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

Arthur M. Keller
Stanford University

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 each client cache based on queries executed on the central database resident 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*. On the other hand, *cache currency* deals with the effect of updates at the central database on the client caches. We examine the various performance trade-offs and optimization issues involved in addressing the questions of cache currency and completeness using predicate descriptions, and suggest solutions that promote good dynamic behavior. Expected benefits of our approach over commonly used relational server-side and object ID-based or page-based client-side 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 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 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 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 using object IDs have been recently investigated [Wilkinson 1990, Wang 1991, Carey 1994]. In these schemes, storage, retrieval, and maintainance of cached objects at client sites is 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 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 1991, Carey 1994, Franklin 1993] — these approaches require access to distributed, high contention indexes in order to 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 an object-ID or page-based caching scheme is to use indexes defined on the database at the server, as is done in some object-oriented databases. If there exist server indexes 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 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 will likely 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 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 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 degraded as access patterns change over time.

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, 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; it is not however 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 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 several optimizations applicable in our 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 outline the details of our scheme in Section 3. Implementation issues and trade-offs at the client and server sites are addressed in Sections 4 and 5 respectively. Work currently in progress and future extensions are discussed in Section 6. Finally, we summarize our contributions in Section 7.

2 Related Work

Our caching scheme is reminiscent of *predicate locks* used for concurrency control [Eswaran 1976], 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 [Gray 1993]. 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 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 [Bannerjee 1981] for centralized systems.

Query containment [Sagiv 1980] is a topic closely related to the cache completeness question. Query evaluation on a set of *derived relations* is examined in [Larson 1987]. Independence of datalog queries from updates is investigated in [Levy 1993]. Efficient maintenance of materialized views has also been the subject of much research [Blakeley 1986, Ceri 1991, Gupta 1993], and is very 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 [Blakeley 1989]. 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, [Roussopoulos 1991] proposes a view caching scheme that uses the notions of *extended logical access path* and *incremental access methods*. A workstation-mainframe database architecure based on *view cache* techniques is proposed in [Roussopoulos 1986]. Simulated performance of related schemes in a client-server environment is studied in [Delis 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 1992] reports 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 amongst the clients and the server.

The work of Kamel and King [Kamel 1992] deals with intelligent database caching, but is meant for applications that have a pre-determined set of queries requiring repetitive reevaluation, as for integrity constraint checking. 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.

Rule systems for triggers and active databases [Hanson 1992, Stonebraker 1986] 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, detecting the triggering 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 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 aims to reduce 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 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 Section 4) 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 sub-tuples 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.

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. On the other hand, 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 at the implementation level for reasons of efficiency. 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 representing 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:

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

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. Thinking that an object is not in the cache when in fact it is 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 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[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 $Q_i^C \leq Q_i^E$, and $\bigcup_{1 \leq k \leq Q_i^C} P_{ik}^C \in CCD_i \implies \bigcup_{1 \leq j \leq Q_i^E} P_{ij}^E \in ECD_i$.

The symbol $\implies$ denotes the material implication operator, and in the context of query predicates has the following meaning: if $Q \implies P$, 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 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. Thus, if all EMPLOYEE tuples in department 100 have been fetched through separate point queries,

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

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

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 [Blakeley 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 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.

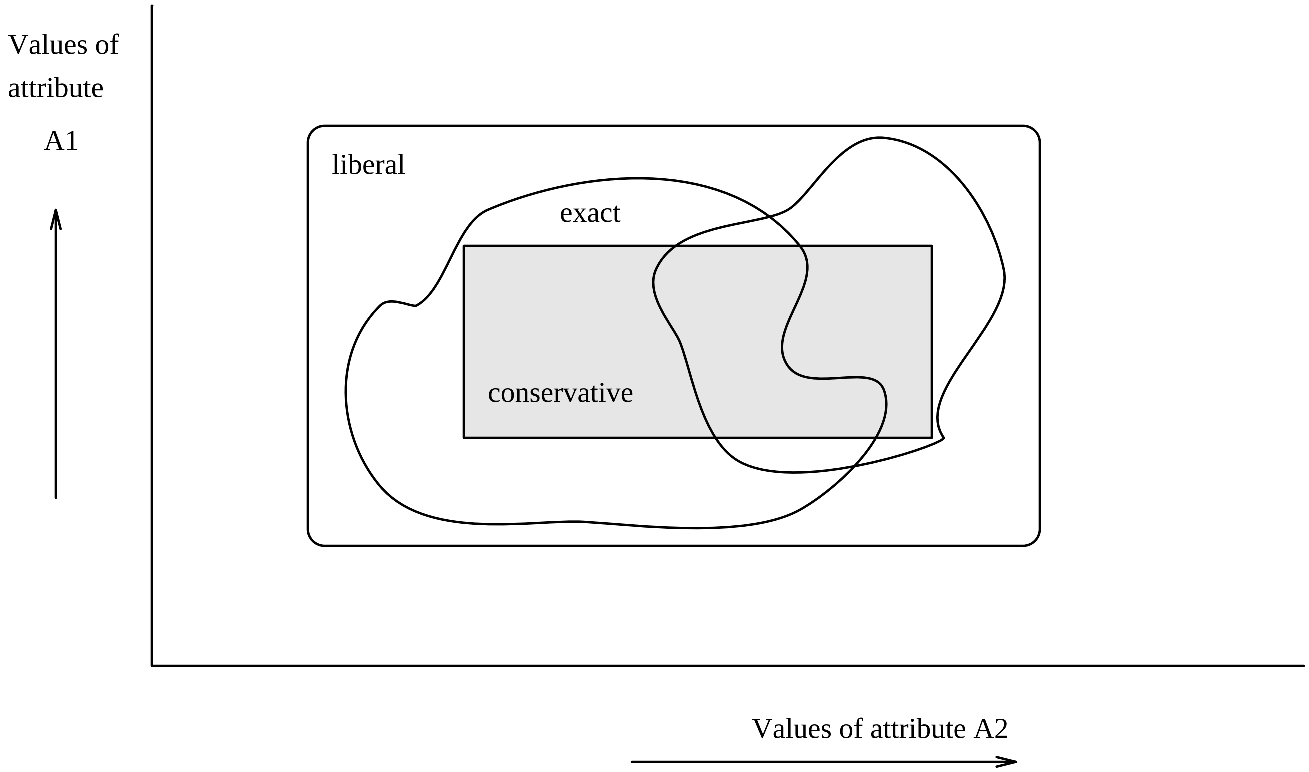


Figure 1. Exact, conservative, and liberal cache descriptions for a relation R(A1, A2).

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. It 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 1990] for object ID-based caches). We assume that the server supports conventional two-phase tuple-level read/write locking (it may or may not provide protection against phantoms).

Whenever queried data is 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 before 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 would 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 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

[2] This is necessary since a query within a particular transaction must be able to see all (as yet uncommitted) updates made by that transaction before the query is evaluated.

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 at the server is done for all cache misses. All locks held by a transaction are released at the server and at the client if the commit is successful, or if the transaction aborts. Note that in order to correctly maintain the currency of a cache, the server must be kept informed of all point and range query predicates cached by the client past a transaction boundary (i.e., past its 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 (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.

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. 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, assuming concurrency control is performed 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.

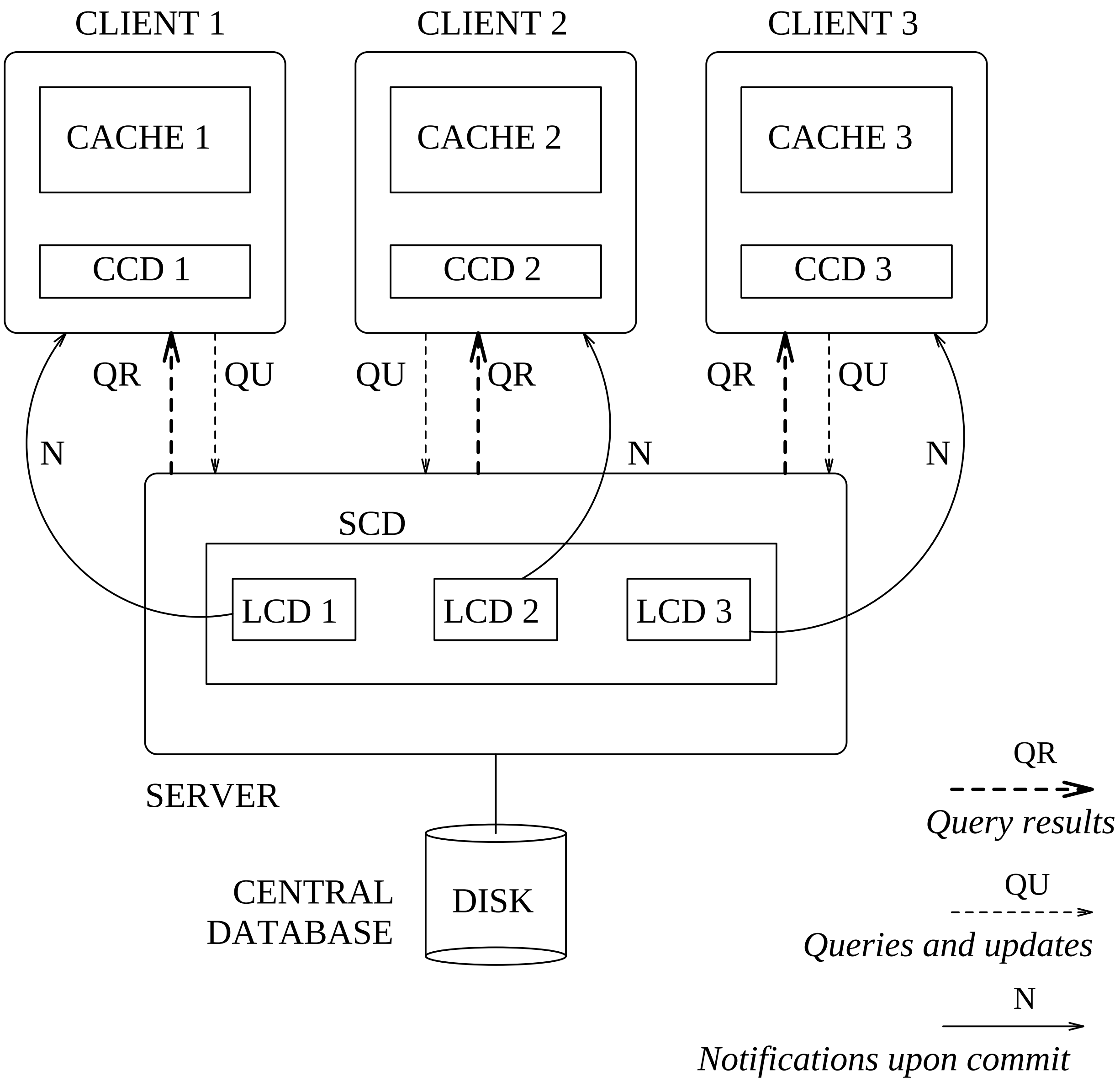


Figure2. Architecture of a predicate-based client-side caching system

- **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, then 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.[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 disjoint from 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. This case are similar to the disjoint one above. One possible optimization is to *trim* the query before submission to eliminate tuples

[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.

or attributes available locally at the client. Trade-offs involved in this type of optimization are discussed briefly in later sections.

- **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 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 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 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. 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 ($dept\_id \geq 300$) be cached at C_i. If EMPLOYEE tuples having ($dept\_id = 500$) 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.

- **One or more tuples are updated at client C_i using query predicate Q.**

  The actions taken by client C_i and the effects on CCD_i and SCD in this case 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.

- **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 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.

- **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 is contained in $P, P \in LCD_i$. 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 actions are also possible, e.g., the client may choose to flush from CCD_i all predicates invalidated by the insertion (e.g., join predicates).[5]

As per 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 the predicate of any locally evaluated range query. A *lazy* strategy may be adopted by the client in processing notifications, in that those messages that apply to relations not yet accessed by the current transaction may be deferred until an access actually happens. Pending notifications may 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 variable range query predicate ($dept\_id =$ 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 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 a priori to the server, and be 'upgraded' or 'downgraded' as access patterns change over time.

[5]The tuples corresponding to the invalidated predicates may or may not be removed immediately from the cache.

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

- **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 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].

- **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 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

[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.

may be screened out by comparing its cached predicates with Q and the modified predicate (exact algorithms appear in [Blakeley 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 at the local site if invalidation of cached predicates is to be avoided. That is, tuples that now satisfy a cached predicate as a result of an 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 must be discarded by C_i. Precise screening of updated tuples can be done locally at a client site and not at the central server, thus distributing some work in maintaining cached results to individual clients.

Apart from the above 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 Section 4.

4 Design Issues and Trade-Offs at a Client

A client must locally examine and evaluate queries, as well as process update notifications from the server. We briefly consider below some performance trade-offs and optimization opportunities at a client site. Details of these issues are beyond the scope of this paper. We use CCD below to denote the cache description at any client in the system.

4.1 Maintaining the CCD

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[9]; they can be used until the server is informed 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

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

4.2 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 1991] 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 1990], but extended 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. One feature of our system is that we allow range queries with *variable* limits, as in the join predicate *dept_id* = DEPARTMENT.*dept_id* on the EMPLOYEE relation. Such predicates cannot be indexed effectively; however, the number of such predicates involving a particular attribute is limited by the number of *feasible* joins that this attribute and the relation participates in. Since for most practical applications this number is quite low (often under 10), we do not feel this is a major problem. Further details of our indexing schemes are beyond the scope of this paper.

Since the major motivation for our caching scheme is effective re-use 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 (EMPLOYEE $\bowtie$ DEPARTMENT) with no attributes projected out. If the client encounters a subsequent query for all

[9]Reference counts (that are associated to all tuples for the purpose of space management) will need to be decreased for these tuples. Please see Section 4 for a discussion of space management.

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 such referential integrity and non-NULL constraints on attribute domains can be quite effective.

4.3 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 main memory. We also do not maintain cached query results in the form of materialized views (as mentioned earlier, they are mapped into sub-tuples of individual relations). Thus, efficient local evaluation of frequent queries involving joins or many tuples indicate 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.

4.4 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.

4.4.1 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 1990] to determine 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 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 to locally access and read the tuples in P_i if cached
U_i: Cost to locally update the tuples in P_i if cached
r_i: Frequency of usage of predicate P_i at the client
u_i: Frequency of updates by other clients that affect tuples within the scope of predicate P_i

The expected cost per unit time, T_i, for 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 is $B_i = r_i F_i - (r_i R_i + u_i U_i)$.

4.4.2 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. The CCD *must* be changed to account for purging of tuples and predicates, whereas such 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.

Once a predicate P is chosen for elimination, a delete command with query predicate P can be executed on the cache. 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 to 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.

4.5 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 like 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 parts of a submitted query that are locally available as "optional"), with the client appropriately skipping or performing the local evaluation step.

4.6 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, 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 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 sub-tuples of participating relations (possibly with some non-key attributes projected out), and stored in local partial copies of the database relations, thus 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.

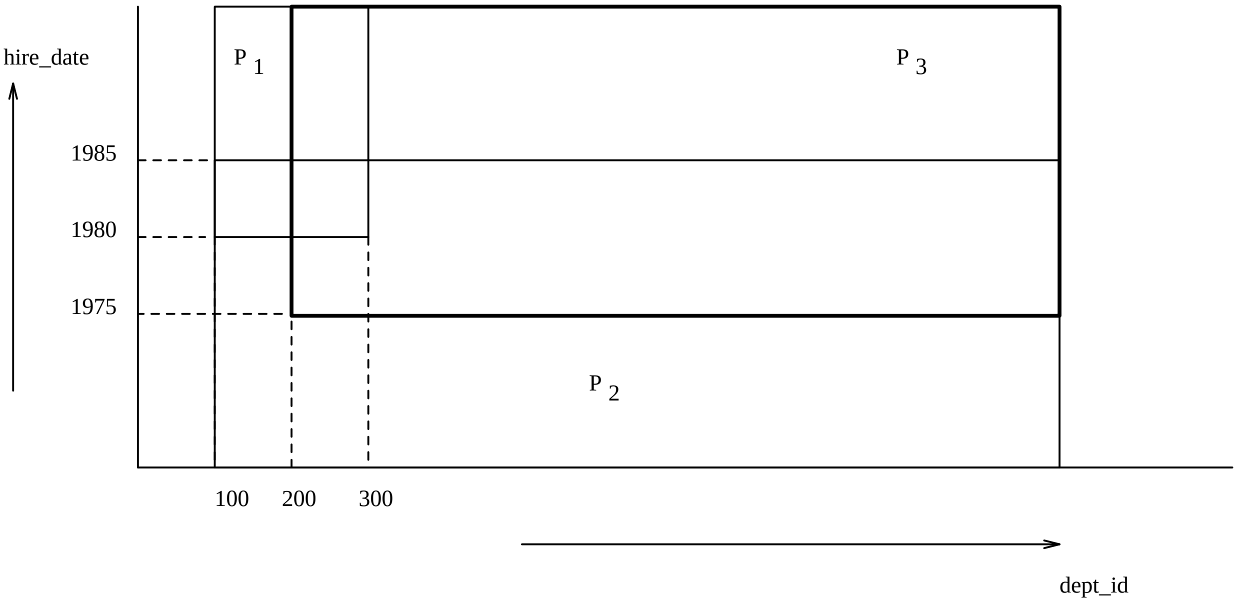
As a simple example of a situation where query augmentation is appropriate for a query predicate, consider a query ($dept\_id \neq 100$) on the relation DEPARTMENT. Suppose that there are 50 tuples in the current instance of the DEPARTMENT relation, and that they are subject to change very infrequently. 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 locally answer any subsequent queries involving all DEPARTMENT tuples, e.g., the tuples satisfying ($director$ = 'Smith').

4.7 Predicate Merging

As a final optimization strategy, we again consider the issue of reducing the number of predicates in a cache description. Although the discussion below is with respect to the client site, this strategy is applicable at both client and server sites. 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. Such simplification results in a more compact cache description, and hence higher efficiency in examining queries at the client, and in processing notifications at both the client and the server.

Consider a cached predicate P_1: $((100 \leq dept\_id \leq 300)$ AND $(hiredate \geq 1980))$ for the EMPLOYEE relation. If two new predicates P_2: $((dept\_id \geq 100)$ AND $(hiredate \leq 1985))$, and P_3: $((dept\_id \geq 200)$ AND $(hiredate \geq 1975))$ are subsequently added to the cache description, the union of the three predicates can be reduced to a single equivalent predicate $(100 \leq dept\_id)$.

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. Purely algebraic techniques have exponential complexity, since all possible subsets of the set of predicates should be considered in determining the applicability of an algebraic rule. We use a *predicate proximity graph* to speed up the process of detecting mergeable predicates. This graph records *proximity* relationships among cached predicates, which are the nodes of this graph. An arc between two nodes implies that the corresponding predicates intersect in at least one dimension of the attribute space. Figure 3 shows the predicates and the predicate proximity graph for the example above.



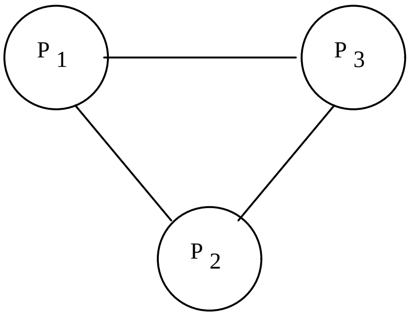


Figure 3. Three mergeable predicates on the EMPLOYEE relation and their proximity graph

5 Design Issues and Trade-offs 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 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.

5.1 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 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 Section 3.4). 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. Notification would still be required to support incremental refreshing of the cached data.

5.2 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. Hence, it is very important to control the *degree* of 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.

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. Indexing mechanisms for facilitating examination of SCD may themselves be approximate. For example, a "coarse" grid can be used to partition the domain space of a relation, and a partition can be 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 partition with intersecting cached predicate identifiers, and separately maintain descriptions of the corresponding predicates, including the identifiers of clients that cache them. Detailed examination of a predicate may be done at times of light load, and may be skipped otherwise.

5.3 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 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 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.

Caching privileges may be specified statically if the access pattern is known a 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.

6 Ongoing and Future Work

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 test the viability of our approach, based on a prototype of a predicate-based client-side caching system for a relational database with 4 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 large numbers of clients and queries are also in progress.

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

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 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.

8 Acknowledgements

We would like to thank Catherine Hamon for many useful discussions, and Marianne Siroker for her help in the preparation of this paper.

References

[Bannerjee 1981] J. Bannerjee, J. Jordan, and R. Batman, "Precision Locks," *ACM SIGMOD Int. Conf. on Management of Data*, 1981.

[Blakeley 1986] 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.

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

[Boral 1984] 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.

[Carey 1991] 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.

[Carey 1994] 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.

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

[Delis 1992] 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.

[Eswaran 1976] 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.

[Franklin 1993] 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.

[Gupta 1993] 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.

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

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

[Hanson 1990] 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.

[Kamel 1992] N. Kamel, and R. King. "Intelligent Database Caching Through the Use of Page-Answers and Page-Traces," *ACM Transactions on Database Systems*, Vol. 17, No. 4, December 1992, 601-646.

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

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

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

[Roussopoulos 1986] N. Roussopoulos and H. Kang, "Preliminary Design of ADMS±: A Workstation-Mainframe Integrated Architecture for Database Management Systems," *Proceeding of* *the 12th Intl. Conf. on Very Large Data Bases*, Kyoto Aug.1986, pp.355-362.

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

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

[Sellis 1987] T. Sellis, "Intelligent Caching and Indexing Techniques for Relational Database Systems", Technical Report CS-TR-1927, University of Maryland, October 1987.

[Sellis 1991] 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.

[Stonebraker 1986] 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.

[Stonebraker 1990] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedures, Caching, and Views in Data Base Systems," *ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, NJ, May 1990, 281-290.

[Wang 1991] 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.

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