The VLDB Journal (1996) 5: 35–47

The VLDB Journal
© Springer-Verlag 1996

A predicate-based caching scheme for client-server database architectures

Arthur M. Keller[1] **and Julie Basu**[2,3]

[1] Stanford University, Computer Science Department, Gates Bulding 2A, Stanford, CA 94305-9020, USA
[2] Stanford University, Computer Science Department, Stanford, CA 94305-9020, USA
[3] Oracle Corporation, 500 Oracle Parkway, Box 659413, Redwood Shores, CA 94065, USA

Edited by Henry F. Korth and Amit Sheth. Received November 1994 / Accepted April 21, 1995

Abstract. We propose a new client-side data-caching scheme for relational databases with a central server and multiple clients. Data are loaded into each client cache based on queries executed on the central database at the server. These queries are used to form predicates that describe 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*. A separate issue, *cache currency*, deals with the effect on client caches of updates committed at the central database. We examine the various performance tradeoffs and optimization issues involved in addressing the questions of cache currency and completeness using predicate descriptions and suggest solutions that promote good dynamic behavior. Lower query-response times, reduced message traffic, higher server throughput, and better scalability are some of the expected benefits of our approach over commonly used relational server-side and object ID-based or page-based client-side caching.

Key words: Caching – Relational databases – Multiple clients – Cache completeness – Cache currency

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 clients are mostly disjoint. In effect, such caching of locally pertinent and frequently used data constitutes a form of dynamic data replication, whereby each client dynamically defines its own data space of interest.

In typical commercial relational databases with client-server configurations (Oracle 7 Server Concepts Manual 1992), 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 therefore 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 rapid growth in the performance of workstations, the validity of this assumption becomes questionable. It is increasingly common to find clients that are high-end workstations with the server being a mainframe or a minicomputer. Such clients are capable of performing intensive computations on their own, using the database as a remote resource that is 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 using object IDs have been recently investigated (Wang and Rowe 1991; Wilkinson and Neimat 1990). In these schemes, storage, retrieval, and maintainance of cached objects at client sites are done based on object IDs. Such caching 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 or an object class, e.g., through a WHERE clause in a SELECT-FROM-WHERE SQL statement, cannot be handled locally in these systems. Similar observations can be made for the page-based caches presented in Carey et al. (1991, 1994) and Franklin (1993) – these approaches do not address the question of associative query execution. We believe this issue is very important for caching in relational systems, where associative queries are common and are indeed one of the major reasons for their success.

Associative access may be supported in an object-ID or page-based client cache by using indexes defined on the database at the server, as is done in some object-oriented databases. Relevant index pages can be used in either a centralized or a distributed manner to answer an associative query. In the centralized scheme, index pages are managed solely by the server and cannot be cached by clients. A

Correspondence to: A.M. Keller

client submits each associative query to the server, which responds with a list of qualifying object IDs. The client then locally checks its cache for object availability and fetches missing data objects or pages as necessary. The centralized index scheme requires communication with the server for all index-based queries and index updates and may cause the server to become a bottleneck in the system. A distributed alternative is to allow clients to fetch, cache, and use index pages locally. This approach requires the enforcement of a consistency-control protocol on index pages. Because an index page relates to many more objects compared to a data page, index pages generally have very high contention and may be subject to frequent invalidation or update. Distributed index maintenance is therefore likely to be expensive even in systems that have low-to-moderate update activity, causing increased network traffic and slower response times.

In our approach, database index pages need not be referenced to answer associative queries locally. Instead, 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 remote 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 can have its own locally maintained 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 a 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 such as locking the accessed objects at the server). Another query represented by the predicate *job*='manager' asking for all managers can only be partially 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 new optimization decision that can potentially speed up data transmission and query processing.

The situation is more complex if the cached data are 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 either be recomputations or incremental, i.e., performed either by cached query re-execution or by differential maintenance methods. Which method performs best depends very much on the characteristics of the database system environment, 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. Cached query results may also have their method of maintenance upgraded or downgraded as access patterns change over time.

Examination and maintenance of cached tuples via predicate descriptions entail 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 tradeoffs 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, and 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; however, it is 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 the approximation can only be *liberal* in nature, that is, occasionally notifying a client of an irrelevant update is not a problem, but failure to notify a client that its cached object has changed can result in significant error. The conservative and liberal approximations must be applied carefully, so that they do not produce persistent negative impacts on system performance.

Apart from the above approximation techniques, we investigate local query evaluation by predicate containment reasoning and several optimizations applicable in that context. *Predicate=indexing* mechanisms are used to speed up the examination of predicate descriptions and the retrieval of cached tuples. Simplification of cache descriptions through *predicate merging* and *query augmentation* can help reduce long-term caching costs (although the 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. Appropriately extended, our scheme is also applicable in the contexts of object-oriented and distributed databases with client-server architectures.

The paper is organized as follows. Section 2 gives an overview of related work. We present a formal model in

Sect. 3, and describe the details of our scheme in Sect. 4. Implementation issues and tradeoffs at the client and server sites are addressed in Sections 5 and 6, respectively. Finally, we summarize our contributions, discuss work currently in progress and outline future plans in Sect. 7.

2 Related work

Our caching scheme is reminiscent of *predicate locks* used for concurrency control (Eswaran et al. 1976), where a transaction can request a lock on all tuples of a relation that satisfy a given predicate. Predicate lock implementations have not been very successful, mainly due to their execution cost (Gray and Reuter 1993) and because they can excessively reduce concurrency. 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 can cause fewer transactions to execute concurrently and thus reduce overall system performance. In contrast, our caching scheme supports predicate-based notification that is more optimistic, in that two transactions using cached predicates at different clients conflict (and are notified by the server of the conflict) only when a tuple in the intersection of shared predicates is actually updated or inserted. A similar scheme called *precision locks* was proposed in (Jordan et al. 1981) for centralized systems.

Query containment (Sagiv and Yannakakis 1980) is a topic closely related to the cache completeness question. Query evaluation on a set of *derived relations* is examined in Larson and Yang (1987). Efficient maintenance of materialized views has also been the subject of much research (Blakeley et al. 1986; Ceri and Widom 1991; Gupta et al. 1993) and is related to the cache currency 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 question has much in common with the elimination of updates that are irrelevant for a view (Blakeley et al. 1989). The above techniques of query containment and materialized view maintenance, though relevant for our scheme, have mostly been designed for relatively static situations with small numbers of queries or pre-defined views. Performance problems in handling large numbers of dynamic queries and views in a client-server environment have not been considered in these papers.

As noted in the Introduction, client-side data caching has been investigated in several recent studies (Wilkinson and Neimat 1990; Carey et al. 1991, 1994; Wang and Rowe 1991; Franklin et al. 1993; Lomet 1994; Adya et al. 1995). However, the issue of associative query execution, which is an important consideration for relational databases, has not been examined in any of the above works.

Among other related work, Roussopoulos (1991) proposes a view-caching scheme that uses the notions of *extended logical access path* and *incremental access methods*. A workstation-mainframe database architecture based on *view cache* techniques is described in Roussopoulos and Kang (1986). Simulated performance of related schemes in a client-server environment is studied in Delis and Roussopoulos (1992). In the "Enhanced Client-Server" system investigated in this work, query results retrieved from the server(s) are cached on local disks of client workstations. 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. Rather predictably, Delis and Roussopoulos (1992) report that fetching incremental update logs from the server(s) was found to be a bottleneck with increasing number of clients and updates and examines a log-buffering scheme to alleviate the problem. 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 among the clients and server.

A caching subsystem that can reason with stored relations and views is proposed in the *BrAID* system (Sheth and O'Hare 1991) 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 predicate-based *partial indexing* scheme for materialized results of procedure-valued relation attributes was outlined in Sellis (1987). The ideas are applicable in conventional database systems also, but were not developed and explored in that context, or in the context of client-server architectures. The work of Kamel and King (1992) deals with intelligent database caching, but is meant for applications that have a predetermined set of queries requiring repetitive reevaluation, such as for integrity constraint checking.

In the area of natural languages, queries are typically issued in context, allowing elliptical expression, such as "Where is the meeting that you are attending?" followed by "How long are you staying there?" or "In which hotel are you staying?"

The work of Davidson (1982) considered these elliptical queries and what they mean for databases, since the context is necessary to have well-formed queries. King (1984) used the knowledge of context as an opportunity for optimization, as formalized here, taking advantage of the data cached from earlier queries to reduce the search space for the elliptical successor queries, much in the spirit of this work.

Rule systems for triggers and active databases (Hanson and Widom 1993) are related to our notification scheme, in the sense that for such systems efficient identification of applicable rules is desired for each database update. Such rules are generally specified in the *condition-action* or *event-condition-action* format, where the *condition* part is expressed as a predicate over one or more relations. Hence, detection of firing of a rule involves determining satisfiability of predicates, and efficiency issues similar to ours arise for such systems. One difference is that for our 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 always propagating the tuples modified by the update. Therefore, unlike rule systems in active databases, we

require that the notification system employed by the server handle not only single modified tuples, but also general predicates involved in update commands.

3 Our approach

We propose a predicate-based client-side caching scheme that aims to reduce query-response times and network traffic between the clients and the server by attempting to answer queries locally 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 of simultaneous transactions 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 Sect. 5.6) is in many cases a viable technique for reducing long-term costs of maintaining cached query results. One major performance benefit is that tuples need not be stored in duplicate. Query results to be cached may be split up into constituent subtuples of participating relations (possibly with some non-key attributes projected out), and stored in local partial copies of original database relations.

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. Our scheme can be easily extended to handle such commands, and future work will explore this issue.

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 non-key attributes as well. Point=query predicates arise frequently during navigation among tuples of different relations using foreign key references, 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. In contrast, a range=query predicate over one or more relations specifies either a single value or a value range for one or more attributes, and in general has zero, one, or more tuples in its target set. For example, *salary* $\geq$ 50000 is a valid range-query predicate on the EMPLOYEE relation.

Note that a point query is a special case of a range query; we distinguish between the two only because they are processed somewhat differently at the implementation level for reasons of efficiency. A query predicate with a join condition, e.g., EMPLOYEE.$dept_id$ = DEPARTMENT.$dept_id$, is also a special instance of a range-query predicate. Join-query predicates may in general have one or more attribute value ranges specified in terms of other join attributes.

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 representing the ith client, $1 \leq i \leq n$. Let Q_i^E, where the superscript E denotes *exact* and $Q_i^E \geq 0$, be the actual number of query predicates such that the results of all queries corresponding to these predicates are cached at client C_i. We denote by P_{ij}^E the query predicate corresponding to the jth query result cached at client C_i. The superscript E in P_{ij}^E represents the fact that these are the exact forms of cached query predicates, without any approximations applied. Other information related to a query may be associated with its query predicate, e.g., the list of *visible* attributes retained after a projection operation on the tuples selected by a WHERE clause.

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:

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

In a real-life scenario, query predicates may be quite complex, and performance problems may arise if precise predicate-containment reasoning is done. 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. Thinking that an object is not in the cache when in fact it is there will result in re-fetching the object from the remote source; while the approach may be inefficient, correct answers are nonetheless produced. As long as data thought to be in the cache are 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.[1] Let Q_i^C denote the number of predicates in CCD_i. Formally,

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

where

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

[1] CCD_i may also be thought of as the *client* cache description, since it is used only by client C_i.

The superscript C in P_{ik}^C and Q_i^C stands for *conservative*. 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.

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 effect 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. Therefore, if all EMPLOYEE tuples in department 100 have been fetched and cached 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 only inefficient and not incorrect and will not recur if the range-query predicate gets cached in CCD_i.

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 still present in the cache and can be used to answer point queries (as discussed below, the server will notify the client of changes to these tuples, so their local usage cannot result in erroneous operation). 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 and uses it to generate notifications as transactions commit updates. Since the server handles all n clients, each of which may be caching tens or hundreds of query results, 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, but must generate all necessary notifications. It is 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,

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

where

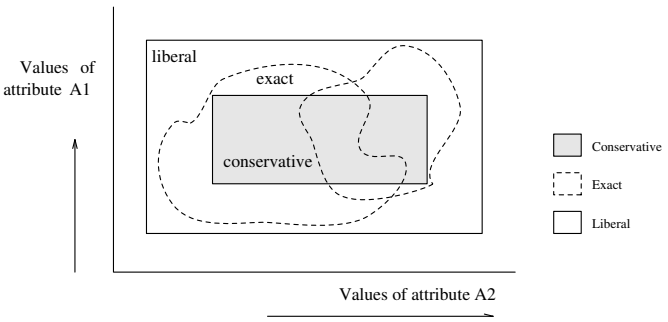

Fig. 1. Exact, conservative, and liberal cache descriptions for a relation R(A1,A2)

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

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 instance of liberal notification may happen when a tuple that could possibly 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 (Blakeley et al 1989), 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 prevailing at the client site (further details on cached joins are provided in Sect. 4.2.6).

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.

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. Client-cache contents and predicate descriptions at the client and server sites also change dynamically over time, as query results are cached or purged by the clients. We now consider the effects of the different events that may occur in the system, in the context of a sample concurrency control scheme defined below in Sect. 4.1. The architecture of a client-server system supporting predicate-based client-side caching is shown in Fig. 2.

4.1 Concurrency control

Several different forms of concurrency control can be employed in our caching scheme. For the purposes of this

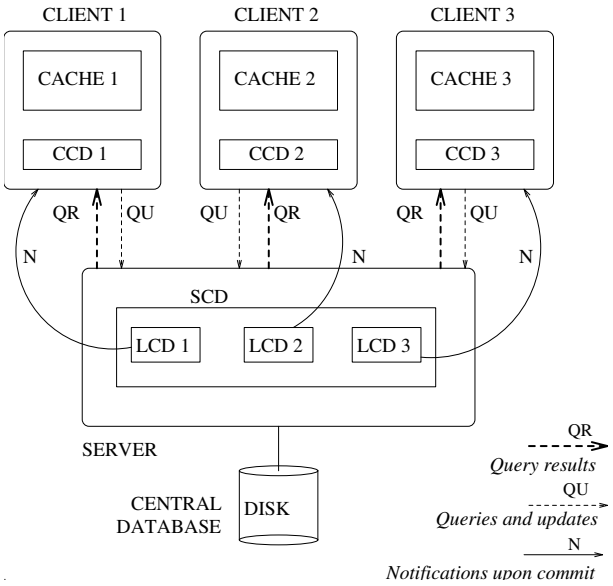

Fig. 2. Architecture of a predicate-based client-side caching system

section, we assume the following protocol. The scheme attempts to minimize unnecessary aborts of transactions while reducing communication with the server and is suitable for systems without overly contentious data-sharing among clients [it is similar in many respects to the notify-locks-based cache-consistency algorithm studied in Wilkinson and Neimat (1990) for object ID-based caches]. We assume that the server supports conventional two-phase tuple-level read/write locking, but it may or may not provide protection against phantoms.

Whenever queried data are locally available, a client optimistically assumes that its cache is up to date – the transaction operates on local copies of tuples, and locks are not obtained immediately from the server, but only recorded locally at the client (however, as we discuss below, the scheme is not purely optimistic). If a query is 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 yet been informed. Tuples accessed by the remote query and by the (uncommitted) updates are read-locked and write-locked, respectively, in the usual two-phase manner at the server during remote fetches. The uncommitted updates are also recorded as such by the server and made visible to the remote query as it executes.[2]

A remote-query submission may also be accompanied by *deferred* read-lock requests for tuples read locally since the last communication with the server. Such locking can help reduce aborts of transactions due to concurrent conflicting updates by other clients and subsequent notification, without incurring much extra cost (since they are "piggy-backed" on remote requests). Note that these lock requests are only for those tuples that have been accessed via the local cache and not through a remote fetch within the transaction.

A commit at the client sends all remaining updates (and possibly any deferred read-lock requests) to the server. One can observe that if a tuple was first read or written locally, and subsequently locked at the server during a remote fetch or upon a commit request, then there is a window of time between locally accessing the tuple and acquiring a lock on it at the server where it may have been modified by other transactions. 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 and the deferred updates are posted to the central database after locking tuples as necessary. As discussed in Wilkinson and Neimat (1990), this checking can be done by assigning a sequential message number to every message sent from the server to a client and by a handshake between the client and the server before the commit is declared to be successful. Notification messages are issued by the server upon each successful transaction commit, and these messages may abort the current transaction at a notified client if its read/write tuple sets or query predicates for locally evaluated range queries conflict with the updates at the central database.

The above protocol thus handles cache hits and misses differently – for hits, an incremental notification-based *semi-optimistic* scheme is employed, whereas normal two-phase locking is done at the server for all cache misses. The reason for the difference is that a cache miss always implies communication with the server, which is utilized also to lock any fetched tuples. When a transaction commits or aborts, all locks held by it are released at the server and at the client. However, cached tuples need not purged upon commit or abort and may be retained in the cache for intertransaction reuse. In order to maintain the currency of a cache correctly, the server must be kept informed of all point- and range-query predicates cached by the client 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 will 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 (e.g., by locking an index or key range), 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.

4.2 DML operations

Below we consider the effects of various DML operations that may be performed by clients. The discussion below is with respect to the cache at the ith client C_i. Apart from DML operations, space constraints may prompt a client to purge some tuples in its cache and alter its cache description. The effect of such an action in discussed briefly in Sect. 5.

4.2.1 Query submission at client C_i

Consider a SELECT-PROJECT-JOIN query with predicate Q that 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, then the tuple(s) satisfying Q

[2] Transmission of local updates along with remote queries is necessary since a query within a particular transaction must be able to see the effects of all (as yet uncommitted) updates made by that transaction when the query is evaluated at the remote server site.

may be found using locally defined and maintained indexes (see Sect. 5.2) 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.[3]

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 tuples satisfying Q (if any) are locally accessed in the cache. There is no effect on either CCD_i or SCD.
- Q is *independent* (Rissanen 1977) of the union of the predicates in CCD_i. The tuples in this case must be remotely fetched from the server. As outlined in the concurrency control protocol above, the request for remote execution is accompanied by any tuples locally read or updated by the transaction since the last communication with the server. The server locks the tuples appropriately and also records the uncommitted updates before executing Q, locking the tuples accessed by it, 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 these 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).[4]
- Q is partially contained in the union of the predicates in CCD_i. As in the preceding case, the query can be executed remotely at the server. One possible optimization is to *trim* the query before submission to eliminate tuples or attributes available locally at the client. Tradeoffs involved in this type of optimization are discussed briefly in later sections.

4.2.2 Tuple insertion at client C_i

When a tuple is inserted by a transaction running at client C_i, it is placed locally in the cache with an *uncommitted* tag. If the transaction later commits successfully at the server, the new tuple is inserted into the central database, incorporated into SCD, and the *uncommitted* tag is removed at C_i. We have assumed here that the new data are likely to be pertinent for C_i and hence 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).

4.2.3 Tuple deletion at client C_i

Assume that one or more tuples are deleted at client C_i using query predicate Q. If all tuples satisfying Q are not locally available in the cache, the procedure outlined in Sect. 4.2.1 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. 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 is cached past the transaction boundary (even though its tuples may have been deleted), 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 and avoiding a trip to the remote server. For example, let all EMPLOYEE tuples with *dept_id* $\geq$ 300 be cached at C_i. If EMPLOYEE tuples satisfying *dept_id* = 500 are now deleted by a transaction, the assertion that the cache holds all EMPLOYEE tuples with the property *dept_id* $\geq$ 300 is still valid after the deletion. A subsequent query with predicate *dept_id* = 500 can be evaluated locally, producing 0 tuples in its result set.

4.2.4 Tuple update at client C_i

If one or more tuples are updated at client C_i using query predicate Q, then the actions taken by client C_i and the effects on CCD_i and SCD are 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 completion of 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.

4.2.5 Transaction commit at client C_i

When a transaction commits at client C_i, the server is informed of all local updates that have not yet been communicated to it. The propagation of updates 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 on tentatively updated data are removed from the cache by C_i; otherwise, the changes are undone. In either case, all locks held by the transaction are released at the server, *after* the server has updated SCD to record all new predicates and tuples cached by the client beyond the transaction.

4.2.6 Tuple insertion at client C_j

Suppose that a transaction running at client C_j, $i \neq j$, inserts a tuple t. 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

[3] 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.

[4] 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.

the insertion. Client C_i will be notified of the change, along with the new tuple, if t is contained in some P, where $P \in LCD_i$. If C_i is notified, it inserts the new tuple into its cache whenever it satisfies any predicate in CCD_i, and discards it otherwise. No changes are necessary to either CCD_i or SCD. Alternative courses of action are also possible, e.g., the client may choose to extend a nearby existing predicate to include the new tuple, or flush from CCD_i all predicates invalidated by the insertion (e.g., join predicates).[5]

According to the concurrency protocol adopted in this paper, a transaction running at client C_i must be aborted if the inserted tuple falls within the purview of any cached predicate that has been used to evaluate locally one or more queries within the transaction. A *lazy* strategy may be adopted by the client in processing notifications, in that those messages that apply to relations and predicates not yet accessed by the current transaction may be deferred until an access actually happens. Such pending notifications can be processed after the transaction commits and before the next transaction is allowed to commence.

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 join-query predicate EMPLOYEE.$dept_id$ = DEPARTMENT.$dept_id$ on the EMPLOYEE relation. Also assume that the client has no other cached predicates for the EMPLOYEE relation. Now suppose that a new EMPLOYEE tuple with a non-NULL *dept_id* field is 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 choose to liberally inform the client of the new tuple. One option at the client is to invalidate the join result (the invalidation may be temporary, with possibly a differential refresh upon demand from a subsequent query). Alternatively, if all DEPARTMENT tuples are locally available (possibly from other cached query results involving the DEPARTMENT relation), then the differential join is *autonomously computable* (Blakeley et al. 1989) at the client.[6] The new EMPLOYEE tuple is either discarded or cached, depending on whether the result of this local computation is empty. A desired maintenance method for a cached query result may be specified à priori to the server and be upgraded or downgraded as access patterns change over time.

4.2.7 Tuple deletion at client C_j

Let one or more tuples be 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 primary keys of deleted tuples, or for large-sized deletions, simply the delete command itself. 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. A transaction running at C_i must be aborted if any tuples it read locally get deleted due to the notification.[7]

4.2.8 Tuple update at client C_j

If one or more tuples are updated using query predicate Q at client C_j, $i \neq j$, then the updated tuples, or simply the corresponding update command for large-sized updates, are sent to the server when or before the transaction issues a commit request at the client. The server posts the changes to the central database if the commit is successful. The notification procedure is more complex than that for the delete case above, since the values of the changed tuples both *before* and *after* the update must be considered to determine which client caches are affected. The set of clients to be notified by the server depends not only on the query predicate, but also on the updated attributes.

If the number of updated tuples is not too large, the update can be treated as a single delete command, followed by insertion of new tuples.[8] The procedures outlined above for deletion and insertion then apply. If many tuples are affected by the update, then this option may be too expensive. In this case, if a *modified predicate* describing tuples after the update can be easily computed, updates irrelevant for a client may be screened out by comparing its cached predicates with Q and the modified predicate [exact algorithms appear in Blakeley et al. (1989)]. Notification may again be in terms of updated tuples or simply the update command that is to be executed on the local cache. In the latter case, additional checks must be made to ensure that the update is *autonomously computable* (Blakeley et al. 1989) at the local site if invalidation of cached predicates is to be avoided. For example, consider a client C_i that caches the query predicate *salary* $\geq$ 50 000 for EMPLOYEE tuples. Now suppose that the *salary* field of all EMPLOYEE tuples is updated, say, by giving everyone a 5% raise. The entire update command can be propagated to C_i and be executed on its cache. However, C_i must still ensure that it has all tuples satisfying the cached predicate *after* the update is effected. That is, tuples that now satisfy *salary* $\geq$ 50 000 as a result of the update either must have been present in the client cache before the update or else they must now be transmitted to the client; otherwise, the cached predicate must be invalidated.

Note that all updated tuples that no longer satisfy any cached predicate should be discarded by C_i, or else LCD_i at the server must be augmented to include them. Precise screening of each updated tuple with respect to the cache can be done locally at a client site instead of at the central server, thereby distributing some work in maintaining cached results to individual clients. The tradeoff is between local computation as opposed to global communication.

[5] The tuples corresponding to the invalidated predicates may or may not be removed immediately from the cache. They may still be used individually for answering point queries locally.

[6] Note that detection of such autonomously computable updates must be based on the exact or conservative client-cache description.

[7] We allow negated terms in query predicates, but do not consider queries involving the difference operator.

[8] A deletion-insertion pair will work as far as the cache maintenance algorithms are concerned. However, the separation of updates into deletions and insertions can confuse constraints and triggers.

5 Design issues and tradeoffs at a client

A client must locally examine and evaluate queries, as well as process update notifications from the server. We consider below some performance tradeoffs and optimization questions that pertain to a client site. Details of these issues are beyond the scope of this paper.

5.1 Determining cache completeness

The cache description at a client, though conservative, may grow to be quite complex as more query predicates are locally cached. We use *predicate=indexing* techniques (Sellis and Lin 1992) 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 Hanson et al. (1990), with extensions to handle general range=query predicates. A one-dimensional index is dynamically defined on an attribute of a relation whenever cached predicates can be organized efficiently along that dimension. Predicate-indexing schemes will be examined in detail in our future work.

Since the major motivation for our caching scheme is effective reuse of local data, 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 of the query 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 is 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 EMPLOYEE.*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 such referential integrity and non-NULL constraints on attribute domains can be quite effective. Techniques developed for semantic query optimization (Bertino and Musto 1992; King 1984) are applicable in this context.

5.2 Evaluation of queries on the local cache

Queries may need to be executed locally in the cache, e.g., to answer a query posed by a transaction or in response to an update command in a notification message. It is not a requirement of our system that all cached tuples be in the main memory. We also do not maintain cached query results in the form of individual materialized views (as mentioned earlier, they are mapped into sub-tuples of database relations). Thus, 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 will depend on data usage at individual clients, and may in particular be different from those in place at the server.

5.3 Effective management of space

We briefly consider the various issues in managing cached tuples at a client site. Further work remains to be done in this area. We use CCD in this section to denote the cache description at any client in the system.

5.3.1 To cache or not to cache

Upon fetching in a new set of tuples from the server, a client is faced with a choice of whether or not to cache these tuples past the termination point of the current transaction. The client performs an approximate predicate-based cost-benefit analysis similar to that outlined in Stonebraker et al. (1990) to estimate the long-term benefits of caching the tuples in the new predicate. The algorithm is based on the LRU scheme and extended to take into account the sizes of cached tuple sets and anticipated future usage patterns. The following client parameters need to be taken into consideration (note that one-time costs of updating the cache descriptions, storing the tuples in the cache, etc. are not taken into account, since they do not affect the long-term benefits):

S_i: Size of the result set for the ith cached predicate P_i
F_i: Cost of fetching tuples satisfying P_i from the server
R_i: Cost of accessing and reading the tuples in P_i if cached locally
U_i: Cost of maintaining the tuples in P_i if cached locally
r_i: Frequency of usage of predicate P_i at the client
u_i: Frequency of updates by other clients that affect tuples covered by predicate P_i

The expected cost per unit time, T_i, of the caching the tuples in ith predicate P_i is:

$$T_i = \begin{cases} r_i R_i + u_i U_i & \text{if } P_i \text{ is cached} \\ r_i F_i & \text{if } P_i \text{ is not cached} \end{cases}$$

Thus, the expected benefit B_i of caching predicate P_i locally is:

$$B_i = r_i F_i - (r_i R_i + u_i U_i).$$

Notice that the above analysis represents only a client's view of the costs and benefits of caching a predicate. A cost model that applies to the server or to the entire system may be developed along similar lines. Work in progress addresses the formulation of such analytical cost models to estimate response times and server throughput.

5.3.2 Reclaiming space

Whenever space needs to be reclaimed, predicates and tuples are flushed using a predicate-ranking algorithm. The *rank* N_i of predicate P_i is defined as the benefit per unit size:

$$N_i = B_i / S_i.$$

Cached predicates may be sorted in descending order of their ranks. At any stage, only those predicates in the cache that have ranks equal to or higher than a certain cutoff rank may be kept in the cache.

Once a predicate P is chosen for elimination, a delete command with query predicate P can be executed on the cache to determine which tuples are candidates for deletion. Not all tuples satisfying P can be deleted, however, since a tuple may be purged from the cache only if it does not satisfy *any* cached predicate for the relation. For this reason, a reference count is associated with a cached tuple to indicate the number of predicates in CCD that are currently satisfied by it. A tuple may be flushed from the cache only when its reference count is 0. Also, recall that tuples retrieved through point queries may be accessed directly via a primary key index instead of CCD. Individual tuple usage must be tracked using standard LRU techniques to reclaim their space.

5.4 Maintaining the CCD

Reclaiming space based on predicate usage as discussed above will result in predicates being purged from the client-cache description. The CCD *must* be changed to account for purging of tuples and predicates, whereas such a change *may* optionally be reflected on the SCD at the server. Presence of extra predicates at the server can at most lead to an increase in irrelevant notifications to the client. Hence, updating SCD in this case can be treated as a low-priority task to be performed at times of light load and preferably before issuing too many notifications.

Whenever new tuples are cached, the associated CCD *may* optionally be augmented with one or more predicates that describe all or a subset of the new tuples. These tuples may remain in the client cache irrespective of whether the CCD is updated, as long as the SCD is augmented to include the tuples. This method ensures correct operation, since all necessary notifications will be generated.

Additionally, conservative approximations may be applied to the client-cache description as the number of cached query predicates increases and as access patterns change over time. For example, if a cached predicate P has not been used in a certain period of time, it may be dropped from CCD using an LRU policy (or reduced to a more conservative version). However, individual tuples satisfying P may or may not be flushed from the client cache, although reference counts will need to be decremented by 1 for these tuples; these tuples can be locally reused until resource constraints prompt the client to inform the server that P is to be flushed from *SCD*. Thus, because of our liberal notification scheme, only asynchronous coordination with the server is required for purging of predicates from a *CCD*. Incorrect operation can never result as long as all operations on all client and the server cache descriptions always obey the constraint

$(\forall i, 1 \leq i \leq n)(CCD_i \subseteq ECD_i \subseteq LCD_i)$.

5.5 Query trimming

Whenever cached predicates partially overlap with a query predicate at a client, there are two possible courses of action. The query may either be executed partially in the local cache and the missing tuples obtained through a remote fetch, or the query can be submitted to the server in its original form. The query predicate can in the first case 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. If the query predicate overlaps multiple predicates in CCD, then the query may be trimmed in more than one way. It is an optimization decision whether and how to trim the query, involving factors such as cost estimates of evaluating and transmitting the trimmed versus untrimmed result sets, communication costs, and update activity on the cached data. The decision may be left to the server (by annotating as "optional" parts of a submitted query that are locally available), with the client appropriately skipping or performing the local evaluation step. Some strategies for query trimming have been explored in the context of *BrAID* (Sheth and O'Hare 1991).

5.6 Query augmentation

Query augmentation is an interesting optimization strategy that can be explored in our caching scheme. A query predicate or the set of attributes projected by a query may be *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; (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 our cached query predicates 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 subtuples of participating relations (possibly with some non-key attributes projected out) and stored in local partial copies of the database relations, thereby simplifying their access and maintenance.

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 might never be referenced by future queries. The cost-benefit analysis of query augmentation involves examining the nature (e.g., selectivity, complexity, and size) of the query predicate, the data distribution, size and access paths of the relation, space availability in the cache, etc.

As a simple example of a situation where query augmentation is appropriate, consider the query predicate $dept_id \neq 100$ on the relation DEPARTMENT. Suppose that there are 50 tuples in the current instance of the DEPARTMENT relation, and that they are subject to change very infrequently. The query predicate in this case may be augmented to remove the restriction on $dept_id$. Thus, the client fetches all 50 tuples intead of 49; the cost of transmitting and storing one extra tuple is negligible, as is the maintenance cost in this particular scenario. Savings include faster execution at

the server and faster query examination at the client, since the check step is eliminated. The pre-fetch will also help answer any subsequent queries involving all DEPARTMENT tuples locally, e.g., the tuples satisfying $director$ = 'Smith'.

5.7 Predicate merging

The problem of handling large number of predicates in a cache description may arise at both client and server sites. We suggest predicate merging as an optimization method to address this issue. Two or more predicates in a cache description may be replaced with a single predicate whenever the *merged* predicate is equal to the union of the spaces covered by the individual predicates. In fact, since the cache description at a client can be conservative, the merged predicate at a client site can be a subset of the union of the constituent predicates. Similarly, the server cache description is allowed to be liberal, and therefore a merged predicate at the server can be a superset of the union of the individual predicates. Such predicate merging and simplification can produce a more compact cache description and can thereby improve the efficiency of examining queries at the client and of processing notifications at both the client and the server. The tradeoff is that communication between the clients and the server may increase due to conservative or liberal approximations applied to the cache descriptions in the process of merging predicates.

As an example, consider a cached predicate

$P_1 : (100 \leq dept_id \leq 300) \quad AND \quad (hiredate \geq 1980)$

for the EMPLOYEE relation. If two new predicates

$P_2 : (dept_id \geq 100) \quad AND \quad (hiredate \leq 1985),$

and

$P_3 : (dept_id \geq 200) \quad AND \quad (hiredate \geq 1975)$

are subsequently added to the cache description, the union of the three predicates can be reduced to a single equivalent predicate

$P_4 = P_1 \cup P_2 \cup P_3 = dept_id \geq 100.$

Well-established algebraic techniques exist for merging predicates – using distributive, associative, and commutative laws of the boolean AND and OR operators, two or more predicates may be replaced with an equivalent single one. However, purely algebraic techniques have exponential complexity, since all possible subsets of the set of predicates have to be considered in determining the applicability of an algebraic rule. Future work will investigate the use of auxiliary indexing mechanisms to speed up the process of detecting mergeable predicates.

Now consider the effect of predicate merging on tuple-reference counts that are maintained for space management. If there is no overlap among the predicates being merged, no special action is necessary; otherwise, reference counts must be appropriately adjusted for those tuples in the intersection of any two or more predicates being merged. One way of updating reference counts is to determine for each tuple in the final merged predicate the number of constituent predicates that are satisfied by it. The reference count for such a tuple should be decremented by an amount that is one less than this number in order to correctly account for predicate overlaps. This scheme incurs a one-time overhead at the time of predicate merging. The expense may be deemed to be unnecessary in a scenario where predicates are frequently merged but not purged as often, in which case the following alternative scheme may be employed. A *predicate=merge history graph* may be maintained for each client cache to record which predicates were merged to produce new predicates. When space needs to be reclaimed by purging a predicate that was previously generated from other predicates, the constituent predicates can be individually executed to decrement by one the reference counts of cached tuples that they each satisfy. Following the usual rule, a tuple can be removed from the cache if its reference count drops to zero during this process. The flushing of a predicate from a *CCD* must also update the predicate-merge history graph as necessary.

6 Design issues and tradeoffs at the server

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 generates notifications. The server performance therefore determines to a large extent the overall performance of the system. Below we consider some major issues at the server site.

6.1 Concurrency control

We would like to emphasize that many different 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 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 Boral and Gold (1984), the correctness of various combinations of these strategies can be proved.

In this paper, we have assumed a *semi-optimistic* concurrency-control scheme for cache hits (see Sect. 4.1). For clients with contentious data sharing, one possible modification to this scheme is to reduce the optimism and always lock all locally 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 such locking, since we have object IDs or relation keys for all cached tuples. Predicate-based notification would still be required to support incremental refreshing of the cached data.

6.2 Issuing liberal notifications

The server uses liberal descriptions of client caches to generate notifications. If notification is over-liberal, it results in

wasted work at client sites, and if too detailed, may have prohibitive overhead at the server. Hence, it is important to control the *degree* of liberal approximation at the server. Ideally, the server should be able to adapt the degree of approximation according to its current workload. Lighter loads may allow precise screening of relevant updates with respect to cached predicates, while notification may be more liberal at times of high load in order to distribute some of the work involved in maintaining cache currency to the clients. In instances of liberal notification, it is assumed that precise update screening is more expensive than communication with a client. Future work will investigate the tradeoffs between the precision of update screening performed by the server and the amount of communication with clients.

Predicate indexing and merging mechanisms similar to those at client sites can be employed at the server to facilitate generation of notifications from the SCD which, though liberal, may grow to be arbitrarily complex with increasing number of clients. Indexes to speed up the examination of SCD may use a "coarse" grid to partition the domain space of a relation. A certain partition can be (liberally) marked as being in use by the appropriate client whenever a cached point or range-query predicate intersects the partition. A more adaptable approach is to mark a grid partition with references to cached predicates that overlap with it and to maintain separately exact descriptions of all query predicates, including which clients cache them. Association of predicate references instead of client identifiers with a grid partition would permit detailed examination of cached predicates to be done in times of light load. During heavy loads, checking the actual predicates may be skipped, resulting in an even more liberal notification to a client.

6.3 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 given relation (or a specific portion of the relation) is updated and the client is caching some tuples in it past the boundary of the current transaction. The client may reuse such cached data with the understanding that the data might be out of date. If currency of the data is important, the query should be resubmitted at the server.

Caching privileges may be specified statically if the access pattern is known à priori or can be anticipated. If the unit of specification of caching rights is an entire relation, the scheme works in a manner similar to the usual authorization mechanisms for performing a selection or update on a relation. Permission to cache a query result is checked at the server when a query is submitted by a client, the client being informed of the outcome along with the answer tuples. The right to cache may also be granted on parts of a relation by defining predicates that specify attribute ranges where caching is prohibited for the client. Processing of such rather detailed caching rights would be approximate in the sense that the server may denote an entire query result as "not eligible for caching" if there is any intersection of the query space with a region where caching is disallowed for the client.

7 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 reuse 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 questions 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 will be employed to furnish our scheme with good dynamic properties.

Any successful implementation must be based on a good conceptual structure and design. In this paper, we have attempted to address some major conceptual and design issues. We are developing an experimental testbed to evaluate the viability of our approach, using a prototype of a predicate-based client-side caching system for a relational database with four clients and a central server. Detailed design of our experiments is currently in progress. Simulation studies to compare the performances of alternative caching schemes against ours for larger numbers of clients and queries are also planned.

Apart from the planned performance studies, many other important issues remain unexplored in this paper. Work currently in progress addresses implementation questions on suitable predicate-indexing techniques, optimization strategies, performance tuning, local index creation, and effective management of space by a client. Development of analytical system models, heuristics for effective conservative and liberal approximations of cache descriptions, and intelligent query-containment algorithms for determining cache completeness are topics for future efforts.

Acknowledgements. We would like to thank Kurt Shoens, Gio Wiederhold, and the anonymous referees for their helpful comments. We also thank Catherine Hamon for many useful discussions. This work was inspired by the caching issues introduced in the context of *view=objects* in the Penguin project (Barsalou et al. 1991).

References

1. Adya A, Gruber R, Liskov B, Maheswari U (1995) Efficient optimistic concurrency control using loosely synchronized clocks. Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, Calif, May
2. Barsalou T, Siambela N, Keller AM, Wiederhold G (1991) Updating relational databases through object-based views. Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, Colo, May
3. Bertino E, Musto D (1992) Query optimization by using knowledge about data semantics. Data Knowl Eng 9:121–155
4. Blakeley JA, P-A Larson, Tompa FW (1986) Efficiently updating materialized views. Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, May

5. Blakeley JA, Coburn N, Larson P-A (1989) Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, ACM Trans Database Syst 14:369–400
6. Boral H, Gold I (1984) Towards a self-adapting centralized concurrency control algorithm. Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Mass, May
7. Carey M, Franklin M, Livny M, Shekita E (1991) Data caching tradeoffs in client-server DBMS architecture. Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, Colo, May
8. Carey M, Franklin MJ, Zaharioudakis M (1994) Fine-grained sharing in a page server OODBMS. Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis, Minn, May
9. Ceri S, Widom J (1991) Deriving production rules for incremental view maintenance, Proceedings of the Seventeenth International Conference on Very Large Data Bases, Barcelona, Spain, September
10. Davidson J (1982) Natural language access to databases: user modeling and focus. Proceedings of the CSCSI/SCEIO Conference, Saskatoon, Canada, May
11. Delis A, Roussopoulos N (1992) Performance and scalability of client-server database architectures. Proceedings of the Eighteenth International Conference on Very Large Data Bases, Vancouver, Canada, August
12. Eswaran KP, Gray JN, Lorie RA, Traiger IL, The notions of consistency and predicate locks in a database system. Commun ACM 19:624–633
13. Franklin MJ (1993) Caching and memory management in client-server database systems. Ph.D. Thesis, Technical Report No. 1168, Computer Sciences Department, University of Wisconsin-Madison
14. Franklin MJ, Carey MJ, Livny M (1993) Local disk caching in client-server database systems. Proceedings of the Nineteenth International Conference on Very Large Data Bases, Dublin, Ireland, August
15. Gray J, Reuter A, Isolation concepts. In: Transaction processing: concepts and techniques. San Mateo, Calif, Morgan Kaufmann, 1993, pp 403–406
16. Gupta A, Mumick IS, Subrahmanian VS (1993) Maintaining views incrementally. Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, May
17. Hanson EN, Chaabouni M, Kim CH, Wang YW (1990) A predicate matching algorithm for database rule systems. Proceedings of the ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May
18. Hanson EN, Widom J (1993) Rule processing in active database systems. Int J Expert Syst Res Appl 6:83–119
19. Jordan JR, Banerjee J, Batman RB (1981) Precision locks. Proceedings of the ACM SIGMOD International Conference on Management of Data, Ann Arbor, Mich, April, pp 143–147
20. Kamel N, King R (1992) Intelligent database caching through the use of page-answers and page-traces. ACM Trans Database Syst 17:601–646
21. King JJ (1984) Query optimization by semantic reasoning. University of Michigan Press, Ann Arbor
22. Larson P-A, Yang HZ (1987) Computing queries from derived relations: theoretical foundation. Research report CS-87-35, Computer Science Department, University of Waterloo
23. Lomet D, (1994) Private locking and distributed cache management. Proceedings of the Third International Conference on Parallel and Distributed Information Systems, Austin, Tex, September
24. Oracle 7 Server Concepts Manual (1992), Oracle Corporation, December
25. Rissanen J (1977) Independent components of relations. ACM Trans Database Syst 2:317–332
26. Roussopoulos N (1991) The incremental access method of view cache: concepts, algorithms, and cost analysis. ACM Trans Database Syst 16:535–563
27. Roussopoulos N, Kang H (1986) Preliminary design of ADMS±: a workstation-mainframe integrated architecture for database management systems. Proceedings of the Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, August
28. Sagiv Y, Yannakakis M (1980) Equivalences among relational expressions with the union and difference operators. J ACM, 27:633–655
29. Sellis T (1987) Intelligent caching and indexing techniques for relational database systems. Technical Report CS-TR-1927, Computer Science Department, University of Maryland, College Park, Md,
30. Sellis T, Lin C-C (1992) A geometric approach to indexing large rule bases. In: Advances in database technology – EDBT '92. 3rd International Conference on Extending Database Technology Proceedings. Pirotte A, Delobel C, Gottlob G (eds.), Springer, pp 405–420
31. Sheth AP, O'Hare AB (1991) The architecture of BrAID: a system for bridging AI/DB systems. Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan, April
32. Stonebraker M, Jhingran A, Goh J, Potamianos S (1990) On rules, procedures, caching, and views in data base systems. Proceedings of the ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May
33. Wang Y, Rowe LA (1991) Cache consistency and concurrency control in a client-server DBMS architecture. Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, Colo, May
34. Wilkinson K, Neimat M-A (1990) Maintaining consistency of client-cached data. Proceedings of the Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, August