Scalable Processing of Read-Only Transactions in Broadcast Push

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

Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh
panos@cs.pitt.edu

Abstract

Recently, push-based delivery has attracted considerable attention as a means of disseminating information to large client populations in both wired and wireless settings. In this paper, we address the problem of ensuring the consistency and currency of client read-only transactions in the presence of updates. To this end, additional control information is broadcast. A suite of methods is proposed that vary in the complexity and volume of the control information transmitted and subsequently differ in response times, degrees of concurrency, and space and processing overheads. The proposed methods are combined with caching to improve query latency. The relative advantages of each method are demonstrated through both simulation results and qualitative arguments. Read-only transactions are processed locally at the client without contacting the server and thus the proposed approaches are scalable, i.e., their performance is independent of the number of clients.

1. Introduction

In traditional client/server systems, data are delivered on demand. A client explicitly requests data items from the server. Upon receipt of a data request, the server locates the information of interest and returns it to the client. This form of data delivery is called *pull-based*. In many wireless settings, such as in satellite and cellular networks, the server machine is provided with a relative high-bandwidth channel which supports broadcast delivery to all mobile clients located inside the geographical region it covers. This facility provides the infrastructure for a new form of data delivery called *push-based* delivery. Broadcast is supported in wireline networks as well. In push-based data delivery, the server repetitively broadcasts data to a client population without a specific request. Clients monitor the broadcast and retrieve the data items they need as they arrive on the broadcast channel.

Push-based delivery is central to an increasingly important range of applications that involve dissemination of information to a large number of clients. Dissemination-based applications include information feeds such as stock quotes and sport tickets, electronic newsletters, mailing lists, road traffic management systems, and cable TV. Important are also electronic commerce applications such as auctions or electronic tendering. Recently, information dissemination on the Internet has gained significant attention (e.g., [7, 19]) as well.

In this paper, we address the problem of preserving the consistency and currency of client read-only transactions, when the values of broadcast data are updated at the server. To this end, control information is broadcast that enables the validation of read-only transactions at the clients. We propose various methods that vary in the complexity and volume of control information, including transmitting invalidation reports, multiple versions per item, and serializability information. Caching at the client is also supported to decrease query latency. The performance of the methods is evaluated and compared through both qualitative arguments and simulation results. In all the methods proposed, consistency is preserved without contacting the server and thus the methods are scalable; i.e., their performance is independent of the number of clients. This property makes the methods appropriate for highly populated service areas.

Providing transactional support tailored to read-only transactions is important for many reasons. First, the great majority of transactions in dissemination systems are read-only. Then, even if we allow update transactions at the client, it is more efficient to process read-only transactions with special algorithms. That is because consistency of queries can be ensured without contacting the server. This is important because even if a backchannel exists from the client to the server, this channel typically has small communication capacity. Furthermore, since the number of clients supported is large, there is a great chance of overwhelming the server with clients' requests. In addition, avoiding

contacting the server decreases the latency of client transactions. The proposed methods are applicable in wired as well as in wireless settings.

In most current research, updates have been treated in the context of caching (for example, [5], [2], and [12]). In this case, updates are considered in terms of local cache consistency; there are no transactional semantics. Transactions and broadcast were first discussed in the Datacycle project [8] where special hardware is used to detect changes of values read and thus ensure consistency. The Datacycle architecture is extended in [3] for the case of a distributed database where each database site broadcasts the contents of the database fragments residing at that site. More recent work involves the development of new correctness criteria for transactions in broadcast environments [18] and the deployment of the broadcast medium for transmitting concurrency control related information so that part of transaction management can be undertaken by the clients [4].

The remainder of this paper is organized as follows. In Section 2, we introduce the problem and in Section 3, we propose various methods for processing read-only transactions. The methods are extended to support caching in Section 4. In Section 5, the performance of the methods proposed is compared through both qualitative arguments and simulation results. Finally, in Section 6, we offer conclusions and present our plans for future work.

2. Read-Only Transactions and Broadcast

2.1. The Broadcast Model

The server periodically broadcasts data items to a large client population. Each period of the broadcast is called a broadcast *cycle* or *bcycle*, while the content of the broadcast is called a *bcast*. Each client listens to the broadcast and fetches data as they arrive. We assume that all updates are performed at the server and disseminated from there. Clients access data from the broadcast in a read-only mode. We do not make any particular assumptions about transaction processing, i.e., concurrency control or recovery, at the server.

Clients do not need to listen to the broadcast continuously. Instead, they can tune-in to read specific items. 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 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 data (see for instance [11]). In this case, the client first gets this information from the broadcast and use it in subsequent reads.

The smallest logical unit of a broadcast is called *bucket*. Buckets are the analog to blocks for disks. Each bucket has a header that includes useful information. 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. 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. Consistency of Read-Only Transactions

We assume that the server broadcasts the content of a database. A database consists of a finite set of data items. A database state is typically defined as a mapping of every data 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 [6].

While data items are being broadcast, transactions executed at the server may update their values. We assume that the content of the broadcast at each cycle is guaranteed to be consistent. In particular, we assume that the values of data items that are broadcast during each bcycle correspond to the state of the database at the beginning of the cycle, i.e., the values produced by all transactions that have been committed by the beginning of the bcycle. Thus, a read-only transaction that reads all its data within a single bcycle can be executed without any concurrency overhead at all. We make this assumption for clarity of presentation.

Since the set of items read by a transaction is not known at static time and access to data is sequential, transactions may read data items from different bcasts, that is values from different database states. As a simple example, consider the transaction that corresponds to the following program: *if a > 0 then read b else read c*, where b and c precede a in the broadcast. Then, a client transaction has to read a first and wait for the next bcycle to read b or c. We define the *span* of a transaction T, $span(T)$, to be the maximum number of the different bcycles from which T reads data.

Since client transactions read data from different cycles, there is no guarantee that the values they read are consistent. 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 [17]. We assume that each server transaction preserves database consistency. Thus, a state produced by a serializable execution (i.e., an execution equivalent to a serial one [6]) of a number of server transactions produces a consistent database state. The goal of the methods presented in this paper is to ensure that the readset of each read-only transaction corresponds to such a state.

3. Read-Only Transaction Processing

3.1. The Invalidation-Only Method

Each bcast is preceded by an invalidation report in the form of a list that includes all data items that were updated at the server during the previous bcycle. For each active read-only transaction R, the client maintains a set $RS(R)$ of all data items that R has read so far. At the beginning of each bcast, the client tunes in and reads the invalidation report. A read transaction R is aborted if an item $x \in RS(R)$ was updated, that is if x appears in the invalidation report. Clearly,

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

Proof. In [14].

With the invalidation-only method, a read-only transaction R reads the most current values, in particular the last values written by transactions committed by the beginning of the bcycle at which R commits. The increase in the size of the broadcast is equal to $\lceil \frac{u\,k}{b} \rceil$, where u is the number of items updated, k is the size of the key and b the bucket size.

3.2. Multiversion Broadcast

The invalidation-only method is prone to starvation of queries by update transactions. To minimize the number of invalidated and aborted read-only transactions, older versions of data items are retained temporarily. In particular, the server, instead of broadcasting the last committed value for each data item, maintains and broadcasts multiple versions per item. Multiversion schemes, where older copies of items are kept for concurrency control purposes, have been successfully used to speed-up processing of online read-only transactions in traditional pull-based systems (e.g., [13]). Let c_0 be the bcycle during which a client transaction R performs its first read operation. During c_0, transaction R reads the most up-to-date value for each data item, that is, the value having the largest version number. In later cycles, R reads the value with the largest version number c_n, such that $c_n \leq c_0$. If such a value exists, R proceeds, else R aborts.

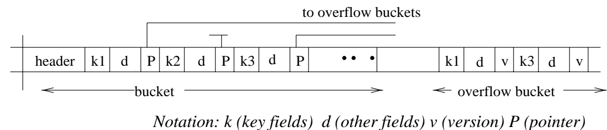

Figure 1. Multiversion broadcast

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

Proof. In [14].

In terms of currency, the data items read by R correspond to the database state at the beginning of c_0. If for each data item, all its S previous values, i.e., the values during the previous S bcycles, are available, where S is the maximum transaction span among all read-only transactions, then, all read-only transactions can proceed successfully by reading older versions of data when necessary. If, instead, the server broadcasts V older versions, for some constant $V < S$, then some read-only transactions may be aborted. A server can either maintain a constant number V of versions per item or just the different values of each item during the V previous cycles (that may be less than V). In any case, the number V of older versions that are retained can be seen as a property of the server. In this sense, a V-multiversion server, i.e., a server that broadcasts the previous V values, is one that guarantees the consistency of all transactions with span V or smaller. The amount 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.

Multiversion Broadcast Organization. There are various ways to organize a multiversion broadcast [14]. One approach is to broadcast old versions at the end of the bcast. In particular, instead of broadcasting with each data item all its versions, a single version, the most recent one, is broadcast along with a pointer. The pointer points to the older versions of the item, if any, that are broadcast at the end of the bcast in *overflow* buckets (Figure 1). This way, for each data item, the offset of its position in the bcast from the beginning of the bcast remains fixed. Thus, the server needs not recompute and broadcast an index at each bcycle. Instead, the client may use a locally stored directory to locate the first appearance of a data item in the broadcast and, if needed, follow the pointer to locate older versions in the overflow bucket. The drawback is that long-running read-only transactions that read old versions are penalized since they have to wait for the end of the bcast. However, transactions that are satisfied with current versions do not suffer from a similar increase in latency.

Let v be the size of the version number, k the size of the key, d the size of the other attributes and u the mean num-

ber of updates per bcycle and S the maximum transaction span. The size of the data buckets is $D(k+d+P)$, where P is the size of the pointer, while the total size of the overflow buckets is $B = \lceil \frac{u(S-1)(k+v+d)}{b} \rceil$. The pointer is kept as the offset of the beginning of the overflow bucket from the end of the bcast, and thus be analog to the number of overflow buckets, in particular $P = log(B)$. To allocate less space for version numbers, instead of broadcasting the number of the bcycle during which a version was created, we broadcast the difference between the current bcycle and the bcycle during which the version was created, i.e., how old the version is. For example, if the current bcycle is cycle 30, and a version was created during bcycle 27, we broadcast 3 as the version of the data value instead of 27. Then, $log(S)$ bits are sufficient for v.

3.3. Serialization-Graph Testing

Both the invalidation-only and the multiversion schemes ensure that transactions read consistent values, i.e., values produced by a serializable execution, by enforcing transactions to read values that correspond to the content of a single bcast. However, it suffices for transactions to read values that correspond to any consistent database state not necessarily one that is broadcast. To this end, we use a conflict serialization graph testing (SGT) method.

The serialization graph for a history H, denoted $SG(H)$, is a directed graph whose nodes are the committed transactions in H and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of T_i's operations precedes and conflicts with one of T_j operations in H [6]. According to the serialization theorem, a history H is serializable iff $SG(H)$ is acyclic. We assume that each transaction reads a data item before it writes it, that is, the readset of a transaction includes its writeset. Then, in the serialization graph, there are two types of edges $T_i \rightarrow T_j$ between any pair of transactions T_i and T_j: *dependency* edges that express the fact that T_j read the value written by T_i and *precedence* edges that express the fact that T_j wrote an item that was previously read by T_i.

In brief, the SGT method works as follows. Each client maintains a copy of the serialization graph locally. The serialization graph at the server includes the transactions *committed* at the server, while the local copy at a client site includes in addition all active read-only transactions issued at the site. At each cycle, the server broadcasts any updates of the serialization graph. Upon receipt of the updates, the client integrates them into its local copy of the graph. A read operation at a client is accepted only if it does not create a cycle in the local serialization graph. Else, the issuing transaction is aborted. The serialization graph at the server is not necessarily used for concurrency control at the server, instead a more practical method, e.g., two-phase locking, may be employed.

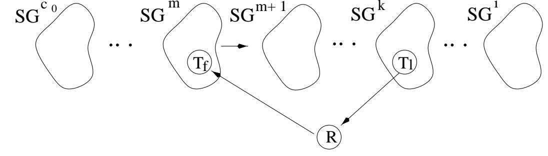

Figure 2. At bcycle $i+1$, R reads x from T_l committed during bcycle k ($c_0 \leq k \leq i$). T_f committed during bcycle m ($c_0 \leq m \leq i$) overwrote an item previously read by R.

Implementation of the SGT Method. Next, we describe an implementation of the SGT method based on the assumption that histories are strict. A history is *strict* if no data may be read or overwritten until the transaction that previously wrote into it terminates. The SGT method is applicable to other cases as well but with additional overhead. At the beginning of bcast $i+1$, the server broadcasts the following control information:

- the *difference from the previous serialization graph*
  In particular, the server broadcasts for each transaction that was committed during bcycle i, a list of the transactions with which it conflicts, i.e., it is connected through a direct edge.
- an *augmented invalidation report*
  The report includes all data written during bcycle i along with an identification of the first transaction that wrote each of them during bcycle i.

In addition, the content of the broadcast is augmented so that along with each item, the identification of the last transaction that wrote it is also broadcast.

At the beginning of each bcycle $i+1$, each client tunes in to obtain the control information and updates its local copy SG of the serialization graph to include any additional edges and nodes. In addition, the client adds precedence edges for all its active read-only transactions as follows. Let R be an active transaction and $RS^i(R)$ be the set of items that R has read so far. For each item x in the augmented invalidation report such that $x \in RS^i(R)$, the client adds a precedence edge $R \rightarrow T_f$, where T_f is the first transaction that wrote x during bcycle i. Although R conflicts with all transactions that wrote x during bcycle i, it suffices to just add one edge to T_f (see [14] for a proof).

When R reads an item y, a dependency edge $T_l \rightarrow R$ is added in SG, where T_l is the last transaction that wrote y. The read operation is accepted, only if no cycle is formed. It can be shown that it suffices to just add one edge $T_l \rightarrow R$ instead of adding edges $T' \rightarrow R$ from all transactions

T' that wrote y. To prove that the SGT method detects all cycles that include a read-only transaction R, we will use the following lemma. Let SG^i be the subgraph of SG that includes only the transactions committed during bcycle i.

Lemma 1 *Let c_0 be the first broadcast cycle during which an item read by R is overwritten.*

(a) During broadcast cycle $i + 1$, the only type of cycle that can be formed that includes R is of the form $R \rightarrow T_{j_1} \rightarrow T_{j_2} \rightarrow \ldots T_{j_k} \rightarrow R$, where for any $T_{j_p} \in SG^m$, it holds $c_0 \leq m \leq i$.

(b) The SGT method detects all such cycles.

Proof. In [14].

Figure 2 shows graphically the formation of such a cycle.

Theorem 3 *The SGT method produces correct read-only transactions.*

Proof. In [14].

Regarding the database state seen by R, R is serialized before all update transactions that overwrote items read by R and after all update transactions from which R read from. This is similar to the approximation criterion of [18] when applied to read-only transactions. The main difference is that to implement the method, we broadcast control information in a form of a serialization graph, while they broadcast information per pair of data items.

Space Efficiency. Instead of keeping a complete copy of the serialization graph at each client, by Lemma 1, it suffices to keep for each read-only transaction R only the subgraphs SG^m with $m \geq c_0$, where c_0 is the bcycle when the first item read by R was invalidated, i.e., overwritten. Thus, if no items are updated, there is no space or processing overhead at the client. Furthermore, at most S subgraphs are maintained, where S is the maximum transaction span of queries at each client. Also, by Lemma 1, we need keep only the outgoing edges of R.

The volume of control information is considerable. Let tid be the size of a transaction identifier, N the maximum number of transactions committed per bcycle, and c the maximum number of operations per server transaction. We assume that transaction identifiers are unique within each bcycle, thus it suffices to allocate $log(N)$ bits per transaction identifier when the bcycle is known. To distinguish between transactions at different bcycles, a version number is broadcast indicating the bcycle at which the transaction was committed; the size of such version number is $log(S)$ bits, since only the last S bcycles are relevant. The size of the broadcast data is $\lceil \frac{D(d+k+log(N))}{b} \rceil$, while the size of the invalidation report is $\lceil \frac{u(k+log(N))}{b} \rceil$. Since, there are at most c operations per transaction, each transaction participates in at most c conflicts with other transactions. Thus, the difference from the previous graph has at most $N\,c$ edges. The total size of the difference is: $\lceil \frac{cN(log(N)+(log(S)+log(N)))}{b} \rceil$, assuming that we broadcast pair of conflicting transactions where the first transaction in the pair is a newly committed transaction, and the second one any previously committed transaction with which it conflicts. If we broadcast the control information at the end of the previous bcast, then the offset of each item from the beginning of each bcast remains fixed and a locally stored directory can be used.

4. Caching

To reduce latency in answering queries, clients can cache items of interest locally. Caching reduces not only the latency but also the span 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 cycles. We assume that each page, i.e., the unit of caching, corresponds to a bucket, i.e., the unit of broadcast. In the presence of updates, the value of cached items may become stale. There are various approaches to communicating updates to the client. We assume invalidation combined with a form of autoprefetching [2]. Other approaches are also applicable. In particular, we assume that at the beginning of each bcycle, 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. The invalidated pages remain in cache. When the new value of an invalidated page appears in the broadcast, the client fetches the new value and replaces the old one. Thus, a page in cache either has a current value (the one in the current bcast) or is marked for autoprefetching. The cache invalidation report is similar to the invalidation reports in our methods. However, the two reports differ in granularity. The cache invalidation report includes pages (buckets) that have been updated, whereas the query-processing invalidation report includes data items.

The proposed read-only transaction processing techniques can be easily extended to accommodate caching. For the invalidation-only scheme, each read first checks whether the item is in cache. If the item is found in cache and the page is not invalidated, the item is read from the cache. Otherwise, the item is read from the broadcast. A simple enhancement to the above scheme is to extend the cache so that along with each item, it also includes the bcycle during which the item was inserted in the cache. Let R be a query and u_0 the first bcycle at which an item $x \in RS(R)$ is invalidated. Instead of aborting R, R is marked abort and continues operation as long as old enough values are found in cache. In particular, a read operation is accepted if the item is in cache and has version number c, $c < u_0$. We call this method invalidation-only with versioned cache.

Table 1. Performance model parameters

Server Parameters		Client Parameters	
D (BroadcastSize)	1000	ReadRange (range of client reads)	250
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 (10)
ServerReadRange	1000	S (transaction span)	varies
N (number of server transactions)	10		
Offset (update and server-read access deviation)	0	**Cache**	
u (Number of updates at the server)	50 - 500 (50)	CacheSize	125
c (Number of operations per server trans)	(u + 4 * u) / N	Cache replacement policy	LRU
k (size of the key field)	1 unit	Cache invalidation	invalidation + autoprefetch
d (size of the other fields)	5 * k		
b (bucket size)	d units		

Theorem 4 *The invalidation-only with versioned cache method produces correct read-only transactions.*

Proof. In [14].

To support the multiversion broadcast method, the cache must also include the version number for each item. Analogously, for the SGT method, the cache must be extended to include for each item the last transaction that wrote it; information that is broadcast anyway. In addition, each time an item is read from the cache, the same test for cycles as when the item is read from the broadcast is performed.

5. Performance Evaluation

5.1. 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 $BroadcastSize$. We assume for simplicity a flat broadcast organization in which the server broadcasts cyclicly the set of items. The client accesses items from the range 1 to $ReadRange$, which is a subset of the items broadcast ($ReadRange \leq BroadcastSize$). Within this range, the access probabilities follow a Zipf distribution with a parameter $theta$ to model non-uniform access. Access patterns become increasingly skewed as $theta$ increases. The client waits $ThinkTime$ units and then makes the next read request. Similarly, updates at the server are generated following a Zipf distribution. 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.

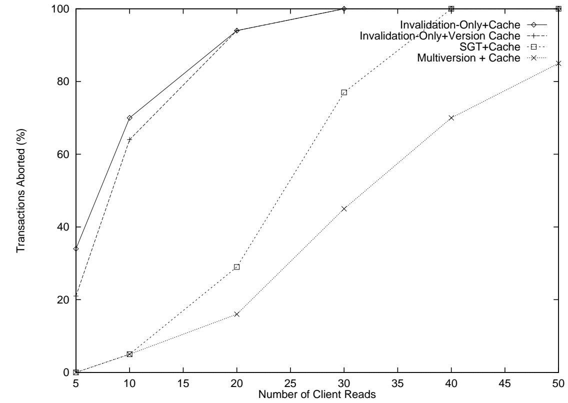

Figure 3. Abort rate with the number of read operations per client transaction

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 $BroadcastSize$, 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. Table 1 summarizes the parameters. Values in parenthesis are the default.

5.2. Comparison of the Methods

Concurrency. Updates at the server may invalidate data values read by read-only transactions and cause them to be aborted and reissued anew. Figure 3 depicts the abort rate for the schemes presented with caching at the client. Caching reduces the number of transactions aborted since it reduces their span and thus the probability of invalidation. For the multiversion scheme, a total of three versions per item is maintained. The abort rate also depends on the update rate and the overlap between the client read and the server update pattern. For results refer to [14].

Broadcast Size. The increase of the broadcast volume is an important measure of the efficiency of the proposed schemes, in terms of bandwidth. Furthermore, the volume of the broadcast data affects the response time of client transactions. Since access to data is sequential, the larger the volume of the broadcast, the longer the clients need to wait until the data of interest appear on the channel. Figure 4 shows the increase of the broadcast size as a function of the maximum transaction span and the number of updates

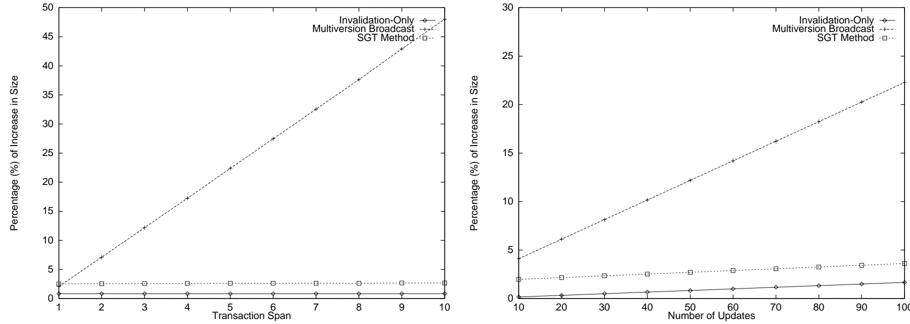

Figure 4. Increase in the size of the broadcast: (left) with the transaction span (for $U = 50$ updates per bcycle) (right) with the number of updates (for $span = 3$)

using the formulas developed in the previous sections.

Latency. We quantify latency, the mean duration of read-only transactions, as the mean number of bcycles per transaction. Besides reading control information at each bcycle, from the methods presented, multiversion broadcast imposes an additional increase in latency, since long-running read-only transactions wait for old versions to appear at the end of the bcast. For quantitative results refer to [14].

Currency Read-only transactions can be classified based on their currency requirements [10]. *Currency requirements* specify what update transactions are reflected by the data read by read-only transactions. Table 2 summarizes the currency properties of read-only transaction in each of the methods.

Disconnections. The techniques presented differ on whether they require active clients to monitor the broadcast continuously. Raising the continuous monitoring requirement is desirable in various settings. For example, in the case of mobile clients, operation relies on the finite power provided by batteries, and since listening to the broadcast consumes energy, selective tuning is required. Besides, access to the broadcast is monetarily expensive, and thus minimizing access to the broadcast is sought for. Finally, client disconnections [16] are very common when the data broadcast are delivered wirelessly.

In the invalidation-only scheme, a client has to tune-in at each and every cycle to read the invalidation report. Otherwise, it cannot ensure the correctness of any active read-only transaction. In multiversion broadcast, clients 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 $V - s_R$ broadcast cycles in any V-multiversion broadcast. The tolerance to disconnections 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 k, $k > V$, cycles, this value will be available to any read-only transactions for more than V cycles. The SGT method does not tolerate any client disconnections. If a client misses a broadcast cycle, it cannot anymore guarantee serializability. Thus, any active read transactions must be reissued anew. To increase tolerance to disconnections, version numbers could be broadcast. In this case, a read operation is accepted iff its version number is smaller than the version of the last broadcast that the transaction has listen to. This guarantees that the client has all the information required for cycle detection. In all the schemes, periodic retransmission of control information can increase their tolerance to intermittent connectivity. For instance, an invalidation report of the items updated during the last w bcycles may be broadcast to allow clients to resynchronize. Finally, caching improves tolerance to disconnections.

6. Conclusions and Future Work

We have presented a suite of processing techniques to provide support for consistent queries for broadcast push in both wired and wireless settings with mobile or stationary clients. The techniques are scalable in that their performance is independent of the number of clients. We have compared the proposed techniques both quantitatively and through simulation and show their relative advantages. The proposed techniques can be extended in various ways. First, we may raise the assumption that the values broadcast at each bcycle are those at the beginning of the cycle. Second, possible refinements of the proposed schemes refer to the supported granularity. For example, invalidation reports may include buckets instead of items. Finally, another possible extension is to consider a broadcast-disk organization [1], where specific items are broadcast more frequently than others, i.e., are placed on “faster disks”. Along this line, in [15] we address the problem of determining the optimal fre-

Table 2. Summary

	Invalidation-Only	Multiversion Broadcast	SGT Method	Caching
Concurrency (percentage of transactions accepted)	Minimum	Maximum (depends on number of versions)	Moderate (depends on the trans activity at the server)	(depends on the cache size)
Processing Overhead	Small	Moderate	Considerable (includes maintaining SGs at both the server and the client)	Small
Size (increase of the broadcast volume)	depends on the update rate (1% for U = 50 updates and span = 3)	depends on the update rate and the span (12 % for U = 50 updates and V = 3)	depends on the activity at the server (2.5 % for N = 10 server trans and U = 50 updates)	(small, for transmitting invalidation reports for buckets)
Latency (number of bcycles)	Not affected	Increases for long transactions	Not affected	Decreases
Currency (database state seen by the clients)	The state when the last read is performed	The state when the first read operation is performed	A state between the first and the last operation	(for invalidation with versioned caching) the state when an item previously read is overwritten for the first time
Tolerance to Disconnections	None	Some, depends on the individual transaction's span and the update rate	None, unless additional information is broadcasted	Some, depends on the update rate and the cache size

quency for transmitting old versions.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. In *Proc. of SIGMOD*, 1995.

[2] S. Acharya, M. J. Franklin, and S. Zdonik. Disseminating Updates on Broadcast Disks. In *Proc. of VLDB*, 1996.

[3] 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.

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

[5] D. Barbará and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proc. of SIGMOD*,1994.

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

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

[8] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib. The Datacycle Architecture. *CACM*, 35(12), December 1992.

[9] 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. Conf. on Data Engineering*, 1997.

[10] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database. *ACM TODS*, 7(2), 1982.

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

[12] 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), 1997.

[13] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and Flexible Methods for Transient Versioning to Avoid Locking by Read-Only Transactions. In *Proc. of SIGMOD*, 1992.

[14] E. Pitoura and P. K. Chrysanthis. Scalable Processing of Read-Only Transaction in Broadcast Push (extended version). Tech. Report TR: 98-26, Univ. of Ioannina, Computer Science Dept, 1998. Also available at:www.cs.uoi.gr/~ pitoura/pub.html.

[15] E. Pitoura and P. K. Chrysanthis. Exploiting Versions for Handling Updates in Broadcast Disks. Tech. Report TR: 99-02, Univ. of Ioannina, Computer Science Dept, 1999.

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

[17] R. Rastogi, S. Mehrotra, Y. Breitbart, H. F. Korth, and A. Silberschatz. On Correctness of Non-serializable Executions. In *Proc. of ACM PODS*, 1993.

[18] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient Concurrency Control for Broadcast Environments. In *ACM SIGMOD*, 1999.

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