value produced before cycle v_i, that is the value that the item had at DS^{v_i-1}. □.

In the multiversioning with invalidation method, R reads the values as of the beginning of the currency interval of its first invalidation v_i, as opposed to the multiversioning method, in which R reads the values that correspond to v_0 (Figure 3). Clearly, multiversioning with invalidation permits better currency than simple multiversioning but at the cost of broadcasting invalidation reports.

3.2 Updates and Caching

To reduce latency in answering queries, clients can cache items of interest locally. Caching reduces the latency of transactions, since transactions find data of interest in their local cache and thus need to access the broadcast channel for a smaller number of times. We assume that each page, i.e., the unit of caching, corresponds to a bucket, i.e., the unit of broadcast. Next, we outline how multiversioning can be used in conjunction with caching.

In the presence of updates, items in cache may become stale. There are various approaches to communicating updates to the clients. Invalidation combined with a form of autoprefetching was shown to perform well in broadcast delivery [2]. The server broadcasts an invalidation report, which is a list of the pages that have been updated. This report is used to invalidate those pages in cache that appear in the invalidation report. These pages remain in cache to be autoprefetched later. In particular, at the next appearance of the invalidated page in the broadcast, the client fetches its new value and replaces the old one. We assume this kind of cache updates in this paper. Other techniques, such as selectively propagating frequently accessed pages [2] that may outperform autoprefetching, should be easily combined with our techniques as well.

To support multiversioning, items in cache also have version numbers. For reading items from the cache, we have to perform the same tests regarding their version numbers as when reading items from the broadcast. To ensure that items in cache are current, the propagation of cache invalidation reports must be at least as frequent as the propagation of invalidation reports for data items. This way, a cached page is either current (i.e., corresponds to the value at the current currency interval) or is marked for auto-prefetch.

3.3 Other Issues

The multiversioning methods can be easily enhanced to handle deletion and insertion of items. When an item is deleted, we create a new version with version number the currency interval, say v, of its deletion and a special field indicating that the item is deleted. A transaction R beginning at v_i with $v_i \geq v$ (or invalidated at v_i if multiversioning with invalidation is used) will read the version with version number v and find out that the item has been deleted. Previous transactions with $v_i < v$ will read versions with smaller version numbers as desired. Similarly, when an item is inserted, we add a version with version number the interval of its insertion.

Another issue is that of the granularity of versions. Instead of maintaining versions of items, it is possible to maintain versions of buckets. Similarly, it is possible to set the invalidation report at the bucket level as well. In this case, to implement the invalidation method, instead of maintaining for each transaction the set of items it has read, we maintain the corresponding set of buckets.

Central to multiversioning is the number of versions maintained per data item. We may always keep the k most current values for each item resulting in a fixed increase in size. Alternatively, we may keep only the different versions of each item during the last k currency intervals and discard older values. In addition, to allocate less space for version numbers, instead of maintaining the absolute number of the currency interval, we can maintain the difference between the current interval and the interval during which the value was updated, i.e., how old the value is. For example, if the current currency interval is interval 30, and the version corresponds to currency interval 27, the version number is set to 3 instead of 27. In this case, $log(S_{max})$ bits are sufficient for version numbers.

Finally, we consider two possibilities for the storage of previous versions. In the *clustering approach*, all versions of the same item are maintained in consequent locations. In the *overflow approach*, older versions are stored separately from the current versions in overflow buckets that are appropriately linked to the current versions.

4 Multiversion Broadcast

With multiversion broadcast, the server, instead of broadcasting the last committed value for each data item, maintains and broadcasts multiple versions for each data item. The number k of older versions that are retained can be seen as a property of the server. In

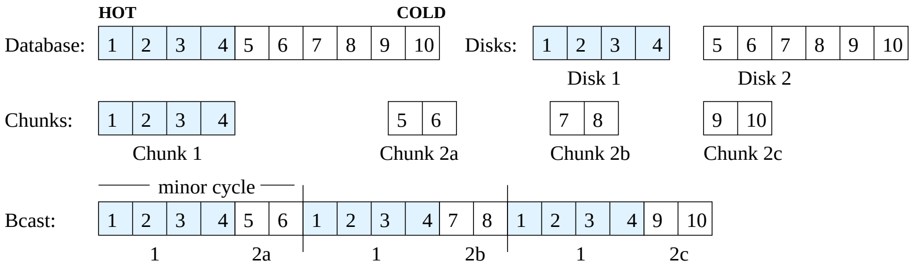

Figure 4: Multiversion broadcast

this sense, a k-multiversion server, i.e., a server that broadcasts the previous k values, is one that guarantees the consistency of all transactions with span k or smaller. Transactions with larger spans can proceed at their own risk; there is a possibility that they will be aborted. The amount k of broadcast reserved for old versions, can be adapted depending on various parameters, such as the allowable bandwidth, feedback from clients, or update rate at the server.

There are two interrelated problems with multiversion broadcast. The first is how to organize the broadcast, that is where to place the old versions. The other is determining the optimal frequency of transmitting versions. In other words, if we consider a broadcast disk organization [1], where specific items are broadcast more frequently than others (i.e., are placed on "faster disks"), at what frequency should old versions be broadcast?

To describe the broadcast disk organization, we will use an example; for a complete definition of the organization refer to [1]. In a broadcast disk organization, the items of the broadcast are divided in ranges of similar access probabilities. Each of these ranges is placed on a separate disk. In the example of Figure 4, buckets of the first disk $Disk_1$ are broadcast three times as often as those in the second disk $Disk_2$. To achieve these relative frequencies, the disks are split into smaller equal sized units called chunks; the number of chunks per disk is inversely proportional to the relative frequencies of the disks. In the example, the number of chunks is 1 (chunk 1) and 3 (chunks 2a, 2b and 2c) for $Disk_1$ and $Disk_2$ respectively. Each bcast is generated by broadcasting one chunk from each disk and cycling through all the chunks sequentially over all disks. A minor cycle is a sub-bcycle that consists of one chunk from each disk. In the example of Figure 4, there are three minor cycles.

4.1 Clustering

Following the clustering approach, one way to structure the broadcast is to broadcast all versions of each item successively. Thus, older versions of hot items (chunk 1 in Figure 5) are placed along with the current values of hot items on fast disks, while versions of cold data (chunks 2a, 2b and 2c) are placed on slow disks. Consequently, clustering works well when each transaction may access any version of an item with equal probability.

The size of each disk, and thus the size of its chunks, is increased to accommodate old versions. The number of chunks per disk, however, remains fixed. The overall increase in the size of the bcast depends on how the hot data items are related to the items that are frequently updated. The increase is the largest when the hot items are the most frequently updated ones since their versions are broadcast more frequently during each bcycle.

Regarding indexing, items are still broadcast in the same disk and disk chunk, however their relative position inside the chunk changes due to the increase of the chunk size. One approach is to broadcast older versions at a special location inside each chunk, e.g., at the end, and chain them to the current versions.

4.2 Overflow Bucket Pool

With the overflow approach, older versions of items are broadcast at the end of each bcycle. In particular, one or more additional minor cycles at the end of each broadcast is allocated to old versions (Figure 5).

Regarding indexing, the offset of the position of the current value of each item in the broadcast from the beginning of the bcast remains fixed. Thus, the server needs not recompute and broadcast an index at each broadcast cycle. Instead, the client may use a locally stored directory to locate the first appearance of a data item in the broadcast. To locate old versions, since their position in the broadcast is fixed, an index can be broadcast before the minor cycle carrying the overflow bucket pool. A transaction that needs to locate old versions first tunes in to read this index. Alternatively, we can maintain a pointer along with the current version of each item pointing to its older version in the overflow pool. After reading an item, if a transaction needs an older version, it uses the pointer to locate it in the overflow bucket.

In the overflow approach, long-running read-only transactions that read old versions are penalized since they have to wait for the end of the bcast to read such versions. However, transactions that are satisfied with current versions do not suffer from a similar increase in latency. On the contrary, in the clustering approach, the overhead in latency due to the increase in the broadcast size is equally divided among all transactions.

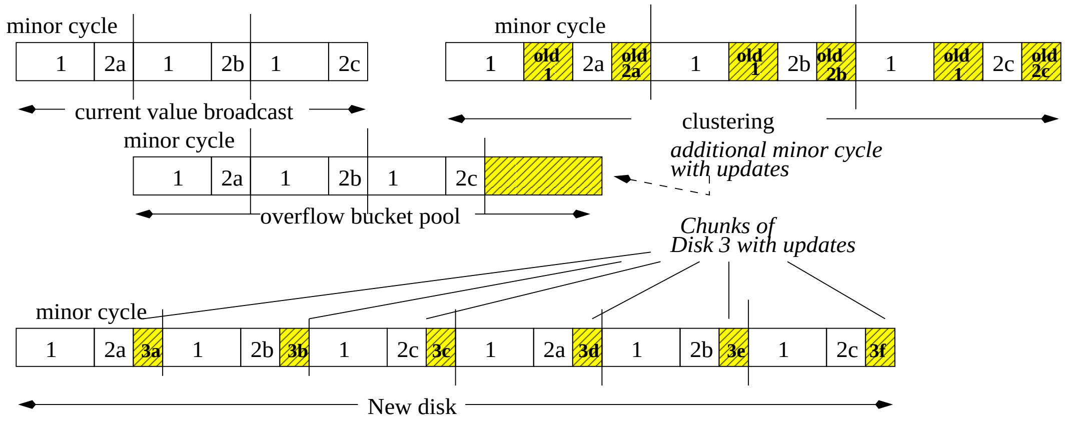

Figure 5: Multiversion broadcast with multiple versions

A drawback of this approach is that the introduction of the additional minor cycle affects the relative speed of each disk. Another problem is that the space allocated to old versions is fixed; it is a multiple of the size of a minor cycle. To avoid this restriction, older versions can be placed on the slowest disk. In this case, the size of the slowest disk and the size of its chunks are increased to accommodate old versions. Old versions are placed on those chunks of the disk that are broadcast last. Again, the relative speed is affected.

4.3 Old Versions on New Disk

With this approach, a new disk is created to hold any old versions. The relative frequency of the disks with the current versions is maintained, by simply multiplying their frequency by a positive number m, so that the slow disk that carries the old versions is m times slower than the disks with the current versions. For instance, say we have a broadcast organization for the current versions consisting of three disks: $Disk_1$ with speed 4, $Disk_2$ with speed 2, and $Disk_3$ with speed 1. We create a new disk $Disk_4$ where we place the old versions. Assume that we want the current versions to be broadcast three times ($m = 3$) more frequently than old versions. Then, we adjust the relative frequencies of the disks as follows: $Disk_1$ now has speed 12, $Disk_2$ has speed 6, $Disk_3$ has speed 3, and $Disk_4$ has speed 1. With this approach, the size of each bcast is also multiplied by m.

Figure 5 shows yet another example. A new disk $Disk_3$ with 6 chunks is created for the old versions. Current items are broadcast twice ($m = 2$) as frequent as old versions. The relative frequency of the two disks is maintained; items of $Disk_1$ are broadcast three times as frequent as items of $Disk_2$. The resulting bcast is twice the size of the original plus the extra space for the old versions.

To locate older versions of items, pointers may be kept along with their current versions. This approach is adaptive. Old versions are placed on faster disks when there are many long-running transactions and on slower disks when most transactions need current values.

5 Multiversion Caching

In multiversion caching, the client cache is used to provide an alternative storage medium for older versions of data items. A version is associated with each cache entry. When an item is updated at the server, its cache entry is not updated; instead, a new entry is inserted in cache for the new version. Thus, for a data item, there may be multiple entries with different version numbers. We assume that the cache replacement policy is such that, the following always hold:

Page Replacement Invariant: For each data item, the versions cached are the most recent ones.

Then, we can either use the multiversioning method or the multiversioning with invalidation method. It is also possible to avoid broadcasting version numbers along with items. In this case, the version number associated with each cached item is the currency interval during which the item was inserted in the cache. This value is larger or equal to the currency interval during which the item was actually updated. In this case, we get the following variation of the multiversioning with invalidation method. Until its first invalidation at currency interval say v_i, a transaction R reads items from the broadcast. After v_i, R only reads items from the cache. In particular, R continues operation as long as there are versions in cache with version numbers $v < v_i$, that is versions inserted in cache prior to the invalidation.

With multiversion caching, the effective cache size is decreased, since part of the cache is used to maintain old versions of items. However, for long-running transactions that read old versions, there may be some speed-up, since older versions may be found in cache. Whereas, k (the number of older versions broadcast) in the multiversion broadcast, is a property of the server, in multiversion caching, k (the number of versions kept in cache) is a characteristic of each client. Transactions at different clients may have varying spans. In this case, it is the client's responsibility to adjust the space in cache allocated to older versions, based on the size of its cache, the requirements and types of its read-only transactions, or other local parameters.

5.1 Garbage Collection

Instead of maintaining in cache all k oldest versions, it is reasonable to maintain only the useful ones. A version is *useful* if it may be read by some invalidated transaction. In this case, the page replacement invariant is revised accordingly.

Page Replacement Invariant(revised): for each data item, the versions cached are the most recent useful ones.

We assume that the multiversioning with invalidation method is used, but the same also holds for the simple multiversioning method, if we just consider begin points of transactions in the place of invalidation points. Let $IL = \{R_i, R_{i+1}, \ldots, R_{i+n}\}$ be the set of all active invalidated transactions, that is all transactions for which one of the items they have read was subsequently updated. Let v_{R_j} be the currency interval that corresponds to the database state read by R_j (that is, R_j was invalidated for the first time at $v_{R_j}+1$). Transactions appear in the list in ascending order of invalidation, that is, R_i is the transaction that was invalidated first.

When an item in cache is updated, the version in cache is invalidated and the new value of the item is autoprefetched. Instead of always maintaining the previous version, that is the version in cache with the largest version number v', we maintain this version only if there is a possibility that it will be read, that is, if there is an R_j in IL that may read v'. This can be tested as follows. Recall that $v_{R_{i+n}}$ is the most recent invalidation point. We discard v', if $v_{R_{i+n}} < v'$. This is because in this case v' is not useful: transactions that will be invalidated in future bcycles will read the newly inserted version, while transactions in the current IL will read versions with version numbers smaller than v'.

Furthermore, when a transaction R_m in IL finishes (aborts or commits), for each item x in cache, we delete all versions that were useful only to R_m. In particular,

Condition for Discarding Versions
When a transaction R_m completes its operation, a version of x with version number v_o is discarded, if all three of the following conditions hold:
(1) there is a version of x with version number v such that $v > v_o$ (i.e., v_o is not the current version),
(2) if R_m is not the most recently invalidated transaction (i.e., $m \neq i+n$), then there is a version with version number v_l such that $v_o < v_l \leq v_{R_{m+1}}$, and
(3) if R_m is not the transaction that was invalidated first $(m \neq i)$, then $v_o > v_{R_{m-1}}$.

We will show that the above conditions are correct (no useful versions are deleted) and optimal (all non-useful versions are deleted).

Correctness: We will show that it is not possible that any transaction will read v_o. Case (a): For each R_j in IL, with $v_{R_j} > v_{R_m}$, it holds $v_{R_j} \geq v_{R_{m+1}}$, thus R_j would not read v_o but a version $v \geq v_l$ where v_l is the version of Condition (2) above. Case (b): For each R_j in IL, with $v_{R_j} < v_{R_m}$, it holds $v_{R_j} \leq v_{R_{m-1}}$, thus R_j would not read v_o, but from Condition (3) above, it will read an item with version number $v_{R_{m-1}}$ or smaller. Case (c): Since from Condition (1), v_o is not the most recent version, any transaction that will be invalidated in the future will not read v_o but a more current version.
Optimality: We will show that if any of the three conditions does not hold useful versions may be deleted. Case (a): Assume that we delete a version v_o for which there is no version in cache with version number v, such that $v > v_o$, then v_o is the current value and may be read by a transaction invalidated at some future point. Case (b): Assume that we delete a version v_o for which there is no v_l, such that $v_o < v_l \leq v_{R_{m+1}}$, then v_o is useful at least to R_{m+1}. Case (c): Assume that we delete a version v_o, such that $v_o \leq v_{R_{m-1}}$, then R_{m-1} may read v_o.

5.2 Page Replacement

When older versions are maintained in cache, the page replacement policies must be revised. An approach that offers flexibility is to divide the cache space into two parts: one that maintains current versions and one that maintains older ones. In this case, different cache replacement policies can be used for each part of the cache. This approach provides for adaptability, since the percentage of cache allocated to older versions can be adjusted dynamically. The most suitable organization for this approach is overflow buckets with old versions placed on the old version part of the cache.

Another approach is to apply a global policy, that is to replace the page with the overall smallest probability of being accessed without considering version numbers. Clustering works better with this approach. Finally, to maintain the invariant, when a version of item x with version number v is selected for replacement, we must in addition discard from the cache all x's versions with version numbers $v' < v$.

6 Multiversion Warehouse

In this scenario, the client stores the data of interest locally. Data are defined and maintained using views defined over base relations. When the base data are updated the view becomes stale. Updating the view to reflect changes of base data is called view maintenance.

View maintenance is a well known and studied problem (e.g., see [14] for a survey) which is beyond the scope of this paper. Here, we only focus on views in terms of broadcast and versioning.

In the broadcast setting, for scalability reasons, we assume that the server is stateless. In particular, the server cannot maintain any views in lieu of its clients. Furthermore, the server is not aware of the views maintained at its clients. In addition and in contrast to [25], we assume that there is no direct communication from clients to the server. Specifically, the client cannot ask the server to compute the view.

One main advantage of the broadcast model is that the base relations are available to clients without any storage overhead. In fact, the base relations are on air and thus using them to recompute the view is not expensive in terms of communication messages. To maintain the view, we create a client transaction that we call the view maintenance transaction. The view maintenance transaction has two parts: the first part, called view-query, recomputes the view, while the second part, called view-updater, installs the updates in the local view.

The view-query part is executed as a normal client read-only transaction concurrently with any query processing at the client. The view-query recomputes the view to take into account any updates. Depending on the currency requirements, any of the versioning, invalidation or multiversioning methods can be used by the view-query. For instance, if the most-up-to-date values as of the commitment of the view-query are required, then the view-query must use the invalidation method. The view-query can either recompute the view from scratch or use an incremental technique. Furthermore, a locally stored index can be used to speed-up the processing of the query and decrease its span. The view-updater installs the updates at the client. To allow reads at the client to proceed concurrently with the view updater, two versions of data may be kept along the lines of [20].

With this simple view maintenance scheme, the server is not aware of the fact that clients maintain views and thus there is no associated overhead at the server. Furthermore, there is no need to modify the content of the broadcast. An important issue is that of the currency of the locally maintained view that can be decoupled from the currency of the broadcast data. The maintenance transaction may run periodically or when updates occur. In the second case, the invalidation reports can be used to trigger the execution of view maintenance.

7 Related Issues

7.1 Disconnections

In many settings, it is desirable for clients not to monitor the broadcast continuously as for example, in the case of clients carrying portable devices and thus seeking for reducing battery power consumption. Further, access to the broadcast may be monetarily expensive, and thus minimizing access to the broadcast is sought for. Finally, client disconnections are very common when the data broadcast are delivered wirelessly. Wireless communications face many obstacles because the surrounding environment interacts heavily with the signal, thus in general wireless communications are less reliable and deliver less bandwidth than wireline communications. In such cases, clients may be forced to miss a number of broadcast cycles.

In general, versioning frees transactions from the requirement of reading invalidation reports. When there are no versions, a transaction can not tolerate missing any invalidation reports. Furthermore, with multiversioning, client transactions can refrain from listening to the broadcast for a number of cycles and resume execution later as long as the required versions are still on air. In general, a transaction R with $span(R) = s_R$ can tolerate missing up to $k - s_R$ currency intervals in any k-multiversion broadcast. The tolerance of the multiversion scheme to intermittent connectivity depends also on the rate of updates, i.e., the creation of new versions. For example, if the value of an item does not change during m, $m > k$, currency intervals, this value will be available to read-only transactions for more intervals. Multiversion caching further improves tolerance to disconnections. In this case, disconnected operation is supported, since a read-only transaction can proceed without reading data from the broadcast, as long as appropriate versions can be found in cache.

7.2 Update Transactions

While read-only transactions can proceed without contacting the server, update transactions must communicate their updates to the server for certification. Multiversion concurrency control for update transactions is also possible. Actually, it is easy to provide *snapshot isolation* introduced in [7] and supported by a number of databases vendors. To this end, we outline an implementation of the *first-committer-wins* method [7].

Regarding reads, update transactions at the client proceed like read-only transactions; if their reads are invalidated, they are aborted. Regarding updates, values of items updated at the client are maintained locally and transmitted to the server for certification. They are incorporated and included in subsequent broadcast intervals only if certified successfully.

Specifically, a client update transaction T records the currency interval v_{init} during which T performed its first read (or in the case of invalidation reports, the currency interval $v_{inval}-1$, where v_{inval} is the currency interval of T's first invalidation). Further, it records the currency interval v_{commit} in which it completes its operation. When T completes its operation, the client sends to the server the list of items $WS(T)$ written by

T and their values, the commit interval v_{commit}, and the initial interval v_{init}.

At the server, T is certified and committed, if for all transactions T' with v'_{commit} in $[v_{init}, v_{commit}]$ $WS(T) \cap WS(T') = \emptyset$. To perform this test, we simply check the current version numbers of the items written by T.

Snapshot isolation is not equivalent to serializability. For example, it suffers from the write skew anomaly, e.g., two transactions read two items x and y and each modifies one of them resulting in a violated constraint between x and y. However if all update transactions transform the system from one consistent state to another, snapshot isolation will guarantee consistent reads. To ensure serializability (e.g., one-version serializability [8]), stronger tests are required for update transactions, such as also checking their readsets.

8 Performance Evaluation

In this section, we evaluate the performance of multiversion methods with respect to various parameters.

8.1 The Performance Model

Our performance model is similar to the one presented in [1]. The server periodically broadcasts a set of data items in the range of 1 to $NoItems$. We assume a broadcast disk organization with 3 disks and relative frequencies 5, 3 and 1. The client accesses items from the range 1 to *ReadRange*, which is a subset of the items broadcast (*ReadRange* $\leq$ $NoItems$). Within this range, the access probabilities follow a Zipf distribution. The Zipf distribution with a parameter $theta$ is often used to model non-uniform access. It produces access patterns that become increasingly skewed as $theta$ increases. The client waits $ThinkTime$ units and then makes the next read request.

Updates at the server are generated following a Zipf distribution similar to the read access distribution at the client. The write distribution is across the range 1 to *UpdateRange*. We use a parameter called *Offset* to model disagreement between the client access pattern and the server update pattern. When the offset is zero, the overlap between the two distributions is the greatest, that is the client's hottest pages are also the most frequently updated. An offset of k shifts the update distribution k items making them of less interest to the client. We assume that during each bcycle, N transactions are committed at the server. All server transactions have the same number of update and read operations, where read operations are four times more frequent than updates. Read operations at the server are in the range 1 to $NoItems$, follow a Zipf distribution, and have zero offset with the update set at the server.

The client maintains a local cache that can hold up to $CacheSize$ pages. The cache replacement policy is LRU: when the cache is full, the least recently used page is replaced. When pages are updated, the corresponding cache entries are invalidated and subsequently autoprefetched. The currency interval is a bcast. Table 1 summarizes the parameters that describe the operation at the server and the client. Values in parenthesis are the default.

8.2 Performance Results

Due to space limitations, we only present some representative results to show the applicability of the method. Figure 6 shows the increase of the size of the broadcast using each one of the proposed multiversion broadcast organization schemes. In all experiments, we used the simple multiversion schemes (without invalidation reports). For the clustering approach, the increase depends on the offset. The increase is the maximum when the hot items are the most updated ones (*Offset* = 0), while it is minimum when the frequently updated items are cold and thus their versions are placed on slow disks. For the new disk approach, the size of the broadcast is doubled from the case of no versions. However, the current value of each item appears twice as often as in the no version case, thus, it is as if we had an additional bcycle. The increase shown for this case is only that for broadcasting versions on the slowest disk.

Figure 7 shows the decrease of transactions aborted due to updates. With the overflow pool approach, transactions have to wait for the end of the broadcast to locate old versions, thus their span increases as does their probability of abort. For the new disk approach, since the broadcast size is effectively double the size of the other methods, for the same update rate the updates are 200 and 100 correspondingly. However, these updates appear in the broadcast very late (the currency interval is that of a bcast, thus in this case, it is also two times larger than in the other two). Thus, we pay for the increase in concurrency, by reading less current data.

Regarding caching, using part of the cache space to keep old versions results in a very small increase in concurrency of long running transactions. This is because less space is allocated to current versions and transactions have to read items from the broadcast, thus their span increase and so does their abort rate. Thus, our conclusion is that it is better to keep older versions in secondary memory than in cache. In this case, garbage collection results in a dramatic decrease of the space required to maintain old versions (e.g., for maintaining up to 3 old versions per item in cache, a same size secondary storage is sufficient).

9 Conclusions

Data dissemination by broadcast is an important mode for data delivery in data intensive applications. In this

Server Parameters		**Client Parameters**	
No of Items Broadcast	1000	ReadRange (range of client reads)	500
UpdateRange	500	theta (zipf distribution parameter)	0.95
theta (zipf distribution parameter)	0.95	Think Time (time between client reads in broadcast units)	2
Offset (update and client-read access deviation)	0 - 250 (100)	Number of reads per quey	5 - 50 (20)
Number of updates at the server	50 - 500 (50)	S (transaction span)	varies
Currency interval	bcast		
Broadcast Disk Parameters		**Cache**	
No of disks	3	CacheSize	125
Relative frequency: Disk1, Disk2, Disk3	5, 3, 1	Cache replacement policy	LRU
No of items per range (disk) Range1, Range2, Range3	75, 175, 750	Cache invalidation	invalidation + autoprefetch

Table 1: Performance Model Parameters

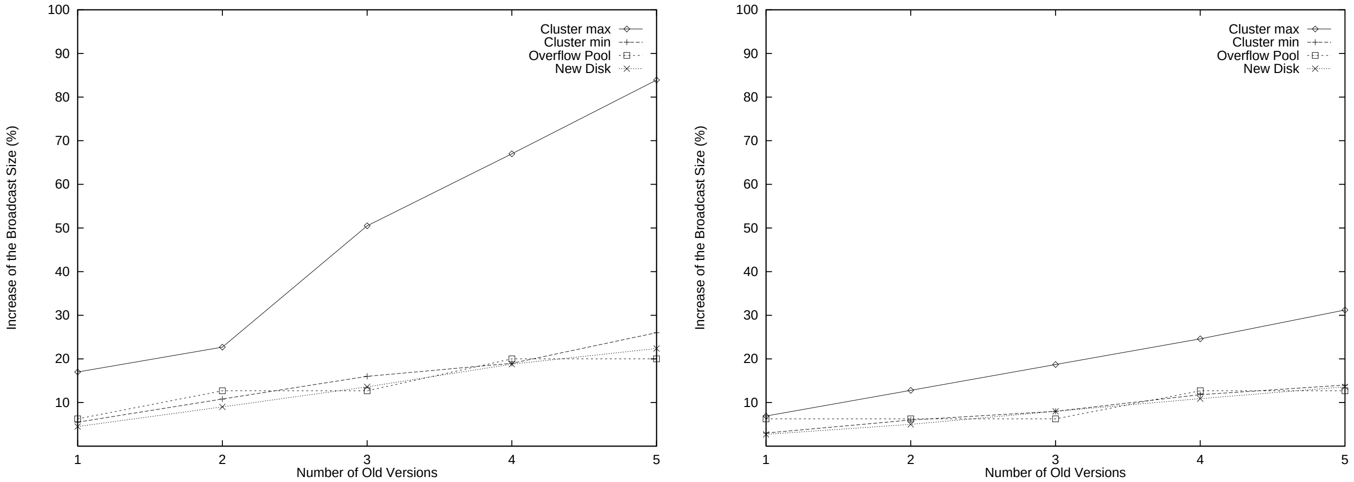

Figure 6: Increase of the broadcast size. For the figure at the left updates are set to 100 per bcast, while for the figure at the right to 50.

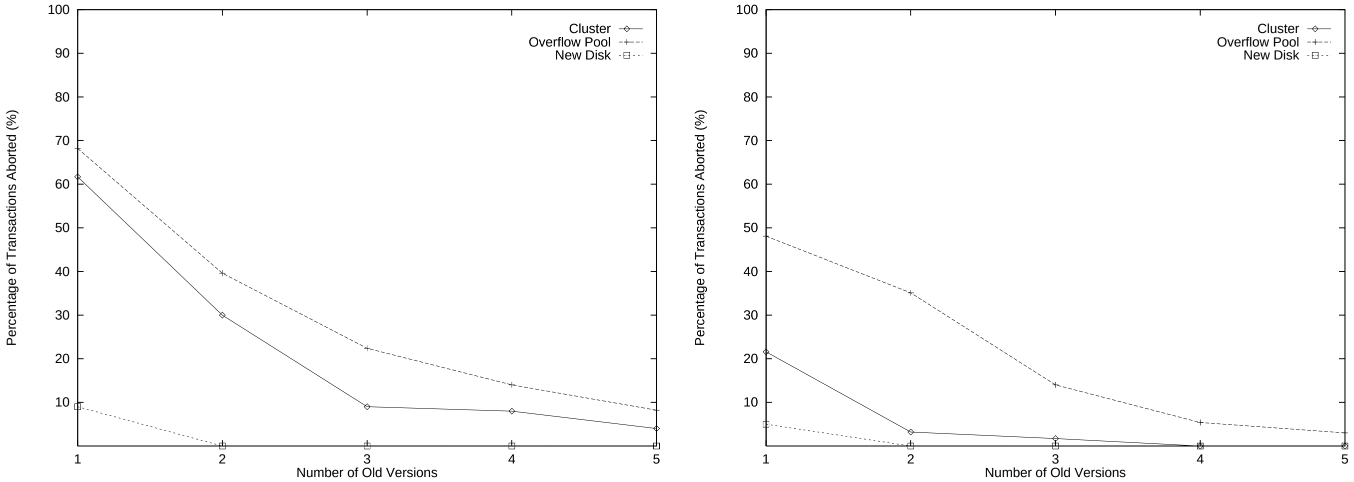

Figure 7: Abort Rate. For the figure at the left updates are set to 100 per bcast (in this case, when no versions are maintained, the abort rate is 88.5%) while for the figure at the right the updates are set at 50 per bcast (in this case, when no versions are maintained, the abort rate is 83%).

paper, we propose maintaining multiple versions to increase the concurrency of read-only transactions in the presence of updates. Invalidation reports are also used to ensure the currency of reads. The approach is scalable in that it is independent of the number of clients. Performance results show that the overhead of maintaining versions can be kept low, while providing a considerable increase in concurrency.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 199–210, 1995.

[2] S. Acharya, M. J. Franklin, and S. Zdonik. Disseminating Updates on Broadcast Disks. In *Proc. of the 22nd Int'l Conf. on Very Large Data Bases*, pp. 354–365, 1996.

[3] A. H. Ammar and J. W. Wong. The Design of Teletext Broadcast Cycles. *Performance Evaluation*, 5(4), 1985.

[4] S. Banerjee and V. O. K. Li. Evaluating the Distributed Datacycle Scheme for a High Performance Distributed System. *Journal of Computing and Information*, 1(1), 1994.

[5] D. Barbará. Certification Reports: Supporting Transactions in Wireless Systems. In *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, 1997.

[6] D. Barbará and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–12, 1994.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil., and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–10, 1995.

[8] P. A. Bernstein, V. Hadjilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] A. Bestavros and C. Cunha. Server-initiated Document Dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3):3–11, 1996.

[10] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib. The Datacycle Architecture. *Communications of the ACM*, 35(12):71–81, 1992.

[11] A. Datta, A. Celik, J. Kim, D. VanderMeer, and V. Kumar. Adaptive Broadcast Protocols to Support Efficient and Energy Conserving Retrieval from Databases in Mobile Computing Environments. In *Proc. of the 13th IEEE Int'l Conf. on Data Engineering*, pp. 124–133, 1997.

[12] M. J. Franklin and S. B. Zdonik. A Framework for Scalable Dissemination-Based Systems. In *Proc. of the OOPSLA Conf.* , pp. 94–105, 1997.

[13] D. Gifford. Polychannel Systems for Mass Digital Communication. *Communications of the ACM*, 33(2):141–150, 1990.

[14] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Data Engineering Bulletin*, 18(3):3–18, June 1995.

[15] T. Imielinski, S. Viswanathan, and B. R. Badrinanth. Data on Air: Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, 1997.

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

[17] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and Flexible Methods for Transient Versioning to Avoid Locking by Read-Only Transactions. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 124–133, 1992.

[18] E. Pitoura and P. K. Chrysanthis. Scalable Processing of Read-Only Transactions in Broadcast Push. In *Proc. of the 19th IEEE Int'l Conf. on Distributed Computing Systems*, 1999.

[19] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.

[20] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *Proc. of the 1997 SIGMOD Intl. Conf. on Management of Data*, pp. 393–404, 1997.

[21] R. Rastogi, S. Mehrotra, Y. Breitbart, H. F. Korth, and A. Silberschatz. On Correctness of Nonserializable Executions. In *Proc. of ACM Symposium on Principles of Database Systems*, pp. 97–108, 1993.

[22] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient Concurrency Control for Broadcast Environments. In *ACM SIGMOD Int'l Conf. on Management of Data*, 1999.

[23] J. Wong. Broadcast Delivery. *Proc. of the IEEE*, 76(12), 1988.

[24] T. Yan and H. Garcia-Molina. SIFT – A Tool for Wide-area Information Dissemination. In *Proc. of the 1995 USENIX Technical Conf.* , pp. 177–186, 1995.

[25] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proc. of the 1995 SIGMOD Intl. Conf. on Management of Data*, pp. 316–327, 1995.