A Predicate-based Caching Scheme for Client-Server Database Architectures

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

Julie Basu
Stanford University
and
Oracle Corporation

Abstract

We propose a new client-side data caching scheme for relational databases with a central server and multiple clients. Data is loaded into a client cache based on queries, which are used to form predicates describing the cache contents. A subsequent query at the client may be satisfied in its local cache if we can determine that the query result is entirely contained in the cache. This issue is called 'cache completeness'. On the other hand, 'cache currency' deals with the effect of updates at the central database on the client caches. We examine various performance and optimization issues involved in addressing the questions of cache currency and completeness using predicate descriptions. Expected benefits of our approach over commonly used object ID-based caching include lower query response times, reduced message traffic, higher server throughput, and better scalability.

1 Introduction

This paper addresses the issue of data caching in client-server relational databases with a central server and multiple clients that are individually connected to the server by a local area network. The database is resident at the server and transactions are initiated from client sites, with the server providing facilities for shared data access. Dynamic local caching of query results at client sites can enhance the overall performance of such a system, especially when the operational data spaces of the clients are mostly disjoint.

In typical commercial relational databases with client-server configurations [16], caching aims to avoid disk traffic and is done on the server side only, based on buffering of frequently accessed disk blocks or pages. The assumption is that clients are low-end workstations that are likely to be overloaded by local data processing; their function is accordingly limited to transmission of SQL queries across the network to the server, and presentation of the received results to the user. However, with the continuing precipitous drop in the price-performance ratio for workstations, the validity of this assumption comes into question. It is increasingly common to find clients that are high-end workstations with the server being a mainframe or a minicomputer. Such intelligent clients are capable of performing intensive computations on their own, using the database as a remote resource that is to be accessed only when necessary. Increased local functionality and autonomy can potentially lead to less network traffic, improved utilization of local computing power, faster response times and higher server throughput, as well as better scalability.

Several techniques to provide caching facilities at client sites have been recently investigated [21, 22]. All of these schemes use the notion of object IDs to store, retrieve and maintain cached objects at client sites. Such caching schemes can only support ID-based operations like *ReadObject* and *UpdateObject* within transactions; an associative query that accesses database objects using a predicate on a relation, e.g., through a WHERE clause of a SELECT-FROM-WHERE SQL statement, cannot be handled in these schemes. Similar observations can be made for the page-based caching schemes presented in [4, 5, 9] — none of these schemes can reuse locally cached data for associative query execution. We feel this is a major drawback of these caching schemes, since associative queries are very common in relational databases, and are indeed one of the major reasons for their success.

One possible way of supporting associative access in object-ID or page-based caching is to use indexes defined on the database at the server, as is done in some object-oriented databases. If server indexes exist that can determine the target objects or pages for an associative query, the client can fetch and examine the relevant index pages to see if current versions

*Arthur Keller's e-mail address is `ark@cs.stanford.edu`.

of the desired objects or pages are locally available. However, even in systems that have low to moderate update activity, index pages generally have very high contention, and may be subject to frequent invalidation or update. The need to reference server index pages locally is thus likely to cause increased network traffic and slower response times.

In our approach, queries executed at the server are used to load the client cache, and *predicate descriptions* derived from these queries are stored at both the client and the server to examine and maintain the contents of the cache. If a client determines from its local cache description that a new query is not completely computable locally, then the query (or a part of it) is sent to the server for processing. The result of this query is optionally added to the client cache, whose description is updated appropriately. On the other hand, a locally computable query is executed by the client on its cached data (the effect of such local query evaluation on concurrency control is discussed later). Each cache may also have its own local indexes or access paths to facilitate local query evaluation. To ensure the currency and validity of cached data, predicate descriptions of client cache contents are used by the server to notify each client of committed updates that are possibly relevant for its cache.

Consider, for example, an employee database managed by a central server, in which table EMPLOYEE(*emp_no, name, job, salary, dept_id*) records a unique employee number and other details of each employee. Suppose that a client caches the result of a query for all employees in department 100, along with a predicate description ($dept\_id = 100$) for these tuples. Assuming that no update at the server has affected these EMPLOYEE tuples, a subsequent query at the same client for all managers in department 100, i.e., those employees that satisfy (($dept\_id = 100$) AND (*job*='manager')), can be answered using the cache associatively, and without referencing server index pages or communicating with the server (except, if deemed necessary, for purposes of concurrency control like locking the accessed objects at the server). Another query represented by the predicate (*job*='manager') asking for all managers can only partially be answered from the cache. In this case, the database could be requested either for all managers, or only for those not in department 100. This choice is an important optimization decision that can potentially speed up data transmission and query processing.

The situation is more complex if the cached data is out-of-date as a result of updates committed at the server. There are several choices for maintaining the currency of data cached at a client: automatic refresh by the server as transactions commit updates, invalidation of appropriate cached data and predicates, or refresh upon demand by a subsequent query. Both automatic and by-demand refresh procedures may again either be recomputations or incremental, i.e., performed either by cached query re-execution or by differential maintenance methods. Which method performs best depends on the characteristics of the database system such as the volume and nature of updates, pattern of local queries, and constraints on query response times. In our scheme, the maintenance method adopted is allowed to vary by client, and also for different query results cached at a client. A client may have results of frequently posed queries automatically refreshed, and may choose to invalidate upon update what is perceived as a random query result.

Examination and maintenance of cached tuples via predicate descriptions entails determining satisfiability of predicates, and concerns about overhead and scalability may naturally arise over reasoning with large numbers of predicates in a dynamic and *real-time* caching environment. In this paper, we attempt to address the practical design issues and trade-offs that pertain to this environment, with the conceptual structure as our primary focus.

To reduce the complexity of the reasoning process, we allow approximate algorithms that might sometimes err causing inefficiency, but can never produce incorrect results. A cache description used for determining *cache completeness* (i.e., whether a query can be completely or partially evaluated locally) need not be exact, but it can be *conservative*. In other words, data claimed to be in the client cache must actually be present in it, so that local query evaluation does not produce incomplete results; it is however not an error if an object residing in the cache is re-fetched from the server. Another description of a client's cache is maintained by the server for alerting the client of changes to its cached objects (the *cache currency* issue). This description can also be approximate, but can only be exact or *liberal*. That is, occasionally notifying the client of an irrelevant update is not a problem, but failure to notify the client that a cached object has changed can result in significant error.

Apart from the above approximation techniques, we investigate several optimizations applicable in our context. *Predicate indexing* mechanisms, simplification of cache descriptions through *predicate merging* and *query augmentation* are used to reduce caching costs (details of these techniques are beyond the scope of this paper). The expected net effect is a decrease in query response times and increase in server throughput compared to other systems, and improved scalability with respect to the number of clients.

The paper is organized as follows. Section 2 gives an overview of related work. We outline the details of our scheme in Section 3. Implementation issues and trade-offs are addressed in Section 4. Finally, we summarize our contributions, work in progress, and future plans in Section 5.

2 Related work

Our caching scheme is reminiscent of predicate locks used for concurrency control [8], where a transaction requests a lock on all tuples of a relation that satisfy a given predicate. Predicate lock implementations have not been very successful, mainly because they are pessimistic in nature and because of their execution cost [11]. The pessimism arises because two predicates intersecting in the attribute space but without any tuples in their intersection for the particular database instance will nonetheless prevent two different transactions from simultaneously locking these predicates. This rule protects against phantoms, but reduces the allowable concurrency. Our caching scheme on the other hand can support predicate locks that are more optimistic, in that two transactions at different clients conflict (and are notified by the server of the conflict) only when a tuple in the intersection of shared predicates already exists or is actually updated or inserted.

Query containment [18] is a topic closely related to the cache completeness question. Query evaluation on a set of *derived relations* is examined in [15]. Efficient maintenance of materialized views has also been the subject of much research [2, 6, 10, 14], and is related to the issues examined in this paper. For example, our update notification scheme involves detecting whether an update has an effect on a client cache, and this has much in common with the elimination of updates that are irrelevant for a view [1]. The above techniques of query containment and materialized view maintenance, though directly applicable in our scheme, have mostly been designed for relatively static situations with small numbers of queries or pre-defined views. Performance issues in handling large numbers of dynamic queries and views in a client-server environment have not been addressed in these papers.

Among other related work, [17] proposes a view caching scheme that uses the notions of *extended logical access path* and *incremental access methods*. Results on the simulated performance of this scheme (and some variations) in a client-server environment are in [7]. Update logs are maintained by the server(s), and each query against cached data at a client results in an explicit refresh request to the server(s) to compute and propagate the relevant differential changes from these logs. In contrast, we follow an incremental and flexible notification strategy at the server, and attempt to split the workload of refreshing cached results more evenly amongst the server and the clients.

Rule systems for triggers and active databases [12, 20] are related to our notification scheme, in the sense that given a database update, efficient identification of applicable rules is desired. This requires determining satisfiability of predicates, and efficiency issues similar to ours arise for such systems. One difference is that for caching, notification by the server can afford to be approximate as long as it is liberal. Additionally, our notification scheme has the capability of directly propagating certain update commands to relevant clients for local execution on cached data, instead of propagating the tuples modified by the update. Therefore, unlike rule systems in active databases, we require that the notification system employed by the server handles not only 'point inputs' corresponding to single tuples, but also general query predicates over the attributes of a relation.

3 Our approach

We propose a predicate-based client-side caching scheme that reduces query response times and network traffic between the clients and the server by attempting to locally answer queries from the cached tuples using associated predicate descriptions. The database is assumed to be resident at the central server, with users originating transactions from client sites. Each client executes transactions sequentially, with at most one transaction active at any time (concurrency control at individual clients can be incorporated in our scheme, but is not considered in this paper).

3.1 Class of queries

Queries specify their target set of objects using *query predicates*, as in the WHERE clause of a SELECT-FROM-WHERE SQL query. We allow general SELECT-PROJECT-JOIN queries over one or more relations, with the restriction that the keys of all relations participating in a query must be included in the query result. We feel this is not overly restrictive, since a query posed by the user that does not satisfy this constraint may optionally be *augmented* by a client to retrieve these keys from the server. User-transparent query augmentation (discussed in Section 4) is in many cases a viable technique for reducing long-term costs of maintaining cached query results.

Transactions may also execute insert, delete, and update commands on a single relation. Insert commands that use subqueries to specify inserted tuples are not considered in this paper.

Query predicates specified as above are classified as either *point query* or *range query* predicates. A point query predicate specifies a unique tuple (that may or may not exist) in a single relation, by conjunctively specifying exact values for all attributes that constitute its primary key, and possibly values for other nonkey attributes as well. Point query predicates arise frequently during navigation among tuples of different relations using foreign keys, e.g., a query about a tuple in the relation DEPARTMENT(*dept_id, dept_name, director*) based on the matching value in the foreign key *dept_id* of an EMPLOYEE tuple. A range query predicate may specify either a single value or a value range for one or more attributes, and may in general have zero, one, or more tuples in its target set. Note that a point query is a special case of a range query; we distinguish between the two only because they are processed differently for efficiency reasons. A general SELECT-PROJECT-JOIN query predicate is treated as a collection of range query predicates on the participating relations with some ranges being *variable*, i.e., specified in terms of join attributes of other relations.

3.2 A formal model of predicate-based caching

We now formalize our terminology using the usual predicate calculus notation. Suppose that there are n clients in the client-server system, with C_i being the ith client, $1 \leq i \leq n$. Let Q_i^E be the number of query predicates such that the results of all queries corresponding to these predicates are cached at client C_i, where $Q_i^E \geq 0$. We denote by P_{ij}^E the query predicate corresponding to the jth query result cached at client C_i. The superscript E in Q_i^E and P_{ij}^E stands for *exact*, to represent the fact that these are the precise count and exact forms of cached query predicates respectively. Other information related to a query may be associated to its query predicate, e.g., the list of *visible* attributes retained after a projection operation on the tuples selected by a WHERE clause condition.

Definition 1: An *exact cache description* ECD_i for the ith client C_i is defined to be the set of exact query predicates P_{ij}^E corresponding to all query results cached at the client:

$$ECD_i = \{P_{ij}^E \mid 1 \leq i \leq n,\ 1 \leq j \leq Q_i^E\}.$$

In a real-life scenario, query predicates may be quite complex, and performance problems may arise if precise reasoning methods are followed. To alleviate such problems, we introduce the notion of *approximate* cache descriptions. A client may use a simpler but *conservative* version of the exact cache description for determining cache completeness. As long as data thought to be in the cache is actually present in it, local evaluation of queries will produce correct and complete answers.

Definition 2: A *conservative cache description* CCD_i for the ith client C_i is a collection of zero or more predicates P_{ik}^C such that the union of these predicates is contained in the union of the predicates in the exact cache description ECD_i for the client. Let Q_i^C denote the number of predicates in CCD_i. Formally,

$$CCD_i = \{P_{ik}^C \mid 1 \leq i \leq n,\ 1 \leq k \leq Q_i^C\},$$

where $Q_i^C \leq Q_i^E$ and $\bigcup_{1 \leq k \leq Q_i^C} P_{ik}^C \in CCD_i \Longrightarrow \bigcup_{1 \leq j \leq Q_i^E} P_{ij}^E \in ECD_i$.

The symbol $\Longrightarrow$ denotes the material implication operator, and in the context of query predicates has the following meaning: if $Q \Longrightarrow P$, then the result of the query corresponding to predicate Q is contained in and is computable from the result of the query corresponding to predicate P. Note that the number of predicates Q_i^C in CCD_i is defined to be possibly smaller, and never greater, than Q_i^E; this restriction corresponds to our informal notion that CCD_i is simpler than ECD_i.

One example of conservative approximation of a query predicate is its simplification by discarding a disjunctive condition. For other possible differences between ECD_i and CCD_i, consider some cached EMPLOYEE tuples that were fetched through a number of point queries, as well as through a few range queries. CCD_i may consist only of the range query predicates, whereas ECD_i includes all cached predicates, both point and range. CCD_i is thus simpler than ECD_i, having eliminated point query predicates. The result of this approximation is that cached results of point queries are not taken into consideration when addressing the cache completeness question for range queries, likely speeding up the reasoning process. Thus, if all EMPLOYEE tuples in department 100 have been fetched through separate point queries, a range query on the EMPLOYEE relation with query predicate (*dept_id* = 100) will result in re-fetching these tuples from the server. Such a remote access is inefficient, but not incorrect in any way.

It is important to note that the conservative approximation pertains to the cache description only, and not to the cache contents. In the above example, single EMPLOYEE tuples cached through point queries are still locally available at client C_i. Although these tuples cannot be accessed through CCD_i, they are present in the cache and can be used to answer point queries. A conventional index based on the primary key of the relation EMPLOYEE may be constructed locally at the client to speed up the processing of point queries.

Let us now consider the cache currency issue. The server maintains a consolidated predicate description of all client caches for issuing notifications as transactions commit updates. Since the server handles all n clients, it is crucial to control the complexity of issuing notifications using such descriptions. For this purpose, we propose the use of *liberally approximate* client cache descriptions at the server that *cover* the exact descriptions of the client caches. Such liberal descriptions are expected to be simpler than the exact ones, and must generate all necessary notifications. It is however at most inefficient and not an error if a client is occasionally informed of an update at the database that is irrelevant for its local cache.

Definition 3: A *liberal cache description* LCD_i for the ith client C_i is a set of zero or more predicates P_{ik}^L such that the union of these predicates contains the union of the predicates in the exact cache description ECD_i for the client. Let Q_i^L denote the number of predicates in LCD_i. Formally,

$$LCD_i = \{P_{ik}^L \mid 1 \leq i \leq n,\ 1 \leq k \leq Q_i^L\},$$

where $Q_i^L \leq Q_i^E$ and $\bigcup_{1 \leq j \leq Q_i^E} P_{ij}^E \in ECD_i$ $\Longrightarrow \bigcup_{1 \leq k \leq Q_i^L} P_{ik}^L \in LCD_i$.

Definition 4: The *server cache description SCD* is the collection of liberal cache descriptions for all n clients of the server:

$$SCD = \{LCD_i \mid 1 \leq i \leq n\}.$$

By the above definitions, attributes projected out in a query *must not* be part of a conservative description, but *may* optionally be part of the liberal one. Thus, one example of a redundant notification occurs when a client is notified of a change to a relation attribute that it does not cache. Another case of liberal notification is possible when a tuple that might affect a cached join result is inserted at the central database. The server may be able to eliminate a tuple that is *unconditionally* irrelevant for the join [1] (i.e., irrelevant independent of the database state); however, determining whether the tuple actually affects the join result for the particular database state requires more work. The client may in this case be informed of the inserted tuple, and can subsequently take actions based on local conditions (further details on cached joins are provided in Section 3.4).

Figure 1 shows a pictorial representation of the exact, conservative, and liberal cache descriptions for some cached query predicates for a single relation R with two attributes A1 and A2.

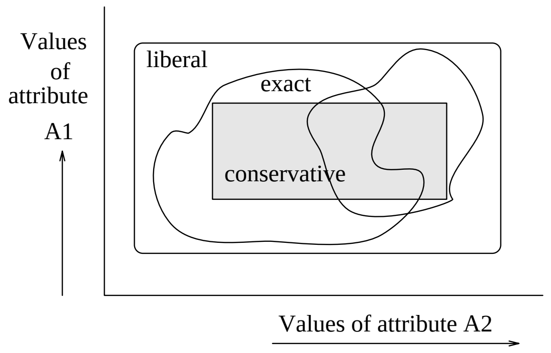

Figure 1. Exact, conservative, and liberal cache descriptions for a relation R(A1, A2)

3.3 Concurrency control

Several different forms of concurrency control can be employed in our caching scheme. For the purposes of this paper, we assume the following protocol. Whenever queried data is locally available, a client assumes that its cache is up-to-date — the transaction operates on the local copy, and a lock is not obtained immediately from the server. On the other hand, if a query not computable locally, it is submitted to the server. A request for remote query execution is accompanied by any local (uncommitted) updates of which the server has not been informed. All objects accessed by a query and the uncommitted updates are locked in the usual two-phase manner at the server during remote fetches. Notification messages from the server may abort an uncommitted transaction whose read/write sets or query predicates conflict with updates committed by other clients. Thus, cache hits and misses are handled differently — for hits, an incremental notification-based *semi-optimistic* scheme is employed, whereas two-phase locking at the server is done for cache misses. This protocol aims to minimize unnecessary aborts of transactions while reducing communication with the server.

A commit at the client sends all remaining updates to the server. In order to prevent synchronization errors due to network delays, the server must ensure that the client has seen its most recent notification message before the commit is confirmed. As discussed in [21], this can be done using a message numbering scheme. All locks held by a transaction are released if the commit is successful, or if the transaction aborts. Note that the server must be aware of predicates cached past a transaction boundary (i.e., past a commit or abort), which in effect act as predicate-based notify locks. If transactions are serializable in the original database, they would remain serializable in this concurrency control scheme (a formal proof is beyond the scope of this paper). Also, if the original database provides protection against phantoms, the same behavior carries over to this scheme. In fact, phantom protection is provided for all locally cached predicates in our scheme through predicate-based notification.

3.4 Effects of database operations

Database operations executed by transactions may affect the contents of the central database, as well as the contents and descriptions of local caches. We now consider different events that may occur in the system, assuming concurrency control as outlined in Section 3.3 above. The discussion below is with respect to the cache at the ith client C_i. The architecture of a sample client-server system supporting predicate-based client-side caching is shown in Figure 2.

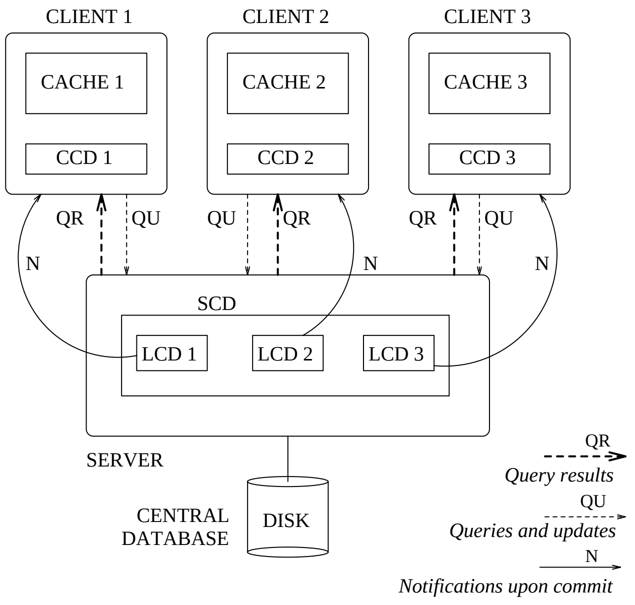

Figure 2. Architecture of a predicate-based client-side caching system

A SELECT-PROJECT-JOIN query with predicate Q is submitted at client C_i: If Q is a point query predicate on a single relation, or can be split up into point queries on several relations, the tuple(s) satisfying Q may be found using local indexes on primary keys of the relation(s) cached at C_i. If the tuples are found, and have all selected attributes visible, we use them. If not, we have to determine whether the tuples exist, i.e., whether Q is contained in the scope of the cache, so we treat it as a range query and handle it as described below.[1]

If Q is a range query predicate, then Q is compared against CCD_i. Three different situations may arise:

- Q is computable from the union of the predicates in CCD_i. In this case, all of the tuples satisfying Q (if any) can be locally found in the cache. There is no effect on either CCD_i or SCD.
- Q is disjoint from the union of the predicates in CCD_i. The tuples in this case must be remotely fetched from the server. The request for remote execution is accompanied by any tuples locally updated by the transaction that have not yet been communicated to the server. The server records the uncommitted updates and locks the updated tuples before executing the query, locking the tuples accessed by Q, and returning the result to C_i. The new tuples are placed in the cache at C_i, with CCD_i being *optionally* augmented. If the tuples are cached past the transaction boundary, SCD *must* be updated before the locks on the tuples are released at the server (upon transaction commit or abort). It is not necessary to update SCD at the time the tuples are fetched; locking tuples at the server provides the usual level of isolation from concurrent transactions.
- Q is partially contained in the union of the predicates in CCD_i. This case are similar to the disjoint one above. One possible optimization is to *trim* the query to eliminate tuples or attributes available locally at the client.

A tuple is inserted at client C_i: The tuple is locked at the server after making an uncommitted insertion at the central database (or alternatively, incorporated into the SCD). It is also inserted locally in the cache with an *uncommitted* tag. If the transaction later commits successfully at the server, the new tuple is unlocked and committed at the central database, incorporated into the SCD, and the *uncommitted* tag is removed at C_i. We have assumed here that the new data is likely to be pertinent for C_i, and thus cached by it past the boundary of the current transaction. The insertion of this tuple may also affect caches of clients other than C_i (this case is discussed below).

One or more tuples are deleted at client C_i using query predicate Q: If all tuples satisfying Q are

[1]Note that it is useful to record that a predicate is cached even when there are no tuples satisfying the predicate, since it can be used to determine locally that the result of a (point or range) query is empty.

not locally available in the cache, the procedure outlined above for a selection query is followed for Q. All or only the missing tuples in Q are fetched from the server after locking them, and locally cached. CCD_i may be optionally augmented. All tuples satisfying Q are then deleted locally in the cache, but marked as *uncommitted.* The server is informed of the deleted tuples upon transaction commit (or earlier, if a remote query is submitted before the commit). The *uncommitted* tag is removed from the cache if commit is successful. If the predicate Q for the tuples is cached past the transaction boundary (although its tuples may have been deleted), the SCD must be updated before the locks on the tuples are released. Retaining predicates in CCD_i whose tuples have been deleted can potentially reduce query response times at the client by allowing local determination of the fact that a subsequent query result is empty.

One or more tuples are updated at client C_i using query predicate Q: This case is similar to the deletion case above, except that the update may move some tuples from one cached predicate to another predicate (which may or may not already be cached at C_i), depending upon the attributes updated. If such tuples are cached beyond the transaction, either the individual tuples or a single *modified predicate* describing tuples after the update *must* be inserted in SCD, and *may* optionally be inserted into CCD_i.

A transaction commits at client C_i: The server is informed of all local updates that have not yet been communicated to it. This can either be in the form of updated tuples and corresponding update commands, or for large-sized updates, in the form of update commands only (to minimize network traffic and message processing costs). C_i is notified of the result of the commit operation. If the commit was successful (according to the concurrency control protocol), the *uncommitted* tags for tentatively updated data are removed from the cache by C_i; if not, the changes are undone. In either case, all locks held by the transaction are released at the server, *after* the server has updated the SCD to record all new predicates and tuples cached by the client beyond the transaction.

A tuple t is inserted at client C_j, $i \neq j$: The tuple is inserted into the database when the current transaction at C_j commits. The server checks SCD to determine which clients other than C_j are affected by the insertion. Client C_i will be notified of the change, along with the new tuple, if $t \Longrightarrow P, P \in LCD_i$. If C_i is notified of the change, it inserts the new tuple into its cache whenever it satisfies any predicate in CCD_i, and discards it otherwise. No changes are required to CCD_i or SCD. Alternative actions are also possible, e.g., the client may choose to flush from CCD_i all predicates invalidated by the insertion.

Note that a new tuple may be sent over to a client because it possibly participates in a cached join result. Consider a client that has cached the join result (EMPLOYEE $\bowtie$ DEPARTMENT) through the variable range query predicate (*dept_id* = DEPARTMENT.*dept_id*) on the EMPLOYEE relation. Suppose that a new EMPLOYEE tuple with a non-NULL *dept_id* field is now inserted at the server. Whether a notification is absolutely necessary is dependent on the differential join involving the new tuple. The differential join may either be computed at the server prior to issuing a notification, or the server may instead liberally inform the client of the new tuple. One option at the client is to invalidate the join result (with possibly a differential refresh upon demand from a subsequent query). Alternatively, if all DEPARTMENT tuples are locally available, then the differential join is *autonomously computable* [1] at the client (detection of such autonomously computable updates must be based on the exact or conservative client cache description). The new EMPLOYEE tuple is either stored or discarded depending on the result. A desired maintenance method for a cached query result may be specified a priori to the server, and be 'upgraded' or 'downgraded' as access patterns change over time.

One or more tuples are deleted using query predicate Q at client C_j, $i \neq j$: Tuples satisfying Q are deleted from the database when the current transaction at C_j commits successfully. The server must again notify clients other than C_j that are affected by the deletion, by comparing Q with SCD. Client C_i will be notified of the deletion if $\exists P \in LCD_i$ such that $(Q \cap P \neq \phi)$. The notification message may consist of object IDs of deleted tuples, or for large-sized deletions, simply the delete command. If client C_i is notified, it must execute the delete command on its cache. No changes are required to CCD_i or LCD_i.

One or more tuples are updated using query predicate Q at client C_j, $i \neq j$: The updated tuples, or simply the corresponding update command for large-sized updates, are sent to the server when or before the transaction commits. The server performs the changes on the central database if the commit is successful. The notification procedure is more complex than that for the delete case above, since tuples both *before* and *after* the update must be considered to determine which client caches are affected.

If the number of updated tuples is not too large, the update can be treated as a delete command, followed by insertion of new tuples. The procedures outlined above for deletion and insertion then apply. If many tuples are involved in the update, and a *modified predicate* describing tuples after the update can be easily computed, updates irrelevant for a client may be filtered out by comparing its cached predicates with Q and the modified predicate (exact algorithms appear in [1]). Notification may be in terms of updated tuples or simply the update command that is to be executed on the local cache. In the latter case, the server must ensure that the update is autonomously computable at the local site if invalidation of cached predicates is to be avoided. That is, tuples that now satisfy a predicate as a result of an update must have been at the client before the update or be transmitted to the client; otherwise the predicate must be invalidated.

Apart from the above operations, space constraints may prompt a client to purge some tuples in its cache and alter its cache description. We do not consider this issue in this paper.

4 Design issues and trade-offs

We briefly discuss below some performance and optimization concerns that may arise in a predicate-based caching scheme. Details of these issues are beyond the scope of this paper.

4.1 Client-side considerations

A client must locally examine and evaluate queries, as well as process update notifications from the server. The main issues at a client site are as follows.

Determining cache completeness: The cache description at a client, though conservative, may grow to be quite complex as more query results are locally cached. We use *predicate indexing* [19] and predicate merging techniques to efficiently support examination of a client cache description in answering the cache completeness question. Our predicate indexing mechanisms are similar to those proposed in [13], extended to handle general range query predicates.

It is important that the process of determining cache completeness be intelligent. Normally, query containment algorithms do not take into account application-specific semantic information like integrity constraints. Consider the cached join result (EMPLOYEE $\bowtie$ DEPARTMENT) with no attributes projected out. If the client encounters a subsequent query for all EMPLOYEE tuples, the general answer to the query containment question is that only a subset of the required tuples are cached locally. However, if it is known that all employees must have a valid department, then there are no *dangling* EMPLOYEE tuples with respect to this join, and the join predicate ($dept\_id$ = DEPARTMENT.$dept\_id$) on EMPLOYEE may simply be replaced by the predicate "TRUE" (i.e., all EMPLOYEE tuples appear in the join result). Although using general semantic information may be too complex, simplification of query predicates using referential integrity and non-NULL constraints on attribute domains can be very effective.

Local query evaluation: Queries may need to be processed locally in the cache, e.g., to answer a query posed by a transaction or to execute an update command in a notification message. It is not a requirement of our system that all cached tuples be in main memory. Efficient local evaluation of frequent queries involving joins or many tuples may require that appropriate indexes be constructed locally at a client site for either main memory or secondary storage. These local access paths may depend on data usage at individual clients, and may in particular be different from those in place at the server.

Query trimming: Whenever a query partially intersects cached predicates, the query predicate can be *trimmed* (by removal or annotation of locally available parts) before submission to the server. Trimming a query can potentially reduce the time required to materialize a query result at the client. It is an optimization decision whether and how to trim the query, involving factors like cost estimates of evaluating trimmed and untrimmed result sets, communication costs, and update activity on the cached data.

Query augmentation: Query augmentation is an optimization strategy in which a query predicate or the set of attributes selected by a query is *augmented* by a client before submission to the server so as to make the query result more *suitable* for caching. Possible benefits of query augmentation are: (1) simplification of the query predicate and cache descriptions at both the client and the server, thereby reducing costs of determining cache completeness and currency, (2) local processing of a larger number of future queries, due to pre-fetching of tuples and the augmented predicate, and (3) augmenting a query result to include any missing primary keys of participating relations, thus allowing a user query to conform to the restrictions imposed on cached queries for reasons of implementation efficiency.

A major performance benefit of adding relation keys to a query is that cached information need not be stored in duplicate, or in the form of individual query results. Tuples to be cached may be split up into constituent sub-tuples of participating relations (possibly with some non-key attributes projected out), and stored in local partial copies of the database relations.

The main costs of query augmentation are: (1) possibly significant increase in result set size and response time for the query, and (2) wastage of server and client resources in maintenance and storage of information that may never be referenced by future queries.

4.2 Server-side considerations

The server supports a number of clients, and manages the central repository of data. It executes remote queries submitted by clients, controls access to shared data, maintains client cache descriptions and issues notifications. The server performance thus dictates to a large extent the overall performance of the system. Below we consider some major issues at the server site.

Concurrency control: We emphasize that many forms of concurrency control can be supported in our framework, possibly even varying by client depending on the requirements specific to a site. For example, either two-phase or optimistic locking could be used, with locks being specified either in terms of predicates or object identifiers. Using notify locks in conjunction with an optimistic scheme adds another dimension, in that the behavior of a purely optimistic scheme becomes *semi-optimistic*, since a transaction may abort before it reaches its commit point if notified of committed updates to the objects in its read/write set. Using techniques outlined in [3], we can prove the correctness of various combinations of these strategies.

For clients with contentious data sharing, one possible modification to the concurrency control scheme adopted in this paper is to eliminate the optimism, and lock all accessed objects at the server. Thus, a query would not be re-executed at the server if the result is locally available, but the objects involved in answering the query would be locked at the server. Note that it is possible to do this locking, since we have object IDs for all cached tuples. Notification would still be required to support incremental refreshing of the cached data.

Issuing liberal notifications: The server uses liberal descriptions of client caches to issue notifications. If notification is over-liberal, it results in wasted work at client sites, and if too detailed, may have prohibitive overhead. It is very important to control the *degree* of approximation at the server, and ideally have it vary by server workload. Lighter loads may allow precise screening of relevant updates with respect to cached predicates. Predicate indexing and merging mechanisms similar to those at client sites are employed at the server to facilitate handling of cache descriptions that, though liberal, may grow to be arbitrarily complex with increasing number of clients.

Caching privileges: To control replication of database hotspots, and to avoid runaway notification costs at the server as the number of clients increases, clients may be granted *caching rights* to a relation or to a part thereof. Denial of caching rights to a client implies that no notification message will be sent to the client when the relation (or a specific portion) is updated when caching past the boundary of the current transaction is denied for the client. The client may re-use such cached data as long as it knows that the data might be out-of-date. If currency of the data is important, the query should be re-submitted at the server.

5 Summary and conclusions

In this paper, we have introduced the concept of client-side caching based on query predicates. A major advantage is associative access to the contents of a cache, allowing effective re-use of cached information. Increased autonomy at client sites, less network traffic and better scalability are a few other expected benefits over object ID-based caching schemes. We have examined design and performance issues relating to cache completeness and cache currency in order to determine the practicality of using our scheme in a dynamic caching environment. Approximate reasoning on cache descriptions, suitable query processing and update propagation techniques, and predicate indexing and merging mechanisms are employed to furnish our scheme with good dynamic properties.

Simulation studies to compare the performances of alternative caching schemes against ours are currently in progress. To prove the viability of our approach, we are also developing a prototype caching system for a relational database with 4 clients and a central server. Future work includes implementation of cache approximations, predicate indexing, local index creation, and effective management of space by a client.

Acknowledgements

This effort was supported in part by the Microelectronics Manufacturing Science and Technology project as a subcontract to Texas Instruments on ARPA contract number F33615-88-C-5448 task number 9. The work of Julie Basu was supported in part by Oracle

Corporation. We thank Catherine Hamon for many useful discussions. We also thank Marianne Siroker for her assistance in preparing this paper.

References

[1] J.A. Blakeley, N. Coburn, and P. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM TODS*, Vol. 14, No. 3, 1989.

[2] J. Blakely, P. A. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views," *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., May 1986.

[3] H. Boral and I. Gold, "Towards a Self-Adapting Centralized Concurrency Control Algorithm," *ACM SIGMOD Int. Conf. on Management of Data*, Boston, Ma, May 1984.

[4] M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architecture," *ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991.

[5] M. Carey, M.J. Franklin, and M. Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS," *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MI, May 1994.

[6] S. Ceri and J. Widom, "Deriving Production Rules for Incremental View Maintenance," *17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991.

[7] A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *18th Int. Conf. on Very Large Data Bases*, Vancouver, B.C., Canada, 1992.

[8] K.P. Eswaran, J.N. Gray. R.A. Lorie, and I.L. Traiger, "The Notion of Consistency and Predicate Locks in Database System," *CACM*, Vol. 19, No. 11, November 1976.

[9] M.J. Franklin, M.J. Carey, and M. Livny, "Local Disk Caching for Client-Server Database Systems," *19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993.

[10] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining Views Incrementally," *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., May 1993.

[11] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., 1993.

[12] E.N. Hanson, "Rule Condition Testing and Action Execution in Ariel," *ACM SIGMOD Int. Conf. on Managing Data*, San Diego, CA, June 1992.

[13] E.N. Hanson, M. Chaabouni, C.H. Kim, and Y.W. Wang, "A Predicate Matching Algorithm for Database Rule Systems," *ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, NJ, May 1990.

[14] A.Y. Levy and Y. Sagiv, "Queries Independent of Updates," *19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993.

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

[16] *Oracle 7 Server Concepts Manual*, Oracle Corporation, December 1992.

[17] N. Roussopoulos, "The Incremental Access Method of View Cache: Concepts, Algorithms, and Cost Analysis," *ACM TODS*, Vol. 16, No. 3, September 1991.

[18] Y. Sagiv and M. Yannakakis, "Equivalences Among Relational Expressions with the Union and Difference Operators," *JACM*, Vol. 27, No. 4, October 1980.

[19] T. Sellis and C.-C. Lin, "A Study of Predicate Indexing for DBMS Implementations of Production Systems," Technical Report, University of Maryland, College Park, MD, February 1991.

[20] M. Stonebraker, T. Sellis, and E. Hanson, "An Analysis of Rule Indexing Implementations in Data Base Systems," *Proceedings of the First Annual Conference on Expert Database Systems*, April 1986.

[21] K. Wilkinson and M.-A. Neimat, "Maintaining Consistency of Client-Cached Data," *16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990.

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