Concurrency and Replica Control for Constraint-based Database Caching

Joachim Klein

Databases and Information Systems, Department of Computer Science,
University of Kaiserslautern, Germany
jklein@cs.uni-kl.de

Abstract. Efficient and dynamic reallocation of data is a major challenge of distributed data management, because current solutions require constant monitoring and manual adjustment. In contrast, future solutions should provide autonomic mechanisms to achieve self-tuning and exhibit high degrees of flexibility and availability. Constraint-based database caching (CbDBC) is one of the most ambitious approaches to reach these goals, because it is able to dynamically respond to workload changes and keep subsets of data near by the application. In turn, caching of data always generates replicas whose consistency needs to be controlled—for reasons of data independence, transparent for both application and underlying DBMS. Hence, such a task can best be approached by a middleware-based solution.
This paper discusses challenges arising when distributed replicas are synchronized within CbDBC. We compare proposals using eager and lazy update propagation and review their feasibility within middleware-based realizations. Because constraints have to be maintained by the cache, they restrict the implementation of concurrency control mechanisms. Therefore, we explore, as a novel contribution, the far-reaching influence of these constraints.

1 Motivation

Similar to concepts used for *Web Caching*, database caching keeps subsets of records close to applications, which allows local and, hence, faster execution of declarative queries. In contrast to Web caching only supporting ID-based queries, database caching services set-oriented requests and must, therefore, verify that the predicates used by SQL queries can be evaluated, i.e., that their *predicate extensions* [12] are contained in the cache. To this end, constraint-based database caching (CbDBC) uses simple constraints (cp. Section 2), which need to be fulfilled at any time and allow to decide whether or not a predicate extension is completely kept. Many database vendors have extended their systems with similar but less flexible ideas [1,2,21]. The most important competitor approach uses materialized views to determine the predicate completeness of records cached [16]. A big challenge of these approaches is replica control, because caching of data always implies the existence of distributed replicas. In addition, database

caching has to guarantee transaction properties, so that the employed concurrency control mechanism is of special importance. Because database caching has to solve exactly the problems occurring in (partially) replicated environments, the research results of this area are used as a starting point to choose an appropriate solution for CbDBC. Section 3 inspects the results and clarifies which methods can be used for replica control and concurrency control (CC) for database caching.

But, there is a major difference between caching and replication: the content of a cache is managed dynamically, which is a great advantage. The caching system can try to limit the number of replicas (regarding one data item) on its own, so that the number of caches that need to be updated remains small, even when many caches coexist. However, dynamic organization is the biggest problem. The constraints used to determine completeness become inconsistent, if updates are made. Therefore, applying an update requires additional refreshment or invalidation steps to guarantee consistency. Section 4 describes the problems arising for the constraint-based database caching approach.

Our CbDBC prototype ACCache [5] is realized as a middleware-based solution and, up to now, independent of a specific database system. We try to preserve this property and, hence, we explore the feasibility of various middleware-based approaches (cp. Section 5). But in doing so, we rely on the concurrency control of the underlying database system. First, we try to realize lazy update propagation being highly desired, before we explore eager approaches. Regarding schemes with lazy update propagation, we demonstrate that a middleware-based solution is only realizable providing limited functionality, so that just read-only transactions can be executed . Based on this observation, we explore eager approaches (cp. Section 5.2) which enable the cache to accelerate any read statement (also of writer transactions). Eager solutions, however, have to use RCC value locks (introduced in Section 4) to speed-up commit processing, which are not needed in lazy solutions.

The following section describes the constraint-based approach in more detail and repeats the most important concepts just for comprehension.

2 Constraint-based Database Caching

A constraint-based database cache stores records of predicate extensions in so-called *cache tables*. The records are retrieved from a primary database system called *backend*. Each cache table T belongs to exactly one backend table T_B and, hence, their definitions are equivalent, except for foreign key constraints, which are not adopted. The tables and constraints maintained by a cache are represented as a so-called *cache group*.

A CbDBC system uses two different types of constraints to determine which predicate extensions are completely contained in the cache: Referential Cache Constraints (RCCs) and Filling Constraints (FCs). Both are defined using the fundamental concept of *value completeness*.

Definition 1 (Value completeness). *A value v is value-complete (or complete for short) in a column $T.a$ if and only if all records of $\sigma_{a=v}T_B$ are in T.*

An RCC $S.a \rightarrow T.b$ is defined between a source column $S.a$ and a target column $T.b$ (not necessarily different from $S.a$) to ensure value completeness in $T.b$ for all distinct values v in $S.a$. Please note, value completeness is just given for the target column. This allows, e. g., to execute an equi-join $S \bowtie_{a=b} T$ if value completeness is given for a value v in S (let us say for v in $S.c$), so that $\sigma_{S.c=v}(S \bowtie_{a=b} T)$ delivers the correct result. In the reverse case (value completeness is given for a value v in T), this join is could produce incomplete results[1].

Definition 2 (Referential cache constraint, RCC). *A referential cache constraint $S.a \rightarrow T.b$ from a source column $S.a$ to a target column $T.b$ is satisfied if and only if all values v in $S.a$ are value-complete in $T.b$.*

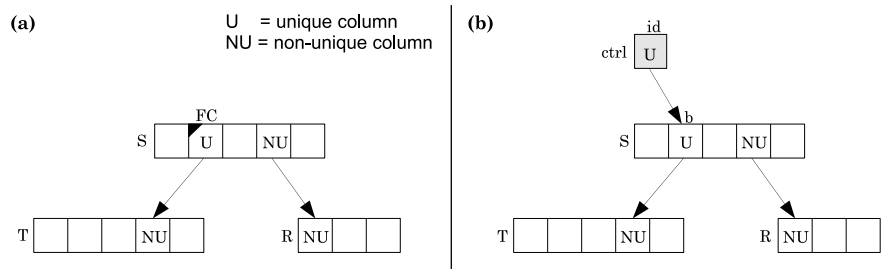

Fig. 1. The main components of a cache group and its internal representation.

An FC is defined on a single column (e. g., $S.b$) and determines when a value v needs to be loaded completely. The loading is initialized as soon as a query refers to v explicitly (e. g., through $\sigma_{S.b=v}S$) and v is in a set of values to be loaded (called candidate values [12]). To implement the behavior of FC $S.b$, we internally use a so-called *control table* (*ctrl*) and an RCC $ctrl.id \rightarrow S.b$ (cp. Figure 1). Conceptually, we put the value v in the id column of the control table. This violates RCC $ctrl.id \rightarrow S.b$ and, hence, triggers maintenance, i.e., all records $\sigma_{b=v}S_B$ have to be made available. In doing so, new RCC source-column values in S arrive at the cache. This may violate outgoing RCCs which need to be satisfied again. In this way, depending on the value we put into the control table, the cache tables need to be filled-up in a consistent way.

Because of the special importance of values kept in an RCC source column, we denote such values as *control values* [15]. As illustrated, the presence of a control value (e. g., v) demands the availability of (recursively) dependent records. Hence, we denote this set of records as *closure* of the control value v.

[1] One reason that shows that an RCC is different from a foreign key.

Definition 3 (Closure of a control value). *Let v be a control value of RCC $S.a \rightarrow T.b$ and, thus, $I = \sigma_{a=v}T_B$ the set of records that have to be value-complete. The closure of v is the set of records $C(v) = I \cup C(v_i)$, $\forall v_i \in V(I)$, where $V(I) = (v_1, ..., v_n)$ denotes the control values included in I.*

Both constraint types (FCs and RCCs) are used to determine if a set of records currently stored in the cache is value-complete. Values of unique columns are implicitly, i.e., always complete. With help of this simple concept, it becomes possible to decide if predicates are completely covered by the cache and, hence, whether queries can be executed locally or not.

3 Preliminary Considerations

The main challenge regarding a replicated database environment is replica control. Typically used to control read-intensive workloads[2], our approach is based on a "read one replica write all (available) replicas" (ROWA(A)) schema. In the seminal paper of Gray et al. [8], replication techniques are classified by two parameters. The first parameter specifies where updates can be executed, at a primary copy or everywhere.

With database caching, caches temporarily hold data from a primary data source maintaining the consistency of all data items [12]. This so-called backend defines the schema that is visible to the user, whereas the caches as in-between components remain transparent. In contrast to replication, a primary copy approach fits into such a system architecture in a natural way, where the backend performs all updates and propagates them to the caches (if needed).

However, the given system architecture extremely complicates an *update-everywhere* solution. An important problem is that a cache cannot decide if an update violates constraints defined at the backend database, because it does not store any foreign-key constraints, check constraints, definitions for tables not present, triggers, or other information needed. All this meta-information had to be available to perform updates, so that cache maintenance could be done locally. In addition, *update everywhere* requires complex concurrency control or conflict resolution [8]; therefore, we strongly recommend the use of a primary copy approach where updates are always forwarded to the backend.

The second parameter introduced in Gray's paper [8] describes when replicas are refreshed, which can be done in *eager* or *lazy* fashion. In eager approaches, the changes of a transaction are propagated to all replicas before commit, whereas in lazy approaches the propagation may take place after commit. Because eager solutions delay transaction execution and lazy solutions have to deal with consistency problems, the most recent approaches (e. g., [3,11,14,17,22]) are designed as interim solutions, providing a well-defined isolation level and a so-called hybrid propagation, where, on the one hand, transactions accessing the same replicas do coordinate before commit (eager) and, on the other hand, successful commit of a replica is acknowledged to the client (lazy update propagation). In Section 5, we

[2] This can be generally assumed in scenarios where database caching takes place.

illustrate that, using database caching, it is possible to apply lazy update propagation without consistency problems. Hence, we can use an approach where commit is acknowledged to the client as soon as the transaction updates are committed at the backend database, while caches are updated lazily.

However, update propagation must be combined with an adequate CC mechanism [17,22]. As the most important requirement when choosing or rather developing a tailor-made CC policy for CbDBC, the chosen mechanism should preserve a *caching benefit*, i. e., the performance gained from local query evaluation should not be outbalanced by cache maintenance. If the approach needs to access remotely maintained caches or the backend to perform read statements, the caching benefit will be compromised. For that reason, read accesses should never be blocked, e. g., to acquire distributed read locks as needed in a distributed two-phase locking (D2PL) approach. Another main problem for CC is that database caching is designed to cache data near to applications and, hence, caches are often allocated far away from the original data source, only reachable via wide-area networks. For that reason, a comparatively high network latency has to be anticipated to send CC messages (ca. 50–200 ms) and, thus, they must be avoided if possible.

If we scan recent research for CC approaches that fulfill these basic requirements, we find that only optimistic CC schemes and approaches using snapshot isolation (SI)[3] [4,7] as its isolation level are sufficient (cp. Section 5). Another possibility is to allow inconsistencies [9,10], but, first, the level of inconsistency needs to be defined by developers within SQL statements and, second, such cache systems do not scale if a high isolation level is required.

The preceding discussion clarifies that the basic assumptions and requirements of CbDBC dramatically decrease the number of viable approaches (for replica control and concurrency control). Moreover, it should be easy to combine the chosen approach for replica control with CC. Hence, our solutions provide SI which facilitate a simple integration with eager and lazy update propagation schemes.

In the following section, we examine the specific challenges that need to be solved if we implement update propagation, i. e., from the backend to the caches involved, within CbDBC.

4 Update Propagation

The process of propagating updates consists of three main tasks: gathering changed records (capture), identifying and informing the caches which are affected by changes (distribution), and accepting changes at the cache (acceptance).

[3] SI is a multi-version concurrency control mechanism presenting to a transaction T the DB state committed at $EOT(T)$. It does not guarantee serializable execution, but it is supplied by Oracle and PostgreSQL for "Isolation Level Serializable" or in Microsoft SQL Server as "Isolation Level Snapshot".

Capture. ACCache can be used on top of any relational database system and relies just on the SQL interface to implement its functionality. Provided by most database systems [13,18,19,20], triggers or appropriate capture technology for changed data is thus necessary. Eager approaches (cp. Section 5.2), therefore, have to gather all changes of a transaction before commit. Lazy approaches, in turn, allow to collect changes after commit. Such an approach can be handled much more efficiently by log-sniffing techniques, which may be even processed concurrently. The collected data must be provided as a *write set (WS)* that includes at least the following information for each transaction: transaction id (*XId*), begin-of-transaction (*BOT*) timestamp, end-of-transaction (*EOT or commit*) timestamp, and all records changed. Furthermore, each record is described by an identifier (*RId*), the type of values included (*VType* := new values or old values), the type of DML operation (*DMLType* := insert, update, or delete), and the values themselves. For each update, two records (with old and new values) are included.

Distribution. For each record in a WS, we need to decide whether or not some of the connected caches have to be informed about the change. For this task, ACCache provides a dedicated service (Change Distributor, CD) running at the backend host. Depending on the meta-information maintained, a fine-grained or just a coarse-grained solution is possible. We distinguish the following levels of meta-information to be maintained:

- *Cache Information.* In this case, CD only knows that connected caches exist. Therefore, its only option is to ship the whole WS to all of them. In most cases, this level is not recommended, because, typically, just a few tables of a schema are of interest and, hence, appear in a cache.
- *Table Information.* This level additionally keeps for each cache the names of the cached tables. Therefore, selective shipment of the changed records is possible.
- *Cache Group Information.* It expands table information by storing all cache group definitions at the backend. Providing no additional help for the CD service, all changed records still have to be shipped to a cached table. However, we differentiate between this level and table information, because the backend can effectively assist loading policies using this additional information (cp. *prepared loading* in [15]).
- *Perfect Information.* In this case, CD maintains a special hash-based index to determine if a record is stored in a cache or not. With this support, it is possible to check each record of the WS and ship just the currently stored ones.

Obviously, perfect information enables a fine-grained selection of the records that need to be shipped to a cache. But the meta-information maintained must be continuously refreshed and needs to be consistent to ensure correctness. In summary, perfect information unnecessarily stresses the backend host and, therefore, we suggest using table information or cache group information.

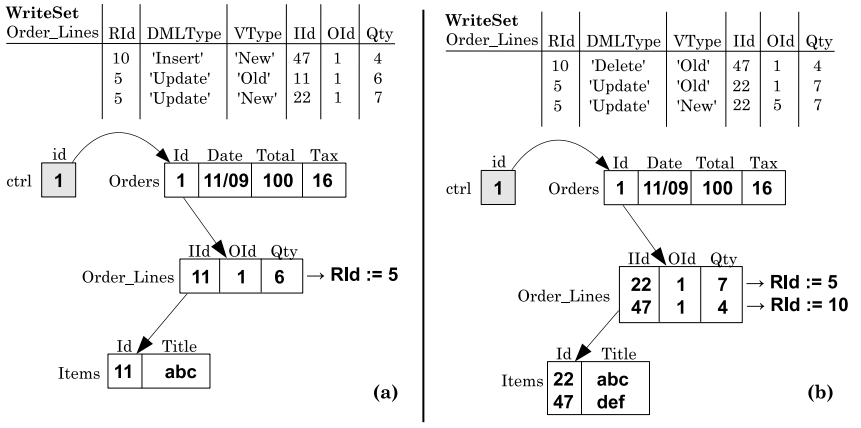

Fig. 2. Arrival of new control values (a), records losing their dependencies (b)

Acceptance. The most critical part of CbDBC is the dynamic and concurrent acceptance of changes. Using "normal" replication, it is sufficient to reproduce the changes of the primary copy to refresh a replica. Within CbDBC, a change, for example, of value v to w may violate constraints and, therefore, changes in a cache must be hidden until all constraints are satisfied. Because we internally model FCs through RCCs (and control tables, cp. Section 2), we only need to observe RCC violations. The only situation, where RCCs can be violated, occurs when new control values reach the cache, i. e., during an update or insert. Figure 2a gives an example for this situation. The WS of table *Order_Lines* includes a new record with $RId = 10$ that inserts the control value $IId = 47$ and an update for the record with $RId = 5$ changing the control value 11 to 22. Hence, RCC $Order\_Lines.IId \rightarrow Item.Id$ is no longer satisfied and the closures of 22 and 47 need to be loaded first, before the WS can be accepted. In all other cases, the RCCs remain valid, but some records may be unloaded, if no incoming RCC implies their existence (cp. Figure 2b). Here, the delivered WS claims the deletion of record with $RId = 10$ and signals an update from $OId = 1$ to $OId = 5$ for the record with $RId = 5$. After acceptance of these changes, the closure of $Orders.Id = 1$ is empty and, thus, all records in the tables *Order_Lines* and *Items* should be unloaded.

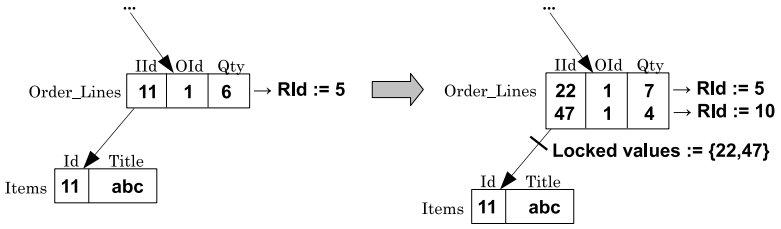

Fig. 3. Accepting of changes (WS from Figure 2a) using RCC value locks.

Because the load of closures may be time consuming [15], commit processing may be considerably delayed. To overcome this problem, we use RCC value locks which indicate that, for a given RCC, some of the control values are not value-complete at the moment. This allows us to accept updates as soon as all locks are set (cp. Figure 3, where the same WS is applied as shown in Figure 2a). These locks accelerate the processing of write sets, which is very important for eager concurrency control schemes. However, RCC locks constrain the execution of joins and, hence, using them with lazy approaches is not recommendable.

5 Concurrency Control

As described in Section 3, the most important requirement when choosing an appropriate CC mechanism is to preserve the caching benefit. Middleware-based solutions (like ACCache) are implemented on top of existing CC policies provided by the underlying database system and have to regard their special properties. Most restricting, a transaction T accessing these underlying systems has only access to the latest transaction-consistent state valid at $BOT(T)$. Hence, we denote this state as *latest snapshot*.

Observing current research activities, the simplest and, hence, most likely the best way to preserve the caching benefit is to allow read accesses without taking further actions, i. e., without retrieval of read locks, setting of timestamps, or collecting of read sets (e. g., for an optimistic CC policy). This has been possible since we know about the very powerful properties of SI, where reads are never blocked. It allows a cache to execute any read statement from any transaction without gathering information about elements read. However, to provide SI for database caching, the same snapshot has to be maintained for all statements of a transaction. Hence, this so-called *global snapshot* needs to be provided by a cache in combination with the backend.

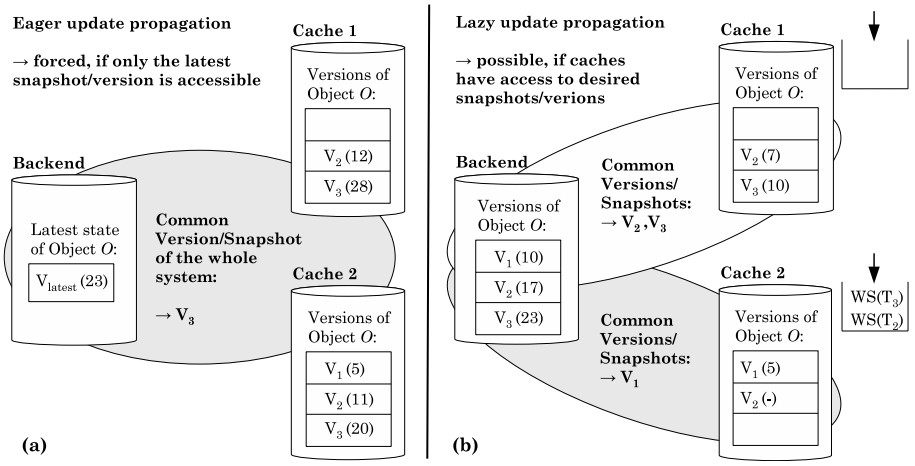

Fig. 4. Providing the same versions/snapshots either eager (a) or lazy (b).

This basic requirement can only be realized if either backend and caches provide always the latest snapshot (eager, cp. Figure 4a) or the caches have access to required snapshots at the backend (lazy, cp. Figure 4b).

In Figure 4, the point in time (represented through an integer value) when a version is locally committed is given in parentheses. Because cache and backend use their own *local* CC mechanisms, the timestamps assigned to the same version by both sides will differ. A cache is always supported by local CC mechanism providing SI and, hence, it maintains multiple versions. In the model for eager update propagation (given by Figure 4a), all caches have to accept a transaction's WS logically at the same time. During lazy update propagation (reconsider Figure 4b), caches maintain a queue of WSs that have to be applied in FIFO (first in first out) order. The queue of `Cache1` is currently empty and `Cache2` has to accept at first $WS(T_2)$ and after that $WS(T_3)$.

In all further explanations, we mark a read-only *user transaction* T_j with a subscript *sr* (e. g., T_{sr1}), if it executes just a single read, and with *mr*, if the transaction consists of multiple read statements. Write transactions are not differentiated any further.

Each *user transaction* is executed by a cache C_i and the backend where C_i is the cache that took control over the user transaction. Hence, for each user transaction T_j, the cache maintains a *cache transaction* T_j^{ca} to access its local data source (i. e., the cached data) and a *backend transaction* T_j^{be} to access the backend data. A cache transaction or a backend transaction not initiated by the user is simply marked with its purpose (e. g., T_j^{load} is used to load new cache contents).

Regarding correctness, it is sufficient to prove that each user transaction T_j realized with the aid of T_j^{ca} and T_j^{be} accesses the same snapshot S_i, because all changes are synchronized by the backend database.

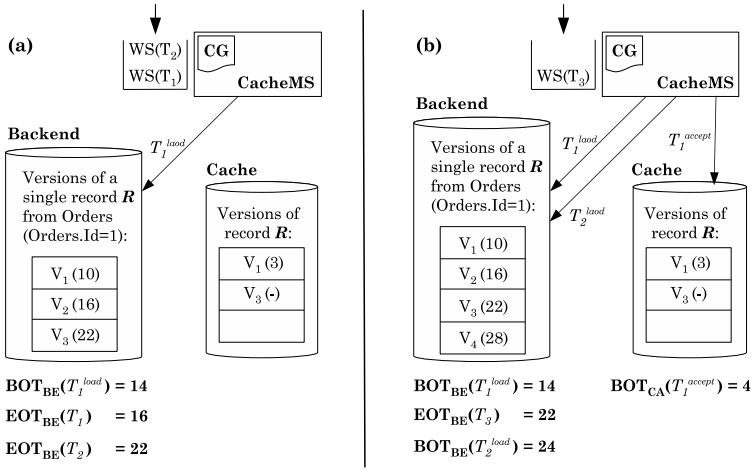

Fig. 5. Refreshing of T_1^{load} to T_2^{load}.

5.1 Lazy update propagation

To allow lazy schemes, the only solution is to keep a backend transaction T^{load} open that allows reading the latest snapshot provided by the cache. Regardless of the problems of realizing a user transaction (i. e., commit cannot be processed without losing the link to the right snapshot), the *refresh* of such connections (i. e., switching to the next transaction representing a new snapshot) is critical. In addition, T^{load} must be used to execute read statements of user transactions that accesses the backend to reach the correct snapshot and because the commit of T^{load} is not permitted, the cache can just execute read-only user transactions.

Assume a cache retains a backend transaction T_1^{load} that reads a snapshot representing the state before transactions T_1 and T_2 are finished (cp. Figure 5a). To refresh T_1^{load} (e. g., after a short period of time), the cache creates a new transaction T_2^{load} representing the state after commit of T_1 and T_2. Given that the cache can arrange T_2^{load} (i. e., it can determine that $\mathrm{BOT}(T_2^{load}) > \mathrm{EOT}(T_1)$ and $\mathrm{BOT}(T_2^{load}) > \mathrm{EOT}(T_2)$), the cache has to accept the changes of $\mathrm{WS}(T_1)$ and $\mathrm{WS}(T_2)$ (e. g., through a cache transaction T_1^{accept}), before switching to T_2^{load} is possible (cp. Figure 5b).

In addition, while changes are accepted, new records need to be loaded (cp. Section 4). The loading is still performed by T_1^{load} and, thus, newly loaded records can recursively be affected by changes within $\mathrm{WS}(T_1)$ and $\mathrm{WS}(T_2)$. Hence, each record loaded must be checked against $\mathrm{WS}(T_1)$ and $\mathrm{WS}(T_2)$ to ensure that all changes get accepted correctly. Only if all changes within $\mathrm{WS}(T_1)$ and $\mathrm{WS}(T_2)$ have been processed completely or the loading over T_1^{load} gets shortly suspended, T_1^{accept} can be committed and T_2^{load} can be used for further processing. T_1^{load} is released as soon as no user transaction requires it anymore (i. e., if just user transactions T_j with $\mathrm{BOT}(T_j) > \mathrm{BOT}(T_2^{load})$ are executed by a cache).

Absolutely impossible is the usage of databases that apply pessimistic CC. The retained transaction will cause deadlocks and after an induced abort the state needed is no longer accessible.

Single-read transactions. To overcome the problem of accessing the same snapshot at cache and backend, we can try to allow only single-read transactions at the cache. We have to limit transactions to execute just one read statement, because further statements may need backend access. This approaches restrict the usage of the cache but, even in that case, lazy update propagation cannot be realized, as our following example shows.

Figure 6 illustrates the situation mentioned before. `Cache1` provides the transaction-consistent state (snapshot) before T_1 was committed, because the WS of T_1 is still available in the queue of WSs to be processed. For that reason, the transactions T_{sr6} and T_{sr7} are logically executed before T_1. `Cache2` has already applied $\mathrm{WS}(T_1)$ and, hence, T_{sr9} is after T_1 and before T_2. Assuming that backend and caches provide serializability for their local transactions, the transactions are also globally serializable, because writers are synchronized at the backend. This concept appears to be realizable in a simple way. Caches

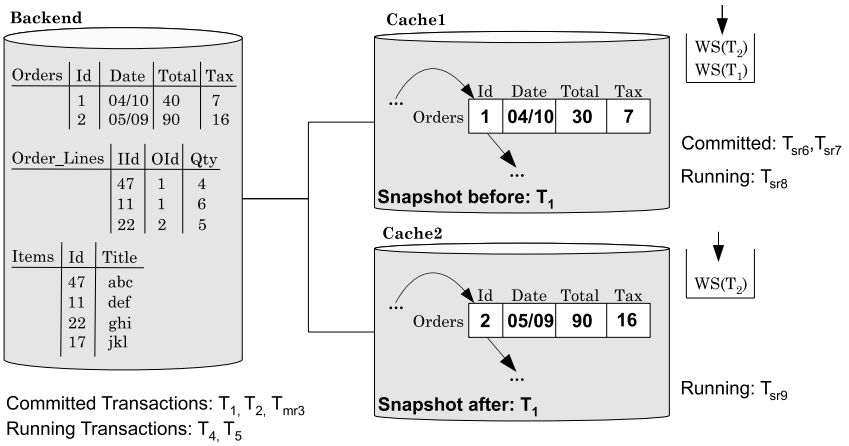

Fig. 6. Caches only executing single-read transactions.

could be lazily refreshed by accepting the WSs delivered as explained in Section 4. However, global serializability is only guaranteed if caches always provide a transaction-consistent state for its locally executed transactions.

The following example clarifies that this is not possible if only the latest snapshot is accessible at the backend, because, then, necessary loading operations (e. g., triggered through the standard filling behavior or during update acceptance) access different snapshots.

Considering Figure 7, we assume that the transaction T_1 increases the total amount of order 1 from 30 to 40 and of order 2 from 70 to 90. If we sum up the total amounts of order 1 and 2 before T_1 starts, we see a consistent state of 100; after commit of T_1 the sum of 130 is correct. The cache shown in Figure 7 has already loaded order 1 and starts accepting changes in the WS of T_1. Hence, at that moment, it provides the state before T_1. In this state, two single-read transactions run before the acceptance of WS(T_1) are finished. The first one, T_{sr1}, selects the order 2 that is not kept by the cache but triggers the loading of it. We assume that the loading occurs immediately and loads order 2, but now, the loading operation accesses the snapshot after T_1, because this is the latest snapshot and T_1 already committed at the backend. If the second transaction T_{sr2} now sums up the total amount of order 1 and 2, the answer given by the cache is 120, which was never a consistent state at the backend.

As a result, we conclude that a middleware-based solution with lazy update propagation can only be implemented on top of databases providing SI. Middleware-based approaches cause a lot of limitations: Changes of multiple write sets must be accepted together, only one transaction can be kept to access the correct snapshot, concurrently loaded records need to be checked against write sets, and the cache can just perform reads of read-only transactions.

Therefore, to support all read statements (also of writer transactions) at the cache using a middleware-based solution, building an eager solution is mandatory.

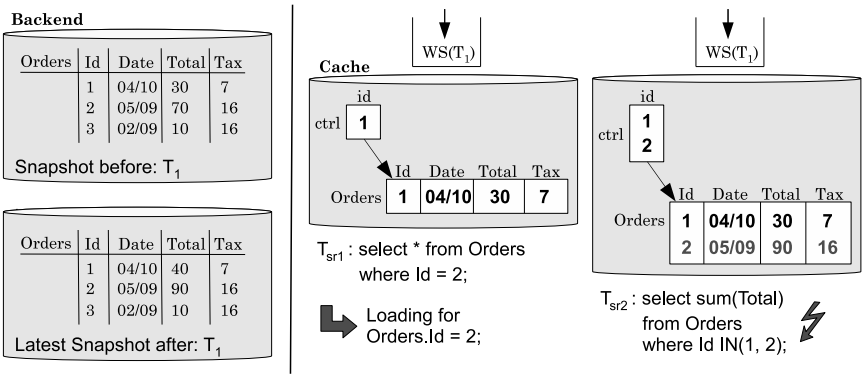

Fig. 7. Loading operations construct an inconsistent state and cause chaos.

5.2 Eager Update Propagation

It is well known that eager solutions do not scale well, but within database caching they have important advantages: As backend, they allow any kind of database system providing SI[4] and their realization does not cause changes at the backend. In addition, if the locality at caches is very high[5], updates mostly affect only a very small number of caches (in the best case only one), so that commit processing performance is acceptable even when many caches co-exist. Subsequently, we explain the tailor-made commit procedure based on the well-known two-phase commit (2PC) protocol. It can be easily integrated with the update propagation described in Section 4 to reach a fully working concurrency control. However, beyond the limited scalability, the realization of eager approaches pose problems that impede the cache.

Commit Processing. A 2PC protocol synchronizes the replicas before commit and, thereby, guarantees that the backend and all caches provide only the latest snapshot to the user. The most ambitious challenge is to prepare the caches, so that the subsequent abort or commit message can be safely executed. But first of all, we respond to error processing which can be substantially simplified because cache databases do not need to be durable.

It allows us to commit transactions even if errors occur within the preparation phase at caches. After sending the WSs to the affected caches, the backend defines a period of time (timeout) in which the caches have to answer. If a cache signals a failed prepare or is not answering, it is invalidated. That means, all affected transactions are aborted. In the simplest case, a reinitialization is

[4] For eager methods, a database system using pessimistic CC could also serve as backend, in principle, but it may cause deadlocks that potentially affect performance. In addition, its use limits the level of isolation, because transactions running in the cache do not acquire read locks.

[5] Using database caching, the system can try to reach this state through an adaptive reorganization.

enforced (i. e., purging or restarting) to achieve a consistent state at failed caches. At the end, the transaction and all caches in the *prepared state* can commit.

If the coordinator crashes, all running user transactions whose statements need to be redirected to the backend are automatically aborted. Transactions that were currently in the prepare phase are aborted after restarting the backend. If the backend is available again, normal processing can be continued without further adjustments. The last scenario nicely shows the improved fault tolerance of the global system, because the data kept by the caches remains accessible in case of a coordinator failure.

Preparation of Changes. When a cache receives a `Prepare` request of a user transaction (e. g., T_1), it has to process the corresponding WS(T_1) as fast as possible to shorten commit processing. Hence, loading of all records, needed to remedy violated RCCs (cp. Section 4), would imply excessive costs. Moreover, if an `Abort` message is received later, records were unnecessarily loaded. For that reason, we start a cache transaction T_1^{accept} that accepts the changes and assigns RCC value locks (cp. Figure 3) to all invalidated control values. As a result, the control values locked are invisible for probing, i. e., they can not be used to determine the completeness of records in target columns of RCCs checked. Furthermore, an RCC holding value locks can not be used to perform an equi-join.

As soon as all RCC value locks are applied, the cache sends the `Ready` signal to the backend and waits for the `Commit`/`Abort` message. If an `Abort` is received, the cache simply aborts T_1^{accept} and removes all RCC value locks previously assigned. After a `Commit` instruction, T_1^{accept} gets committed, too, and a loading process is initiated for each RCC value lock, which reconstructs value completeness for the control value and removes the lock.

Since the cache management system provides SI for its locally executed transactions, the changes and locks assigned through T_1^{accept} are only visible for local transactions T_i^{ca}, whose BOT(T_i^{ca}) > BOT(T_1^{accept}), so that transactions with BOT(T_i^{ca}) < BOT(T_1^{accept}) are not hindered.

Problems. Our proposal indicates how eager solutions should be designed for CbDBC. To cover the entire spectrum of approaches, we explored other opportunities as well [6], but all of them need to invalidate records or control values, which straiten the usage of cache contents.

But, furthermore, all proposals suffer from another important disadvantage: All participating instances of the system have to logically commit at the same time. If the backend would first commit (e. g., after having received all `Prepare` messages), the updates of a transaction T at the backend are immediately visible. But at the cache side, if the `Commit` instruction has not been received yet, T's snapshot is still present. Hence, a user transaction (e. g., T_2) initiating such a situation may access different snapshots (cp. Figure 8a). Of course, this problem can be simply avoided by redirecting requests from T_2 to the backend using T_2^{be} after the cache sends `Ready` to the backend. And after receiving `Commit`, a cache

transaction T_2^{ca} can be initialized and used (cp. Figure 8b). But cache usage is again prevented for transactions initiated in the meantime.

In summary, eager approaches do not scale and, thus, they should not be used within wide-area networks.

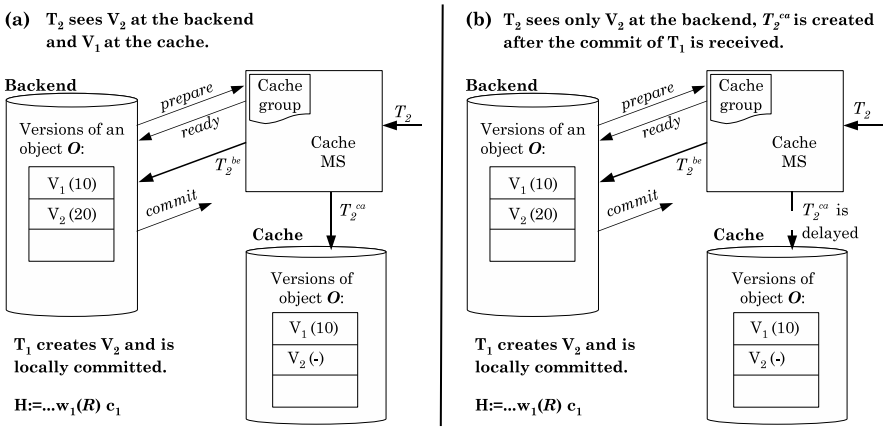

Fig. 8. T_2 can still access different snapshots (a) and is thus redirected (b).

6 Summary and Future Work

Using database caching, preserving the caching benefit is a superordinated requirement. In order to fulfill it, we can only use CC policies that allow to read the cached data without further coordination steps (e. g., accessing the backend database to acquire read locks). Multi-version concurrency control mechanisms providing SI (attached to the most recent database systems) offer almost perfect properties to realize this high concurrency level in database caching and especially in CbDBC. Middleware-based realizations are indeed implementable, but our observations clarify that they have some restrictions. The preferred lazy update propagation only supports read-only transactions and using eager update propagation, cache contents need to be temporarily locked. As a big drawback, backend systems using pessimistic CC policies may seriously affect transaction processing, because update acceptance in a cache triggers load operations that may cause deadlocks.

With help of our examination, we found out that an integration of lazy update propagation in a CC mechanism providing SI is realizable, but a middleware-based solution is challenging and shows restrictions, because backend transactions can only access the latest snapshot. Hence, we start implementing a closer integration of both CC mechanisms used at backend and cache, so that the cache has an influence of selecting the right snapshot for backend transactions.

References

1. Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: Cache Tables: Paving the Way for an Adaptive Database Cache. In: VLDB Conf. (2003), 718–729
2. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: DBProxy: A Dynamic Data Cache for Web Applications. In: ICDE Conf. (2003), 821–831
3. Amza, C., Cox, A., Zwaenepoel, W.: Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Sites. In: Proceedings of the Fourth Middleware Conference (2003)
4. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A Critique of ANSI SQL Isolation Levels. In: SIGMOD Conference (1995), 1–10
5. Bühmann, A., Härder, T., Merker, C.: A Middleware-Based Approach to Database Caching. In: ADBIS Conf., LNCS 4152, Springer (2006), 182–199
6. Braun, S.: Implementation and Analysis of Concurrency Control Policies for Constraint-Based Database Caching (in German). Master's thesis, TU Kaiserslautern (2008), http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/users/jklein/documents/_2008_Braun_.DA.pdf
7. Fekete, A.: Snapshot Isolation. In: Ency. of Database Systems (2009), 2659–2664
8. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The Dangers of Replication and a Solution. In: SIGMOD Conference (1996), 173–182
9. Guo, H., Larson, P.A., Ramakrishnan, R.: Caching with "Good Enough" Currency, Consistency, and Completeness. In: VLDB, VLDB Endowment (2005), 457–468
10. Guo, H., Larson, P.A., Ramakrishnan, R., Goldstein, J.: Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. In: SIGMOD, ACM, New York, NY, USA (2004), 815–826
11. Holliday, J., Agrawal, D., Abbadi, A.E.: The Performance of Database Replication with Group Multicast. In: FTCS (1999), 158–165
12. Härder, T., Bühmann, A.: Value Complete, Column Complete, Predicate Complete – Magic Words Driving the Design of Cache Groups. The VLDB Journal **17**(4) (2008) 805–826
13. IBM: InfoSphere Change Data Capture (2009), URL http://www-01.ibm.com/software/data/infosphere/change-data-capture/
14. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., Alonso, G.: Improving the Scalability of Fault-Tolerant Database Clusters. International Conference on Distributed Computing Systems (2002) 477
15. Klein, J., Braun, S.: Optimizing Maintenance of Constraint-Based Database Caches. In: ADBIS (2009), 219–234
16. Larson, P., Goldstein, J., Zhou, J.: MTCache: Transparent Mid-Tier Database Caching in SQL Server. In: ICDE Conf. (2004), 177–189
17. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware-based Data Replication providing Snapshot Isolation. In: SIGMOD (2005), 419–430
18. Microsoft Corporation: SQL Server 2008–Change Data Capture (2009), URL http://msdn.microsoft.com/en-us/library/bb522489.aspx
19. Oracle Corporation: Data Warehousing Guide–Change Data Capture (2009), URL http://download.oracle.com/docs/cd/E11882_01/server.112/e10810.pdf
20. The PostgreSQL Global Development Group: Postgresql 8.4.3 Documentation–Triggers (2009), URL http://www.postgresql.org/files/documentation/pdf/8.4/postgresql-8.4.3-A4.pdf
21. The TimesTen Team: Mid-tier Caching: The TimesTen Approach. In: SIGMOD Conf. (2002), 588–593
22. Wu, S., Kemme, B.: Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In: ICDE (2005), 422–433