Degrees of Transaction Isolation in SQL*Cache: A Predicate-based Client-side Caching System

Julie Basu
Stanford University
and
Oracle Corporation

Arthur M. Keller *
Stanford University
Computer Science Department
Palo Alto, CA 94305-2140

May 15, 1996

Abstract

A caching scheme that uses query predicates to cache data on the client-side in a client-server relational database system was presented in [15]. The client-side cache (henceforth referred to as a *SQL*Cache*), loads query results dynamically in the course of transaction execution, and formulates a *cache description* based on the query predicates. SQL*Cache is *associative* in nature, in that it supports *content-based* reasoning and local execution of SQL queries on the cached data, thereby attempting to reduce query response times and increase server throughput. Local caching involves cache consistency maintenance and complicates transaction concurrency control — in this paper, we examine serializability questions that arise in such a system. We first analyze the issues in supporting different (0/1/2/3) degrees of isolation that client transactions may specify, and describe algorithms to achieve these isolation levels. Then, we propose new levels of isolation that take into account the distributed nature of transaction execution, in terms of the *lag* of locally cached data with respect to the server database and discrepancies between local and remote reads. The SQL*Cache scheme is currently being implemented using a main memory database as the client-side cache, and commercial database servers as the backends.

1 Introduction

In an earlier paper [15], we proposed a predicate-based client-side caching system for client-server relational databases, that allows inter-transaction reuse of cached data. The primary design goals of such an associative cache, called a SQL*Cache, are twofold: (1) to reduce network traffic and query response times, and (2) to increase server throughput. Data is dynamically loaded from the server database into a SQL*Cache based on queries submitted by the clients, and the current cache contents are described by predicates derived from the query predicates. Thus, SQL*Cache essentially supports client-side caching of multiple views, each of which is the result of dynamic query computation in the process of transaction execution.

*For more information please write to `ark@cs.stanford.edu`.

When a transaction submits a query, it is intercepted locally by SQL*Cache, and compared (conservatively, as described in [15]) against the cached predicates. If the query result is found to be contained in the cached predicates, a cache *hit* occurs, and SQL*Cache executes the query locally on the cached data. A cache *miss* causes transmission of the query to the remote server, followed by query evaluation at the server site, the results of which may or may not be cached locally upon return to the client. In this paper, we examine serializability issues that can arise due to query evaluation on locally cached data.

Transactions may also perform update operations, and questions of concurrency control must be addressed in order to provide isolation guarantees to client transactions. Additionally, maintaining the currency of the cached data requires the propagation of committed updates from the remote server, in an asynchronous (or deferred) way but possibly interleaved with the read and write operations of the currently running client transaction. As discussed in later sections, the timing of update propagation is closely related to the issues of transaction consistency and the concurrency model of the server database; the *when* and *how* of cache maintenance directly influences the isolation level of client transactions.

The rest of this paper is organized as follows. In Section 2, we provide an overview of the architecture of a SQL*Cache system. Section 3 reviews related work. In Section 4, we formalize the notions of cache consistency and currency for SQL*Cache. The issues in supporting different degrees of transaction isolation in the presence of a SQL*Cache are analyzed in Section 5. Section 6 proposes extended isolation levels that are appropriate for such client-server environments. Finally, in Section 7, we summarize our contributions, and outline ongoing work and future plans.

2 SQL*Cache Architecture

Client-side caching can offer substantial performance benefits by utilizing the local computing and storage capacity of today's intelligent clients. The SQL*Cache scheme is designed to employ the processing power of the clients of relational database systems, by dynamically loading into a client-side cache the results of queries executed by transactions, and by reusing the cached data for subsequent transactions. The aim is to reduce network traffic and query response times, thereby increasing server throughput.

2.1 Overview of System Configuration

We consider a central server and multiple clients that are individually connected to the server by a local area network. The relational database is resident at the server and transactions are initiated from client sites, with the server providing facilities for shared data access. A SQL*Cache server-side subsystem is present at the central server, and a client-side subsystem exists at each client site. The architecture of a SQL*Cache system for a single client is shown in Figure 1.

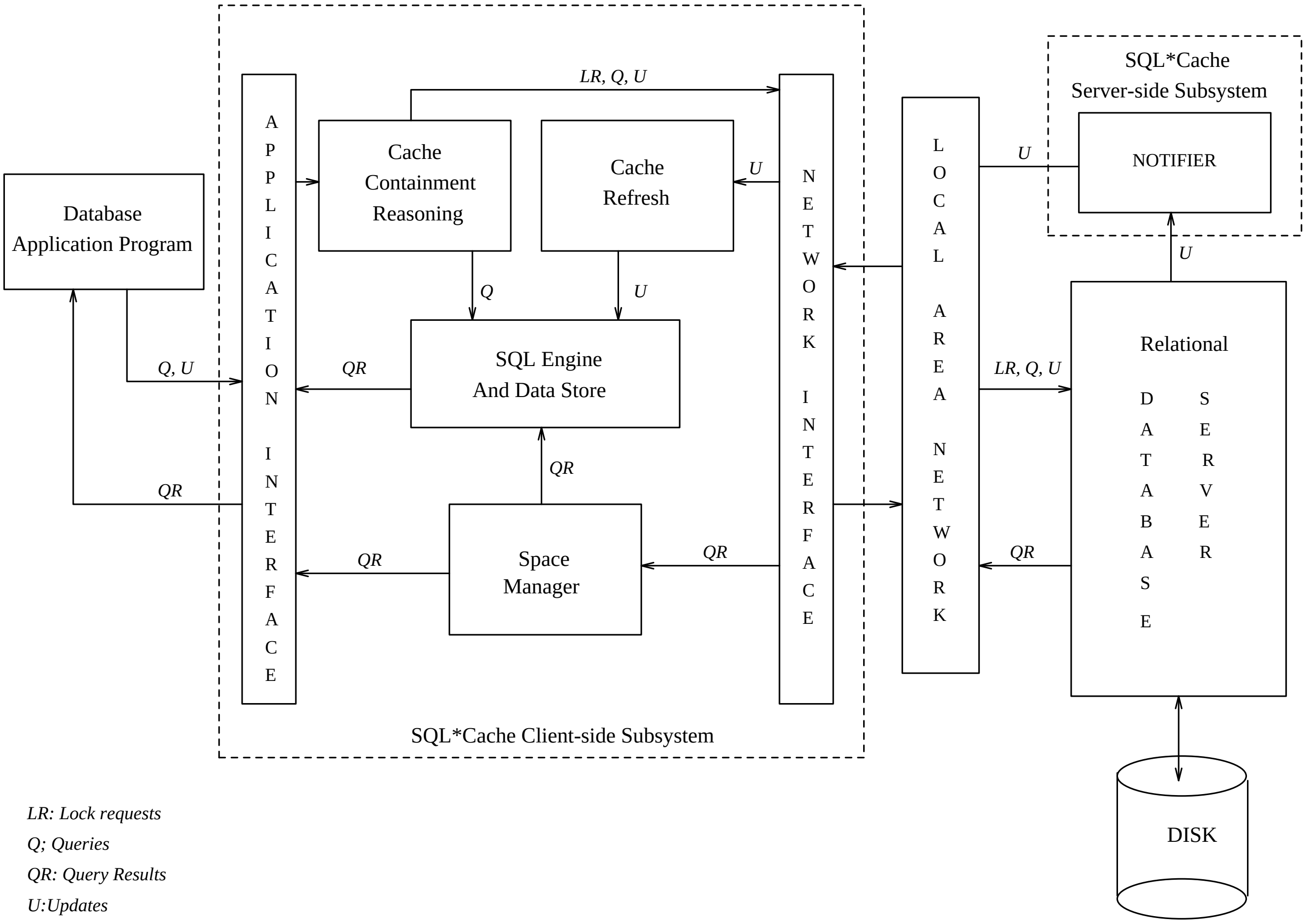

Figure 1. Architecture of a SQL*Cache System

2.2 SQL*Cache Subsystem on the Client-Side

SQL*Cache performs several functions at the client site — it decides whether a query result should be cached, how to reclaim space when the cache is full, how to execute queries locally, and most importantly, whether a posed query can be answered using the cached results. Associated to each client-side SQL*Cache subsystem is a SQL*Cache manager, which consists of four components:

- A cache containment reasoning subsystem,
- A local SQL engine with an associated data store,
- A space manager for loading and discarding query results, and
- An update notification handler for cache refresh operations.

Cache Containment Reasoning

The function of the cache containment reasoning system is to 'conservatively' compare each user query against the current SQL*Cache description, and return a Yes/No answer denoting whether the query result is contained in the cache or not. If the answer is Yes, then the query is submitted to the local SQL engine, which evaluates the query on the cached data.

Local SQL Engine and Data Store

The local SQL engine supports transactional semantics, and consists of a SQL parser, a query plan optimizer, and a query executor. For cache hits, query execution plans are constructed using locally defined indexes on the cached relations, and the query is executed on the cached data (after obtaining any necessary locks from the server, as described later in the isolation algorithms). These local indexes may be different from those in place at the server, and may, in particular, be defined on different sets of attributes, depending on data usage and access patterns at a client.

For local caching, we consider the class of SELECT-PROJECT-JOIN queries on database relations. A SQL*Cache is described in terms of query predicates that are derived from the user queries; however, query results are not directly stored in a 'materialized' form, as is done for materialized views. Rather, the schema of the cache reflects the schema of the database, with the cached relations containing a subset (corresponding to the stored query predicates) of the base relation tuples. A cached tuple may also be a subtuple of a base relation tuple, if not all attributes have been selected by the associated query. If a user query is to be cached, the list of selected attributes is 'augmented' as necessary to include relation keys and all other attributes appearing in the query, and the result inserted into the appropriate local relation(s). This implementation scheme aims to reduce storage and maintenance costs, by avoiding the caching of duplicate information and by simplifying update propagation. Details of our implementation choices are discussed in [15].

Space manager

It is the function of the space manager to determine whether a new set of tuples will be cached locally. This analysis is based on the anticipated frequency of usage of the query result and on space availability. Predicates and associated tuples may also need to be purged from the cache in order to reclaim space, and the cache descriptions at the client and the server changed accordingly. As described in [15], reclamation of space is done using a reference counting scheme, so that a cached tuple can be purged only when it is not referred to by any predicate in the client cache description.

Update notification handler

The update notification handler propagates updates that have previously been committed at the server to the cache. Incremental view maintenance techniques [12] are used to maintain the cached tuples and query predicates. Such incremental refresh of the cache may require inserting, updating, or deleting cached tuples, as well as modification of the cache description. For example, a cached query predicate may have to be invalidated upon an incremental update, or auxiliary information to refresh cached result be fetched from the server. Details of cache maintenance can be found in [15]. Note that timing issues are critical for update propagation, in that they affect the level of transaction consistency that is supported by the client. This issue will be the focus of later sections.

2.3 SQL*Cache Subsystem on the Server-side

In addition to the SQL*Cache manager at the client site, a server to client update propagation system known as the Notifier operates at the server site and communicates with the SQL*Cache manager process at the client site in order to support maintenance of the tuples in SQL*Cache.

Client Subscriptions

Each SQL*Cache can *subscribe* to a set of predicates on database relations, and register its subscription with the server-side notifier. In order that the client is informed of all relevant updates, a predicate to be cached must first be registered with the notifier before tuples corresponding to the predicates can be cached.

The Notifier

SQL*Cache managers are clients of the notifier running at the server site. It is the responsibility of the notifier to keep track of updates committed at the server, and to propagate the updates to its clients based on their registered subscriptions. The notifier may perform some filtering of updates, so as to eliminate updates that are irrelevant for a client cache. In general, the notification scheme follows a 'liberal' policy, whereby each SQL*Cache may receive some irrelevant notifications, but is guaranteed to receive all relevant ones.

3 Related Work

A client-side caching scheme related to SQL*Cache is the *ViewCache* technique employed in the ADMS± system, which uses the notions of *extended logical access path* and *incremental access methods* [18, 19]. One major difference of SQL*Cache with ADMS± is that our update propagation scheme is based on notifications of committed updates instead of server update logs. Additionally, the important issues of transaction isolation and consistency are not addressed by the ADMS± system.

A caching subsystem that can reason with stored relations and views is proposed in the *BrAID* system [21] to integrate AI systems with relational DBMSs. Some aspects of *BrAID* that pertain to local query processing, such as query subsumption and local versus remote query execution, are very relevant for our system. However, consistency maintenance of multiple client caches in the presence of database updates is an important issue not addressed in this work.

A recent paper [13] on data integration defines notions of consistency for virtual, materialized, and hybrid views of database relations. These concepts are also applicable in the context of SQL*Cache; however, there are significant differences in the frameworks of the two systems, since [13] only considers read-only queries interleaved with update propagation; in contrast, SQL*Cache supports read-write transactions at the client site. Another major difference is that we analyze SQL*Cache consistency in terms of the well-established 0/1/2/3 isolation levels, by dividing the notion of consistency into two parts: 'cache consistency' and 'transaction consistency'. Cache consistency defines certain basic properties that the cache must satisfy in order for transaction execution to be 'meaningful'; transaction consistency on the other hand may be selected by the client transaction according to its requirements for isolation, and be enforced by the caching system.

Among other related work, algorithms for query containment [20] and query evaluation using materialized views [16] are adopted for our SQL*Cache containment reasoning process. Efficient maintenance of materialized views has been the focus of much research [12], and is related to the cache currency issues examined in this paper. Client-side data caching has been investigated in several recent studies [7, 22, 24]; however, the effect on transaction consistency of local execution of associative queries has not been examined in these works.

Many similarities exist between SQL*Cache and replicated or distributed database systems [4, 8, 23]. However, the SQL*Cache scheme with its centralized server model is in many ways different than multiple databases operating in a distributed environment. Firstly, caching is performed dynamically in SQL*Cache based on submitted queries. There is no 'global' schema or a replication scheme defined a priori. The various client caches operate autonomously, and coordination among the clients occurs only through the central server. Secondly, unlike a replicated or distributed database system, SQL*Cache does not provide full-fledged database services at the client site. Among other things, this implies that the client-side caches are not involved in 2-phase commit protocols with the server, and do not support local crash recovery. Lastly, we have the important difference that coupling of the local caches with the database server is 'tight', and the final commit must take place at the server. Therefore, the central database is the 'single point of truth' as far as durability of data is concerned (the 'D' in the transactional ACID property [10]).

Despite the above differences, a major similarity between the SQL*Cache scheme and distributed database systems is that queries may be executed at multiple sites, which are a client site and the server site for SQL*Cache. This mixture of query execution sites complicates the issue of transaction concurrency control. Below we analyze concurrency control questions using the 0/1/2/3 degrees of isolation framework, as proposed in [11] and recently formalized in [2]. Based on our analysis, we introduce extensions to these degrees of isolation to take into account the 'lag' of the cache with respect to the database. We first introduce some terminology regarding the consistency and 'freshness' of a cache.

4 SQL*Cache Consistency and Currency

Consistency and currency are important issues that must be addressed for any caching system. Consistency of a cache normally implies a correspondence of the cache contents with the source of the cached data. Transaction consistency is a related but separate issue that is central for SQL*Cache, since such consistency (represented by the 'C' in the transactional 'ACID' property [10]) is an essential concern for database applications. Existence of a SQL*Cache on the client-side undoubtedly affects the data read or written by a transaction, and therefore its isolation and consistency. This section introduces terminology that will be used later to analyze precisely these effects.

4.1 Consistency of a Cache with Respect to a Database

The cache may be perceived as a transient and 'past' replica of the database — updates that have been committed at the central server will generally not appear in the cache until a certain time interval (depending on network delays and on the update propagation scheme) elapses. We define some terms below to formalize this notion.

Let C represent a client-side cache that stores data fetched from the server, and T be a transaction currently running at the client. Let t denote the current time as measured at the server.

Definition 1. State of a Cache with Respect to a Database

Let D_t be the set of committed tuples present in a database D at time t. Then, D_t denotes the *state* of the database D at time t. Let D_t^C be the subset of D_t that corresponds to the set of query predicates stored in a client cache C. D_t^C is called the **coherent state** of the cache C at time t with respect to the database D.

Let C_t be the set of committed tuples in the cache at the time instant t; C_t is called the **actual state** of the cache at time t. Note that we do not consider uncommitted updates, if any, of transactions in progress at the server or at the client. □

Definition 2. SQL*Cache Consistency

Updates committed at the database by time t may not all be reflected in a SQL*Cache until some time $t + \delta t$, due to varying amounts of physical delay δt in update propagation. The actual state C_t of a cache C at time t is therefore different from its coherent state D_t^C. For cache behavior to be 'meaningful', we impose the following **cache consistency** restrictions (P1) and (P2):

- **P1.** C_t must equal the coherent state of the client cache at some previous instant of time $t - \delta t$. That is,

$$C_t = D_{t-\delta t}^C, \tag{1}$$

  where δt is a positive and variable quantity arising from delays in update propagation. Additionally,

- **P2.** The state of a client cache can only *move forwards in time in the same order* that the updates are committed at and propagated from the server. That is, successive actual states of a cache must correspond to its successive coherent states, albeit with some finite amount of time lag. Hence, any two actual states C_{t1} and C_{t2} of the cache must be related to their corresponding coherent states as follows:

$$(C_{t1} = D_{t3}^C) \ AND \ (C_{t2} = D_{t4}^C) \ AND \ (t1 < t2) \longrightarrow \ (t3 < t4). \tag{2}$$

□

Although the above restrictions P1 and P2 on the actual state C_t of a cache may seem fairly straightforward, they have some concrete implications for the update propagation scheme:

- Only committed updates may be propagated to the client-side caches.
- Incremental update propagation on the cache occurs in the same order as the committing of the updates to the database. Therefore, the notifier at the server site, the network, as well as the client must all preserve the order of updates.
- Since the database provides the atomicity (all-or-nothing) property of transactions, the state of the cache must also be 'transaction-consistent' at any given instant. Therefore, all updates committed by a single transaction that are possibly relevant for a cache C must be sent across the network 'batched' by transaction ID, and must be applied to the cache C such that all of them become visible to a client transaction at the same instant (effectively), or that none of them do. That is, maintenance operations to propagate updates must obey transactional semantics at the client site, and must be appropriately grouped into *maintenance transactions.*
- Maintaining the cache may require changes not only to the cached data, but also to the cache description itself [15]. Each relevant predicate in the client cache description must be examined to determine whether it is possibly invalidated by the set of updates being processed, as may happen for cached join predicates.[1,2] All cache maintenance operations, such as obtaining auxiliary information from the server to refresh cached results or altering the cache description, that result from the propagation of updates of a single transaction must be performed as part of the maintenance transaction itself.

Note that we have not yet specified *when* the updates are propagated to the cache. Should they be applied as a client transaction executes? What are the *legal* sequences of interleaving the maintenance transactions and read/write operations of a client transaction ? As we shall see in subsequent sections, answers to these questions are closely related to the isolation levels of client transactions, to the concurrency model employed by the server, and whether pessimistic or optimistic algorithms are followed for concurrency control.

4.2 Currency of a Cache with Respect to a Database

Let u_t^C be a set of updates at time t that have been committed at the server since the last refresh of a cache C. Let U_t^C be the subset of u_t^C that would actually affect the query results cached in C (i.e., those that fall under the scope of some query predicate cached by C). To formally define the notion of cache currency, we assume that a unique timestamp is associated with each update. This timestamp is the commit time of the transaction which executed the update, and is assigned by the server upon successful transaction commit.

[1]Detection of such an invalidation may not necessarily cause a predicate (and its associated tuples) to be purged from the cache; instead, the client may choose to refresh the data for a frequently used predicate by querying the server for auxiliary information.

[2]Detecting whether a tuple affects a cached predicate is similar to the concept of 'precision locks' [14], which were proposed as a more efficient variant of predicate locks [6].

Definition 3. Lag of a SQL*Cache

The *lag* of a cache C at time instant t, denoted by Δ_t^C, is a measure of the *freshness* of the cached data with respect to the database, and is defined to be equal to the following amount (in units of time)

$$\Delta_t^C = t - t_{min},$$

where t_{min} is the minimum of the timestamps of all updates in the set U_t^C. □

A value of 0 at time instant t for the lag of a SQL*Cache denotes that the cache contents at that instant are exactly the same as the associated subset of the database at time t, or equivalently, that the actual and coherent states of the cache are equal at instant t. The first update (in timestamp order) in the set U_t^C causes the lag Δ_t^C to become non-zero. This definition of lag is related to the notion of *Guaranteed Freshness* in [13], in that both are measures of currency of data maintained outside the database.

Observe that even if some updates committed at the database are missing from the cache, it does not necessarily affect a transaction T currently running at the client; this is because the set of updated rows in U_t^C may be disjoint from the read set (and from the query predicates, if phantoms are considered) of the transaction T. This observation leads to the following definition.

Definition 4. Lag of a SQL*Cache with Respect to a Transaction

Let the start time of the client transaction T be T_s and the end time be T_e, and let readSet(T) denote the set of data items (or predicates, if phantoms arising from predicate-base reads are considered) read by T. For any item d belonging to readSet(T) and locally cached by C, let $last(d)$ denote the commit timestamp of the last maintenance transaction that updated d in the cache before T locally read d. Now assume that d is updated again at the server at time $next(d)$, where $next(d) \leq T_e$, upon which the value of item d as read by T is out-of-date by an amount $(next(d) - last(d))$.

Then, the lag Δ_C^T of a SQL*cache C with respect to a client transaction T, denoted by Δ_C^T, is

$$\Delta_C^T = \forall d \in (readset(T))\ (maximum(next(d) - last(d))).$$

□

As defined above, the lag Δ_C^T of a SQL*cache C with respect to a client transaction T is a measure of the maximum divergence over the duration of the transaction of the actual states of the cache from its coherent states, considering only those updates at the server that affect local reads made by transaction T. This notion of lag will be utilized later in Section 6 to define extended isolation levels in the presence of data caching.

5 Isolation of Transactions Using SQL*Cache

This section presents a qualitative analysis of the various issues in supporting a hierarchy of isolation levels [10] for client transactions that use a SQL*Cache. First, we enumerate our assumptions about client transactions, and define the abstract *units* of execution at the client and the server. We will develop algorithms using different sequences of these execution units to achieve different isolation levels. Theorem 1 states an important result prohibiting the caching of query results produced by transactions of degrees 0, 1, or 2, and leads to the definition of *view-consistency.* Intermediate levels of isolation between degrees 2 and 3, such as 2^o view-consistency, are then considered; such levels of isolation are common in commercial databases because they offer higher concurrency and better performance compared to purely serializable systems. Finally, we describe SQL*Cache algorithms to achieve 3^o isolation for lock-based 3^o servers.

5.1 Execution of Client Transactions

Listed below are the assumptions we make about client transactions and update propagation in the SQL*Cache scheme described in this paper. In many cases, a restriction could be relaxed at the expense of additional system complexity.

- **A1.** At most one read-write client transaction T may execute at a client site at any given time instant. In this paper, we do not consider the case of multiple client transactions utilizing the same SQL*Cache simultaneously.[3]

- **A2.** Upon submission of an application program, a local transaction is started at a client site, and a remote transaction is also initiated at the server database. The remote transaction lasts until the client transaction terminates (commits or aborts), and executes the commands submitted for remote execution. The commit of the transaction at the client results in a commit request for the remote transaction at the server. As described later, certain *optimistic* concurrency control algorithms may require the server to verify with the client that all relevant notifications have been processed before a commit is declared successful. Depending on the outcome of the remote commit, the local transaction at the client either commits or aborts.

- **A3.** The abstract *units* of transaction execution at the client and at the server are assumed to be the ones listed in Table I. Note that they utilize timestamp information generated by the server; however, only the time at the server is of importance — no global time synchronization is required amongst the clients, or between a client and the server. Also, the read and write locks are assumed to be 'long duration' locks, in that they are held until the associated transaction terminates (commits or aborts).

- **A4.** In this paper, we assume that updates are handled as follows. Write requests submitted by client transactions are first routed to the server, which sets the appropriate write locks

[3] Once the consistency issues in supporting a single client transaction per SQL*Cache at a time are analyzed and well-understood, we plan to extend it to multiple concurrent transactions at the client. This extension would require some local concurrency control mechanism at the client site in addition to the scheme in place at the server, and is the subject of future work.

Table I: Abstract Units of Execution at the Cache and at the Database

Execution Unit	Result	Meaning
$contains(C, Q)$	Yes/No	Whether the description for cache C contains query Q.
$do\_cache(C, Q)$	Yes/No	Whether the manager for cache C wants to cache the result of query; depends upon several factors such as space, estimated benefit of caching Q etc.
$load(C, Q)$	Load cache	Cache C stores the result of query Q and updates cache description accordingly.
$refresh(C, t)$	Refresh cache	Cache C propagates via maintenance transactions updates from its notification queue upto and including the updates of the transaction with commit timestamp t.
$flush(C, D)$	Flush cache	Sends any local updates in cache C to database D.
$purge(C, U)$	Purge predicates	Purges from the description of cache C the predicates describing the sets of tuples both *before* and *after* the update U.
$read(C, Q)$	Local read	Evaluates query Q locally on the cache C.
$write(C, U)$	Local update	Executes update U locally on the cache C.
$read\_lock(C, Q)$	Local read lock	Sets local long duration read lock on the predicate for query Q.
$abort(C, T)$	Abort transaction	Aborts the transaction T at cache C and undo its effects.
$commit(C, T)$	Commit transaction	Commits the transaction T at cache C and make its effects visible externally.
$read(D, Q, t)$	Remote read	Executes query Q on the remote database D, and returns a timestamp t which is the commit timestamp of the last update notification sent to the client at the 'effective' time of execution of the remote read operation. This effective time varies depending on the concurrency model of the server.
$write(D, Q, t)$	Remote write	Executes query Q on the remote database D and returns a timestamp t which is the commit timestamp of the last update notification sent to the client at the 'effective' time of execution of the remote update operation.
$read\_lock(D, Q, t)$	Remote read lock	Sets long duration read locks for query Q at the database D (without actually evaluating the query), and returns a timestamp t which is the commit timestamp of the last update notification sent to the client at the time the read lock operation completed.
$write\_lock(D, Q, t)$	Remote write lock	Sets long duration write locks for update U at the database D, and returns a timestamp t which is the commit timestamp of the last update notification sent to the client at the time the write lock operation completed.
$abort(D, T)$	Abort transaction	Aborts transaction T at database D and undo its effects.
$commit(D, T)$	Commit transaction	Commits transaction T at database D and make its effects visible externally.

and returns to the client the set of tuples to be updated.[4] Before execution is resumed at the client, the cache may need to be refreshed appropriately. The new set of tuples are cached and then updated locally, possibly causing predicates in the cache description (as perceived by the transaction) to also be updated. The locally modified tuples are flushed to the database 'piggy-backed' along with the request for the next remote operation.

- **A5.** We adopt the semantics for standard SQL [1], which requires that uncommitted updates made by an 'in-flight' transaction be visible to itself as it executes.

  By assumption (A4), any local updates are flushed to the remote server before a remote operation. If the the database follows standard SQL semantics, the uncommitted changes of the transaction will be visible to itself for all remote operations. These uncommitted updates are also visible to the transaction running at the client, but do not affect the committed state of the cache.

5.2 0^o, 1^o, and 2^o Isolation Levels

With 0^o isolation, a transaction is not allowed to update a data item while it is being updated by a different transaction; that is, 'dirty' writes are prohibited [10]. For an isolation level of 1^o, also known as Read Uncommitted, a transaction is not allowed to overwrite a data item until a transaction that is in the process of modifying the same item terminates (commits or aborts). Formal definitions of these consistency levels appear in [2]. Transactions of 1^o or less are normally restricted to be read-only, as chaotic behavior may result otherwise [10]. Also, many commercial databases [17] do not support consistency levels of 0^o or 1^o, in that only committed data is exposed to other transactions.

The rule for 2^o (also known as Read Committed) isolation specifies that a transaction has 1^o isolation, and additionally that it is allowed to read committed data only (except for its own updates). This definition of 2^o isolation is equivalent to the notion of *strictness* defined in [5].

By assumption (A4), write locks are always obtained at the server before performing any local updates. Therefore, dirty writes are precluded in the SQL*Cache scheme. Only committed updates are propagated to a SQL*Cache, and by assumption (A1), the cache supports a single client transaction at a time. The question of reading uncommitted data from other concurrently executing transactions therefore does not arise for any local reads. 2^o isolation is obtained by default for local reads, and hence the isolation level of a 0^o, 1^o, or 2^o transaction is determined by the isolation level of its remote operations.

There is however an important issue that still remains to be examined — for remote operations, even a 2^o isolation level does not guarantee a 'consistent' snapshot of the database in that a query can see data that was committed after the query started and even as it executes. That is, a single query involving a join could possibly produce a result that is inconsistent with any actual committed state of the database. If such a query result is cached upon return to the client, the

[4]This update handling scheme is similar to the *SELECT-FOR-UPDATE* command in SQL database systems.

cache consistency property (P2) defined above will no longer be valid. This observation leads to the following important result:

Theorem 1. *Query results produced by read-only or read-write transactions of isolation levels 0, 1, and 2 cannot be cached for inter-transaction reuse without the possibility of violating the cache consistency property P2 that the actual state of the cache must always correspond to a past coherent state.*

Proof. To see why the above statement is true, let us assume the contrary. We will give a counter-example with a 2^o transaction (which by definition is also 1^o and 0^o).

Let the cache initially be empty. Suppose the transaction now submits a query Q: $DEPT \bowtie PROJECTS$ which gets submitted to the server. Assume that two update transactions U_1 and U_2 are operating concurrently at the server site, inserting new *PROJECT* tuples corresponding to departments 1 and 2 respectively.

2^o isolation does not isolate Q from seeing committed updates even *as it executes*. Let us assume that the U_1 commits its changes to the *PROJECTS* table at time t_1, *after* Q has read the relevant portion (associated to department 1) of the table. Let U_2 commit its changes at time t_2, *before* query Q reaches the affected data (i.e., for department 2) during the evaluation of the join. In this case, the result for Q may have the updates for U_2 but not those of U_1 even though U_1 committed earlier than U_2. Therefore, the result for Q does not correspond to any actual committed state in the database, and if cached, would violate the cache consistency property P2. □

Note that 0^o, 1^o, and 2^o transactions may use a previously warmed-up SQL*Cache to perform local reads on cache hits. The only restriction is that results of remote queries executed by transactions of these isolation levels cannot be cached locally without violating SQL*Cache consistency. Maintenance transactions for propagation of updates to the cache can be arbitrarily interleaved with the execution of 0^o, 1^o, and 2^o client transactions; if the maintenance transactions behave in accordance with the cache consistency criteria stated in section 4.3, such incremental refresh of the cache will not violate the consistency rules for 2^o and lower isolation levels.

5.3 Consistency Levels Between Degrees 2 and 3

Although degree 3 serializability is a desirable property of transactions, it reduces the amount of available concurrency and hence the throughput of a database system. In practice, reduced levels of isolation ranging between degrees 2 and 3 are quite popular for commercial applications [3]. We now show that for such intermediate isolation levels, a necessary criterion for caching a query result is that of *view-consistency* of the query.

By Theorem 1, there is an inherent risk of creating an inconsistent cache if the results of remote queries executed at 2^o or lower consistency levels are stored. Subsequent update propagation to an inconsistent may cause further inconsistencies, and finally result in chaos. In order for a query result to be 'meaningful' for caching purposes and for inter-transaction reuse, the query must operate on a *transaction-set consistent* [17] view of the database during its evaluation.

Definition 5. 2^o **View-Consistency**

Let V be the set of transactions that have committed at the server by time t_V, and let the start and end times of a query be q_s and q_e respectively. A query is said to be *view-consistent with respect to time* t_V if it perceives data written by the *same* set V of committed transactions over the entire period $[q_s, q_e]$ of its execution. A transaction is said to be 2^o *view-consistent* if has 2^o isolation and all of its queries are individually view-consistent. □

2^o view-consistency is 'stronger' than 2^o, since it prevents the Read Skew anomaly defined in [3]. Thus, if each transaction preserves the database constraints, and the database had no constraint violations to begin with, then a view-consistent query will always see a consistent database state. However, repeatable reads are not provided at the transaction level, and the *lost update* phenomenon [10] also cannot be ruled out for 2^o view-consistent transactions; therefore, their serializability is not guaranteed. With respect to caching, the following statement holds:

Theorem 2. *A SQL*Cache can support* 2^o *view-consistent isolation level for the local operations of a client transaction, as long as all previously cached queries and all remote operations of the transaction are executed with* 2^o *view-consistent or higher level of isolation at the server.*

Proof. The execution sequence (in terms of the abstract units defined in Table I) for supporting 2^o view-consistent transactions on a SQL*Cache appears below:

Algorithm 1. 2^o **View-Consistent Isolation Using SQL*Cache.**

```
for each command E submitted by a transaction T
do {  if E is a query Q
      then {  if contains(C, Q)                  /* cache hit */
              then  {  read(C, Q); }             /* local read */
      else {                                     /* cache miss */
                flush(C, D);                     /* flush local updates */
                read(D, Q, t_Q);                 /* Q is view-consistent at time t_Q */

                if do_cache(C, Q)
                then {  refresh(C, t_Q);         /* sync cache */
                        load(C, Q);  }           /* store result */
           }
      }
      else if E is an update U
      then {
              flush(C, D);               /* flush local updates */
              write_lock(D, U, t_U);     /* remote write lock at t_U */
              read(D, U, t_U);           /* read tuples to be updated */
              refresh(C, t_U);           /* sync cache */
              load (C, U);               /* store new tuples */
              write(C, U);               /* update tuples locally */
              purge(C, U);               /* purge updated predicates */
```

```
    }
    else if E is an abort
    then {  abort(D, T);
            abort(C, T); }
    else if E is a commit
    then {  commit(D, T, T_e);        /* commit at database */
            refresh(C, T_e);          /* refresh cache */
            commit(C, T); }           /* commit at client */
}
```

Correctness of the above theorem can be deduced from the following argument: (1) In the above algorithm, local caching of 2^o view-consistent query results preserves the cache consistency property (P2) by first making the actual state of the cache equal to its appropriate coherent state, (2) Maintenance transactions propagate committed updates only, thereby providing at least 2^o isolation for local operations, and (3) The additional requirement of view-consistency for local reads is also satisfied, since the cache is not refreshed while evaluating a local query. □

Notice that we have not yet examined the the effects of cache currency on transaction consistency. Local reads on the cache perceive a past state of the database, whereas remote operations may see a more current one. For isolations lower than 3^o, alternating between the present and the past states of the database does not violate the consistency requirements as stated in [11]; in particular, the repeatable read property does not apply. Non-serializable transactions could therefore be said to be unaffected by stale data reads, whereas 3^o transactions are stricter in this respect (since all reads must be repeatable irrespective of whether they are local or remote). We must keep in mind however that the framework of the 0/1/2/3 isolation hierarchy was originally formulated in the context of single-site transactions. For some non-serializable transactions that use local caching, it may still be desirable to have the capability to specify an acceptable level of cache currency as well as the acceptability of alternating between past and present database states within the scope of a single transaction. Later in this paper we introduce extensions to the isolation hierarchy for handling these specific requirements.

Intermediate levels of isolation between 2^o and 3^o are very common in commercial database systems, because of performance benefits over pure 3^o systems. [5] Examples of such isolation levels are *Snapshot* isolation in [1] and *Read Consistency* isolation in Oracle [17]. Both of these schemes involve some form of multi-version concurrency control [5], and both are at least 2^o view-consistent in terms of the anomalies they prevent [3]. Such multi-version implementations of concurrency control are generally based on query or transaction start-timestamps [5], and therefore integrate well with client-side caching schemes like SQL*Cache.

[5] In fact, the SQL standard [1] defines each SQL statement as atomic, effectively creating a serializable subtransaction at the start of each statement.

5.4 3^o Isolation (Serializable)

Degree 3 isolation requires that the transaction have 2^o isolation, and additionally that all its reads are *repeatable* during the period of its execution [10].

Support for repeatable reads at the level of individual data items provides level of isolation that is slightly lower than 3^o, and is sometimes referred to as 2.99^o [10]. To be completely serializable, the repeatable read property must also be satisfied by the *sets* of data items returned by the predicate-based commands of a transaction, as in the case of $SELECT - FROM - WHERE$ SQL queries. This property is also known as *phantom protection* [10], where the appearance or disappearance of *phantom* records from any predicate-based read is prevented over the duration of the transaction. Essentially, a 3^o transaction has 'complete' isolation from the activities of other concurrently executing transactions.

Remote queries executing with 3^o isolation perceive a 'constant' database state and obviously possess the view-consistent property discussed above. Results of these queries can be cached and reused, but some timing issues must still be addressed in answering the following important question: What should be the mechanism to enforce serializability of client transactions in the presence of a SQL*Cache?

The answer depends on many factors, including the concurrency control model of the server database and the requirements of client transactions. The general strategy we use to achieve 3^o isolation for a client transaction T is to make the actual state of the cache the same as its coherent state with respect to the database before allowing any local operation on the cache. T will therefore perceive the same database state for remote reads and writes as for local ones. This goal can indeed be achieved with the minor extensions to server functionality as listed in Table I.

First, we define some terminology for the levels of *optimism* in concurrency control. A mechanism for concurrency control is called:

- *Pessimistic*, if it *prevents* the violation of the specified isolation constraints; lock-based concurrency control systems are an example of this approach.
- *Optimistic*, if it *detects* the violation of the specified constraints, at some time (possibly upon commit) after the constraints have been violated. Multi-version or timestamp-based concurrency control are examples of optimistic algorithms.

Now assume that the server is lock-based, and that it provides at least 3^o isolation for operations on the database. Enforcement of repeatable reads at the predicate level requires some variant of *predicate locking*, such as *granular* or *key-range* locks [10].

Below we outline two concurrency control schemes, one pessimistic and the other optimistic, that ensure serializability of transactions using a SQL*Cache with a lock-based server. The main difference between the two schemes is that for pessimistic behavior, SQL*Cache obtains read locks from the server even when there is a cache hit, while the optimistic scheme does not; subsequent detection of violation of serializability can cause transactions to abort in the optimistic scheme.

Algorithm 2. Lock-based 3^o Server, Pessimistic SQL*Cache.

```
for each command E submitted by a transaction T
do {  if E is a query Q
      then { if contains(C, Q)                /* cache hit */
             then {
                     read_lock(D, Q, t_Q);    /* Q is lock-protected at time t_Q */
                     refresh(C, t_Q);         /* sync cache */
                     read(C, Q);   }          /* local read */
             else {                           /* cache miss */
                     flush(C, D);             /* flush local updates */
                     read_lock(D, Q, t'_Q);   /* Q is lock-protected at time t'_Q */
                     read(D, Q, t'_Q);        /* remote read */
                     if do_cache(C, Q)        /* will result be cached ? */
                     then {  refresh(C, t'_Q); /* sync cache */
                             load(C, Q);  }   /* store result */
             }
       }
       else if E is an update U
       then {
               write_lock(D, U, t_U);    /* remote write lock at t_U */
               read(D, U, t_U);          /* read tuples to be updated */
               refresh(C, t_U);          /* sync cache */
               load (C, U);              /* store new tuples */
               write(C, U);              /* update tuples locally */
               purge(C, U);              /* purge updated predicates */
       }
       else if E is an abort
       then {  abort(D, T);
               abort(C, T); }
       else if E is a commit
       then {  commit(D, T, T_e);       /* commit at database */
               refresh(C, T_e);          /* refresh cache */
               commit(C, T); }           /* commit at client */
}
```

The algorithm given in Figure 3, although pessimistic and somewhat inefficient in terms of communication requirements with the server, ensures serializability. To see why, observe that before a local read is performed, the actual state of the cache is made the same as its coherent state at the time read locks are obtained at the server.[6] Likewise, the cache and the central database

[6]One point to note in the above algorithm is that for all cache refresh steps of the form $refresh(C, t)$, the client can actually propagate pending updates, if any, with commit timestamps greater than t, since updates committed to the database after time t cannot conflict with the lock-protected read and write operations of the current 3^o transaction. Such refresh 'in advance' is not necessary but is also not incorrect.

are 'in sync' when a set of tuples are cached locally. Therefore, for cache hits, the same set of data items are produced locally as the answer to a submitted query as would be produced at the server. By always locking query predicates at the server, the cache avoids any violation of the repeatable read rule, even for cache hits. □

An optimistic version of the above algorithm may be constructed along similar lines. Due to space constraints, we only sketch the outline of the algorithm below. Cache misses are handled identically; however, cache hits cause local reads to be performed after setting local read locks, without acquiring any locks from the server at that point. These local read locks are sent over to the server on the next remote operation, 'piggy-backed' with the remote request. The server first obtains the *deferred* read locks, if any, accompanying a remote request, and then executes the operation to return the result. Serializability restrictions may be violated due to the delayed read locking at the server. If a conflict of committed updates with local read locks is detected during cache refresh, then the transaction must be aborted.

The commit of the transaction is processed in a manner similar to Figure 3, with an additional step before successful completion of commit at the server; this step involves a quick 'handshake' with the client to verify that all update notifications relevant for the transaction have been processed by the client.

Following a reasoning similar to the pessimistic case, we can prove that the optimistic algorithm outlined above also ensures serializability. In fact, by extending the above algorithm with local predicate-based write locks, it can be shown that serializable transactions can be supported by SQL*Cache even when the server is not 3^o, but only supports 2^o view-consistent isolation. SQL*Cache algorithms can also be formulated for the case of 3^o multi-version servers that use multi-version timestamp ordering (the MVTO protocol in [5]) to enforce serializability. These algorithms and their correctness proofs are beyond the scope of this paper.

6 Extended Isolation Levels for Data Caching

The four isolation levels discussed above were proposed in the context of centralized environments. Query evaluation at both local and remote sites within the scope of a single client transaction, as done in SQL*Cache, requires extensions to these isolation levels. In particular, what is missing is the concept of exposing to the user the possibility of multi-site transaction execution with associated delays in update propagation and resulting discrepancies in data. Our goal is not to provide 'transparent' concurrency control in which the user is unaware of the presence of a client-side cache. Rather, we wish to allow the application to itself select an appropriate level of concurrency control, being aware of the the local caching of data and its implications on transaction consistency. Although the discussion below is with respect to SQL*Cache, the concepts also apply to replicated or distributed database systems supporting multi-site transaction execution [8].

We concentrate on 2^o and higher isolation levels, since 0^o and 1^o are used infrequently. Analogous extensions may be made to the isolation levels lower than 2^o.

Definition 7. 2^o **Read Forward Isolation**

This isolation level is an extension of the basic 2^o isolation level, requiring additionally that alternating local and remote reads should not cause a client transaction to switch between past and present states of the database. That is, all reads, whether local or remote, should present either a constant or a *move-forward-in-time* picture of the database.

Formally, let t_r be the time at which a remote operation completes at the database, and suppose that t_l is the time of the next local read, if any. For 2^o Read Forward isolation to hold, for each such t_r and t_l within a client transaction we must have that $C_{t_l} = D^C_{t_f}$, where $t_f \geq t_r$. □

Therefore, with Read Forward isolation, the actual state of the cache at the time of a local read must correspond to a coherent state of the cache at or after the last remote operation.

Definition 8. 2^o **Lag** L **Isolation**

This isolation level is an extension of the basic 2^o isolation level. It requires 2^o isolation, and imposes an additional requirement that the lag of the cache with respect to the transaction (as defined in Section 4.2) must remain within the specified limit of L at all times during the execution of the transaction. □

Therefore, any local reads performed by a 2^o Lag L transaction may be out-of-date by a maximum of L time units with respect to the server. Remote reads that happen on a cache miss are executed on the server and therefore have no associated lag.

For non-zero values of L, this isolation level assumes that the application can tolerate the data discrepancies, if any, between local and remote reads. As a special case, consider the isolation level 2^o with lag 0. This level of isolation essentially specifies that each local read return the same result as a corresponding remote read. This behavior may be implemented in a variety of ways, including optimistically allowing potentially stale local reads to occur, but aborting the transaction at a later point in time if the specified lag is found to have been violated.

Note that Lag L and Read Forward are orthogonal concepts — it is possible to have Read Forward isolation with or without an associated lag. In fact, it is useful to combine the above extensions with other consistency levels such as the 2^o View-Consistent isolation defined in Section 5.3, to produce 'stronger' consistency levels than the extensions applied to pure 2^o. The notion of Read Forward does not apply to 3^o isolation, since the serializability condition is stricter. However, this restriction can be a distinguishing feature for different optimistic implementations of 3^o isolation. For example, the 3^o optimistic algorithm sketched in Section 5.4 can be subjected to the Read Forward restriction, whereby any remote read will necessitate a refresh of the cache.

6.1 Implementation of the Extended Isolation Levels

Let us consider the changes required to Algorithm 1 for supporting the isolation level 2^o View-Consistent Read Forward. Only the section of the code that is affected is shown below:

Algorithm 3. 2^o **View-Consistent Read Forward Isolation Using SQL*Cache.**

```
...
if the submitted command is a query Q
then {  if contains(C, Q)                       /* cache hit */
        then {  read(C, Q);  }                  /* local read */
        else {                                  /* cache miss */
                flush(C, D);                    /* flush local updates */
                read(D, Q, t_Q);                /* Q is view-consistent at time t_Q */
                refresh(C, t_Q);                /* sync cache to Read Forward */

                if do_cache(C, Q)
                then {  load(C, Q);  }          /* store result */
        }
} ...
```

Now consider an optimistic implementation of the isolation level 2^o View-Consistent Read Forward Lag L. The following modifications are necessary to Algorithm 3. We assume that an enhanced version of the $refresh(C, t)$ execution unit of Table I is available, which returns an error if during a refresh operation any maintenance transaction updates a tuple covered by a local read lock.

Algorithm 4. 2^o **View-Consistent Read Forward Isolation Using Optimistic SQL*Cache.**

```
lagError = FALSE;
if the submitted command is a query Q
then {  if contains(C, Q)                     /* cache hit */
        then {  read_lock(C, Q);              /* local read lock */
                read(C, Q);  }                /* local read */
        else {                                /* cache miss */
                flush(C, D);                  /* flush local updates */
                read(D, Q, t_Q);              /* Q is view-consistent at time t_Q */
                lagError = refresh(C, t_Q);   /* sync cache to Read Forward */
                if (lagError)                 /* lag condition violated */
                then { abort (D, T);          /* abort transaction */
                       abort (C, T);
                       exit;  }
                if do_cache(C, Q)
                then {  load(C, Q);  }        /* store result */
        }
} ...
```

7 Conclusions

The SQL*Cache caching scheme supports query containment reasoning and local query execution on cached data, which are essentially views computed dynamically in the process of transaction execution. In this paper, we have focused on consistency guarantees which must be provided to client transactions that read or update locally cached data. We have analyzed the transaction isolation and concurrency control issues for SQL*Cache. Using the well-known 0/1/2/3 framework for isolation levels, we have demonstrated that a variety of isolation levels can be effectively supported by SQL*Cache. We have also identified the class of isolation levels for which dynamic caching of query results violates the cache consistency criteria. Additionally, we have defined extensions to the original hierarchy of isolation levels; these extensions are appropriate for client-server environments with data caching at the client site, and can be supported by SQL*Cache.

Two prototype implementations of SQL*Cache are under way, one using Sybase and the other using Oracle as the backend servers. In the Sybase implementation, the Open Server mechanism is employed for update propagation and cache consistency maintenance, while the Oracle-based implementation uses snapshot logs (with underlying triggers). In both cases, the client-side data store and SQL engine of SQL*Cache will be implemented using the HP/SmallBase main memory database. Layered on top of the SmallBase SQL interface is a cache containment reasoning system and an update notification handler. Details of our implementation experience will be presented in a forthcoming paper.

Acknowledgements

The work of Julie Basu was supported in part by Oracle Corporation. We gratefully acknowledge the useful discussions with Kurt Shoens of Hewlett-Packard Laboratories, Dallan Quass of Stanford University, and Bob Jenkins of Oracle Corporation.

References

[1] ANSI X3.135-1992, "American National Standard for Information Systems — Database Language — SQL," November 1992.

[2] V. Atluri, E. Bertino, and S. Jajodia, "A Theoretical Formulation of Degrees of Isolation," Technical report, George Mason University, VA, 1995.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," Proceedings of SIGMOD, May 1995, San Jose, CA, pgs 1-10.

[4] Y. Breibart, H. Garcia-Molina, and A. Silberschatz, "Overview of Multidatabase Transaction Management," The VLDB Journal, Vol. 1, No. 2, Oct. 1992, pp. 181-239.

[5] P.A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, Massachusetts, 1987.

[6] K.P. Eswaran, J.N. Gray. R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, 1976, pp. 624-633.

[7] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems," *Ph.D. Thesis*, Technical Report No. 1168, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[8] R. Gallersdorfer and M. Nicola, "Improving Performance in Replicated Databases through Relaxed Coherency," *Proceedings of the Twenty First International Conference on Very Large Data Bases*, Zurich, Switzerland, Sept. 1995, pgs. 445-456.

[9] J. Gray, R. Lorie, I. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM Vol. 19, No. 11, pp. 624-633, Nov. 1978.

[10] J. Gray and A. Reuter, "Isolation Concepts," in *Transaction Processing: Concepts and Techniques*, San Mateo, CA, Morgan Kaufmann Publishers, 1993, pp. 403-406.

[11] J. Gray, R. Lorie, G. Putzolu, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database," in *Readings in Database Systems*, Second Edition, Chapter 3, Michael Stonebraker, Ed., Morgan Kaufmann 1994 (paper was originally published in 1977).

[12] A. Gupta and I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995, pp. 3-18.

[13] R. Hull and G. Zhou, "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches,", Proceedings of SIGMOD, June 1996, Montreal, Canada, pgs. 481-492.

[14] J.R. Jordan, J. Banerjee, and R.B. Batman, "Precision Locks," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Ann Arbor, MI, April 1981, pp. 143-147.

[15] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *The VLDB Journal*, Jan 1996.

[16] P.-A. Larson and H.Z. Yang, "Computing Queries from Derived Relations: Theoretical Foundation," Research report CS-87-35, Computer Science Department, University of Waterloo, 1987.

[17] Oracle Corporation, "Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7," White Paper, Part No. A33745, July 1995.

[18] N. Roussopoulos, "The Incremental Access Method of View Cache: Concepts, Algorithms, and Cost Analysis," *ACM Transactions on Database Systems*, Vol. 16, No. 3, 1991, pp. 535-563.

[19] N. Roussopoulos, C.M. Chen, S. Kelly, "The ADMS Project: Views R Us," *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995, pp. 19-28.

[20] Y. Sagiv and M. Yannakakis, "Equivalences Among Relational Expressions with the Union and Difference Operators," *Journal of the ACM*, Vol. 27, No. 4, 1980, pp. 633-655.

[21] A.P. Sheth and A.B. O'Hare, "The Architecture of *BrAID*: A System for Bridging AI/DB Systems," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991.

[22] Y. Wang and L.A. Rowe, "Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991.

[23] G. Wiederhold and X. Qian, "Consistency Control of Replicated Data in Federated Databases," *Proceedings of the IEEE Workshop on Management of Replicated Data*, Houston, Texas, Nov. 1990, pgs. 130-132.

[24] K. Wilkinson and M.-A. Neimat, "Maintaining Consistency of Client-Cached Data," *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.