PROACTIVE CACHING OF DATABASE QUERIES FOR WEB USE

Kai Risku(*)
TAI Research Centre
Helsinki University of Technology
P.O.B. 9555, FIN-02015 HUT, Finland
Phone: +358-9-4512888
Fax: +358-9-4514958
Email: krisku@cs.hut.fi

Teemu Rinta-aho
Ericsson Research
Oy L M Ericsson Ab
Telecom R&D, FIN-02420 Jorvas, Finland
Email: teemu.rinta-aho@lmf.ericsson.se

(*) = corresponding author.

ABSTRACT

Caching is a widely used technology for improving response times in different contexts. Accessing databases via the World Wide Web is an ever increasing trend, and efficient caching is therefore increasingly important in order to minimize both server load and user wait time. Using distributed databases containing large amounts of data makes solutions involving replication of data to a more centralized server unfeasible. Regardless of whether data has been replicated or not, obtaining data for web use via a database query can still take significant time.

We have built a Java-based metrics visualization tool that is capable of drawing charts based on data acquired via an HTTP-conformant server (MESS) using SQL queries. MESS provides facilities for defining user, group or corporate wide visualization panels, as well as access control to limit the visibility of sensitive data.

In this paper we describe the cache mechanism implemented in MESS to meet the demands on the query response times. The cache mechanism includes a proactive caching algorithm that remembers previously seen queries and updates the server-cached query results before they are

again requested by the clients. The cache algorithm is based on usage patterns to determine when and how often the queries should be refreshed by the server, giving clients a good chance of getting an up-to-date cached result.

The paper also includes a case study performed at Ericsson, where the visualization system has been in use for almost a year. We will show the concrete benefits of proactive query caching versus a more traditional approach.

PROACTIVE CACHING OF DATABASE QUERIES FOR WEB USE

INTRODUCTION

Caching is a common technology applied widely today for improving response times in different types of contexts ranging from processor instruction fetching to the heterogeneous objects of the World Wide Web. Most caching techniques used for WWW are weakly consistent, because they might return stale objects (old versions of an object). Weakly consistent caches usually depend on time-to-live (TTL) values and polling for updated versions (validation) to decide when a cached object must be updated.

With the ever increasing trend of accessing databases via the web, either directly or by generating dynamic webpages containing queried data [Nguyen], traditional web caching mechanisms are not so well suited because they assume the ability to quickly test the validity of cached data. When caching query results, it is not a trivial task to quickly test whether cached data still is valid without special support from the database, because the contents of the database can be modified by other applications.

In our target system, an HTTP-conformant [Berners-Lee] metrics server (MESS) provides data for a Java-based metrics visualization tool (ViCA) by returning results to SQL queries sent by the clients. MESS acts as a middleware component, providing access to both local and distributed databases, and is therefore a natural place to implement caching of database queries.

In this paper we describe our approach towards implementing a cache mechanism in MESS, as well as some initial results from this ongoing work. The cache mechanism includes a proactive caching algorithm that remembers previously seen queries and updates the server-cached query results before they are again requested by the clients. The cache algorithm is based on usage patterns to

determine when and how often the queries should be refreshed by the server, giving clients running SQL queries a good chance of getting an up-to-date cached result.

We base our cache algorithm on the following presumptions:

- The clients only perform read-only database queries, using a relatively small set of reoccurring queries (ViCA uses stored visualization panels with associated SQL queries)
- Typical query results are small enough to be cached in memory (although secondary storage could be used)
- We cannot rely on database notifications to know when data changes, because we are interfacing distributed legacy systems and the query itself can have temporal restrictions (i.e. getting data from the last 10 days)
- Data change and query references exhibit some kind of profile over 24 hour intervals

RELATED WORK

There have been significant research done regarding caching and prefetching in different areas of computing, such as filesystems [Cao,Kimbrel] or memory use in database management systems [Trancoso]. These are low-level techniques implemented in operating systems and the core of database servers, and are not the issue in this paper.

In client-server database architectures, client caching is mostly concerned with maintaining strong consistency and concurrency control [Carey, Wang] when updating data. In our system, we are only performing read-only queries of data updated from else, so update consistency and concurrency is not a concern for us.

Speeding up query processing has also widely been done using periodically updated snapshots [Adiba], materialized views, where the view is updated whenever the database is updated [Gupta]

and even stored queries [Subieta]. Although these problems are very closely related to the problems we are trying to solve, they all are based on a tight integration with the database where updated data directly can interact with the cached data, i.e. they are more concerned with where and how to update the data. In our approach, we simply have an application accessing a database, and we cannot intercept the data being updated or even detect that such has happened, so we are interested in knowing or guessing when to update the data.

Previous work on caching query results, are not really applicable to our problem domain, because they assume knowledge of when (and sometimes what) data has been changed, and the caching algorithms focus on cache admission and replacement [Scheuermann].

Other caching approaches include semantic caching [Godfrey], or caching of chunks [Deshpande], but we chose to not indulge in the inner workings of queries and their result. We do not take queries apart, nor do we try to understand the results of the queries.

Because we implicitly maintain only weak consistency, better ideas can be found in caching for the WWW, which also happen to fit nicely with the protocol we use between our clients and the server. Our server, can act like a WWW-proxy [Luotonen], because all client requests go to the server, which then can either forward the request to a database, or return an answer from the cache. Our approach is very similar to previously seen works on caching of database objects in web servers [Jadav], but again they assume knowledge of when data changes.

THE PROACTIVE CACHE ALGORITHM

Terminology

A query is being *referenced* when a client submits a previously seen query to the server. When the server executes a query in the background (i.e. without having a client waiting for the result) we

say it is being *refreshed*. If the query result differs from the previously cached result, the result has *changed*. A cached result is *stale* if the underlying data has changed, i.e. the result is old.

If a client request can be satisfied from the cache, it is called a *hit*, otherwise it is a *miss*. Removing an item from the cache is called *eviction*.

Problems

A big problem with caching database queries is the cost of validating cached data. In our system, we presumable have no other mechanism for validation than to execute the query and see if the result has changed. This is clearly an unacceptable course of action in response to a client request, so cached data must be proactively refreshed before any client requests it.

The problem then becomes to try to predict both when each query will be referenced and when it might have changed. Because this is a system for humans, where the data typically is generated directly or indirectly by humans (e.g. when visualizing project metrics) or by automated scripts, analyzing use and change patterns with respect to the time of day seems appropriate.

Selecting Queries to Cache

We only cache read-only queries (i.e. that have no side effects) that take longer than T_{min} seconds to run. Queries, together with their result and usage history are persistently stored in the database, but held in server memory for performance. Typical query results are rather small, so memory use is not a problem. On a server restart (which frequently happens, because this is a development system), the cached data can quickly be loaded into memory from the database.

Cache Eviction Policy

Since cached queries use both memory and other resources (CPU, network, database) for keeping the result up to date, they should be discarded from the cache when no longer actively needed. On the other hand, discarding e.g. a slow query can give a severe penalty if the query is later needed. A good eviction policy should consider both resource usage needed to cache a query and the implications of discarding the query. Our algorithm calculates a lifetime for each query based on its use frequency and cost of running the query, so that frequently used costly queries are kept cached longer. If the time since last use exceeds the calculated lifetime, the query is evicted from the cache. We do not yet have enough experience from real use to say if this is a good eviction policy.

Notified Updates

For database modifications performed via MESS (this is possible, but only used for some of the data), we are able to take the naive approach and mark all queries referencing the modified tables as dirty. Queries marked dirty get a fixed high priority, which ensures they are refreshed in the background as soon as the system load permits. In this case we can approximate an optimal execution order by giving quicker queries a higher priority than queries that have taken longer time to execute, i.e. $pri = F + 1/T_e$, where F is the fixed high priority, and T_e is the average time to execute the query. By executing the quicker queries first, we minimize the average waiting time.

Deciding When Queries Should Be Refreshed

A priority value is periodically calculated for each cached query, and queries whose priority value exceed a given threshold are refreshed in the background during server idle times.

We assume that the query result can change at arbitrary times, due to changes in the source data or because of temporal restrictions in the query itself (e.g. getting data for the last 10 days). Since we

can be interfacing distributed legacy systems (~ data warehousing), we cannot rely on triggers or other notification schemes, so other heuristics must be used.

In our system, we can intuitively think that the data is probably updated by humans during their working hours, or more general: the data tend to change at time periods that somehow follow the clock. We try to model the changes using a period of one day, divided into timeslots of a fixed size, e.g. one hour. By analyzing the times when a specific query has been executed and whether its result has changed, we can draw a histogram over the hours of the day with probabilities that the query result will change during each hour.

If a query is executed by the server at times t_1, t_2, t_3 and the result at t_3 differs from the result at t_2, the query result have changed one or more times between t_2 and t_3. The interval $[t_2, t_3]$ is then marked as a *dirty interval*. Every dirty interval is then distributed over the timeslots it spans (with appropriate weighting for partial slots).

Queries without a History

The first 24 hours after a new query has been placed in the cache, there is no point in trying to use the empty histogram. In this case a simpler algorithm is used, which calculates a priority value based on the time since the last update versus the average time between changes in the result.

Queries with a History

Using the change histogram, we can calculate a dirtiness factor D by summing the probabilities from t_u to t, where t_u is the time when the query last was updated and t is the current time.

For a recently added query, the statistical confidence of the histogram is very low, so we add an uncertainty compensation term

$$U = 1/C^k$$

where C is the coverage factor of the interval $[t_u, t]$, i.e. 1 if we have one day worth of experience in the timeslots covered by the interval, and so on. The exponent k is a decay factor that can adjust how fast the uncertainty term should decay as we accumulate days of experience. A value close to one gives a rather slow decay (but is more mathematically correct with regard to statistical confidence), while a larger value makes the uncertainty term diminish faster.

The same histogram technique can track when each query has been referenced and can be used to try to predict when a query is more likely to be used again. We can calculate a predicted reference value R, as e.g. the logarithm of the expected number of references in the interval $[t, t+c]$, where t is the current time, and c is a suitable look-ahead interval.

Now we can calculate a dirtiness (or staleness) accumulation factor A by summing:

$$A = D + R + U$$

where the terms D, R and U can be appropriately scaled to weight their relative significance. Multiplying A by the age of the query result gives us a priority value, where a higher priority value indicates a better candidate for an update operation. We can now, as a background operation in the server, update queries with priority values over some threshold, where the threshold can even be varied depending on the server load.

IMPLEMENTATION

The LUCOS Framework consists of a metrics server MESS, and Java-based visualization clients that communicates with the server using the HTTP/1.0 protocol [Berners-Lee]. The HTTP protocol allows both client-side and server-side caching which our implementation fully supports regarding SQL queries. The query result contains a standard Last-Modified header, denoting the time when the query result last was changed. The client can later send the same query with a condition to only

return data if the result has been changed after the timestamp supplied by the client (known as a conditional GET).

MESS is a multithreaded server implemented in C++, and it accesses the database using ODBC. There is one background thread responsible for garbage collection and cache maintenance and a dynamic number of worker threads responsible for processing client requests (both database queries and other types of requests). New worker threads are spawned on demand, and they are removed if left idle for more than a couple of minutes. The server load imposed by client requests can therefore be roughly approximated by the number of active worker threads.

The server recognizes known queries quickly by calculating a hash value of the query. New queries are given a unique identifier, which is used in log- and tracefiles to identify the query, and also allows us to analyze the behavior of the algorithm more easily.

Database queries are cached in a hashtable in server memory, but also written to the database in case the server needs to be restarted. For each query, a record containing information about the query and its history is kept. The record contains data about when the query was first seen, last referenced, last updated and last changed, as well as the dirtiness and reference histogram. Queries evicted from the cache are removed from memory, but left in the database so whey can be properly identified if they are seen again (resulting in a cache miss).

The background thread periodically recalculates the priority value for each cached query, which determines the position of the query in a priority queue. The background thread then refreshes queries whose priority value exceed a threshold which varies depending on the server load, i.e. the number of active worker threads.

EVALUATION

Many queries seen by the server can be quickly executed, while some queries may take rather long time to execute. By setting an appropriate value for T_{min} in the selection criteria for caching queries, the number of cached queries can be regulated and also concentrates the caching on the slower queries where the gain in response times are better. The value of T_{min} could also be altered at runtime, regulating the size of the cache and the burden of refreshing cached queries. In our implementation we arbitrarily set T_{min} to 0.5 secs, which gave us many small queries in the cache resulting in more test data to evaluate.

In a post mortem analysis of the server logfiles, we can try to determine if clients have been served stale data. At the time of a client request, the server cannot quickly determine if the cached data is stale, so the server has no choice but to give the cached data to the client if it is found in the cache. Suppose the cached data was generated at time t_0, given to the client at time t_1 and the query is refreshed at time t_2. If we at time t_2 determine that the query result has changed, there is a probability

$$p_s = (t_1-t_0)/(t_2-t_0)$$

that the client was served stale data. In fact, since we cannot detect if the data has been updated several times during $[t_0,t_2]$, the probability p_s is most likely an underestimation.

In our development environment (with only a couple of users accessing the server), post mortem analysis of the logfiles for the time period 20-28 May 1999 reveals that of 1068 query hits, only 270 (25.3%) are *potential* ($p_s>0$) dirty reads, i.e. where the query result had changed at the next refresh after they had been referenced. It is impossible to say if the read actually was dirty, but if we assume an even distribution, only half of them were actually dirty reads resulting in a dirty read

rate of 12.6%. Consequently, fresh data was obtained for 87.4% of the requests served from the cache.

In our development environment, no cached query has yet been evicted from the cache due to rather limited use of the system. As a result, we have no evidence yet of the goodness of the eviction policy.

CORPORATE CASE

Oy L M Ericsson Ab is one of the participating companies in the LUCOS pilot program. There is a long tradition of collecting data from various processes in Ericsson, to support product development and other processes.

A lot of databases all over the world contain data from economics to product development details. Combining and visualizing the data has traditionally required a big effort and a lot of manual work. The visualized results have not been enough up-to-date to allow efficient use of history data in steering an ongoing project. A lot of manual work is still done especially on the business side to get the visualizations to reports.

While LUCOS technology offers answers to visualization and product development controllability problems, new problems arise in adapting the technology on an existing large-scale data collection and warehousing infrastructure. Data is distributed over several databases in several countries. Having a unified view to the data gives the possibility to combine existing knowledge on different areas. It is although impossible to replicate the required data to one place because of the amount of it. In Ericsson the problem is solved by using the Sybase Adaptive Server Enterprise DBMS which allows the creation of proxy tables to present remote tables or views. Users can make queries to the proxy tables or views as if they were local. The database engine optimizes and forwards the queries

to appropriate remote databases. Using this technology it is possible to combine the distributed data at the level of basic SQL using joins.

The amount of data is huge. Some data changes all the time, some periodically. The average query response time can easily be over a minute. Most users of the system prefer fast response times to everything else. Caching the results in this kind of environment is therefore essential. The same stored queries are used daily during office hours. During nights there is a lot of time to check and update the cached results. As the data input is not a real time activity, but operated by employees at random times, it is acceptable to have a possibly few hours old value than to wait minutes for the latest value. Although support should be added for explicit fetching of the latest data, perhaps also giving the user an estimate of the time needed for the immediate refreshing of the queries used in the visualization.

The currently installed system proves that the average time to run a query for the first time is around 90 seconds. Giving that a panel may contain several queries, caching proves to be essential. The average time to obtain a cached result is under a second, where most of the time is actually spent transferring and parsing the data. Under a small load MESS is active for around ten percent of the time during a day to serve the clients and to keep the results of around 40 queries updated.

CONCLUSIONS AND FUTURE WORK

We have tried to implement an adaptive algorithm for caching database queries and actively trying to keep the cached data up to date without any special integration with the DBMS. This algorithm is well suited to be implemented in a middleware component providing mostly read access to databases for a number of clients (much like a WWW-proxy [Glassman]) or as a transparent layer on top of a database where only a simple database connection (such as ODBC) is used to interface to the database. In these cases we cannot rely on notification mechanisms to know when data in the

database has changed, and must therefore try to predict when cached data should be refreshed using the change history of the query results.

We think it is inadequate to simply calculate the average time between changes in the query results, because changes do not happen every X seconds, but instead are more likely to happen at certain times during the working day and less likely to happen at night. Our model includes histogram analysis (???or 'temporal analysis'???) of both changes and references by the hour, and use of the histograms to determine when queries should be refreshed. Figure 1 shows an example of the temporal distribution of a query in our development environment.

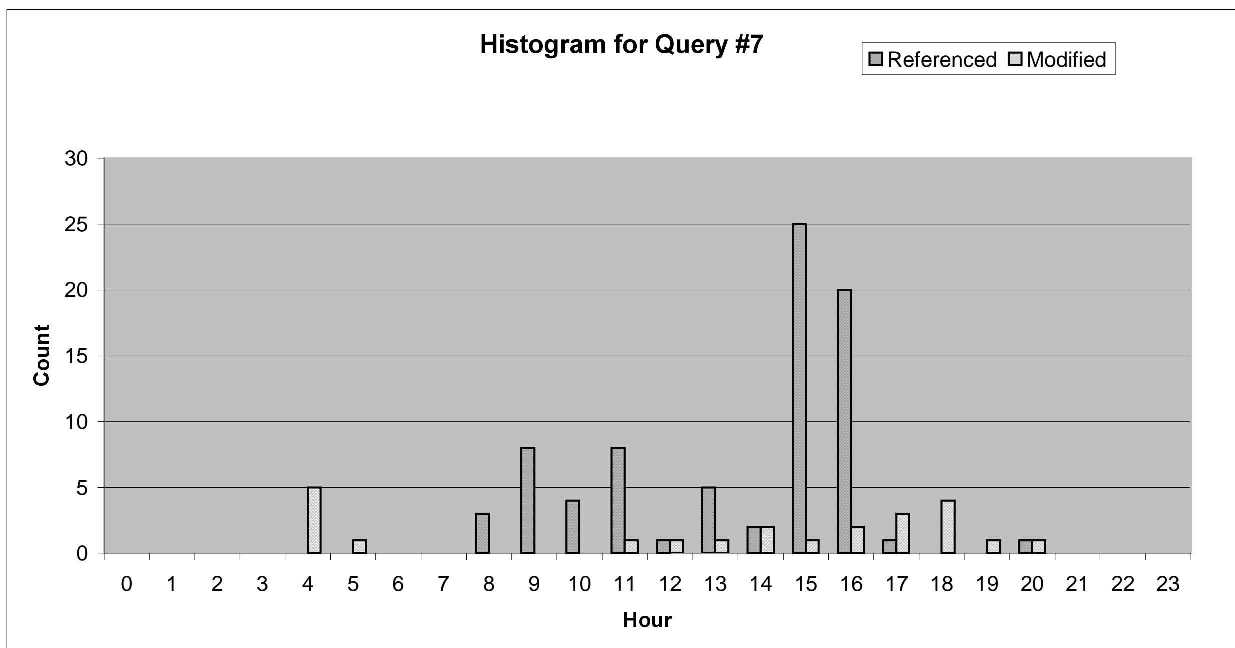

Figure 1. Example histogram

Our algorithm has not yet been extensively used, nor has it been properly analyzed to see if it performs significantly better than simpler approaches. We expect further refinements and tuning to be made as we gain more experiences in using our algorithm in a larger scale. Early findings indicate that our algorithm takes a rather aggressive approach and refreshes queries quite a lot. This is especially true of "young queries" where there is not much history data to utilize, and the built-in

pessimistic uncertainty refreshes the queries often. It is easy to tweak the algorithm to be less pessimistic at the expense of a higher probability to get stale data, although we are currently content with the approach of utilizing the server idle time to do query updates.

The varying freshness demands on the data is still problematic. Some data may be updated more or less continually, while other data can be updated on a monthly basis, and the user visualizing the data can have quite different requirements regarding its freshness. In some cases the data must be very fresh, and the user might be willing to trade response time for freshness. The current implementation does not take this into account, but we intend to extend the client to specify explicit freshness requirements [Dingle] for query results. This should make it easier to adapt the algorithm to fulfill the expectations of the users.

In the future, we will try to correlate the behavior of different queries using common tables. Our algorithm, while analyzing each query separately, actually try to reflect the changes in the underlying database tables (except for temporally bound queries where the result may change over time although the table remains unchanged). When a refresh detects a changed result, the correlation analysis would point out other queries with a high probability of change. This would also lower the validation overhead, because related queries do not need to be refreshed unless the data seem to have changed.

With the introduction of the unique identifiers assigned to each cached query, we are now gathering logs over real use and reuse of database queries, which was not very feasible earlier. As we collect logs over a longer period of time, we will be able to perform trace-driven simulations, which will help improving the algorithm. It is a very slow and cumbersome process to try to tune the parameters in a real environment, because the effects are not immediately seen.

REFERENCES

[Adiba] M. Adiba and B. Lindsay, "Database Snapshots", *Proceedings of the 6th International Conference on Very Large Databases*, pp. 86-91, October 1980.

[Cao] P. Cao, E. Felten, A. Karlin and K. Li, "A Study of Integrated Prefetching and Caching Strategies", *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 188-198, May 1995.

[Carey] M. Carey, M. Franklin, M. Livny and E. Shekita "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proceedings of the 1991 AMC SIGMOD International Conference on Management of Data*, pp. 357-366, May 1991.

[Berners-Lee] T. Berners-Lee, R. Fielding and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0", **RFC 1945**, UC Irvine, May 1996.

[Deshpande] P. Deshpande, K. Ramasamy, A. Shukla and J. Naughton "Caching Multidimensional Queries Using Chunks", *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 259-270, June 1998.

[Dingle] A. Dingle and T. Partl, "Web Cache Coherence", *Proceedings of the 5th WWW Conference*, May 1996.

[Glassman] S. Glassman, "A Caching Relay for the World Wide Web", *Computer Network and ISDN Systems,* 27(2), November 1994.

[Godfrey] P. Godfrey and J. Gryz, "Semantic Query Caching for Heterogeneous Databases", *Proceedings of the 4th KRDB Workshop*, August 1997.

[Gupta] A. Gupta, I. Mumick and K. Ross, "Adapting Materialized Views after Redefinitions", *Proceedings of the 1995 ACM SIGMOD international conference on Management of Data*, pp. 211-222, May 1995.

[Jadav] D. Jadav and M. Gupta, "Caching of large Database Objects in Web Servers", *Proceedings of the 7th International Workshop on Research Issues in Data Engineering*, pp. 10-19, April 1997.

[Kimbrel] T. Kimbrel and A. Karlin, "Near-optimal Parallel Prefetching and Caching", *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pp. 540-549, Oct 1996.

[Luotonen] A. Luotonen and K. Altis, "World-Wide Web Proxies", *Proceedings of the 2nd International WWW Conference*, October 1995.

[Nguyen] T. Nguyen and V. Srinivasan, "Accessing relation databases from the World Wide Web", *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pp. 529-540, June 1996.

[Sheuermann] P. Sheuermann, J. Shim and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager", *Proceedings of the 22nd VLDB Conference*, pp. 51-62, September 1996.

[Subieta] K. Subieta and W. Rzeczkowski, "Query Optimization by Stored Queries", *Proceedings of the 13th VLDB Conference*, pp. 369-380, September 1987.

[Trancoso] P. Trancoso, J-L. Larriba-Pey, Z. Zhange and J. Torrellas, "The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors", *Proceeding of the 13th International Conference on Distributed Computing Systems*, pp. 301-310, May 1993.

[Wang] Y. Wang and L. Rowe, “Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture”, *Proceedings of the 1991 AMC SIGMOD International Conference on Management of Data*, pp. 367-376, May 1991.