

A Middleware System Which Intelligently Caches Query Results

Louis Degenaro, Arun Iyengar, Ilya Lipkind* and Isabelle Rouvellou
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

Abstract

This paper describes how caching was used to improve performance in the Accessible Business Rules framework (ABR) for IBM's Websphere. ABR is a middleware system which enables application writers to build applications where the time and situation-variable parts of their business logic are externally applied entities known as business rules. The cache significantly reduced the number of queries to remote databases by storing query results. A key problem we faced was how to keep the cache current after database updates. This was solved using data update propagation (DUP). Two enhancements we made to DUP were to employ an update strategy which considers the values of database updates in order to perform intelligent cache invalidations and to automatically compute dependencies using compile and run-time analysis. Our techniques can be applied to other caching environments besides ABR. We show how our cache invalidation strategies perform for applications with database updates having queries similar to those in the Set Query benchmark.

1 Introduction

Caching is critical for improving the performance of many middleware applications. In order for an application to benefit from caching, it must repeatedly use data which is expensive to calculate or fetch. By caching such data, the application only needs to calculate or fetch the data once. Whenever the data is needed after it has been cached, the application can fetch the data from the cache instead of recalculating it or fetching it from a remote location.

This paper describes how caching is used to improve performance in the Accessible Business Rules framework (ABR) for IBM's Websphere. ABR is a middleware system which enables application writers to build applications where the time and situation-variable parts of their business logic are externally applied entities known as business rules.

The techniques we have used for caching in ABR can be applied to other applications as well. The General-Purpose Software cache (GPS cache) which we used is designed to be plugged into

*Author's current affiliation: Courant Institute of Mathematical Sciences, New York University, New York, NY

different applications. The GPS cache has also been successfully deployed in a Web server accelerator. The GPS cache has very efficient code for storing data in memory, on disk, or both. It also has optimized support for invalidating objects based on expiration times and for logging cache transactions.

The GPS cache, as applied in ABR, stores the results of queries ultimately made to a database. A key problem with caching query results is determining which queries are affected by changes that occur to the database. In order to keep caches current after database updates, we use an enhanced version of data update propagation (DUP) [1]. A query result may depend on several attributes, and these dependency relationships are represented by an *object dependence graph* (ODG).

DUP has been previously used to cache dynamic Web data. We made two key innovations to DUP for caching in ABR. When attributes change, we consider the old and new values of the attributes in order to determine how to update the cache. This *value-aware* update policy is implemented by annotating edges of ODG's with values based on queries.

When DUP was used for caching dynamic Web data, an application program was responsible for generating the ODG. The second key innovation we made to DUP for ABR was to automatically generate ODG's from the queries within an ABR application.

Our techniques for caching queries in ABR can be deployed in other query-based environments as well. This paper examines how our update policies perform under different update rates for queries similar to those used by the Set Query benchmark.

The remainder of the paper is structured as follows. Section 2 presents an overview of the ABR system which utilized our cache. Section 3 describes the general-purpose software cache used to improve the performance of ABR. Section 4 describes the techniques we used to keep cached data current in the presence of updates. Section 5 discusses the performance of our cache update schemes on applications with queries similar to those in the Set Query benchmark. Section 6 discusses related work. Finally, Section 7 summarizes our main results and conclusions.

2 Overview of the Accessible Business Rules Framework (ABR)

The techniques for intelligent query caching described in this paper were developed to improve performance for the Accessible Business Rule framework (ABR), one of the e-business application frameworks available on IBM's Websphere middleware [3]. ABR enables application writers to build applications where the time and situation-variable parts of their business logic are externally

applied entities called business rules. The structure of the application then matches the built in core behavior with variations specified, managed, and applied externally. Customizable services and features (such as web personalization) can also be built on top of ABR. An ABR rule is a persistent object encapsulating code implementing variable behavior as well as a number of attributes defining the business context in which this behavior applies. ABR defines structured exit points from the main application logic which are referred to as *variability points* or *decision points*. The code in decision points selects the particular business logic (rule or rules) to be executed via a query. A query statement is a constraint on the context attributes of the rule objects which reflect the business criteria and context used to select the rules. Some contexts are fixed and represent a static business situation (i.e. not dependent on run-time data). They are captured in ABR with either a simple direct name (e.g. ComputeRateQuote) or a compound name when the context is hierarchical (e.g. Vehicle::isEligible). Some contexts are situational; the correct rules then partly depend on a business context computed or derived at run time (e.g. category of the customer accessing this web page, season of the year). More details on ABR can be found in [15, 7].

Externalizing business rules from the main code has invaluable benefits for the clarity of the application and its ease of maintenance. However, this externalization, as first implemented, had significant overhead which largely resulted from querying. Performance profiling clearly correlated the performance bottleneck with the overhead introduced by querying the persistent store (typically a database) where the rules are stored.

Because business rules do not change very often, though much faster than the core of the applications they are attached to, the benefits of caching the query results in that context were apparent. However, because queries were dependent on multiple objects in nontrivial ways, there was a need for a general cache invalidation mechanism to avoid using stale objects when caching of query results was desired.

Figure 1 depicts a Websphere system including an ABR Rule Server. This configuration contains three cloned instances of the Rule Application Server distributed on two different nodes (i.e, a physical system on which the application server runtime is installed). Multiple server instances are defined and active on a node, and each server is a single, multithreaded process. The three rule server instances are in a Rule Server Group (i.e. a named collection of server instances) and are all connected to the same database (IBM's DB2). The Rule Server Group is defined for availability and performance reasons, but client applications see only a a single logical server instance. The figure shows two types of clients: a rule administrator client which would typically be within the

same Intranet as the Rule Server and other browser-based clients accessing different rule-based Web applications. Although not shown in the figure, clients may also have caches.

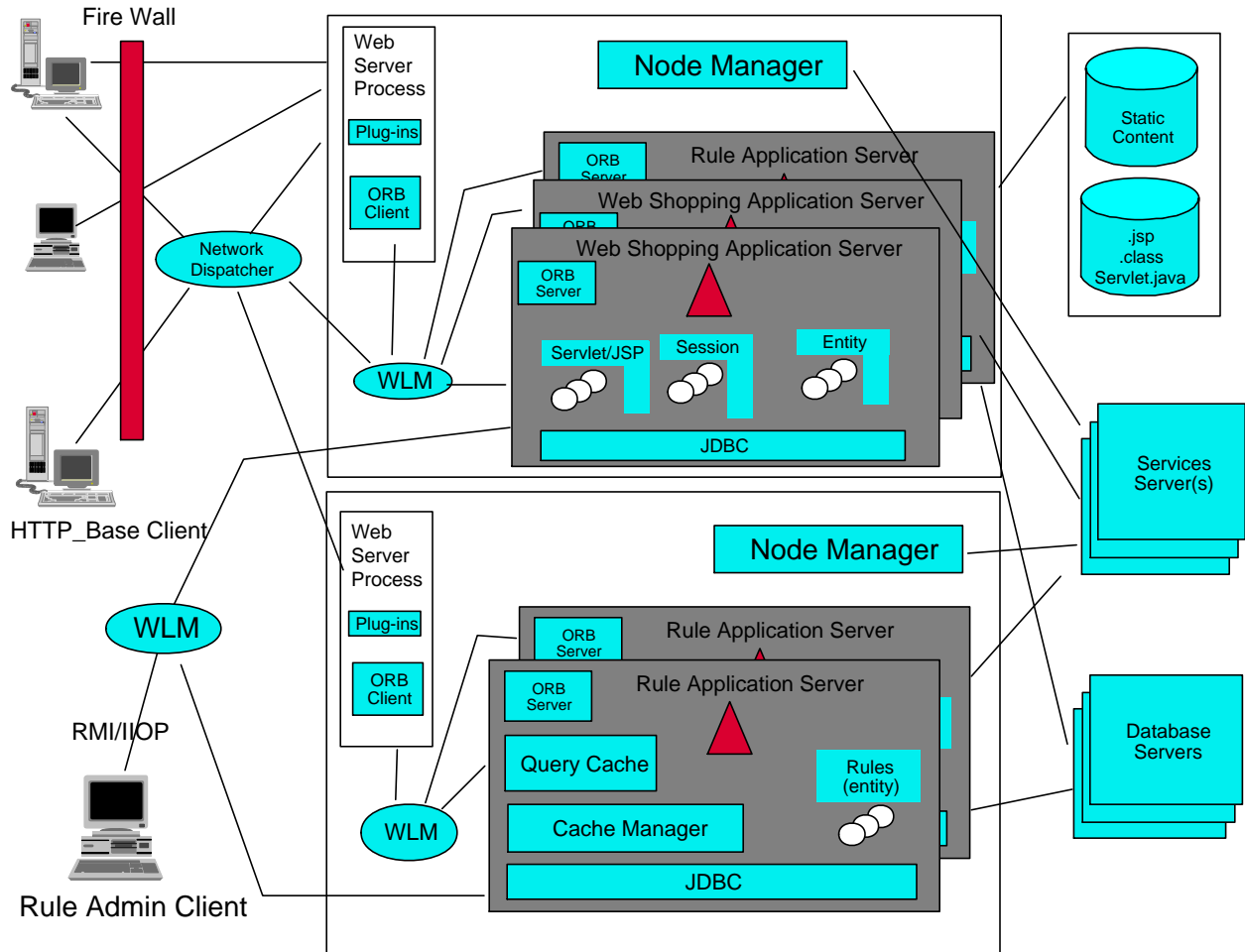


Figure 1: The ABR architecture.

3 Caching Software Used by ABR

Caching is an extremely useful technique for improving performance in a variety of software applications. We have used caching to significantly improve performance of ABR and numerous Web applications [1, 13, 9]. In order to achieve performance improvements for multiple applications, we have implemented a *General-Purpose Software cache (GPS cache)*. The GPS cache is a POSIX-compliant [14] C++ library. In order to use the GPS cache, an application uses the GPS cache application program interface (API) to manage the cache and is linked with the GPS cache library. Applications add, delete, and query the cache via a set of API function calls. The GPS cache has

been used to improve performance in ABR and in a Web server accelerator [13].

Software designers without something similar to a GPS cache would have to write their own caches for each application requiring one. This could require considerable extra effort. A hastily designed cache is likely to have inferior performance and less functionality than the GPS cache.

We would like to see the GPS cache or something similar to it become widely available in operating systems or software development libraries. This would allow software developers to easily improve performance via caching. Ideally, a common API for invoking caching functions could be agreed upon.

The GPS cache can be configured to store data in memory, on disk, or both. A common mode of operation is to use disk as secondary storage for cached data which cannot fit in memory due to the presence of other cached data which are accessed more frequently.

Cached objects can have expiration times associated with them after which they are no longer valid. The GPS cache implements an efficient algorithm for invalidating objects based on expiration times.

The GPS cache implements the data update propagation (DUP) algorithm for invalidating cached objects [1, 10]. This feature is useful for keeping complex data current in the cache. DUP has proved to be extremely useful for caching dynamic Web pages. Future sections of this paper describe how DUP was used for ABR.

The GPS cache allows cache transactions to be logged in a file. In order to reduce the overhead for logging, the frequency with which buffers containing transaction information are flushed to the file system can be varied. If every transaction record is flushed to disk as soon as it is generated, log files will always be up to date and no logs will be lost if the cache process fails. The overhead for immediately flushing every transaction log is substantial, however. An alternative approach is to accumulate several transaction records in a buffer before flushing the buffer to disk. This approach has lower overhead. If the cache process fails, however, transaction records which have not yet been flushed to disk are lost.

4 Cache Invalidation using Data Update Propagation Algorithm

Data update propagation (DUP) determines how cached data are affected by changes to underlying data which determine the current values of the data. For example, if a cache is storing results from querying databases, a method is needed to determine which query results are affected by updates

to the database. Such a method could synchronize caches with databases so that the caches do not contain stale data. Furthermore, the method should associate cached data with parts of the database in as precise a fashion as possible. Otherwise, objects whose values have not changed may be mistakenly invalidated or updated from a cache after a database change. Such unnecessary updates to caches can increase miss rates and hurt performance.

DUP maintains correspondences between *objects* which are defined as entities which may be cached and *underlying data* which periodically change and affect the values of objects. In the ABR system, the objects being cached are query results and the underlying data are parts of the database. We have also employed DUP for caching Web data in which the objects being cached are Web pages.

The system maintains data dependence information between objects and underlying data. When the system becomes aware of a change to underlying data, it examines the dependence information which it has stored in order to determine which cached objects are affected. Caches use dependency information to determine which objects need to be invalidated or updated as a result of changes to underlying data.

Data dependencies between underlying data and objects are represented by a directed graph known as an *object dependence graph (ODG)*, wherein a vertex usually represents an object or underlying data. An edge from a vertex v to another vertex u denoted (v, u) indicates that a change to v also affects u . Node v is known as the *source* of the edge, while u is known as the *target* of the edge. For example, if node $go2$ in Figure 2 changes, then nodes $go5$ and $go6$ also change. By transitivity, $go7$ also changes.

Edges may optionally have weights associated with them which indicate the importance of data dependencies. In Figure 2, the data dependence from $go1$ to $go5$ is more important than the data dependence from $go2$ to $go5$ because the former edge has a weight which is 5 times the weight of the latter edge. Edge weights can be used to quantitatively determine how obsolete a cached object is. In some cases, it is acceptable to keep around a cached object which is not too obsolete. Retaining slightly obsolete versions of cached objects results in better performance than updating or invalidating an object every time it changes.

For most situations in ABR, the object dependence graph is a *simple object dependence graph* having the following characteristics:

- Each vertex representing underlying data does not have an incoming edge.

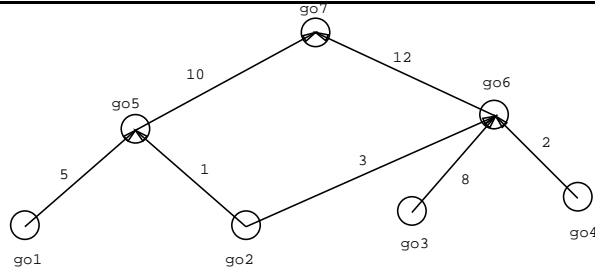


Figure 2: An object dependence graph (ODG). Weights are correlated with the importance of data dependencies.

- Each vertex representing an object does not have an outgoing edge.
- All vertices in the graph correspond to underlying data (nodes with no incoming edges) or objects (nodes with no outgoing edges).
- None of the edges have weights associated with them.

Figure 3 depicts a simple ODG.

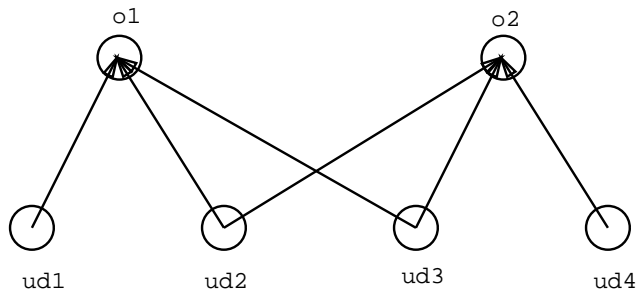


Figure 3: A simple object dependence graph.

4.1 Constructing ODG's from Queries

We now give an example of constructing an ODG from a query. The query:

```
select A where A.x > 2 and A.x < 9 and A.z = B.y
```

would generate the ODG shown in Figure 4. Each class.attribute term in the query has a corresponding vertex in the ODG. Edges are drawn from each class.attribute vertex to the query result objects it affects.

A key enhancement that we have used in applying DUP to ABR over previous implementations of DUP is the use of annotations of graph edges in order to achieve a *value-aware* invalidation scheme. For example, the annotation of the edge originating from the $A.x$ vertex indicates that if $A.x$ changes, query result $Q1$ would only be affected if either:

1. $A.x$ was previously between 2 and 9 and is no longer in this range
2. $A.x$ was previously not between 2 and 9 but now is in this range

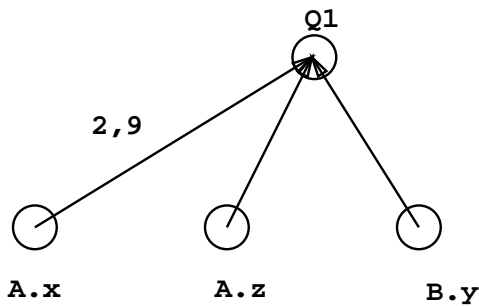


Figure 4: The ODG resulting from the query *select A where A.x > 2 and A.x < 9 and A.z = B.y*. The annotation of the edge from $A.x$ to $Q1$ allows more selective invalidations.

There are no annotations of edges originating from $A.z$ and $B.y$. This indicates that value-aware invalidation is not being used for these edges and any change to $A.z$ or $B.y$ might affect the value of $Q1$.

ODG's constructed in this fashion are stored in the GPS cache. The GPS cache has an efficient algorithm for traversing ODG's in order to locate all cached query results affected by changes to underlying data.

4.2 DUP Implementation in ABR

As mentioned in Section 2, ABR applications contain decision points which query the ABR Rule Server for the collection of rules that presently apply. Once retrieved, these rules are “fired”, resulting in the appropriate application behavior at this decision point (note that the “variable behaviors” encapsulated in rules range from constraint checking to derivation of a particular value). Ordinarily, such a query would be pushed down to the persistent store (typically a database). Caching improves performance by avoiding the persistent store in many cases.

We now describe how ABR handles queries extracted from a particular rule-based Web shopping application. This application serves pages to browsers. The pages are created with “holes” which

get filled with dynamic content (e.g. URLs or images) according to business rules that are managed externally. The selection of the content is typically situational (e.g. category of the customer accessing this web page, season of the year). We focus below on a particular hole which is to be filled with a product promotion based upon classification of the shopper's status into Gold, Silver, or Bronze. The code interacting with the ABR server issues two queries. The first query, Q1, retrieves classifier rules which when fired return the classification(s) of the current shopper. The second query, Q2, retrieves promotion content rules defined for the current shopper classification as established when firing the classifier rules returned by Q1.

Q1 and Q2 are shown below:

```
Q1 :
  SELECT * FROM RULEUSETABLE WHERE
           CONTEXTID      LIKE 'customerLevel'
  AND     TYPE            LIKE 'classifier'
  AND     COMPLETIONSTATUS LIKE 'ready'
```

```
Q2(userClassification):
  SELECT * FROM RULEUSETABLE WHERE
           CONTEXTID      LIKE 'promotion'
  AND     CLASSIFICATION  LIKE $1
  AND     TYPE            LIKE 'situational'
  AND     COMPLETIONSTATUS LIKE 'ready'
```

Q1 is a static SQL statement for which the ODG is completely generated at compile time. Q2 is a parametrized statement (\$1 represents a variable whose value isn't known statically) for which all of the ODG, except for annotations of edges dependent on parameters, is generated at compile time. Such annotations are determined at run-time. In our example, the run-time work is limited to setting a parameter and thus introduces minimal overhead. The ODG generated from Q1 and Q2 are shown in Figure 5.

The ABR Server API offers 23 queries for use in ABR-enabled applications. These queries are constraints on all or a subset of the 13 attributes of the rule. All but one of the queries are similar to the ones shown above and are therefore either static or parametrized. ODG's for dynamic SQL statements can be created from scratch by the system at run-time.

Selective invalidation of the cache is triggered by invalidation code in the attribute setter, creation and deletion methods. The code is automatically generated at compile time. Invalidation tokens need to be partially computed at run-time in order to allow value-aware invalidation. Just as with parametrized queries, much of the token is known statically and the run-time computation

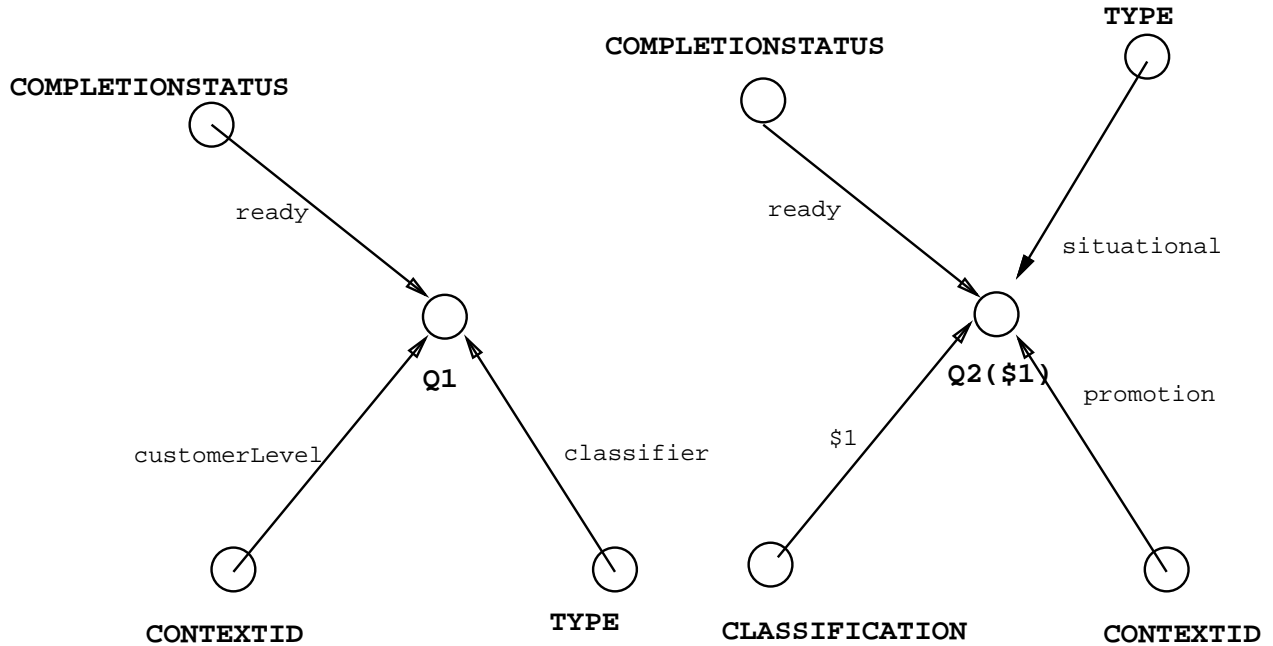


Figure 5: The ODG generated statically from the two ABR queries given in the text. The actual value of “\$1” is determined at run-time.

just involves determining a parameter, a process which usually introduces little overhead.

Figure 6 shows the invalidation code in a set method. In this example, if a new classifier establishing customer level is introduced (e.g. a Platinum level is created, resulting in new classifier rules of ContextId “CustomerLevel”), Q1 will be invalidated. Cached Q2 query results corresponding to the old classification are still valid and don’t need not to be invalidated.

Figure 7 shows how the ABR caching system works for two typical scenarios. The first one is initiated by a client of a rule-enabled application; the second one is initiated by a rule administration client.

In the first scenario, a client invokes the Web Shopping application which runs until it encounters an ABR dynamic content decision point. This triggers a (1) find to be requested for a set of rules to classify the current situation. The query processor attempts a (2) cache lookup to retrieve the requested (3) result from cache and returns it to continue the second phase of the decision point processing for displaying dynamic content based upon classification. Each individual RuleUse returned in the query result is interrogated via (7) get for desired attributes. Then each RuleUse is fired to produce the current classification, which is then used to locate and fire more RuleUses to

```

void RuleUse.setContextId(String inContextId)
{
// Cache begin invalidate

if (!contextId.equals(inContextId)) {
    cache.invalidate("RuleUse.contextId", inContextId);
}

// Cache end invalidate

contextId = inContextId;
...
}

```

Figure 6: The invalidation code in a set method.

Cached Query Results System Overview

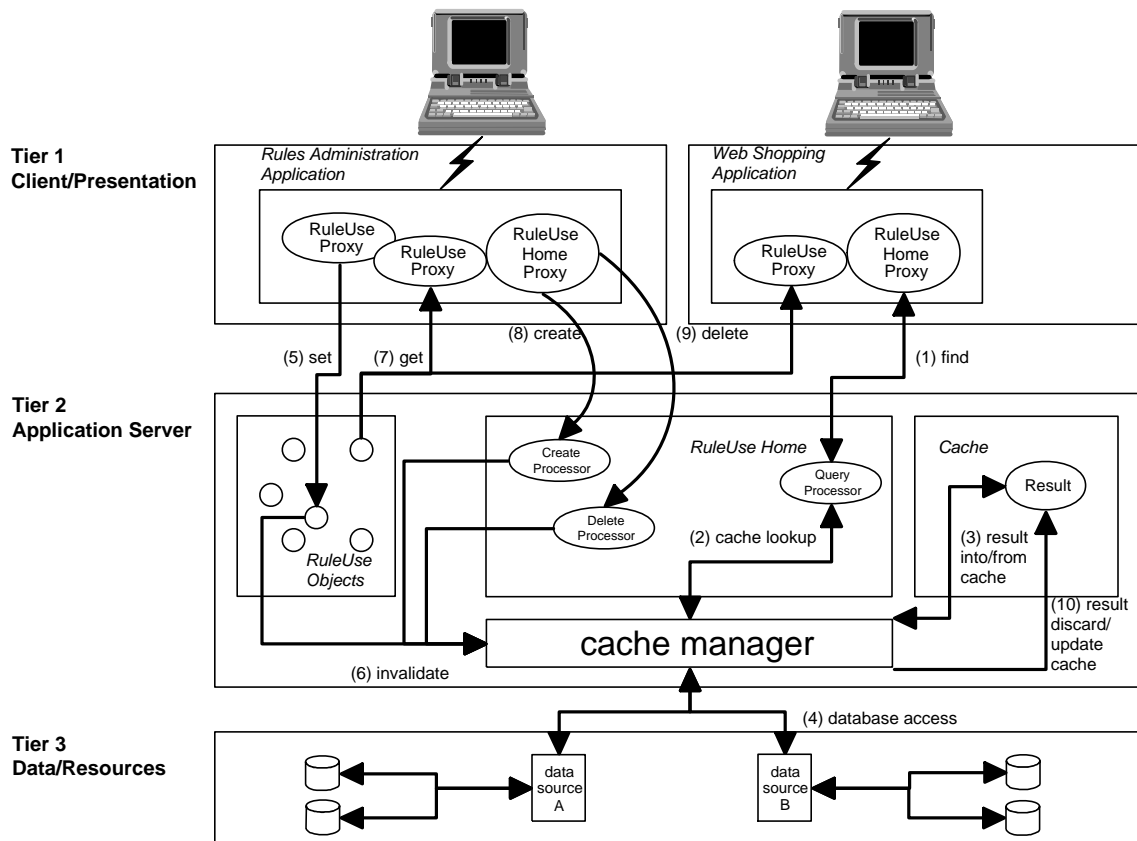


Figure 7: Typical interaction sequences with the ABR caching system.

render the dynamic content. If not found in the cache, then the result is obtained via (4) database access, followed by caching of (3) the result, which causes the appropriate ODG to be constructed. Finally, the result is returned, and processing continues as above. Future requests for the same result will be obtained from the cache until the result is invalidated.

In the second scenario, a rule administration client has already performed one or more queries to obtain collections of RuleUses. Now the administrator decides to change one of the RuleUse attributes via (5) set attribute. This action causes (6) invalidate to occur which will precipitate (10) result discard or update of zero or more dependent query results from the cache, as prescribed by the ODG. A similar invalidation sequence occurs when the administrator decides to (8) create or (9) delete one or more RuleUses.

5 Performance of Query Caching Techniques

Our caching techniques significantly improve the performance of ABR. We do not yet have traces to show the degree to which our techniques improve performance on real applications. We hope to have these soon.

DUP with value-aware invalidations can be applied to a variety of query caching environments and not just ABR. In order for DUP to show significant improvement over conventional caching methods, some query results must change over time. Our approach is particularly important for *set queries* which need to refer to data from a potentially large set of table rows for an answer. Such queries are common in document searching, direct marketing, and decision support.

To determine the quantitative benefits of using our caching techniques, we have run a series of experiments designed to evaluate cache hit rates under different workload scenarios using the Set Query benchmark [8]. This benchmark emphasizes set queries and is designed to model queries encountered in document searching, direct marketing, and decision support. The benchmark includes nine different types of queries. All of them are run against a single table of a million entries that has thirteen different attributes. Each attribute spans a different set of values ranging from 2 to 1,000,000 unique values. Queries involve multiple attributes and ranges of values. A complete list of the queries are contained in the Appendix.

The original benchmark designed to test the performance of the database server did not contain any updates. Since we wanted to show how caches that use our invalidation scheme perform under different update rates, we introduced updates into the mix of transactions.

In general, three types of events can potentially invalidate the results of a query. They include change in the attribute value of a particular object, object creation, and object deletion. From the perspective of the invalidation scheme, object creation and deletion are equivalent to resetting all of the object’s attributes. We varied two factors: the percentage of update transactions in the total mix and the percentage of attributes updated per update transaction. The attributes to update were chosen uniformly from the set of all attributes, while the update value was chosen uniformly from the full range of possible values for a specific attribute.

The experiments were conducted for three different invalidation policies. The first policy (Policy I) invalidated all cached data after any update. Policy II used the basic DUP algorithm described in [1] to invalidate query results. We refer to this policy as being *value-unaware* because it uses only object dependency information without considering the values involved in the update. Policy III, the *value-aware* policy, uses the enhanced DUP algorithm with edge annotations on the ODG (Figure 4).

We now give an illustrative example of how Policies II and III work for a query of type Q3A. Generalization of this example to the other query types is straightforward. Query Q3A is of the form:

```
Q3A: SELECT SUM(K1K) FROM BENCH
      WHERE KSEQ BETWEEN 400000 AND 500000 AND KN = 3;
```

For each $KN \in \{K100K, \dots, K4\}$. Figure 8 shows the ODG for the case where $KN = K100K$. Using Policy II, any update to KSEQ or K100K would cause the cached query result to be invalidated. Using Policy III, the cached query result would only be invalidated if:

1. An update to KSEQ moved it from inside the range of 40000 and 50000 to outside this range or vice versa.
2. An update to K100K changed it from 3 to something else or vice versa.

Figure 9 shows the hit rates for different types of queries. Queries 1, 2A and 2B involve one or two attributes and testing for specific values which explains the high hit rates, especially for the value-aware invalidation scheme. Queries 3A, 3B, 4A and 4B involve ranges of values for combinations of different attributes, and the performance numbers show that our techniques can be effectively used for range type queries as well. Query 5 returns a count of records which fall in cells of a two-dimensional array, determined by the specific values of each of two fields. For this

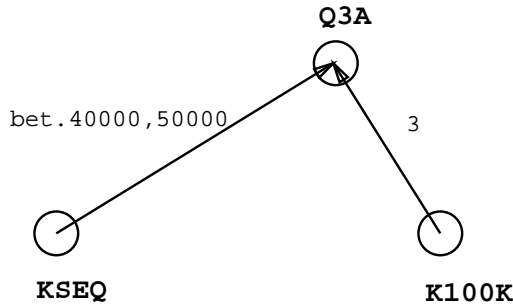


Figure 8: An ODG resulting from a query of type Q3A.

type of query, Policy II and III are equivalent and result in the same hit rates. Finally, queries of type 6 involve relationships between two different attributes ($A.x > A.y$), where both Policy II and III are also equivalent. The difference in hit rate for those queries is explained by the presence of additional conditions of the type exhibited in Queries 1,2,3 and 4.

Overall we see that the value-aware scheme improves performance significantly for single value or range type queries (i.e low selectivity), and both are vastly superior to Policy I.

The experimental results summarized in Figure 10 show the performance of the cache under different update rates. The results demonstrate that that the value-aware policy results in reasonably high hit rates even in the presence of frequent updates.

Figure 11 show the effect of the percentage of attributes modified per update transaction. Here the benefits of using value-aware invalidation increase with the proportion of attributes being updated per transaction.

Figure 12 shows the effect of hot spots. For the data plotted in this figure, 80% of the accesses were uniformly distributed among 20% of the data. The other 20% of accesses were uniformly distributed among the remaining 80% of the data. Updates were uniformly distributed. Only one bar is shown for Policy I because cache hit rates didn't vary much in the presence of hot spots. Policy II and III achieve more significant performance gains in the presence of hot spots than when accesses are uniformly distributed. The advantages of these policies increase with the update rate.

In our system, the cache runs on one machine so the number of invalidations per update does not affect the performance of the system in any significant way. However, distributed caches running on clustered servers or even clients might require some coherence traffic for invalidations. Figure 13 shows the number of invalidations per transaction for Policies II and III. Under Policy I, we assume

Performance for different Query types (update rate fixed at 2%)

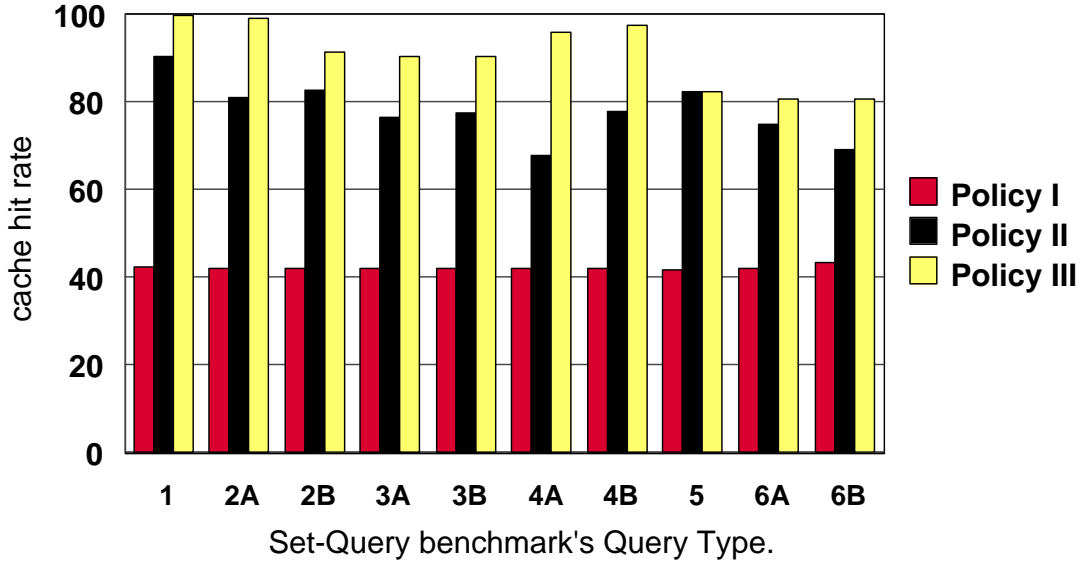


Figure 9: Cache hit rates for different types of queries. Two percent of the transactions are updates. Each update transaction modifies one attribute.

Performance for different update rates (update size fixed at 15%)

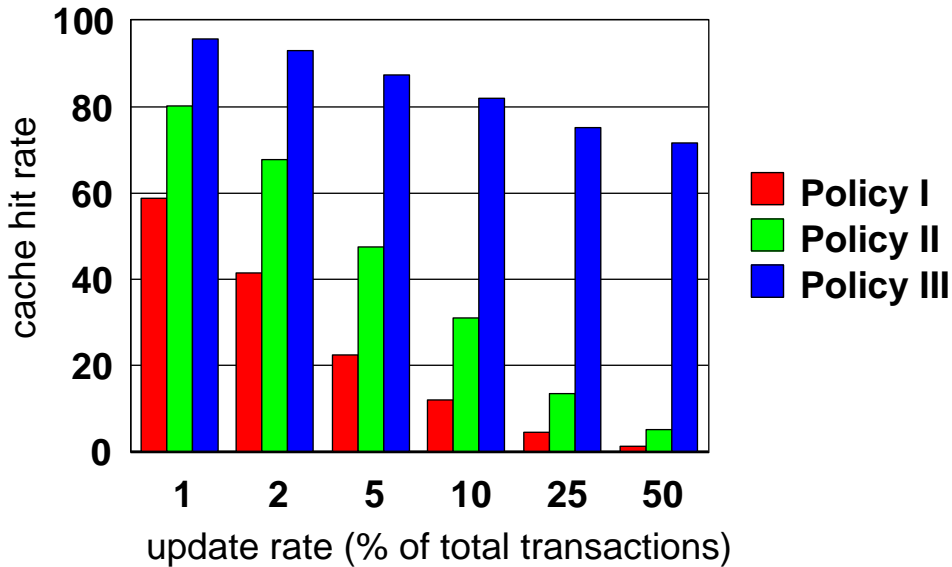


Figure 10: Cache hit rates for different update rates. Update rates are expressed as the percentage of transactions which are updates. Each update transaction modifies two attributes.

Performance for different update sizes (update rate fixed at 2%)

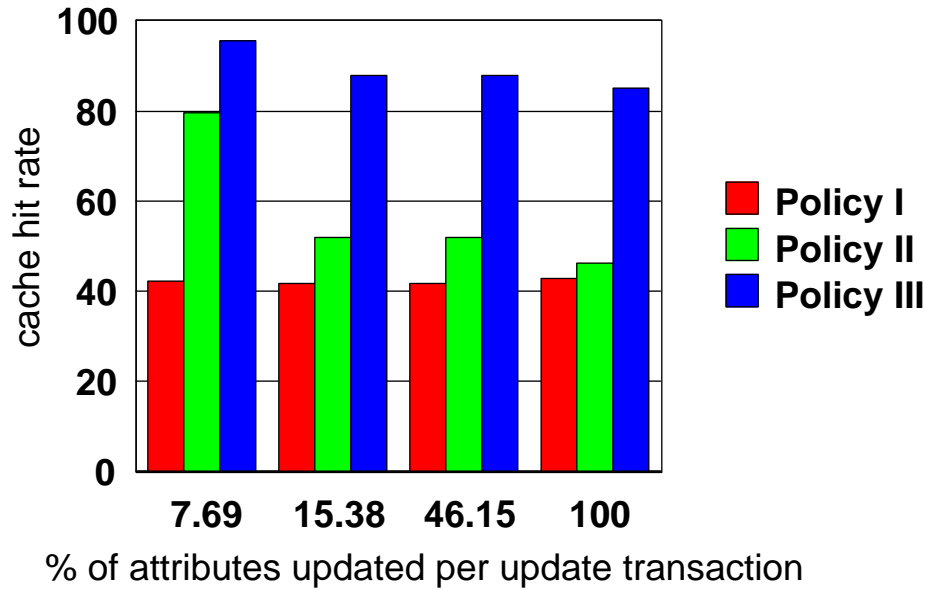


Figure 11: Cache hit rates as a function of attributes updated per update transaction. Two percent of the transactions are updates.

that the cache can be purged completely in a single instruction, so the number of invalidations doesn't affect the coherence traffic. The average number of invalidations per transaction for the two invalidation schemes can be used for predicting the invalidation traffic if a remote cache is used.

5.1 Other Benchmarks

We have also looked at how our cache invalidation techniques affect cache hit rates for the commonly used benchmarks TPC-C and TPC-D [4]. TPC-C models on-line transaction processing applications and has a high percentage of update transactions. We did not see significant improvements in cache hit rates when our methods were applied to TPC-C. TPC-D is a commonly used benchmark which models data warehousing applications. Queries tend to be aggregations of large amounts of data. Updates to such data tend to be done periodically in large batches or not at all. For such situations, having a sophisticated invalidation strategy such as ours is not important.

Hot Spot Effect (80/20)

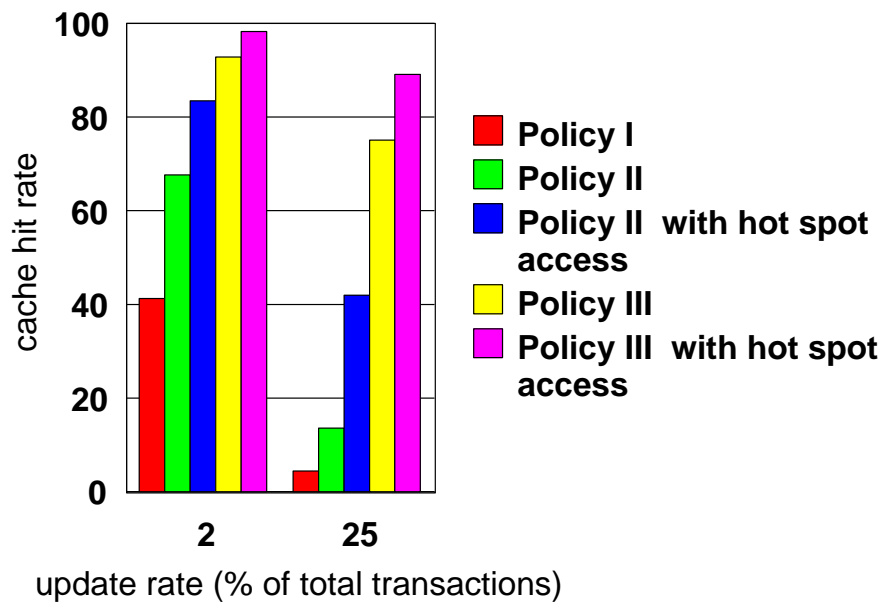


Figure 12: Cache hit rates in the presence of hot spots. Each update transaction modifies two attributes. Since hit rates for Policy I are similar with and without hot spots, only one bar for Policy I is shown.

Number of invalidations for different update rates

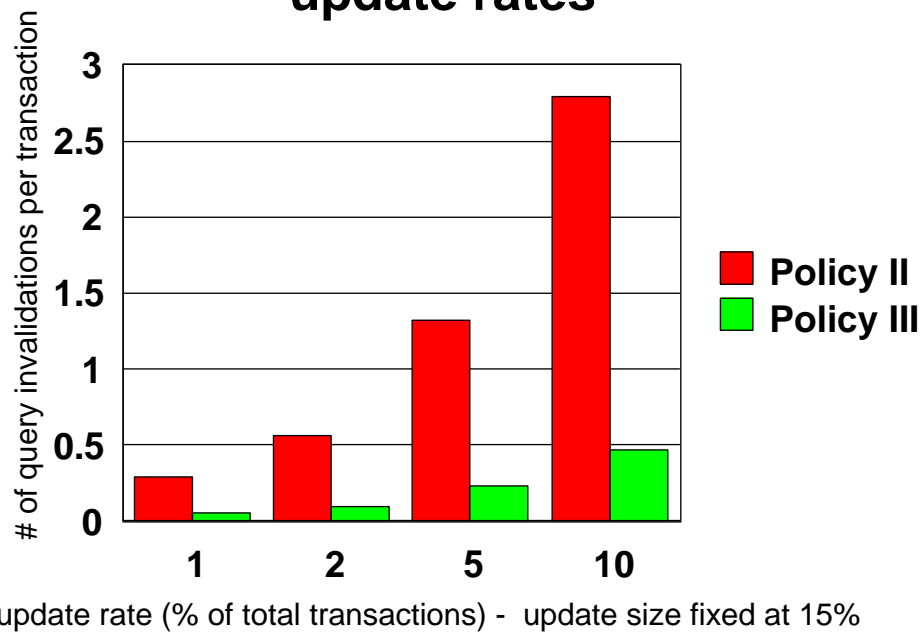


Figure 13: Average number of query invalidations per transaction as a function of update rate. Update rates are expressed as the percentage of transactions which are updates. Each update transaction modifies two attributes.

6 Related Work

The DUP algorithm used to keep caches updated was first deployed for caching dynamic Web data [1]. We have generalized DUP for caching query results which are not necessarily part of Web applications. Previous implementations of DUP require application programs to manually construct object dependence graphs. The DUP implementation for ABR automatically constructs object dependence graphs. We have also extended DUP to employ a value-aware invalidation scheme which improves cache hit rates over previous implementations. A number of previous papers have examined query caching in various contexts [12, 5, 11, 2, 6]. ABR is presented in [15, 7]. Neither of these references discuss caching, however.

7 Summary and Conclusions

We have described how caching is used to improve performance in the Accessible Business Rules framework (ABR) for IBM's Websphere. ABR is a middleware system which enables application writers to build applications where the time and situation-variable parts of their business logic are externally applied entities known as business rules.

The General-Purpose Software cache (GPS cache) used by ABR is designed to be plugged into different applications. The GPS cache, as applied in ABR, stores the results of queries ultimately made to a database. A key problem with caching query results is determining which queries are affected by changes that occur to the database. In order to keep caches current after database updates, we use an enhanced version of data update propagation (DUP). A query result may depend on several attributes, and these dependency relationships are represented by an object dependence graph (ODG).

DUP has been previously used to cache dynamic Web data. We made two key innovations to DUP for caching in ABR. When attributes change, we consider the old and new values of the attributes in order to determine how to update the cache. This value-aware update policy is implemented by annotating edges of ODG's with values based on queries.

When DUP was used for caching dynamic Web data, an application program was responsible for generating the ODG. The second key innovation we made to DUP for ABR was to automatically generate ODG's from the queries within an ABR application.

Our techniques for caching queries in ABR can be deployed in other query-based environments as well. We examined how our update policies perform under different update rates for queries

similar to those used by the Set Query benchmark.

References

- [1] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM'99*, March 1999.
- [2] C. Chen and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Proceedings of the 4th International Conference on Extending Database Technology*, 1994.
- [3] IBM Corporation. IBM Software : Application Development : Component Broker : Overview. <http://www.software.ibm.com/software/ad/cb/>.
- [4] Transaction Processing Performance Council. Welcome to the TPC Main Page! <http://www.tpc.org/>.
- [5] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proceedings of the 22nd VLDB Conference*, 1996.
- [6] A. Delis and N. Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *Proceedings of the 18th VLDB Conference*, 1992.
- [7] D. Ehnebuske, B. Mc Kee, I. Rouvellou, and I. Simmonds. Business Objects and Business Rules. In *Proceedings of the OOPSLA '97 Business Object Workshop*, 1997.
- [8] J. Gray and R. Cattell. *The Benchmark Handbook*. Morgan Kaufmann Publishers, Inc., second edition, 1993.
- [9] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [10] A. Iyengar and J. Challenger. Data Update Propagation: A Method for Determining How Changes to Underlying Data Affect Cached Objects on the Web. Technical Report RC 21093(94368), IBM Research Division, Yorktown Heights, NY, February 1998.
- [11] A. Keller and J. Basu. A Predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5:35–47, 1996.
- [12] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of ACM SIGMOD*, 1999.
- [13] E. Levy, A. Iyengar, J. Song, and D. Dias. Design and Performance of a Web Server Accelerator. In *Proceedings of IEEE INFOCOM'99*, March 1999.
- [14] D. Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, 1991.
- [15] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske, and B. Mc Kee. A Framework to Extend Business Objects with Business Rules. In *Proceedings of OOPSLA*, 1999.

A Appendix: Queries from the Set Query Benchmark

- A COUNT of records with a single exact match condition, known as query Q1:

```
Q1: SELECT COUNT(*) FROM BENCH
     WHERE KN = 2;
```

(Here and in later queries, KN stands for any member of a set of columns. Here,

$$KN \in \{KSEQ, K100K, \dots, K4, K2\}.$$

The measurements are reported separately for each of these cases.)

- A COUNT of records from a conjunction of two exact match condition, query Q2A:

```
Q2A: SELECT COUNT(*) FROM BENCH
      WHERE K2 = 2 AND KN = 3;
```

For each $KN \in \{KSEQ, K100K, \dots, K4, K2\}$

or an AND of an exact match with a negation of an exact match condition: query Q2B:

```
Q2B: SELECT COUNT(*) FROM BENCH
      WHERE K2 = 2 AND NOT KN = 3;
```

For each $KN \in \{KSEQ, K100K, \dots, K4\}$

- A retrieval of data (not counts) given constraints of three conditions, including range conditions, (Q4A), or constraints of five conditions, (Q4B).

```
Q4: SELECT KSEQ, K500K FROM BENCH
     WHERE constraint with (3 or 5) conditions ;
```

- A query where a SUM of column K1K values is retrieved with two qualifying clauses restricting the selection.

```
Q3A: SELECT SUM(K1K) FROM BENCH
      WHERE KSEQ BETWEEN 400000 AND 500000 AND KN = 3;
```

For each $KN \in \{K100K, \dots, K4\}$

- In addition, Query Q3B captures a slightly more realistic (but less intuitive) OR of several ranges corresponding to a restriction of Zip-codes:

```
Q3B: SELECT SUM(K1K) FROM BENCH
      WHERE (KSEQ BETWEEN 400000 AND 410000
             OR KSEQ BETWEEN 420000 AND 430000
             OR KSEQ BETWEEN 440000 AND 450000
             OR KSEQ BETWEEN 460000 AND 470000
             OR KSEQ BETWEEN 480000 AND 500000)
      AND KN = 3;
```

For each $KN \in \{K100K, \dots, K4\}$

The SUM aggregate in queries Q3A and Q3B requires actual retrieval of up to 25,000 records, since it cannot be resolved in index by current commercial database indexing methods; thus, a large data retrieval is assured.

- A query which returns counts of records which fall in cells of a 2-dimensional array, determined by the specific values of each of two fields.

```
Q5: SELECT KN1, KN2, COUNT(*) FROM BENCH
      GROUP BY KN1,KN2;
```

For each $(KN1, KN2) \in \{(K2, K100), (K10, K25), (K10, K25)\}$

- Queries Q6A and Q6B exercise the join functionality that would be needed when data from two or more records in different tables must be combined.

```
Q6A: SELECT COUNT(*) FROM BENCH B1, BENCH B2
      WHERE B1.KN = 49 AND B1.K250K = B2.K500K;
```

For each $KN \in \{K100K, K40K, K10K, K1K, K100\}$

```
Q6B: SELECT B1.KSEQ, B2.KSEQ FROM BENCH B1, BENCH B2
      WHERE B1.KN = 99
      AND B1.K250K = B2.K500K
      AND B2.K25 = 19;
```

For each $KN \in \{K40K, K10K, K1K, K100\}$