Database Change Notifications: Primitives for Efficient Database Query Result Caching

Cesar Galindo-Legaria Torsten Grabs Christian Kleinerman Florian Waas

Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
U.S.A.
{cesarg, torsteng, ckleiner, florianw}@microsoft.com

1 Introduction

Many database applications implement caching of data from a back-end database server to avoid repeated round trips to the back-end and to improve response times for end-user requests. For example, consider a web application that caches *dynamic web content* in the mid-tier [3, 2]. The content of dynamic web pages is usually assembled from data stored in the underlying database system and subject to modification whenever the data sources are modified. The workload is ideal for caching query results: most queries are read-only (browsing sessions) and only a small portion of the queries are actually modifying data. Caching at the mid-tier helps off-load the back-end database servers and can increase scalability of a distributed system drastically.

Figure 1 shows a typical 3-tier architecture of a web services provider including database servers, a farm of web servers which generate the dynamic content, and clients. Servers of the mid-tier layer maintain individual caches.

While caching at the mid tier is very attractive from a scalability perspective, applications cannot arbitrarily trade cache consistency for performance. Depending on the application, stale cache entries may cause user frustration at the least and incorrect application behavior at the worst.

Periodically polling the back-end for changes is in many cases inadequate —if not done frequently enough then data in the cache becomes stale, but if done too frequently then unnecessary processing overhead is incurred.

Different commercial products have addressed this problem by making different trade-offs, but typical solutions tend to be targeted towards specific application scenarios and they cannot easily be re-used. At the same time, some of the techniques could perform in a more robust and efficient manner if implemented inside the database server.

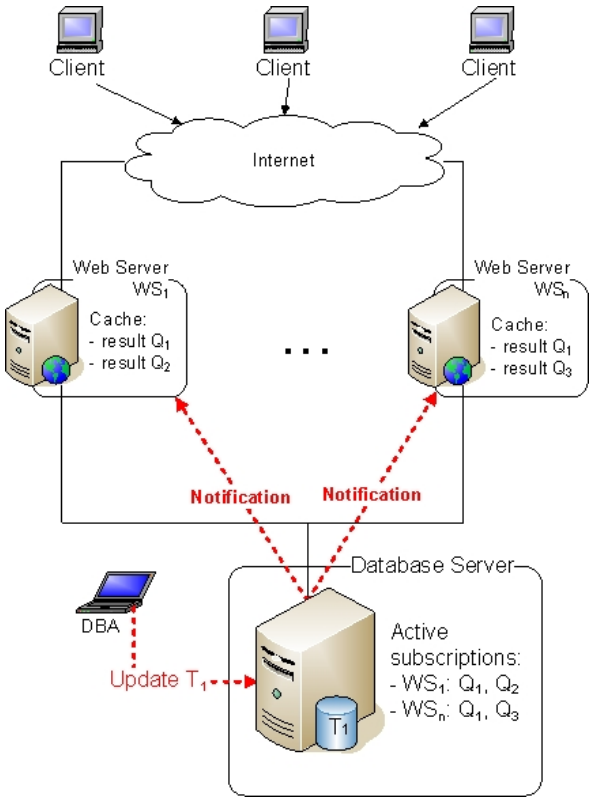

Figure 1: 3-tier web application architecture with mid-tier caching

In this demonstration, we show the Database Change Notification mechanism as implemented in Microsoft SQL Server 2005 Beta 3. This database primitive allows subscription and change detection for query results, and it is integrated in the query processor, where we leverage existing infrastructure and techniques for materialized view maintenance (as suggested in [1]). The demonstration is

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

an example of an online book store and we show how various technologies fit together — Database Change Notifications for subscription and invalidation; Service Broker as the transport layer to reliably route notification messages from the database server to the caches; and ASP.NET to cache web pages in the mid tier.

2 Database Change Notifications

A goal of our work is to factor out common primitives that are required by a wide range of database applications which can benefit from mid-tier or client-side caching. *Database Change Notifications (DCN)* provide server-side primitives that allow clients or applications to *subscribe to query results*, and the server sends notification messages for cache invalidation as soon as these results have changed.

We see two main requirements for DCN. First, *under-invalidation* is unacceptable, i.e. if the result of the subscribed query has changed due to data modification in the server, the system must guarantee that invalidation takes place. And, second, *latency must be low*. This is not only a performance requirement but it also affects consistency across caches.

The important building blocks of DCN are the notions of (1) a subscription request, (2) an active subscription, and (3) a change notification message:

A *subscription request* is a regular (SQL) query with a special annotation[1] which indicates the client's intent to cache the query result.

The database system processes the query as usual and returns the result. In addition, it registers the request to notify the cache of potential query result changes. We refer to the successful registrations as *active subscriptions*.

In case any database operation – ranging from DML queries to maintenance or DDL operations – causes a change to the query results, the database server sends a *change notification message* to the subscriber.

Figure 1 summarizes these concepts and relates them to the different components in a typical web application environment. As the figure shows, web servers WS_1 and WS_n cache query results for queries Q_1, Q_2 and Q_1, Q_3, respectively. These subscriptions are reflected in the set of active subscriptions at the database server. Let us now assume that table T_1 which underlies Q_1 is updated and the result for Q_1 has changed. This fires notification messages for both copies of Q_1's result stored at WS_1 and WS_n which need to be invalidated.

3 Workflow

Database change notifications consist of a number of components inside the database system which are tightly integrated with existing query processor technology.[2] The key concept is to leverage existing materialized view maintenance algorithms. Explaining the individual parts and their interaction can be done best by examining sample workflows.

We distinguish two major scenarios: (1) subscribing to a query result and (2) processing of changes which trigger a change notification. Additional scenarios a comprehensive implementation needs to take into account including DDL or maintenance operations are discussed in the next section.

3.1 Subscribing to a Query Result

In Figure 2 illustrates the workflow for subscribing to query results. Integrated into the query optimizer, the DCN components interact as follows (numbers in the text refer to labels in the figure):

- DCN analyzes the incoming query annotated with a subscription request (1, 2); requests for queries which are not notifiable (e.g. queries which reference non-deterministic functions) are rejected (3).
- For notifiable queries, DCN sets up *templates* and parameter tables as necessary (4). The template serves as a transient materialized view which will trigger the change detection mechanisms of view maintenance; the parameter table stores metadata about the subscribing client incl. a delivery address for the notification message.
- DCN re-writes the query so that at run time parameters are inserted into the parameter table (5) and the rewritten query is optimized as usual (6).
- During query execution, parameters are processed and stored (7) and query results returned as usual (8).

Note that during optimization only the support structures are set up, i.e. the resulting query plan can be cached and re-used exactly the way conventional query plans are cached.

3.2 Change Detection and Notification

Figure 3 outlines the workflow for DCN processing in case of DML operations such as `INSERT`, `DELETE`, `UPDATE`. Again, numbers in the text refer to labels in the figure.

- DCN analyzes the DML – submitted by clients – with the materialized view framework (1, 2).
- The QNM supplies the templates (corresponds to a transient materialized view) (3, 4). The templates contain special operators which call the QNM at run time to fire notifications.
- The optimizer combines/expands the plan as usual for materialized view maintenance (5).

[1] No alteration of the SQL statement is actually necessary; the annotation is part of the protocol and can be controled through language extensions in database programming environment such as ADO .NET

[2] The central component is the Query Notification Manager (QNM), implemented as part of the query optimizer. However, for clarity we depict several internal components such as the subscription store as separate elements in the figures.

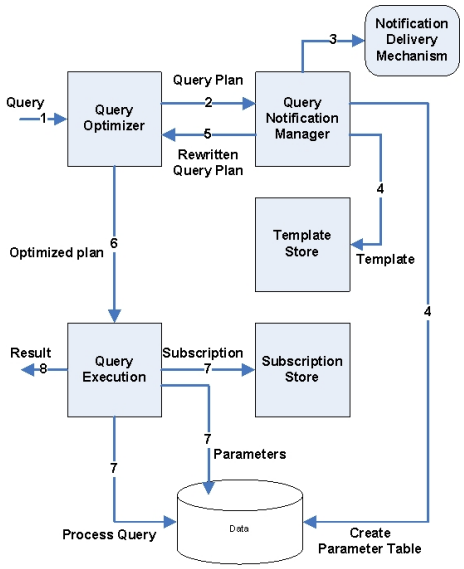

Figure 2: Workflow for subscribing to a query result

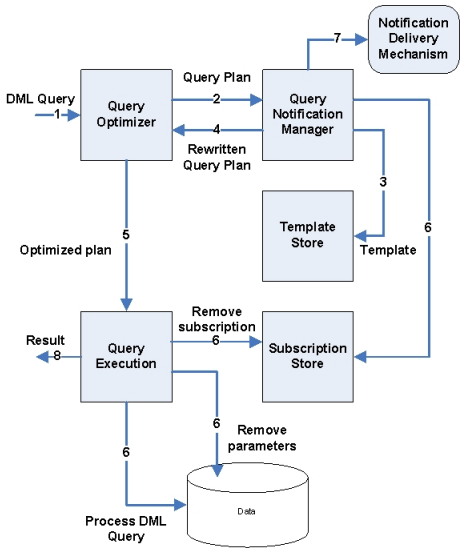

Figure 3: Workflow for change detection and notification

- The modified query plan computes deltas effectively identifying parameter rows which contain the metadata needed for notification delivery and maintains the DCN metadata in the parameter table (6).
- DCN fires notifications (7) and returns results (8).

The fact that all elements of the change detection are part of the query plan ensures that firing the notification is part of the transaction of the DML statement. Moreover, like with the subscribing plan, the query plan for processing changes can be cached.

4 Database Change Notifications & The Real World

DCN utilizes a number of simple principles many of which have been used in another context such as change detection for materialized views. Implementing the feature in an industry strength database system posed a number of additional challenges: Besides the two major scenarios, which we discussed in Section 3, DCN has to be integrated into a multitude of database operations. In this section, we will discuss what it means to implement a feature like DCN in a commercial database system.

One of the major design considerations of DCN was that it be as non-intrusive as possible. For example, we decided not to expose DCN through SQL syntax to avoid that existing applications have to be rewritten. Rather, the subscription request can be passed as a protocol annotation. Another desideratum is that the behavior of an application should not be altered if DCN is activated. This means that DCN cannot not simply raise errors and abort processing of a workload but has to fail gracefully. At the same time, delivery guarantees have to be met.

Analogous to the two principal workflows, we distinguish two categories of additional logic that is required: (1) pertaining to subscription requests and (2) operations that might invalidate query results.

4.1 Invalid Subscription Requests

Not all SQL statements qualify as a subscription request. For example, queries which reference non-deterministic functions such as the random number generator `rand()` cannot be verified by a change detection mechanism. Furthermore, the complexity of the query might be such that change detection for incremental maintenance is unable to determine the changes. In these cases, DCN rejects the subscription request at query time and, instead of setting up the template and parameter tables, sends a notification to the client right away, notifying the subscriber about the failed request. The set of queries which are supported by DCN is tightly coupled to the set of expressions for which materialized views can be generated.

4.2 Invalidation through non-DML Statements

While the dealing with invalid subscription requests could leverage much of the existing technology for materialized views, taking into account all components which might affect query results has proven more difficult.

Obviously, DCN needs to monitor DDL operations on objects for which subscriptions are registered. Operations such as `DROP TABLE` must trigger invalidation of all subscriptions that rely on this object and send appropriate notifications. User DDL operations, such as disabling or even removing an account, are more subtle examples: since the database system impersonates the subscriber when sending the notification, sending notifications on behalf of a user which has been removed from the system may pose a security thread. The correct action is to destroy all pending subscriptions as part of the user DDL. Since DCN can consume significant server resources such as disk space or memory, subscriptions are only valid for a given period of time after

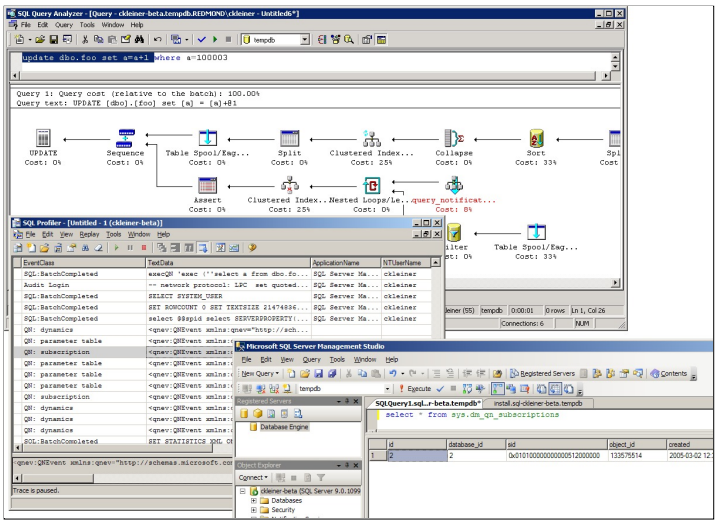

Figure 4: Screenshot with SQL Server tools showing DCN related data: (1) Graphical Showplan of DML query plan, (2) profiler trace, (3) system table with subscriptions

which they are garbage-collected. Support structures such as templates or parameter tables are garbage-collected with a time delay to avoid thrashing if the time-out notification triggered the client to re-query immediately, i.e. parameter tables are only destroyed after having been unused for a grace period of 10 minutes. Finally, DCN needs to be accounted for when it comes to disaster management: In case the server is brought down due to a power shortage or hardware failure, the validity of a subscription can no longer be guaranteed since the database files have not been under the server's control during the down time. Therefore, at server start-up all notifications have to be purged from the system so clients would re-query once the server is up and running again.

5 System Demonstration

This demonstration exercises the complete 3-tier architecture built on Microsoft SQL Server 2005 as database system and ASP.NET as web application server and cache infrastructure. Using an online bookstore as running example, we showcase the implementation of DCN and its integration with the mid-tier layer. We demonstrate subscription processing including query and update plan rewriting, caching query results at the mid-tier, and invalidating cached results through change notifications. Our application illustrates how the query workload on the database server is reduced significantly when enabling DCN as compared to deploying the web service without caching. Another focus of the demonstration will be the tight coupling with the Microsoft SQL Server relational engine and the integration of DCN with the standard toolsets of a commercial database product: DCN can be monitiored and tracked with Microsoft SQL Server Profiler trace events and Dynamic Management Views, i.e. system tables in the database. Tools such as Graphical Showplan let users explore and troubleshoot queries which have been rewritten for change notifications (cf. Figure 4).

6 Summary

In this demonstration we show Database Change Notifications as a core database server primitive to support granular, consistent, and efficient mid-tier caching. Integrating this functionality in the database server provides robustness and efficiency that cannot be achieved through a client-based implementation outside the database.

Our demo application also shows the effective use of various Microsoft technologies working together: DCN for subscription and notification of changes; Service Broker for reliable routing and delivery of messages; and ASP.NET for web page caching in the mid-tier.

References

[1] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View Invalidation for Dynamic Content Caching in Multitiered Architectures. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 562–573, 2002.

[2] P.-Å. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 177–189, 2004.

[3] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 600–611, 2002.