



Processing Read-Only Transactions in Hybrid Data Delivery Environments with Consistency and Currency Guarantees

ANDRÉ SEIFERT and MARC H. SCHOLL

University of Konstanz, Database Research Group, P.O. Box D188, D-78457 Konstanz, Germany

Abstract. Different isolation levels are required to ensure various degrees of data consistency and currency to read-only transactions. Current definitions of isolation levels such as Conflict Serializability, Update Serializability or External Consistency/Update Consistency are not appropriate for processing read-only transactions since they lack any currency guarantees. To resolve this problem, we propose four new isolation levels which incorporate data consistency and currency guarantees. Further, we present efficient implementations of the proposed isolation levels. Our concurrency control protocols are envisaged to be used in a hybrid mobile data delivery environment in which broadcast push technology is utilized to disseminate database objects to a large number of mobile clients and conventional point-to-point technology is applied to satisfy on-demand requests. The paper also presents the results of a simulation study conducted to evaluate the performance of our protocols. According to the simulation results the costs imposed by the MVCC-SFBS protocol, which ensures serializability to read-only transactions are moderate relative to those imposed by the MVCC-SFBUS and MVCC-SFBVC protocols, which provide weaker consistency guarantees. A comparison study reveals that the MVCC-SFBVC scheme outperforms all other investigated concurrency control schemes suitable for mobile database systems.

Keywords: read-only transactions, broadcasting, unicasting, data currency, data consistency

1. Introduction and motivation

Consider applications (e.g., road traffic information services, online auctions, stock market tickers, etc.) that may employ broadcast technology to deliver data to a large number of clients. Most of such applications requiring transactional guarantees initiate read-only transactions. Running such transactions efficiently despite the various limitations of a mobile broadcasting environment is a challenging research topic addressed in this paper.

Irrespective of the environment (central or distributed, wireless or stationary) in which read-only transactions are processed, they have the potential of being managed more efficiently than their read–write counterparts especially if special concurrency control (CC) protocols are applied. Multi-version CC schemes [14,24,33] appear to be an ideal candidate for read-only transaction processing in broadcasting environments since they allow read-only transactions to execute without any interference with concurrent read–write transactions. If multiple object versions are kept in the database system, read-only transactions can read older object versions and, thus, never need to wait for a read–write transaction to commit or to abort in order to resolve the conflict. As with read–write transactions, read-only transactions may be executed with various degrees of consistency. Choosing lower levels of consistency than serializability for transaction management is attractive for two reasons. First, the set of correct multi-version histories that can be produced by a scheduler can be increased and, hence, higher performance (transaction throughput) can be achieved. Further, weaker consistency levels may allow read-only transactions to read *more recent*

object versions. Thus, weaker consistency levels trade consistency for transactional performance and data currency.

While reading current data is necessary for read–write transactions to maintain database consistency during updates, such requirements are not necessary for read-only transactions to be scheduled in a serializable way. That is, read-only transactions can be executed with serializability guarantees even though observing out-of-date database snapshots. Read-only transactions may therefore be allowed to specify various levels of data currency requirements. In order to provide reliable guarantees for the behavior of the database system, we need well-defined isolation levels (ILs) suitable for read-only transactions which guarantee both data consistency and data currency. The ANSI/ISO SQL-92 specifications [9] define four ILs, namely *Read Uncommitted*, *Read Committed*, *Repeatable Read*, and *Serializability*. Those levels do not incorporate any currency guarantees, though, and thus are unsuitable for managing read-only transactions in distributed mobile database environments.

Theory and practice have pointed out the inadequacy and imprecise definition of the SQL ILs [11] and some redefinitions have been proposed in [7]. Additionally, a range of new ILs were proposed that lie between the Read Committed and Serializability levels. The new intermediate ILs were designed for the needs of read–write transactions with only three of them explicitly stating the notion of logical time. The level called *Snapshot Isolation* (SI) proposed by Berenson et al. [11], ensures data currency to both read-only and read–write transactions forcing them to read from a data snapshot that existed by the time the transaction started. Oracle's *Read Consistency* (RC) level [26] provides stronger currency guarantees than Snapshot Isolation by guaranteeing that each

SQL statement in a transaction T_i sees the database state at least as recent as it existed by the time T_i issued its first read operation. For subsequent read operations/SQL statements RC ensures that they observe the database state that is at least as recent as the snapshot seen by the previous read operation/SQL statement. Finally, Adya [6] defines an IL named *Forward Consistent View* (FCV) that extends SI by allowing a read-only (read–write) transaction T_i (T_j) to read object versions created by read–write transactions after T_i 's (T_j 's) starting point, as long as those reads are consistent in the sense that T_i (T_j) sees the total effects of all update transactions it write–read or (write–read/write–write) depends on.

The above mentioned levels are not ideally suitable for processing read-only transactions for a number of reasons. First, all of them are weaker consistency levels, i.e., read–write transactions executed at any of these levels may violate consistency of the database since none of them requires the strictness of serializability. Consequently, read-only transactions may observe an inconsistent database state, if they view the effects of transactions that have modified the database in an inconsistent manner. Inconsistent or bounded consistent reads may not be acceptable for some mobile applications, thus making non-serializability levels that do not ensure database consistency to such transactions inappropriate. Another problem arises from the fact that mobile database applications may need various data currency guarantees depending on the type of application and actual user requirements. The ILs mentioned above provide only a limited variety of data currency guarantees to read-only transactions. All levels ensure that read-only transactions read from a database state that existed at a time not later than the transaction's starting point. Such firm currency guarantees may be too restrictive for some mobile applications. Hence, there is a need for definition of new ILs that incorporate weaker currency guarantees. Moreover, we need to define new ILs that meet the specific requirements of (mobile) read-only transactions.

This paper's contributions are as follows. First, we define four new ILs that provide useful consistency and currency guarantees to mobile read-only transactions. In contrast to the ANSI/ISO SQL-92 ILs [9] and their modifications by Berenson et al. [11], our definitions are not stated in terms of existing concurrency control mechanisms including locking, timestamp ordering, and optimistic schemes, but are rather independent of such protocols in their specification. Second, we have designed a suite of multi-version concurrency control algorithms that efficiently implement the proposed ILs. Finally, we present the performance results of our protocols and compare them. To our knowledge, this is the first simulation study that validates the performance of concurrency control protocols providing various levels of consistency and currency to read-only transactions in a mobile hybrid data delivery environment.

The remainder of the paper is organized as follows. In section 2, we introduce some notations and terminology that is necessary for the formal definition of new ILs. In section 3, we define new ILs especially suitable for mobile read-only transactions by combining both data consistency and currency

guarantees. Implementation issues are discussed in section 4. Section 5 reports on the simulation study and performance tradeoffs. Section 6 contains the conclusions of our work and highlights some direction of our future research activities.

2. Preliminaries

A transaction T_i is a sequence of operations $OP_i = (op_{i1}, op_{i2}, \dots, op_{in})$ where each data operation op_{ij} is either an atomic read or write action. The j th write operation within T_i on object X is denoted $w_{ij}[x_i, v]$, where v is the value written into X_i , and the subscript i assigned to object X is a non-decreasing object version identifier equal to the transaction identifier of T_i . When transaction T_i reads object version X_j as its j th operation that had been written by T_j , we denote this action by $r_{ij}[x_j, v]$. Read and write operations on the same data granules are partially ordered according to $<_i$ and each transaction T_i is associated with three transaction management operations: begin b_i , and commit c_i , or abort a_i , i.e., each transaction either commits or aborts. The set of such primitives executed by T_i are denoted P_i . All transactional operations are recorded in a history H in the (real-time) order in which they are performed. For performance reasons, multiple transactions may be executed concurrently, i.e., operations of different transactions can be interleaved in H . We assume that all data items initially stored in the database had been produced by an initialization transaction T_0 , and are called zero versions. Subsequent transactions that modify a zero version of a data item create a new version and assign their unique transaction identifier to it. Finally, we define the notion of a multi-version history by extending the definition of a single-version history as follows.

Definition 1 (Single-version history). A single-version history SVH of a set of transactions $T = \{T_0, T_1, \dots, T_n\}$ is a partial order $(\Sigma_T, <_{SVH})$ of events such that:

1. $\Sigma_T = \bigcup_{i=1, \dots, n} OP_i \cup \bigcup_{i=1, \dots, n} P_i$.
2. $<_{SVH} \supseteq \bigcup_{i=1, \dots, n} <_i$.
3. If p, q are data operations in SVH and at least one of them is a write operation, then either $p <_{SVH} q$ or $q <_{SVH} p$.

Definition 2 (Multi-version history). A multi-version history MVH of a set of transactions $T = \{T_0, T_1, \dots, T_n\}$ is a single-version history SVH extended by a version function v that maps each read operation r_i to some object version written by a write action w_j that precedes the read operation according to $<_{MVH}$, i.e., like in SVH, an object version may not be read by a transaction until it has been created. Additionally, a version order, denoted \ll , is associated with each committed object in MVH representing a total order among the versions of each object.

For notational convenience, we assume that the version order of an object X in a multi-version history MVH corresponds to the order in which write operations of X occur in

MVH, i.e., whenever write operation $w_{ij}[x_i, v]$ immediately precedes write operation $w_{kj}[x_k, v]$ in MVH, then $x_i \ll x_k$. To determine whether a multi-version history MVH satisfies certain criteria defined by an IL, a subhistory of MVH may need to be considered. The projection P of a multi-version history MVH with respect to a single transaction is given below.

Definition 3 (Transactional projection). Let $\text{top} \in \{r, w, a, c, b\}$ denote a transactional operation that is either a data operation or a transaction management operation. A transactional projection of a multi-version history MVH onto T_i , denoted $P(\text{MVH}, T_i)$, is a subhistory of MVH containing transactional operations $\text{top}(\text{MVH}') := \text{top}(T_i)$, i.e., MVH' includes only the operations issued by T_i .

It is important to note that the projection preserves the relative order of the original operations. To validate the correctness of multi-version histories with respect to ILs defined in section 3, we need to formalize possible direct and indirect data dependencies between transactions.

Definition 4 (Direct write-read dependency). A direct write-read dependency between T_i and T_j exists if there is a write operation w_j which precedes a read operation r_i in MVH according to $<_{\text{MVH}}$ and T_i accesses the object version written by T_j . In what follows, we denote such a dependency wr .

Definition 5 (Direct write-write dependency). A transaction T_i directly write-write depends on a transaction T_j if there exists a write operation w_j which precedes a write operation w_i in MVH according to $<_{\text{MVH}}$, and w_j produces the predecessor object version of some object version written by w_i . We denote write-write dependencies ww .

Definition 6 (Direct read-write dependency). A direct read-write dependency occurs between two transactions T_i and T_j if there is a read operation r_i and a write operation w_j in MVH in the order $r_i <_{\text{MVH}} w_j$ and w_j installs the successor object version of the object version read by r_i . Read-write dependencies are denoted rw .

If the type of dependency between two distinct transactions does not matter, we say that they are in an arbitrary dependency.

Definition 7 (Arbitrary direct dependency). Two transactions T_i and T_j are in an arbitrary direct dependency in MVH, if there exists a direct read-write, write-write or write-read dependency between T_i and T_j .

Definition 8 (Arbitrary indirect dependency). A transaction T_i arbitrary indirectly depends on a transaction T_j in a multi-version history MVH, if there exists a sequence $[T_j \delta T_{k1} \delta T_{k2} \dots \delta T_{kn} \delta T_i (n \geq 1)]$ in MVH where δ denotes an arbitrary direct dependency between two transactions.

3. New isolation levels suitable for read-only transactions

3.1. Why serializability may be insufficient

Serializability is the standard criterion for transaction processing in both stationary and mobile computing. Its importance and popularity is related to the fact that it prevents read-write transactions from violating database consistency by assuring that they always transform the database from one consistent state into another. With respect to read-only transactions, serializability as defined in [12] guarantees that all read-only transactions perceive the same serial order of read-write transactions. Additionally, serializability requires that read-only transactions serialize with each other. However, the serializability criterion in itself is not sufficient for preventing read-only transactions from experiencing anomalies related to data currency.

Example 1. An example illustrating this pitfall is shown in the following (non-serial but serializable) multi-version history:

MVH₁: $b_0 w_0[x_0, 2:40 \text{ pm}] b_1 r_1[z_0, \text{cloudy}]$
 $w_0[y_0, 2:50 \text{ pm}] c_0 w_1[z_1, \text{blizzard}] c_1 b_2$
 $r_2[z_1, \text{blizzard}] r_2[x_0, 2:40 \text{ pm}]$
 $w_2[x_2, 2:50 \text{ pm}] c_2 b_3 r_3[x_0, 2:40 \text{ pm}] b_4$
 $r_4[x_2, 2:50 \text{ pm}] b_5 r_5[z_1, \text{blizzard}]$
 $r_5[y_0, 2:50 \text{ pm}] r_3[y_0, 2:50 \text{ pm}] c_3$
 $w_5[y_5, 3:00 \text{ pm}] c_5 r_4[y_5, 3:00 \text{ pm}] c_4$

History MVH₁ might be produced by a flight scheduling system supporting multiple object versions, which is rather the rule than an exception in mobile distributed database systems. In MVH₁, transaction T_0 is a blind write transaction that initializes the flight status (take-off times) of flights X and Y , respectively, and T_1 is an event-driven transaction initiated automatically by the airport weather station since the weather monitoring system indicates an imminent weather change. Due to the weather forecast the Air Traffic Control Center instantly delays both scheduled flights by 10 minutes. At the same time, two employees of the ground personnel equipped with PDAs query the airport flight scheduling system in response to passengers' requests to check the actual take-off times of flights X and Y (T_3 and T_4). While one of the employees (who invokes transaction T_3) may locate the required data in his local cache, the other (who invokes transaction T_4)

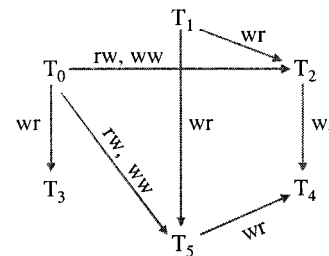


Figure 1. Multi-version serialization graph of MVH₁.

may have to connect to the central database in order to satisfy his data requirements. As a consequence, both persons read from different database snapshots without serializability guarantees being violated, which can be easily verified by sketching the multi-version serialization graph (MVSG) of MVH_1 .

As the above example clearly illustrates, serializability by itself may not be a sufficient requirement for avoiding phenomena related to reading from old database snapshots. This shortage is eliminated in the following subsections.

3.1.1. BOT serializability

Encouraged by example 1, we now define two new ILs that combine the strictness of serializability with firm data currency guarantees. Unlike the ANSI definition of serializability, our definition ensures well-defined data currency to read-only transactions. The existing ANSI specification of serializability and its redefinition by Adya et al. [7] do not contain any data currency guarantees for read-only transactions. Under those levels, read-only transactions are allowed to be executed without any restrictions with respect to the currency of the observed data. We will define our ILs in terms of histories. We associate a directed graph with each newly defined isolation level IL_i . A multi-version history MVH provides IL_i guarantees, if the corresponding graph is acyclic.

Due to space restrictions, we define only such ILs that are especially attractive for the mobile broadcasting environment where clients run data-dissemination applications forced to read (nearly) up-to-date database objects and are expected to be rarely disconnected from the server. Based on some research done on real-time transactions [1,18], we divide data currency requirements into three categories: transactions with strong, firm, and weak requirements. We say that a read-only transaction T_i has *strong* currency requirements, if it is forced to read data that is (still) up-to-date by T_i 's commit time. Since all read operations of T_i must be valid at the end of the transaction's execution, we also specify that T_i runs with End of Transaction (EOT) data currency guarantees. Note that the EOT data currency property requires only that writes of committed read-write transactions must not interfere with operations of read-only transactions, i.e., object updates of uncommitted transactions are not considered by that property.

The *firm* concurrency requirement, in turn, provides slightly weaker currency guarantees. It requires that any data item read by a read-only transaction T_i must be at least as recent as by the time T_i started its execution. Similarly to the strong data currency requirement, the firm criterion is concerned only with objects installed by committed read-only transactions when checking transaction's validity. Firm currency requirements are attractive for the processing of read-only transactions in mobile broadcasting environments for mainly two reasons: first, and most importantly from the data currency perspective, they guarantee that read-only transactions observe up-to-date or nearly up-to-date data objects, which is an important criterion for data-dissemination applications such as news and sports tickers, stock market monitors, traffic and parking information systems, etc. Second, contrary to

the strong currency requirements, they can easily and instantaneously be validated at the client site without any communication with the server.

For some mobile database applications, however, *weak* data currency requirements may suffice. These requirements can be declared in at least two ways. A read-only transaction can be forced to observe a database state the way it existed at a certain point in time t_i before its actual starting point. That is, all object versions read by a transaction must have been up-to-date by the time t_i . A user may also want to observe a transaction consistent state of the database that existed at some point within the time interval $[i, j]$. Despite unquestionable attractiveness of weaker currency requirements especially for applications running on clients with frequent disconnections, we believe that the majority of data-dissemination applications require firm currency guarantees. Thus, in this paper we focus on firm currency guarantees and leave out the extension of known ILs by strong and weak data currency requirements for future study. Prior to specifying a new IL that provides serializability along with firm data currency guarantees, some additional concepts are to be introduced.

As defined so far, a multi-version history MVH consists of two components: (a) a partial order of database events (Σ_T), and (b) a total order of object versions (\ll). Now, we extend the definition of a multi-version history by specifying for each committed read-only transaction a start time order that relates its starting point to the commit time of previously terminated read-write transactions. The association of a start time order with a multi-version history was first introduced in the context of the *Snapshot Isolation* (SI) level definition [11] to provide more flexibility for implementations of this degree. According to the SI concept, the database system is free to choose a starting point for a transaction as long as the selected starting point is some (logical) time before its first read operation. Allowing the system to choose a transaction's starting point without any restrictions is inappropriate in situations where the user expects to read from a database state that existed at some time close to the transaction's actual starting point. Thus, for applications/transactions to work correctly, the database system needs to select a transaction's starting point somehow in accordance with the order of events in MVH. We now formally define the concept of start time order.

Definition 9 (Start time order). A start time order of a multi-version history MVH over a set of committed transactions $T = \{T_0, T_1, \dots, T_n\}$ is a partial order (S_T, \rightarrow_{MVH}) of events such that:

1. $S_T = \bigcup_{i=1, \dots, n} \{c_i, b_i\}$.
2. $\forall T_i \in T, b_i \rightarrow_{MVH} c_i$.
3. If $T_i, T_j \in T$, then either $b_i \rightarrow_{MVH} c_j$ or $c_j \rightarrow_{MVH} b_i$ or $(b_i \rightarrow_{MVH} c_j \text{ and } b_j \rightarrow_{MVH} c_i)$.
4. If $w_i, w_j \in MVH, w_i \ll w_j, c_j \rightarrow_{MVH} b_k$, then $c_i \rightarrow_{MVH} b_k$.

According to statement 1 the start time order relates begin and commit operations of committed transactions in MVH.

Point 2 states that a transaction's starting point always precedes its commit point. Condition 3 states that a scheduler S has three possibilities in ordering the start and commit points of any committed transactions T_i and T_j in MVH. A scheduler S may choose T_i 's starting point before T_j 's commit point or vice versa or, if both transactions are concurrent, neither starts its execution after the other transaction has committed. Condition 4 means that if S chooses T_k 's starting point after T_j 's commit point and T_j overwrites the object installed by T_i then T_i 's commit point must precede T_k 's starting point in any start time order.

For notational convenience, in what follows, we do not specify a start time order for all committed transactions in MVH. Instead, we only associate with each MVH the start time order between read-only and read-write transactions. Now we can define a data currency property required for the definition of the BOT serializability IL.

Definition 10 (BOT data currency). A transaction T_i possesses BOT data currency guarantees if for all read operations invoked by T_i the following invariant holds:

1. If the pair $w_j[x_j]$ and $r_i[x_j]$ is in MVH, then $c_j \rightarrow_{\text{MVH}} b_i$.
2. If there is another write operation $w_k[x_k]$ of a committed transaction T_k in MVH, then either
 - (a) $c_k \rightarrow_{\text{MVH}} b_i$, $x_k \ll x_j$, or
 - (b) $b_i \rightarrow_{\text{MVH}} c_k$.

Note that we ignore transaction aborts in our definition of BOT data currency since subsequent definitions that incorporate this criterion only consider MVHs of committed transactions. On the basis of the BOT data currency property, the serializability IL can be extended as follows.

Definition 11 (BOT serializability). A multi-version history MVH over a set of read-only and read-write transactions is BOT serializable, if MVH is serializable in the sense that the projection of MVH onto all committed transactions in MVH is equivalent to some serial history $\text{MVH}_{\text{serial}}$ and the BOT data currency property holds for all read-only transactions in MVH. (Note that we do not explicitly define data currency guarantees for read operations of read-write transactions since we believe that the data currency requirements implicitly enforced for those reads by the serializability criterion are sufficiently strict for most applications.)

To determine if a given multi-version history MVH satisfies the requirements of the BOT serializability level, we use a variation of the MVSG called start time multi-version serialization graph (ST-MVSG). In this paper, we assume that the reader is familiar with the notion of MVSG, and for details we refer to [12].

Definition 12 (Start time multi-version serialization graph). Let MVH denote a history over a set of read-only and read-write transactions $T = \{T_1, \dots, T_n\}$ and $\text{commit}(\text{MVH})$ represents a function that returns the committed transactions of

MVH. A start time multi-version serialization graph for history MVH, denoted $\text{ST-MVSG}(\text{MVH})$, is a directed graph with nodes $N := \text{commit}(\text{MVH})$ and edges E such that:

1. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$) if T_j arbitrary directly depends on T_i .
2. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$) whenever there exists a set of operations $\{r_i[x_j], w_j[x_j], w_k[x_k]\}$ such that either $w_j \ll w_k$ and $c_k \rightarrow_{\text{MVH}} b_i$ or $b_i \rightarrow_{\text{MVH}} c_j$.

Theorem 1. Let MVH be a multi-version history over a set of committed transactions $T = \{T_1, \dots, T_n\}$. Then MVH is BOT serializable, if $\text{ST-MVSG}(\text{MVH})$ is acyclic.

Proof. See [30]. □

3.1.2. Strict forward BOT serializability

The currency requirements of BOT serializability may not be ideally suited for processing read-only transactions in mobile broadcasting environments for at least two reasons. First, mobile read-only transactions are mostly long running in nature due to such factors as interactive data usage, intentional or accidental disconnections, and/or high communication delays. Therefore, disallowing a long-lived read-only transaction to see object versions that were created by committed read-write transactions after its starting point might be too restrictive. Another reason for allowing "forward" reads is related to version management. Reading from a snapshot of the database that existed at the time when a read-only transaction started its execution can be expensive in terms of storage costs. If database objects are frequently updated, which is a reasonable assumption for data-dissemination environments, multiple previous object versions have to be retained in various parts of the database system. Allowing read-only transactions to view more recent data than permitted by the BOT data currency property is efficient, since it enables purging out-of-date objects sooner, thus allowing to keep more recent objects in the database system. An IL that provides such currency guarantees while still enforcing degree 3 consistency is called *strict forward BOT serializability*. Prior to defining this IL, we formulate a rule that is sufficient and practicable for determining whether a read-only transaction T_i may be allowed to see the (total) effects of an update transaction that committed after T_i 's starting point without violating serializability requirements.

Read Rule 1 (Serializable forward reads). Let T_i denote a read-only transaction that needs to observe the effects of an update transaction T_j that committed after T_i 's starting point as long as the serializability requirement holds. Further, let T_{update} represents a set of read-write transactions that committed after T_i 's starting point but before the commit point of T_j and whose effects have not been seen by T_i ; i.e., $\forall T_k \in T_{\text{update}} (b_i \rightarrow_{\text{MVH}} c_k \wedge c_k \rightarrow_{\text{MVH}} c_j \wedge \text{if } w_k[x_k] \text{ occurs in } P(\text{MVH}, [b_i, c_j]), \text{ then there is no } r_i[x_k] \text{ in } P(\text{MVH}, [b_i, c_j]))$. T_i is allowed to read forward and see the effects of T_j whenever the invariant $\text{ReadSet}(\text{MVH}', T_i) \cap$

$\text{WriteSet}(\text{MVH}', (T_{\text{update}} \cup T_j)) = \emptyset$ is true for the subhistory $\text{MVH}' := \text{P}(\text{MVH}, [b_i, c_j])$, i.e., the intersection of the actual read set of T_i and the write set of all read–write transactions that committed between T_i 's starting point and T_j 's commit point (including T_j itself) must be an empty set. Otherwise, T_i is forced to observe the database state that was valid as of the time T_i started.

Note that in Read Rule 1 the projection onto MVH with regard to the time interval $[b_i, c_j]$ refers to the start time order of transactions in MVH and is independent of the (real-time) order of those events in MVH. Further note that in Read Rule 1 the read set and the write set refer to data objects and not to their dedicated versions. This will be the case throughout the paper if not otherwise specified. In what follows, we denote the fact that T_i is permitted to read forward on the object versions produced by T_j , by $T_i \rightarrow_{\text{sfr}} T_j$.

It can be shown that Read Rule 1 produces only correct read-only transactions in the sense that they are serializable with respect to all committed update transactions and all other committed read-only transactions in a multi-version history MVH.

Theorem 2. In a multi-version history MVH that contains a set of read–write transactions T_{update} such that all read–write transactions in T_{update} , are serializable, each read-only transaction T_i satisfying Read Rule 1 is serializable as well.

Proof. Omitted due to space restrictions. \square

The following new IL incorporates the serializable forward read property, and is defined as follows.

Definition 13 (Strict forward BOT serializability). A multi-version history MVH over a set of read-only and read–write transactions is a strict forward BOT serializable history, if all of the following conditions hold: MVH is serializable, and if the pair $r_i[x_j]$ and $w_j[x_j]$ of a read-only transaction T_i and a read–write transaction T_j is in MVH, then either

- (a) $b_i \rightarrow_{\text{MVH}} c_j$, $w_j[x_j] <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{sfr}} T_j$ and there is no write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $x_j \ll x_k$, $c_k <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{sfr}} T_k$; or
- (b) $c_j <_{\text{MVH}} b_i$ and there is no write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $c_k <_{\text{MVH}} b_i$, $x_j \ll x_k$.

To check whether a given history MVH is strict forward BOT serializable, we use a variant of the MVSG.

Definition 14 (Strict forward read multi-version serialization graph). A strict forward read multi-version serialization graph for a multi-version history MVH, denoted $\text{SFR-MVSG}(\text{MVH})$, is a directed graph with nodes $N := \text{commit}(\text{MVH})$ and edges E such that:

1. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$), if T_j arbitrary directly depends on T_i .
2. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$), whenever there exists a pair of operations $r_i[x_j]$ and $w_j[x_j]$ of a read-only transaction T_i and a read–write transaction T_j such that $w_j \ll w_k$ and $c_k \rightarrow_{\text{MVH}} b_i$.
3. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$), whenever there exists a pair of operations $r_i[x_j]$ and $w_j[x_j]$ of a read-only transaction T_i and a read–write transaction T_j such that $b_i \rightarrow_{\text{MVH}} c_j$, $c_j[x_j] <_{\text{MVH}} r_i[x_j]$, $\neg T_i \rightarrow_{\text{sfr}} T_j$.
4. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$), whenever there exists a pair of operations $r_i[x_j]$ and $w_j[x_j]$ of a read-only transaction T_i and a read–write transaction T_j such that $b_i \rightarrow_{\text{MVH}} c_j$, $c_j[x_j] <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{sfr}} T_j$ and there is a write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $x_j \ll x_k$, $c_k <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{sfr}} T_k$.

Theorem 3. A history MVH is strict forward BOT serializable, if $\text{SFR-MVSG}(\text{MVH})$ is acyclic.

Proof. See [30]. \square

3.2. Update serializability

While the strictness of serializability may be necessary for some read-only transactions, often, however, the use of such strong criteria is overly restrictive and may negatively affect the overall system performance. Even worse, serializability does not only trade consistency for performance, but it also has an impact on data currency. Such drawbacks can be eliminated or at least diminished by allowing read-only transactions to be executed at weaker ILs. Various correctness criteria have been proposed in the literature to achieve performance benefits by allowing non-serializable execution of read-only transactions. While some forms of consistency such as *update serializability/weak consistency* [13,14,17] or *external consistency/update consistency* [13,33] require that read-only transactions observe consistent database state, others such as *epsilon serializability* [34] allow them to view transaction inconsistent data. We believe that the majority of read-only transactions need to see a transaction consistent database state and therefore we focus solely on ILs that provide such guarantees. An IL that is strictly weaker than serializability and allows read-only transactions to see a transaction consistent state is the Update Serializability (US) level which can be formally defined as follows.

Definition 15 (Update serializability). Let us denote the set of committed read–write transactions by $T_{\text{update}} = \{T_1, \dots, T_n\}$ and the projection of MVH onto T_{update} by $\text{P}(\text{MVH}, T_{\text{update}})$. A multi-version history MVH over a set of read-only and read–write transactions is an update serializable history, if for each read-only transaction T_i in MVH the subhistory $\text{MVH}' := \text{P}(\text{MVH}, T_{\text{update}}) \cup \text{P}(\text{MVH}, T_i)$ is serializable. If there are no read-only transactions in MVH, then only the subhistory $\text{MVH}' := \text{P}(\text{MVH}, T_{\text{update}})$ has to be serializable.

Update serializability differs from the serializability IL by allowing read-only transactions to serialize individually with the set of committed read–write transactions in a multi-version history MVH, i.e., it relaxes the strictness of the serializability criterion by requiring that read-only transactions are serializable with respect to committed read–write transactions, but not with respect to other committed read-only transactions.

3.2.1. Strict forward BOT update serializability

Update serializability as defined above allows different read-only transactions to view different transaction consistent database states that result from different serialization orders of read–write transactions. By not requiring that all read-only transactions have to see the same consistent state, more concurrency between read-only and read–write transactions is made possible. However, higher transaction throughput by relaxing the consistency requirement may not be achieved at the cost of providing no or unacceptable data currency guarantees to users. It is obvious, that update serializability lacks any currency requirements, thus we need to extend the update serializability IL by incorporating such guarantees. As data currency and consistency are orthogonal concepts, it is possible to combine update serializability with various types of currency. As before, we concentrate on the BOT data currency type, since we believe that it is frequently required in the mobile environment. Actually there is no need to define a new IL that provides BOT data currency guarantees in combination with update serializability correctness since such a level would be equivalent to the already defined BOT serializability degree. Nevertheless, extending update serializability by the requirement that a read-only transaction T_i must perceive the most recent version of committed objects that existed by T_i 's starting point or thereafter seems to be a valuable property in terms of currency and performance. However, forward reads beyond T_i 's start point should only be allowed, if the update serializability criterion is not violated. In order to determine whether a read-only transaction T_i can safely read forward on some version of object X it wants to read, the following property can be used.

Read Rule 2 (Update serializable forward reads). Let T_i denote a read-only transaction in MVH that requires to observe the effects of a read–write transaction T_j that committed after T_i 's starting point as long as the update serializability requirements are not violated. Further, let T_{update} represent a set of read–write transactions that committed after T_i 's starting point but before the commit of T_j , i.e., $\forall T_k \in T_{\text{update}} (b_i \rightarrow_{\text{MVH}} c_k \wedge c_k \rightarrow_{\text{MVH}} c_j)$. T_i is allowed to read forward and see the effects of T_j , if the invariant $\text{ReadSet}(\text{MVH}, T_i, [b_i, c_j]) \cap \text{WriteSet}(\text{MVH}, T_j) = \emptyset$ holds and there is no read–write transaction T_k in MVH ($j \neq k, i \neq k$) such that $b_i \rightarrow_{\text{MVH}} c_k, c_k \rightarrow_{\text{MVH}} c_j, \neg T_i \rightarrow_{\text{usfr}} T_k$ and T_j arbitrary depends on T_k . Otherwise, T_i is forced to see the database state that was valid at its starting point b_i . We denote the fact that T_i is allowed to read forward to observe the effects of T_j by $T_i \rightarrow_{\text{usfr}} T_j$.

As before, it can be shown that Read Rule 2 produces only correct histories in the sense that each read-only transaction sees a serial order of all committed read–write transactions in a multi-version history MVH.

Theorem 4. In a multi-version history MVH that contains a set of read–write transactions T_{update} such that all read–write transactions in T_{update} are serializable, each read-only transaction T_i satisfying Read Rule 2 is update serializable with respect to T_{update} .

Proof. Omitted due to space restrictions. \square

We can now define a new IL that ensures update serializability correctness along with firm data currency guarantees.

Definition 16 (Strict forward BOT update serializability). A multi-version history MVH over a set of read-only and read–write transactions is strict forward BOT update serializable, if the following condition holds:

1. MVH is update serializable, and if the pair $r_i[x_j]$ and $w_j[x_j]$ of a read-only transaction T_i and a read–write transaction T_j are in MVH, then either
 - (a) $b_i \rightarrow_{\text{MVH}} c_j, c_j[x_j] <_{\text{MVH}} r_i[x_j], T_i \rightarrow_{\text{usfr}} T_j$ and there is no write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $x_j \ll x_k, c_k <_{\text{MVH}} r_i[x_j], T_i \rightarrow_{\text{usfr}} T_k$ or
 - (b) Requirement 2(b) of definition 13 is true.

Again, we determine whether a given history MVH is strict forward BOT update serializable by using a directed MVSG.

Definition 17 (Strict forward read single query multi-version serialization graph). A strict forward read single query multi-version serialization graph for MVH with respect to a read-only transaction T_i , denoted SFR-SQ-MVSG(MVH, T_i), is a directed graph with nodes $N := T_{\text{update}} \cup T_i$ and edges E such that:

1. An edge of type 1 and 2 in SFR-MVSG(MVH) is an edge in SFR-SQ-MVSG(MVH, T_i).
2. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$) whenever there exists a pair of operations $w_j[x_j]$ and $r_i[x_j]$ of a read-only transaction T_i and a read–write transaction T_j such that $b_i \rightarrow_{\text{MVH}} c_j, c_j[x_j] <_{\text{MVH}} r_i[x_j], T_i \rightarrow_{\text{usfr}} T_j$ and there is a write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $x_j \ll x_k, c_k <_{\text{MVH}} r_i[x_j], T_i \rightarrow_{\text{usfr}} T_k$.

Theorem 5. A history MVH is strict forward BOT serializable, if for each read-only transaction T_i the corresponding SFR-SQ-MVSG(MVH, T_i) is acyclic.

Proof. See [30]. \square

3.3. View consistency

View consistency (VC) is the weakest IL that ensures transaction consistency to read-only transactions provided that all read–write transactions modifying the database state are serializable. It was first informally defined in the literature by Weihl [33] under the name *external consistency*. Due to its valuable guarantees provided to read-only transactions, it appears to be an ideal candidate for use in all forms of environments including broadcasting systems. However, as noticed for the conflict serializability and update serializability degree, the definition of view consistency lacks the notion of data currency. We formally define the view consistency level as follows.

Definition 18 (View consistency). Let T_i^{dep} denote a set of committed read–write transactions in MVH that T_i directly and indirectly depends on. A multi-version history MVH over a set of read-only and read–write transactions is view consistent, if all read–write transactions are serializable and for each read-only transaction T_i in MVH the subhistory $\text{MVH}' := \text{P}(\text{MVH}, T_i^{\text{dep}}) \cup \text{P}(\text{MVH}, T_i)$ is serializable.

This IL's attractiveness relates to the fact that all read–write transactions produce a consistent database state and read-only transactions view a transaction consistent database state. However, as with update serializability, there might be a concern that two read-only (or read-only and read–write) transactions executed at the same client can see different serial orders of read–write transactions. Another issue is related to the currency of the data observed by read-only transactions. While the first potential problem can only be resolved by running read-only transactions with serializability guarantees, the latter issue can be compensated by extending the view consistency level by appropriate currency guarantees. As for the update serializability level, there is no need to define a new IL that ensures view consistency correctness in combination with BOT data currency since such an IL would be equivalent to the defined BOT serializability level. However, extending BOT serializability with a forward read obligation that allows read-only transactions to see the effects of read–write transactions as long as the view consistency requirements are not violated appears to be a worthwhile approach. Before we formally define this new IL, we need to formalize a condition that allows us to determine whether a read-only transaction T_i can observe the effects of a read–write transaction T_j that committed its execution after T_i 's starting time.

Read Rule 3 (View consistent forward reads). Again, let T_{update} represent a set of read–write transactions that committed after T_i 's starting point but before the commit point of T_j . T_i is allowed to read forward and see the (total) effects of T_j ($T_i \rightarrow_{\text{vcfr}} T_j$), if the invariant $\text{ReadSet}(\text{MVH}, T_i, [b_i, c_j]) \cap \text{WriteSet}(\text{MVH}, T_j) = \emptyset$ holds and there is no read–write transaction T_k in MVH ($j \neq k, i \neq k$) such that $b_i \rightarrow_{\text{MVH}} c_k$, $c_k \rightarrow_{\text{MVH}} c_j$, $\neg T_i \rightarrow_{\text{vcfr}} T_k$ and T_j write–read or write–write

depends on T_k . Otherwise, T_i is forced to see the database state as it existed by its starting point.

Again, it can be shown that Read Rule 3 produces only syntactically correct histories in the sense that read-only transactions see a transaction consistent database state.

Theorem 6. In a multi-version history MVH containing a set of read–write transactions T_{update} such that all read–write transactions in T_{update} are serializable, each read-only transaction T_i satisfying Read Rule 3 is serializable with respect to all transactions in T_{update} whose effects T_i has either directly or indirectly seen.

Proof. Omitted due to space restrictions. \square

We can now define our new IL that ensures update serializability correctness together with firm data currency guarantees.

Definition 19 (Strict forward BOT view consistency). A multi-version history MVH over a set of read-only and read–write transactions is strict forward BOT view consistent, if the following condition holds:

1. MVH is view consistent, and if the pair $r_i[x_j]$ and $w_j[x_j]$ of a read-only transaction T_i and a read–write transaction T_j is in MVH, then either
 - (a) $b_i \rightarrow_{\text{MVH}} c_j$, $c_j[x_j] <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{vcfr}} T_j$ and there is no write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $x_j \ll x_k$, $c_k <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{vcfr}} x_k$ or
 - (b) requirement 2(b) of definition 13 is true.

To show that a multi-version history MVH provides *strict forward BOT view consistency* guarantees, we associate a corresponding graph with MVH.

Definition 20 (Causal dependency strict forward read single query multi-version serialization graph). A causal dependency strict forward read single query multi-version serialization graph for a multi-version history MVH with respect to a read-only transaction T_i , denoted $\text{CD-SFR-SQ-MVSG}(\text{MVH}, T_i)$, is a directed graph with nodes $N := T_i^{\text{dep}} \cup T_i$, and edges E such that:

1. An edge of type 1 and 2 in $\text{SFR-MVSG}(\text{MVH})$ is an edge in $\text{CD-SFR-SQ-MVSG}(\text{MVH}, T_i)$.
2. There is an edge $T_i \rightarrow T_j$ ($T_i \neq T_j$) whenever there exists a pair of operations $w_j[x_j]$ and $r_i[x_j]$ of a read–write transaction T_j and a read-only transaction T_i such that $b_i \rightarrow_{\text{MVH}} c_j$, $c_j[x_j] <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{vcfr}} T_j$ and there is a write operation $w_k[x_k]$ of a committed transaction T_k in MVH such that $x_j \ll x_k$, $c_k <_{\text{MVH}} r_i[x_j]$, $T_i \rightarrow_{\text{vcfr}} T_k$.

Table 1
Newly defined ILs and their core characteristics.

Newly defined isolation level	Base isolation level	Consistency guarantees	Currency guarantees
BOT serializability	Serializability	Each read-only transaction in MVH is required to serialize with all committed read-write and all other read-only transactions in MVH.	Read-only transactions are required to observe a snapshot of committed data objects that existed by their starting points.
Strict forward BOT serializability	Serializability	Each read-only transaction in MVH is required to serialize with all committed read-write and all other read-only transactions in MVH.	Read-only transactions are required to read from a database snapshot valid as of the time when they started. However, read-only transactions are forced to read "forward" and observe the updates from read-write transactions that committed after their starting points as long as the serializability requirement is not violated by those reads.
Strict forward BOT update serializability	Update serializability [14,17]/ Weak consistency [13]	Each read-only transaction in MVH is required to serialize with all committed update transactions in MVH, but does not need to be serializable with other committed read-only transactions.	Enforces the same currency requirements as the strict forward BOT serializability level with the difference that read-only transactions are obliged to issue forward reads as long as the update serializability requirements are not violated by those reads.
Strict forward BOT view consistency	View consistency/ Update consistency [13]/ External consistency [33]	Each committed read-only transaction in MVH is required to serialize with all committed update transactions in MVH that had written values which have either directly or indirectly been seen by the read-only transaction.	Enforces the same currency requirements as the strict forward BOT serializability level with the difference that forward reads of read-only transactions are enforced whenever the view consistency criterion is not violated by those reads.

Theorem 7. A history MVH is strict forward BOT serializable, if for each read-only transaction T_i the corresponding CD-SFR-SQ-MVSG(MVH, T_i) is acyclic.

Proof. See [30]. □

To conclude this section, table 1 summarizes the main characteristics of the newly defined ILs.

4. Implementation issues

We now propose protocols that implement the newly defined ILs in an efficient manner. First, we illustrate the key characteristics of our envisaged broadcasting environment and present some general design assumptions that underlie the implementation of the ILs.

Data dissemination by using broadcast disks is likely to become the prevailing mode of data exchange in mobile wireless environments. The characteristics of a broadcast disk environment are well known in the literature and therefore we only present some key properties that are relevant for our protocols. For simplicity, we assume a flat broadcast disk that consists of three types of segments: (a) index segment, (b) data segment, and (c) control information segment. To make the data disseminated self-descriptive, we incorporate an index into the broadcast program. We choose $(1, m)$ in-

dexing [22] as the underlying index organization method and broadcast the complete index once within each minor broadcast cycle. To provide cache consistency in spite of server updates, each minor cycle is preceded with a concurrency control report or CCR that contains the read and write sets along with the values of newly created objects of read-write transactions that committed in the last minor broadcast cycle. An entry in a CCR is a 3-tuple $\langle TID, ReadSet, WriteSet \rangle$ where TID denotes a globally unique transaction identifier. Transactions stored in CCR are ordered by their commit time. The data segment contains hot-spot data objects that are of interest to a large number of clients. The rest of the database can be accessed on-demand. To allow clients to communicate with the server, we assume the availability of a back channel.

With respect to the client and server architecture, we assume a hybrid caching system for both system components to improve the performance of our protocols. In a hybrid caching system the cache memory available is divided into a page-based segment and an object-based segment. The server uses its page cache to handle fetch requests from the server and to fill the broadcast disk with pages containing hot-spot objects. The server object cache is utilized to save installation disk reads for writing modified objects onto disk. The latter is organized similar to the modified object buffer (MOB) in [15]. With respect to concurrency control, the server object cache can be used to answer object requests in case a transaction

consistent page is not available from the client's perspective. The client also maintains a hybrid cache scheme to get full advantage of both types. The client page cache is used to keep requested and prefetched database pages in volatile memory. We assume a single version page cache that maintains up-to-date server pages. The client object cache, on the other hand, is allowed to store multiple versions of an object X . To simplify the description of our protocols, we assume that an object X can be either stored in a page P or in the object cache of the client. To judge about the correctness of a client read operation, each page P is assigned a timestamp $TS(P)$ that reflects the (logical) time when an object X resident in P was last updated. Analogous to the page cache, each version of an object maintained in the client object cache is associated with a commit timestamp reflecting the point in time when the version was installed.

4.1. Multi-Version Concurrency Control protocol with BOT Serializability guarantees (MVCC-BS)

In this section, we present an algorithm that provides BOT serializability to read-only transactions. To enforce database consistency, we assume that the state of the mobile database is exclusively modified by transactions that run with serializability requirements. We also assume that clients can only execute a single read-only transaction at a time. The subsequently described algorithm will build the fundamental basis for subsequent protocols that ensure weaker semantic guarantees than serializability. For space considerations, we only cover the case where mobile clients do not suffer from intermittent connectivity and can actively observe the broadcast channel.

Our implementation of the BOT serializability level allows concurrency control with nearly no overhead. For each read-only transaction T_i , the client keeps the following data structures and information for concurrency control purposes: (a) T_i 's startup timestamp, (b) T_i 's read set, and (c) an object invalidation list. The latter contains the identifiers and commit timestamps of objects that were created during the current major broadcast cycle (MBC). Note that all underlying data structures of our CC schemes are chosen for clarity of exposition rather than for efficient implementation.

The server data structures include the hybrid server cache and CCR as described before and the temporary object cache (TOB). The TOB is used to record the modified or newly created object values of transactions that committed during the current minor broadcast cycle. Additionally, the TOB is utilized to store "shadow" versions of transactions that are not yet committed. Whenever a minor broadcast cycle is finished, all versions of committed transactions are merged from the TOB into the MOB and the updated or newly created object versions will be available for the next minor and major broadcast cycle.

Now we describe the protocol scheme by differentiating between client and server operations.

4.1.1. Client operations

1. Read object X by transaction T_i on client C .

(a) T_i issues its first read operation. Assign the number of the current minor broadcast cycle to $STS(T_i)$. Add X to T_i 's read set (RS).

(b) Requested object X is cache-resident in the page or object cache. If the requested object is stored in page P , it can be read by T_i whenever P 's update timestamp $TS(P)$ is smaller than $STS(T_i)$ or if T_i started its operations in the current MBC and there is no entry of X with timestamp $TS_{OIL}(X)$ in the object invalidation list (OIL) such that $STS(T_i) \leq TS_{OIL}(X)$. Otherwise, T_i looks for the entry of object X in the object cache. If some version j of object X is in the object cache, T_i can read X_j if the invariant $TS(X_j) < STS(T_i)$ holds. (Note that there is no need to check whether there is an other version k of object X in the client cache such that $TS(X_j) < TS(X_k)$ and $TS(X_k) < STS(T_i)$ since by assuming that clients run not more than a single read-only transaction at any time only one version of object X with a commit timestamp smaller than the starting timestamp of the read-only transaction may be useful for it and is therefore maintained in the client object cache. The other object versions would only waste scarce memory space and are therefore garbage-collected as mentioned below.) If T_i reads some version of X , add X to $RS(T_i)$.

(c) Requested object X is scheduled for broadcasting. Read index of the broadcast to determine the position of the object on the broadcast. The client is allowed to download the desired object X if the update timestamp of the page P in which X resides is smaller than T_i 's starting point, i.e., $TS(P) < STS(T_i)$ and there is no object version of X in OIL such that $TS(P) < TS_{OIL}(X)$ and $TS_{OIL}(X) < STS(T_i)$. If a consistent object of X cannot be located in the air-cache the client proceeds with (d). Otherwise, it reads the installed version of object X and adds X to $RS(T_i)$.

(d) Requested version of object X is neither in the local cache nor in the air-cache. Send fetch request for object X along with $STS(T_i)$ and T_i 's OIL to the server. The server processes the client request as described below. As a reply the client either receives a transaction consistent copy (with respect to T_i) of a page P which contains the requested object X or otherwise a transaction consistent version of X . If the request cannot be satisfied, the server notifies the client and T_i must be aborted.

2. Concurrency control report processing on client C . CCRs are disseminated at the beginning of each minor broadcast cycle. The client processes the CCR as follows. For each object X included in the write set of a read-write transaction T_j that committed in the last minor broadcast cycle, an entry is added into OIL containing the identifier of object X along with its commit timestamp. Additionally, the contents of the page and object cache is refreshed. If object X kept in page P at client C was updated during the last minor broadcast cycle,

the old version of X is overwritten by the newly created version. Otherwise, the updated version of X is installed into the object cache, if X belongs to C 's hot-spot objects. If a prior version of object X becomes useless for T_i it is discarded from the object cache.

3. Transaction commit. Transaction T_i is allowed to commit, if all read requests were satisfied and no abort notification was sent by the server.

4.1.2. Server operations

1. Fetch request for object X from client C . If the server receives a fetch request for object X from transaction T_i , the server first checks if the page P in which X resides is in the server cache. If P is cache-resident and the startup timestamp of T_i is equal to the number of the current minor broadcast cycle, the server will send page P to C after applying to P all pending server object cache entries. Otherwise, if any of the aforementioned conditions is violated, the server searches for X in the MOB. If it finds an entry for object X such that $TS(X) < STS(T_i)$, the server will send object X to the client. Otherwise, if $STS(T_i)$ equals the number of the current minor broadcast cycle, the server reads P from disk and applies all outstanding modifications recorded in the MOB of objects that reside in P to the page. If a fetch request cannot be satisfied due to consistency reasons, an abort message will be transmitted to the client.

2. Integration of the TOB into the MOB. At the end of each minor broadcast cycle, the newly created and updated versions of objects are merged into the MOB. If objects exist in the MOB, their object values will be overwritten and timestamp numbers will be updated.

3. Filling the broadcast disk server. The server fills the data and index segment of the broadcast disk server at the beginning of each MBC. Thereby, the server proceeds as follows. If the desired page containing hot-spot objects is not in the page cache of the server, it is read into the cache from the disk and thereafter it is updated to reflect all the modifications of its objects recorded in the MOB. At the end of this process, all pages stored in the broadcast disk server are completely up-to-date, i.e., they contain the most current versions of their objects. Further, the server creates an $(1, m)$ -index containing entries for objects scheduled for broadcasting and stores it into the index segment of the broadcast disk. The concurrency control segment of the broadcast disk is updated at the beginning of each minor broadcast cycle. This segment is filled with the CCR as described above.

4.2. Multi-Version Concurrency Control protocol with Strict Forward BOT Serializability guarantees (MVCC-SFBS)

Having described a MVCC scheme that ensures BOT Serializability consistency, we extend this scheme to provide Strict Forward BOT Serializability. Recall that the Strict Forward BOT Serializability level differs from the BOT Serializability degree by requiring that read-only transactions observe

the updates of transactions that committed after their starting point provided that Read Rule 1 is satisfied. To implement the latter requirement, we adopt a technique used by the multi-versioning with invalidation scheme in [28] and associate a read forward flag, or RFF, with each read-only transaction T_i in MVH that indicates whether T_i has read a version of an object that was later modified by a read-write transaction T_j . If such an event occurs, RFF of T_i is set to false and T_j 's commit timestamp is recorded in a variable called read forward stop or RFS. Equipped with the latter information, the scheduler can efficiently determine which versions of requested objects a mobile transaction needs to observe by applying the following read rule.

Read Rule 4. Whenever a read-only transaction T_i wants to read an object X and RFF is set to false, T_i has to read the latest committed object version of X with a timestamp TS that is smaller than $RFS(T_i)$. Otherwise, T_i has to read the most recent object version of X .

4.3. Multi-Version Concurrency Control protocol with Strict Forward BOT Update Serializability guarantees (MVCC-SFBUS)

Recall that the Update Serializability level is less restrictive than the serializability level by allowing different read-only transactions to see different serialization orders of read-write transactions. This weaker requirement affects the forward read behavior of read-only transactions running under Strict Forward BOT Update Serializability. As with the MVCC-SFBS, the mobile client has to determine for each active read-only transaction T_i as to whether it needs to observe the effects of a read-write transaction T_j that committed during T_i 's execution time. To this end, each read-only transaction maintains two additional data structures. First, an object version write prohibition list, or OVWPL, is associated with each read-only transaction T_i . An OVWPL is a set of pairs (OID, CTS) where OID denotes the identifiers of objects whose values T_i is not allowed to see and CTS represents the logical time when the transactions that modified or created the objects committed. The OVWPL of an active read-only transaction T_i is updated whenever a new CCR appears on the broadcast channel. Further, for each active read-only transaction T_i the client maintains an object version read prohibition list OVRPL that keeps track of the objects read by read-write transactions that committed during T_i 's execution time and whose effects may not be seen by T_i . The identifiers of objects created by a read-write transaction T_j along with the corresponding timestamp (in case of the OVWPL) have to be added to the T_i 's OVWPL and OVRPL if any of the following conditions holds:

- C1. $ReadSet(T_i) \cap WriteSet(T_j) \neq \emptyset$.
- C2. $ReadSet(T_j) \cap OVWPL(T_i) \neq \emptyset$.
- C3. $OVRPL(T_i) \cap WriteSet(T_j) \neq \emptyset$.

Condition C1 implies that in order for T_i to read forward on objects written by T_j , the intersection between T_i 's read set

and T_j 's write set must be empty. Otherwise, the read set and write set of T_j must be registered in T_i 's OVRPL and OVWPL, respectively. If T_j has updated object X that is already listed in T_i 's OVWPL, the entry of X is not modified and the protocol proceeds with the next object written by T_j (if there is any). Condition C2 states that T_j may not see any objects contained in T_i 's OVWPL. It ensures that T_i will only see the effects of T_j if there is no write-read dependency between any transaction T_k whose updates are registered in T_i 's OVWPL and T_j . If C2 is violated, T_j 's updates and read objects have to be placed into T_i 's OVWPL and OVRPL, respectively. Condition C3 states that T_j must not have overwritten an object that is included in T_i 's OVRPL. This condition guarantees that T_i will only see the updates of T_j if there exists no read-write dependency between any transaction T_k (that conflicts with T_i and those read operations are included in T_i 's OVRPL) and T_j . Again, if C3 is not satisfied, T_i 's OVRPL and OVWPL must be updated.

A read-only transaction running under Strict Forward BOT Update Serializability sees a correct state of the database if the mobile client obeys the following read rule:

Read Rule 5. Whenever a read-only transaction T_i wants to read object X that is registered in its OVWPL, T_i has to read the latest committed object version of X with a timestamp TS that is smaller than the one of the entry of object X in OVWPL. Otherwise, T_i has to read the most recent object version of X .

Note that Read Rule 5 is applicable to the MVCC-SFBVC protocol as well.

4.4. Multi-Version Concurrency Control protocol with Strict Forward BOT View Consistency guarantees (MVCC-SFBVC)

View Consistency is the weakest IL that provides transaction consistency to read-only transactions and has the potential to maximize the number of forward reads of read-only transactions without violating transaction correctness. To determine whether an active read-only transaction T_i is required to see the updates of a read-write transaction T_j that successfully finished its execution during T_i 's lifetime, the applicability of rule C1 and C2 has to be tested. If both conditions are satisfied and T_i wants to read an object written by T_j , it needs to read the version installed by T_j , if there exists no later version of the respective object that T_i is allowed to observe as well. As the rules given above are a proper subset of those formulated for the MVCC-SFBUS scheme, it is obvious that the MVCC-SFBVC protocol provides strictly stronger currency guarantees than MVCC-SFBUS. Further, it is easy to see that MVCC-SFBVC has lower time and space overheads than MVCC-SFBUS since the former does not need to maintain the OVRPL. Hence, we expect that the MVCC-SFBVC scheme outperforms the MVCC-SFBUS protocol in our performance study.

5. Performance results

The performance study is aimed at measuring the absolute and relative performance of our proposed multi-version concurrency control protocols in a wireless broadcast disk environment. Additionally, we compare our protocols with the previously devised concurrency control schemes [28,32] to detect performance trade-offs between the schemes and their underlying consistency and currency guarantees. We analyze the performance of the new ILs implementations and other protocols using two key metrics, namely *transaction commit* and *transaction abort rate*. We restricted the subsequent analysis to those two metrics due to space restrictions. For an extended version of the performance analysis as well as for the experimental setup of the simulator we refer the interested reader to [30].

5.1. Experimental results of the protocols proposed

All performance results presented are derived from executing 10000 read-only transactions after the system reached its steady state. The results come from artificially generated traces, i.e., they give only an intuition about the performance of our ILs implementations, but may not represent a real application's behavior. Due to space restrictions, we give only a brief interpretation of the experimentally measured results with respect to the aforementioned metrics.

As figure 2(a) shows, the transaction throughput of all protocols decreases along the x -axis as the number of objects accessed by read-only transactions rises. Increasing the transaction length results in longer transaction execution times and hence fewer transaction commits per second. Furthermore, longer read-only transactions might abort at a later point of their execution which results in higher abort costs, thus also reducing the transaction throughput. Additionally, as transaction execution time progresses, the likelihood that object read requests can be satisfied by some component of the database system (client cache, air-cache or server memory) decreases. Thus, apart from increased abort costs, higher abort rates are another consequence of longer transaction execution times. As the tabular results show, the performance difference between the MVCC-SFBVC and the other protocols widens slightly with respect to the MVCC-SFBUS and MVCC-SFBS scheme and significantly with respect to the MVCC-BS protocol with increase in read-only transaction length. The growing performance penalty is caused by a disproportionate increase in the number of messages sent per committed read-only transaction since fewer client cache and air-cache hits occur.

The increase in the transaction abort rate as a function of the transaction length is depicted in figure 2(b). In terms of the abort rate, the relative difference between the protocols decreases when growing the transaction size from 10 to 50 read operations. The reason for the narrowing gap between the protocols is related to a decline in the relative difference in the number of PLEs (defined as the number of data objects that a read-only transaction is forbidden to read forward by its commit point) with increasing transaction length.

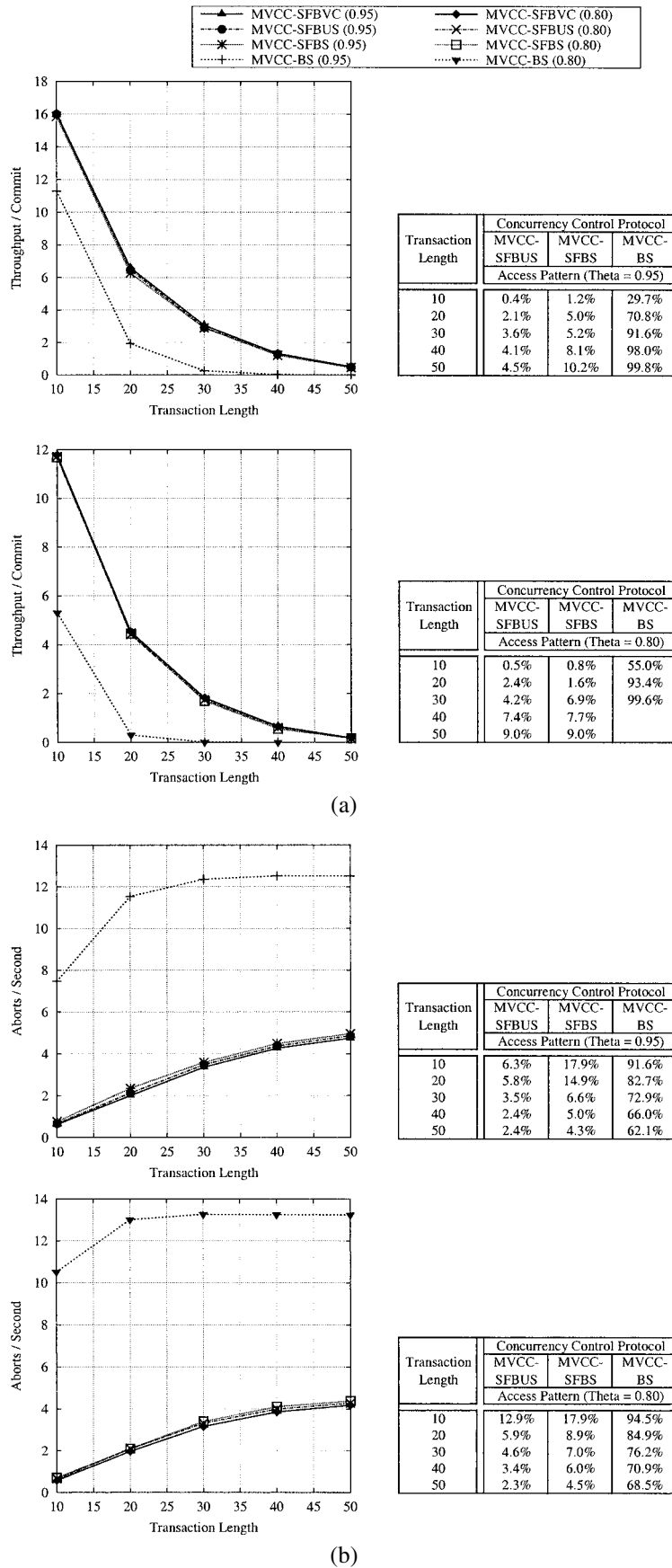


Figure 2. Performance results of the four new ILs' implementations for the baseline settings of the simulator. While graphs show absolute simulation values, tables present the performance penalty of three ILs relative to the best performing CC protocol, namely MVCC-SFBVC.

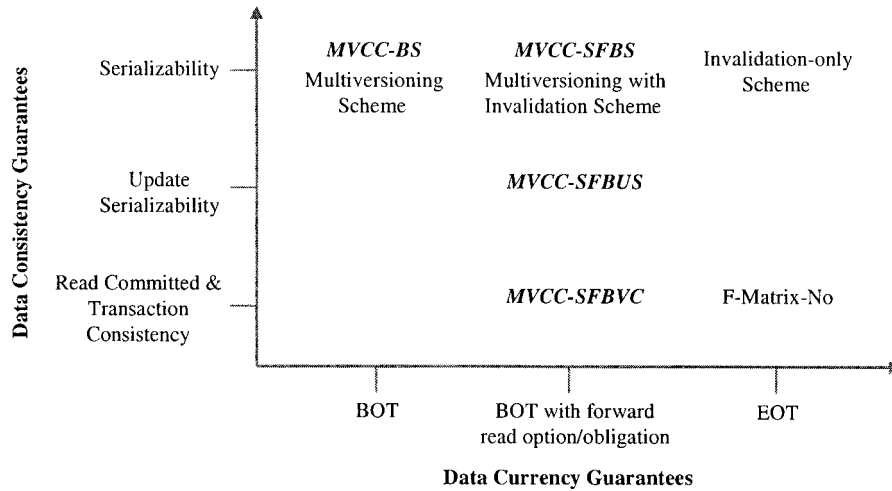


Figure 3. Protocols studied with their respective data consistency and currency guarantees.

5.2. Comparison to existing CC Protocols

Here we present the results of experiments comparing the throughput and abort rate of the best and worst performing protocol, namely MVCC-SFBVC and MVCC-BS, with CC schemes previously proposed in the literature, which are suitable for mobile database systems [27,28,32]. A suite of protocols, namely the *multi-versioning method*, *multi-versioning with invalidation method*, and *invalidation-only scheme*, all providing serializability along with varying currency guarantees to read-only transactions were devised in [27,28]. Out of those protocols, we selected the Invalidation-Only (IO) scheme for the comparison analysis. The other two protocols were left out due to their similarity to the MVCC-BS and MVCC-SFBS schemes, namely, the MVCC-BS protocol ensures the same consistency and currency guarantees as the multi-versioning scheme and the same is valid for the relationship between the MVCC-SFBS and the multi-versioning with invalidation method. Additionally, we compare our protocols with the *APPROX algorithm* [32], which provides View Consistency along with strong currency guarantees to read-only transactions. In [32], two implementations, namely F-Matrix and R-Matrix, were developed for the APPROX algorithm. We have selected a variant of the F-Matrix, called F-Matrix-No, for the comparison analysis, since it showed the best performance results among the four protocols (Datacycle [20], R-Matrix, F-Matrix, and F-Matrix-No) experimentally compared in [32]. F-Matrix-No differs from the F-Matrix protocol by ignoring the cost of broadcasting concurrency control information for each database object, and therefore can be used as a baseline for measuring the best possible performance of the protocol's underlying guarantees.

Figure 3 shows the relationship between our proposed protocols (printed in bold and italics) and the ones published in literature. As the figure indicates, both the IO scheme and F-Matrix-No protocol ensure EOT data currency. Therefore, they are not expected to perform well especially under the 0.95 workload where the clients' access pattern is highly skewed, thus resulting in frequent conflicts.

We now briefly present the results of the comparison study. As shown in figure 4 MVCC-SFBVC turns out to be superior over the other compared protocols. On average, MVCC-SFBVC outperforms the F-Matrix-No by 90.9% for the 0.95 workload and by 85.7% for the 0.80 workload. The average performance degradation of the IO scheme relative to the MVCC-SFBVC protocols is 95.5% for the 0.95 workload and 93% for the 0.80 workload. The reason for the relatively poor performance of F-Matrix-No and IO scheme is related to strong data currency requirements these two protocols impose. The F-Matrix-No performs moderately better than the IO scheme since the former processes read-only transactions with weaker consistency guarantees than the latter. While the IO scheme forces read-only transactions to abort whenever they had read some object version that was later updated by a read-write transaction T_j , the constraints imposed by the F-Matrix-No protocol are less severe. Here, only those read-only transactions need to be aborted which had observed an object version that was later updated by some read-write transaction T_j , and T_j belongs to the set of transactions whose effects have been either directly or indirectly seen by the respective read-only transaction. Thus, the number of transactions potentially aborted by the F-Matrix-No protocol forms a proper subset of the ones aborted by the IO scheme.

6. Conclusion

In this paper, we have presented four new ILs suitable for managing read-only transactions in the broadcasting environment. Further, we have described a suite of MVCC protocols that implement the defined ILs in a hybrid data delivery environment. Finally, the implementations of our defined levels are compared by means of a performance study which experimentally confirmed the hypothesis that protocols with weaker correctness requirements outperform implementations of stronger ILs as long as they enforce the same data currency guarantees. A comparison study has shown that the MVCC-SFBVC scheme is the best concurrency control mechanism for cacheable transactions in mobile broadcasting

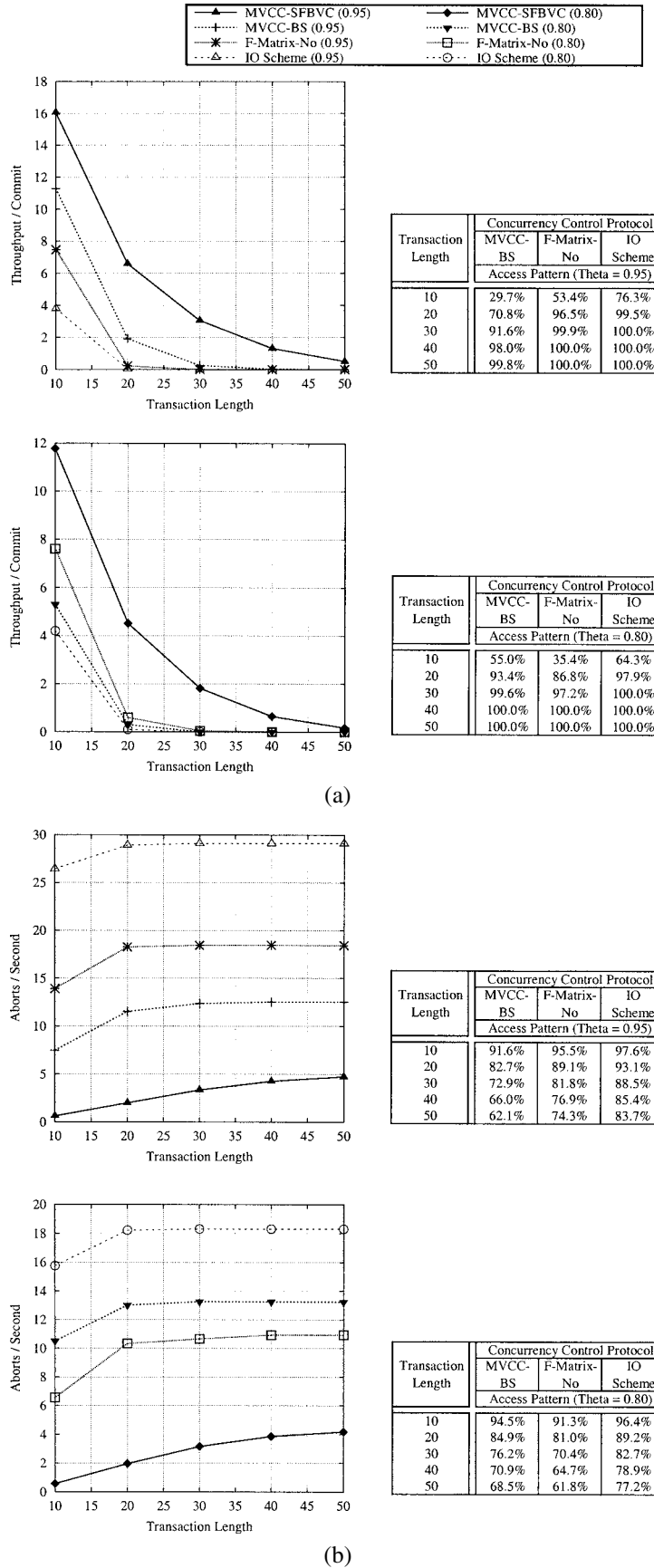


Figure 4. Performance results of the comparison study under the baseline settings of the simulator. While graphs show absolute simulation values, tables present the performance penalty of the MVCC-BS, F-Matrix-No, and IO scheme relative to the MVCC-SFBVC protocol.

environments. Thus, MVCC-SFBVC should always be the first choice for processing read-only transactions in mobile dissemination-based environments whenever read-only transactions are not required to serialize with the complete set of committed transactions in the system. Otherwise, the MVCC-SFBS protocol is to be preferred.

References

- [1] R. Abbott and H. Garcia-Molina, Scheduling real-time transactions: A performance evaluation, in: *VLDB* (1988) pp. 1–12.
- [2] S. Acharya, R. Alonso, M.J. Franklin and S.B. Zdonik, Broadcast disks: Data management for asymmetric communications environments, in: *SIGMOD Conference* (1995) pp. 199–210.
- [3] S. Acharya, M.J. Franklin and S.B. Zdonik, Dissemination-based data delivery using broadcast disks, *IEEE PCM* 2(6) (1995).
- [4] S. Acharya, M. Franklin and S. Zdonik, Prefetching from a broadcast disk, in: *ICDE* (February 1996) pp. 276–285.
- [5] S. Acharya, M. Franklin and S. Zdonik, Balancing push and pull for data broadcast, in: *SIGMOD Conference* (1997) pp. 183–194.
- [6] A. Adya, Weak consistency: A generalized theory and optimistic implementations for distributed transactions, Technical Report MIT/LCS/TR-786, Cambridge, MA (March 1999).
- [7] A. Adya, B. Liskov and P. O’Neil, Generalized isolation level definitions, in: *ICDE*, San Diego, CA (2000) pp. 67–78.
- [8] Anonymous, CPU information guide, <http://www.pocketpccity.com/articles/2001/4/2001-4-1-Palm-size-PC.html> (April 1, 2001).
- [9] ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL (November 1992).
- [10] D. Barbara, Certification reports: supporting transactions in wireless systems, in: *ICDCS* (1997) pp. 466–473.
- [11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil and P. O’Neil, A critique of ANSI SQL isolation levels, in: *SIGMOD Conference* (June 1995) pp. 1–10.
- [12] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Addison-Wesley, Reading, MA, 1987).
- [13] P.M. Bober and M.J. Carey, Multiversion query locking, in: *VLDB*, Vancouver (August 1992) pp. 497–510.
- [14] H. Garcia-Molina and G. Wiederhold, Read-only transactions in a distributed database, *ACM TODS* 7(2) (June 1982) 209–234.
- [15] S. Ghemawat, The modified object buffer: a storage management technique for object-oriented databases, Technical Report MIT/LCS/TR-666, MIT Laboratory for Computer Science (September 1995).
- [16] R. Gruber, Optimism vs. locking: a study of concurrency control for client-server object-oriented databases, Ph.D. thesis, MIT (1990).
- [17] R.C. Hansdah and L.M. Patnaik, Update serializability in locking, in: *International Conference on Database Theory*, Rome, Italy (September 1986) pp. 171–185.
- [18] J.R. Haritsa, M.J. Carey and M. Livny, On being optimistic about real-time constraints, in: *ACM PODS* (1990) pp. 331–343.
- [19] T. Henderson, Networking over next-generation satellite systems, Ph.D. thesis, University of California, Berkeley (1999).
- [20] G. Herman, G. Gopal, K. Lee and A. Weinrib, The datacycle architecture for very high throughput database systems, in: *SIGMOD Conference*, San Francisco, CA (May 1987) pp. 97–103.
- [21] Hughes Network Systems, DirecPC home page (January 2001), <http://www.diracpc.com>
- [22] T. Imielinski, S. Viswanathan and B.R. Badrinanth, Data on air: organization and access, *IEEE TKDE* 9(3) (May/June 1997) 353–372.
- [23] A. Kemper and D. Kossmann, Dual-buffer strategies in object bases, in: *VLDB*, Santiago, Chile (1994) pp. 427–438.
- [24] C. Mohan, H. Pirahesh and R. Lorte, Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions, in: *SIGMOD Conference*, San Diego, CA (June 1992) pp. 124–133.
- [25] E. Mok, H.V. Leong and A. Si, Transaction processing in an asymmetric mobile environment, in: *MDA*, Hong Kong, China (December 1999) pp. 71–81.
- [26] Oracle Corporation, *Oracle8i Concepts, Release 8.1.6*, chapter 24, Data currency and consistency (1999).
- [27] E. Pitoura and P. Chrysanthis, Scalable processing of read-only transactions in broadcast push, in: *ICDCS*, Austin, TX (1999) pp. 432–439.
- [28] E. Pitoura and P. Chrysanthis, Exploiting versions for handling updates in broadcasting disks, in: *VLDB* (1999) pp. 114–125.
- [29] H. Schwetman, CSIM users guide (January 2001), <http://www.mesquite.com/userguidespage.htm>
- [30] A. Seifert and M.H. Scholl, Processing read-only transactions in hybrid data delivery environments with consistency and currency guarantees, Technical Report, No. 163, University of Konstanz (December 2001).
- [31] J. Shanmugasundaram, A. Nithrakasyap, J. Padhye, R. Sivasankaran, M. Xiong and K. Ramamritham, Transaction processing in broadcast disk environments, in: *Advanced Transaction Models and Architectures*, eds. S. Jajodia and L. Kerschberg (Kluwer, Dordrecht, 1997).
- [32] J. Shanmugasundaram, A. Nithrakasyap, R. Sivasankaran and K. Ramamritham, Efficient concurrency control for broadcast environments, in: *SIGMOD Conference* (1999) pp. 85–96.
- [33] W.E. Weihl, Distributed version management for read-only actions, *TSE*, SE-13(1) (January 1987) 55–64.
- [34] K.L. Wu, P.S. Yu and C. Pu, Divergence control for epsilon serializability, in: *ICDCS*, Phoenix, AZ (February 1992) pp. 506–515.



André Seifert is currently Ph.D. student in the Department of Computer Science of the University of Konstanz. He received the Bachelor degree in economic science from the University of Applied Science, Mittweida, Germany, in 1997 and the Master degree in information science from the University of Konstanz, Germany, in 1999. His research interests include replication and transaction management in distributed stationary and mobile environments, load balancing in high performance systems, and consistency management and data integration in intermittently connected systems.
E-mail: Andre.Seifert@uni-konstanz.de



Marc H. Scholl is Full Professor of Computer Science in the Department of Computer and Information Science, University of Konstanz, Germany. Earlier, he has held positions at the University of Ulm, Germany (Associate Professor, 1992–1994) and at ETH Zurich, Switzerland (Assistant Professor, 1989–1992). He received his Diploma (Masters, 1982 and Ph.D. degrees, 1988), both in computer science, from the Technical University of Darmstadt, Germany. His current research interests cover DBMS design and architecture, query processing and optimization, database models and languages, XML databases, and mobile database applications. Dr. Scholl has been serving on PCs and organizations of major international database conferences, he is a member of editorial board of the *VLDB Journal*, and is a member of a number of professional organizations (ACM, IEEE, GI). Currently, he is also Vice-President of the University of Konstanz.
E-mail: Marc.Scholl@uni-konstanz.de