Holistic Application Analysis for Update-Independence

Charles Garrod
charlie@cs.cmu.edu
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213

Amit Manjhi
manjhi+@cs.cmu.edu
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213

Bruce Maggs
bmm@cs.cmu.edu
Carnegie Mellon University
Akamai Technologies

Todd Mowry
tcm@cs.cmu.edu
Carnegie Mellon University
Intel Research Pittsburgh

Anthony Tomasic
tomasic+@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

Current database performance optimizations stop at the border between the database application and the database system, focusing either on improving the performance of just the database system or the application's execution in isolation of the other. We argue that typical database application design enables a more holistic analysis that maintains the relationship between the database and application data. We describe techniques to maintain this relationship and introduce several optimizations to improve the efficiency of Web application execution in a distributed environment. We show that our holistic analysis outperforms traditional non-holistic methods both statically and when used as part of a dynamic, distributed environment for executing Web applications using database caches.

1. INTRODUCTION

A key strength of the paradigm of database management systems (DMBSs) is that it encapsulates all data management tasks into a single infrastructure, separating those tasks from the applications where the data is used. Although this encapsulation is ideal from the perspective of good software engineering and programmatic design, it can cause inefficient performance because relationships among data are not maintained between the data's storage and use. Although some research has attempted to erase the barrier between the application and DBMS, most work to improve system performance focuses on just the DBMS or the database application individually, maintaining the separation imposed by the DBMS paradigm.

In many database applications the interactions between the application and DBMS are clearly specified at compile-time. This is especially true for Web applications, in which database requests are often written as templates with parameters that are filled in at run-time. For many Web applications and database request templates there is a clear relationship between the DBMS data and application data. This fact enables a holistic analysis of the DBMS and the database application: the clear specification of database interactions in an application enable us to maintain the relationships between the DBMS data and application data. Maintaining these relationships allows the automated holistic optimization of the application and DBMS, while still maintaining their programmatic separation.

As an example of the type of data relationships that can be maintained, consider a column within a relation that is specified to contain unique values. Values retrieved from this column are then known to be distinct, which can lead to better optimization of the application's execution. Here we study only a very narrow use of these cross-boundary data relationships: we use them to better determine when the execution of a Web application is not affected by a database update, which we call the *application-update depedence* problem. This problem is especially important for distributed settings where the application is executing at a remote server: if the application is not affected by an update, then the remote server does not need to be notified of the update.

In this paper we describe techniques to maintain the relationship between the DBMS data and application data. We then apply these techniques to the problem of correctly executing Web applications while using database query result caches, showing that our holistic techniques outperform traditional techniques that focus just on the database requests or application alone. Our overall contributions are as follows:

- We introduce static, offline techniques that maintain simple known relationships between DBMS and application data.
- We apply the above techniques to the problem of executing Web applications using database caches in a distributed environment, describing two optimizations that cannot be found without holistic analysis.
- We measure the contribution of our holistic analysis both statically and when used as part of a dynamic, distributed environment that executes Web applications using database caches.

The remainder of the paper is organized as follows. Section 2 describes work related to the holistic analysis of database applications. Section 3 describes our key contribution, showing how we maintain the relationship between database and application data and describing two cases for which holistic analysis can determine that a Web application is not affected by a database update when traditional analyses cannot make this determination. Section 4 describes a distributed database caching system we previously developed that we use here to evaluate our analytical techniques, as well as the results of our evaluation.

2. RELATED WORK

The problem of mapping application data to DBMS data has long been a data management problem. Early approaches such as object-oriented databases and persistent programming languages failed to gain wide acceptance in the database community, and common current practice is still to manually program the database access logic or to use an object-persistence layer (e.g., Hibernate [12] or TopLink [11]) to manage the mapping between application and database data. Recently, Melnik et al. [9] proposed a model in which the programmer explicitly declares a general-purpose mapping between application and database data, independent of the programming model and DBMS used. Our work does not extend the state-of-the-art in developing an application-DBMS data mapping. Instead, our focus is on a novel use of such mappings for holistic performance optimization across the application-DBMS boundary. In our work we exploit only simple, direct mappings between an application's statically-declared database requests and the application data, which we believe are common for Web application database workloads. One key advantage that we have over these approaches is that our mapping analysis is automatic and does not require any additional programmer work.

In an earlier thesis, [8], we use a similar holistic analysis across the application-database boundary to optimize database applications for execution in high-latency distributed environments. There we used holistic analysis to to recompile Web applications to require fewer database requests while generating equivalent output, reducing the average execution time of the applications in high-latency environments. Although both that work and this examine data across the application-DBMS boundary, there we did not explicitly map DBMS data to application data, and our use of the holistic analysis targets different subproblems for executing Web applications in distributed environments.

To test the efficacy of our application-DBMS data mapping we apply our techniques to the the application-update dependence problem, the problem of determining whether an application using a database query result cache is affected by a database update. For a database query result cache, the problem of whether a given cached query result is affected by a particular update is equivalent to the well-studied query-update independence problem. While this problem is undecidable in general, it can be solved for many common types of query-update pairs [1, 5, 7]. Blakeley et al. [1] and Elkan [5] gave efficient methods for determining query-update independence that relied upon showing that the query's result was not derived from data affected upon the update. Levy and Sagiv later gave a more general method of proving independence, reducing the problem of independence to the problem of determining equivalence of datalog programs [7]. The application-update dependence problem is similar to the query-update dependence problem but differs in two key respects. First, a database application often does not specify the exact queries and updates it executes but instead specifies query and update templates which define a range of requests it might execute. The application-update independence problem therefore often reduces to the question of whether some set of database queries is independent of some set of updates. Second, even if an application uses a database query affected by some update, the application might not use the affected data and therefore still be independent of the update.

In [10] we described the notion of dependence analysis for query-update templates in the context of database cache consistency management, but did not explore template dependence analysis formally or extend our work to the notion of application-update independence.

Finally, Challenger et al. previously studied the problem of whether a dynamic Web application was affected by a given database update [4], and several projects extended this idea to identifying the dynamic content fragments affected by particular updates [3, 2]. All of this work uses a very similar notion of application-update dependence that we use here. A key difference between their work and ours, however, is that nobody has previously studied the database application and its associated database requests holistically. By analyzing the relationship between application and database data and maintaining database metadata across the application-database boundary, our analytical techniques can prove application-update independence in some cases where their analyses cannot determine the application-update relationship.

3. HOLISTIC APPLICATION-UPDATE DEPENDENCE ANALYSIS

To show that a database application is independent of a given update we extend the ideas of Blakeley et al. [1] and Elkan [5], showing that all data used by the application is independent of the update. The database requests in many applications consist of a small number of static templates within the application code. Typically, each template has a few parameters that are bound at run-time, with the template and its instantiated parameters defining the database request actually executed. The earlier techniques to show query-update independence extend easily to query and update templates. For many common query templates we can simply determine the data from which an instantiation of the template possibly derives, and for update templates we can similarly determine the data which an instantiation of the template possibly affects. By arguments analagous to those of Blakeley et al. and Elkan, the query template is independent of the update template if no possible instantiated query can derive from data affected by a possible instantiated update. We applied this extension of traditional query-update dependence analysis to query-update template pairs in [10].

It is possible, however, for a query-update template pair to be dependent, but for an application using the query template to still be independent of any updates from the update template. This situation can occur in two ways. First, it is possible that although a query-update template pair is dependent, the application will never use template parameters that result in a dependent query-update pair. Second, it is possible that the application does use parameters that create a dependent query-update pair, but the application only uses a subset of the query result data that was not affected by the update. In both of these cases, determining the independence of the application from the update requires consideration of not just the database requests alone but also how the application issues and uses those database requests.

In our application-update dependence analysis, we refine the traditional approach by applying analytical techniques from optimizing compilers to determine how applications use their database request templates. In particular, we correlate application data to the database data from which it

was derived. This allows us to propagate database metadata through the application and determine constraints on the parameter values actually used to instantiate database templates. All of our dependence analysis occurs offline, using just the static application code and its database request templates. We do not examine or modify the run-time execution of the application in any way.

The remainder of this section describes two classes of query-update template pairs for which application-update dependence analysis can determine that the application is not affected by the update but traditional analysis cannot. We first describe the *unused-data* optimization, in which the query-update pair are dependent but the application does not use any affected data. We then describe the *existing-primary-key* optimization, in which our compile time data analysis can determine that a run time parameter will always be an existing primary key for some relation, allowing us to prove that queries actually instantiated from that template will be independent of insertions to that relation.

3.1 The *unused-data* optimization

In some cases database application programmers retrieve more data from the database than they use in their application. This frequently occurs for requests that do not limit the projection of data retrieved from a row (i.e. "SELECT *"), and can also happen when either the application or database schema evolves and the programmer fails to update the database request to reflect the change. If a subsequent database update affects the unused data, that update will affect the database query result but not the application using the result. This section describes how we detect query result data that is unused by the database application, allowing us to infer that the application is not dependent on some updates that affect the query result.

To apply the *unused-data* optimization, we first use the database schema to expand any wildcard characters (*) in the projection clause to all the fields that are retrieved from the database by the query. We then examine the application's use of the query result set, marking each field in the query result as either used or unused by the application. Applications typically retrieve query result data using either the field name or the numerical index of the data they wish to access. For example, after executing the query "SELECT name, salary FROM emp WHERE id = 42" an application could retrieve the salary using either *getInt("salary")* or *getInt(2)*, so this process is usually straightforward. Sometimes our offline analysis can not determine which field of the result is being accessed, typically when the field is chosen using a variable set at runtime. In such cases we conservatively mark all fields as used to ensure the correctness of the optimization. We then remove any unused fields from the query and use our modified query in traditional query-update dependence analysis with each other update in the application, yielding a potentially finer analysis than was possible before. Note that we use our modified query only in our offline dependence analysis, and do not modify the query that is actually executed by the application at runtime. We do this so that we do not affect the performance characteristics of the application.

3.2 The *existing-primary-key* optimization

The *exisiting-primary-key* optimization applies in situations where we can determine that a query result retrieves data based only on an existing primary key for some relation. In such a case, we then know that the query result is unaffected by insertions to that relation, since primary keys are guaranteed to be unique.

For example, let *emp(id, name, salary)* be a relation with primary key *id*. Consider the following Java code snippet:

```
try {
     PreparedStatement namePs =
       connection.prepareStatement(
       "SELECT name FROM emp WHERE id = ?");
     ResultSet idsResult = statement.executeQuery(
       "SELECT id FROM emp WHERE salary > 10");
     while (idsResult.next()) {
         int empId = idsResult.getInt("id");
         namePs.setInt(1, empId)
         ResultSet nameResult =
           namePs.executeQuery();
         ...
     }
} catch (SQLException e) {
}
```

This code first selects the *id* of all employees with salaries greater than 10, and then selects the name of each of employee based on those *id*s. If there are no interevening deletions or modifications to the primary keys of this relation, then each name query will retrieve the name of an employee already in the database, and thus the name query is unaffected by insertions to the employee relation. In this simple example the code snippet could be replaced by the single query "SELECT name FROM emp WHERE salary > 10". In real applications similar interactions might be broken into multiple queries because the business logic is too complex to easily express as a single SQL query, or intervening user input might affect the application's execution.

To apply the *existing-primary-key* optimization we must first determine that a query's selection parameter is in fact a primary key for some relation, and then prove that the primary key has not been modified or otherwise removed from the database. To accomplish the former we track the relationship between application data and database data, using data propagation techniques similar to those used when applying compiler optimizations. Whenever a variable is assigned a value originating from the database, we label the variable with the database metadata characteristics for the data. For example, in the code snippet above the line *int empId = idsResult.getInt("id");* would allow us to label the variable *empId* with the fact that its value originated from the *id* field of the employee relation. This fact would be propagated to when the value was used in the subsequent query (*namePs.setInt(1, empId)*) at which point the metadata can be used to refine the dependence analysis. This data propagation technique is well-suited to typical Java applications, because variables often have limited scope and direct memory access and pointer arithmetic are prohibited.

When the data source can be propagated to a query parameter we then check whether the data source was the primary key for this query's source relation and whether the parameter constrains the query to match just the row for the primary key. If this is the case, we examine all updates within the application to confirm that no update deletes rows from this relation and also that no update modifies the primary key data for the relation (primary keys are typically immutable). This allows us to conclude that the source data

consists of a primary key that still exists in the relation, and thus that this query is unaffected by any insertions to the relation since primary keys are unique. As described, this technique can only succeed when the primary key for a relation is a single column, since we do not track the relationship between different variables in the application. Our technique could be modified to work for simple cases when the relation has a multi-column primary key, although such a modification might require more data to be tracked about each variable during the data propagation step.

4. EVALUATING APPLICATION-UPDATE DEPENDENCE ANALYSIS

To evaluate the effectiveness of application-update dependence analysis we seek to answer two basic questions. First, to what extent do our improvements apply to existing database applications? Second, what performance advantages can be gained from applying them? To answer these questions we apply application-update dependence analysis to three common database application benchmarks and compare the results to traditional query-update dependence analysis. For these applications we show that our methods can determine the independence of many query-update pairs whose relationship cannot be determined from traditional query-update dependence analysis. We then use the results of these dependence analyses as part of the consistency management infrastructure in a distributed database cache, and show that consistency management requires significantly fewer messages with application-update-derived dependencies than with traditional dependence analysis.

This section is organized as follows. Section 4.1 describes the three benchmark applications we use. In Section 4.2 we apply both application-update dependence analysis and traditional query-update dependence analysis to the benchmarks, comparing the number of possible dependencies that can be ruled out with each technique. Section 4.3 describes Ferdinand, a distributed caching environment we built previously, and Section 4.4 examines the performance of application-update dependence analysis when its dependencies are used by Ferdinand's consistency management system.

4.1 Benchmark applications

To evaluate application-update dependence analysis we use three data-intensive Web applications designed to benchmark the performance of Web and database systems. They are the TPC-W bookstore, the RUBiS auction, and the RUBBoS bulletin board benchmarks. Each benchmark simulates the activity of users as they browse a dynamically generated Web site. Each emulated browser sends a request to a Web server, waits for a reply, and then "thinks" for a moment before sending another request. Eventually each emulated browser concludes its session and another emulated browser is simulated by the benchmark. For each request the Web server generates a reply by executing a Java application, which communicates with a back-end database server as needed. These applications mimic the type of interactions expected for common dynamic Web content, and the database requests they generate are typical of many such applications.

The TPC-W bookstore models the types of interactions expected for users of an online retailer. The application consists of 16 update templates and 29 query templates, acting on 10 database tables. We used a large configuration of the bookstore database composed of one million items and 86,400 registered users, with a total database size of 480 MB.

The RUBiS auction models the interactions of a user of an online auction site. It consists of 11 update templates and 28 query templates, acting on 8 database tables. Our version of the auction database contained 33,667 items and 100,000 registered users, totaling 990 MB.

The RUBBoS bulletin board models an interactive online news site like Slashdot. It consists of 13 update templates and 39 query templates on 8 tables. Our bulletin board database contained 6,000 active stories, 60,000 old stories, 213,292 story comments, and 500,000 users, totaling 1.6 GB.

4.2 Static gains of application-update dependence analysis

To evaluate the coverage of application-update dependence analysis we applied both traditional query-update dependence analysis and each of our dependence analysis improvements to each benchmark. For our evaluation we manually applied the optimizations to each benchmark, being careful to include only cases where we are certain that an automated implementation could perform the necessary analysis. Figure 1 shows the results of our comparison.

The bookstore benchmark contains 464 potential query-update template pairs. A coarse table-level dependence analysis determines that all but 85 query-update template pairs are independent. A finer-grained row- and column-level data analysis eliminates an additional 20 pairs, ruling just 65 to be possibly dependent. Of these 65 pairs, our application-update analysis determines that an additional 26 pairs are independent, an improvement of 40% over the traditional query-update dependence analysis. Of these 26 pairs, 15 possible dependencies were eliminated by our *unused-data* analysis: the query-update templates were possibly dependent, but the application did not actually use any of the possibly affected data. The other 11 pairs were ruled independent by our *existing-primary-key* analysis.

The auction benchmark contains 308 query-update template pairs. All but 82 pairs can be ruled independent by table-level analysis, and all but 48 independent by finer-grained analysis. Application-update dependence analysis finds only 7 additional query-update template pairs that are independent, a 14% improvement over the traditional data analyses. For this benchmark, the *unused-data* optimization is completely ineffective – the application always uses all data in a possibly-affected query – and all gains are from the *existing-primary-key* optimization.

The bulletin board benchmark contains 507 query-update template pairs, and all but 88 of these can be ruled independent by table-level data analysis. Row- and column-level data analysis determines that 65 of these pairs may be dependent, and application-level analysis determines that 19 of those 65 pairs are surely independent, a 29% improvement over the traditional analysis. Of these 19, 4 were found by the *unused-data* optimization and 15 were found by the *existing-primary-key* optimization.

These results demonstrate that the effectiveness of the *unused-data* optimization can vary significantly between applications. For the auction benchmark, all queries are written to retrieve specific columns for rows that match some criteria, and all retrieved information is immediately dis-

	bookstore		auction		bulletin board	
Total (no analysis)	464		308		507	
Table-level analysis	85	(82%)	82	(73%)	88	(83%)
Row- and column-level analysis	65	(24)	48	(41)	65	(26)
Row- and column-level plus *unused-data*	50	(23)	48	(0)	61	(6)
Row- and column-level plus *existing-primary-key*	54	(17)	41	(15)	50	(23)
Row- and column-level plus both methods	39	(40)	41	(15)	46	(29)

Figure 1: Number of possibly dependent query-update template pairs for each benchmark application using various methods of dependence analysis. The parenthesized number is the percent improvement over the previous level of analysis (for all forms of application-update dependence analysis, the previous level is "Row- and column- level").

played or used in the auction's business logic. The bookstore and bulletin board benchmarks, however, both sometimes retrieve data that is not used by the application. In all of these cases, all columns of a row (or set of rows) are retrieved but only some columns are used in the subsequent computation – and any possibly affected columns are not used by the application.

For applications like these, the *existing-primary-key* optimization is highly effective. All of these applications contain interactions where a set of rows is retrieved from the database, the application implements some business logic based on the retrieved data, and additional queries are executed using data from the earlier result set. In many of these cases, our analysis can determine that the subsequent query depends only on data previously existing in the database, and therefore that the query is not affected by insertions to the relevent table. We expect interactions like this to be common, and thus expect that the *existing-primary-key* optimization will apply to a wide range of Web applications.

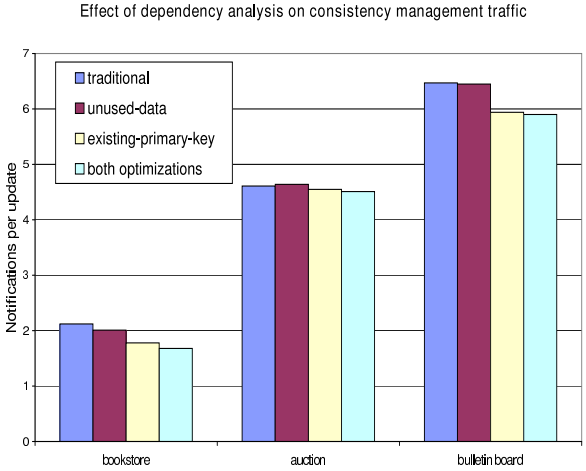

Figure 2: Number of notification messages per update for each benchmark and each dependence analysis.

4.3 The Ferdinand Query Cache

To evaluate the effect of application-update dependence analysis on real systems, we compare the performance of application-update-derived dependencies to traditional query-update dependence analysis when used as part of the consistency management infrastructure of Ferdinand [6], a distributed database caching system we previously developed. This section briefly describes Ferdinand.

Ferdinand is a CDN-like architecture for scaling dynamic Web content. Users connect directly to proxy servers, each of which consists of a Web cache, an application server, and a database query result cache. Each Ferdinand proxy server is a member of a decentralized publish / subscribe infrastructure. When a query result is placed in a Ferdinand proxy cache, the proxy subscribes to topics in the publish / subscribe infrastructure to ensure that the proxy will be notified of any updates that possibly affect the query. When an update is executed at a Ferdinand proxy, notifications are published to any proxy servers containing queries possibly affected by the update; upon receiving an update notification, the proxy server removes from its cache any query results that are possibly affected by the update.

Ferdinand uses a topic-based publish / subscribe system for consistency management. The dependence analysis determines the topics to which a proxy subscribes when it places a query result in its cache and the topics to which a proxy publishes when it executes an update. For a given query the proxy subscribes to a topic for each update template that possibly affects it, unless it is already subscribed to the topic. For some query-update template pairs, the dependence of the pair can be determined from the template parameters at run time even though the dependence is unknown at compile time. In these cases, a subscription might be avoided, or a query might cause a subscription for only updates with the run time parameters that affect the query.

4.4 Dynamic gains of application-update dependence analysis

To evaluate the gains of application-update dependence analysis in an applied setting, we compared the performance of Ferdinand's consistency management system using traditional query-update dependence analysis to the performance when using our methods. For the bookstore and auction benchmarks we executed the application on 8 Ferdinand nodes for 15 minutes at approximately 50% system utilization, starting with warm query result caches, and measured the number of update notifications required by the consistency management system. For the bulletin board benchmark we did the same, but using a Ferdinand system with 12 nodes. The results of this evaluation are in Figure 2.

For both the bookstore and bulletin board benchmarks, the application-update-derived dependencies reduced consistency management traffic by 10% compared to query-update dependence analysis. The improvement for the auc-

tion benchmark is more slight. From the experiments using just the *unused-data* or *existing-primary-key* optimizations, we see that nearly all performance gains come from the *existing-primary-key* optimization. This result is true for the bookstore and bulletin board benchmarks even though the *unused-data* optimization identified many query-update template pairs as independent, when the pairs' relationships could not be determined using traditional dependence analysis or the *existing-primary-key* analysis. This fact indicates that simply measuring the number of dependent query-update template pairs is not sufficient to discern real performance gains, and that any meaningful performance analysis must consider the dynamic execution of the database application. Determining the independence of a frequently executed query or update is much more valuable than determining the independence of a rare database request.

Overall, the value of the *unused-data* optimization is unclear. Its applicability is more variable than that of the *existing-primary-key* optimization, and even when *unused-data* determines that query-update template pairs are independent, for our tested benchmarks those queries and updates seem uncommon in practice. For our benchmarks the *unused-data* optimization most commonly applies to join queries when the application requires all columns from one relation but only several columns from another relation. In such a case a typical programmer might project all columns of the result set (i.e. "`SELECT *`") rather than specify just the needed columns, since all columns are used from one of the relations. If this is the dominant case where the *unused-data* optimization applies, then it is not surprising that its overall effect on performance is slight since join queries are executed less frequently than simpler retrieval and update queries for most typical Web application workloads.

The *existing-primary-key* optimization produced significant gains for all workloads, with the gains relative to its applicability in the workload. We expect this optimization to apply to most Web applications, as it identifies a common data usage scenario: because it applies to simple insertion and row retrieval queries, we expect it to result in similar performance gains for a wide class of applications.

5. CONCLUSIONS

In many environments database requests are executed as part of a client database application. In this paper, we extend the techniques of query-update dependence analysis to instead determine the dependencies between database updates and the applications that execute the database queries. We show how the application and database requests can be analyzed together to infer relationships that cannot otherwise be deterimined using traditional query-update dependence analysis, describing two specific cases where we improve upon earlier dependence analysis. In the first we show how one can sometimes determine that an application is not affected by a database update even though the application uses a query result affected by the update, in cases when the query result is not wholely used by the application. Our second improvement shows how we can determine that some query results are not affected by insertions to the query's relation, by proving that the query result depends only on data already existing in the relation. We then applied our application-update dependence analysis to three benchmark Web applications, showing that our techniques are not merely theoretical but are likely to apply to real database applications. Finally, we executed those Web applications using a distributed query result cache, showing how our application-update dependence analysis required less communication to maintain cache coherence than when traditional query-update dependence analysis was used.

In our latter optimization we introduced a powerful new technique of tracking database metadata across the application-database boundary. Here we used this technique specifically to determine when parameters in a database query originated from the database via an earlier query, allowing us to infer the independence from insertions described above. Tracking the relationship between application data and database data, however, is a new general tool that might yield advancements in many database subfields; we plan to explore this technique further.

6. REFERENCES

[1] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[2] I. Chabbouh and M. Makpangou. Caching dynamic content with automatic fragmentation. In G. Kotsis, D. Taniar, S. Bressan, I. K. Ibrahim, and S. Mokhtar, editors, *iiWAS*, volume 196 of *books@ocg.at*, pages 975–986. Austrian Computer Society, 2005.

[3] J. Challenger, P. Dantzig, A. Iyengar, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Trans. Internet Techn.*, 5(2):359–389, 2005.

[4] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM*, pages 294–303, 1999.

[5] C. Elkan. Independence of logic database queries and updates. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee*, pages 154–160. ACM Press, 1990.

[6] C. Garrod, A. Manjhi, A. Ailamaki, C. Olston, B. Maggs, T. Mowry, and A. Tomasic. Scalable query result caching for web applications. In *VLDB*, 2008.

[7] A. Y. Levy and Y. Sagiv. Queries independent of updates. In R. Agrawal, S. Baker, and D. A. Bell, editors, *VLDB*, pages 171–181. Morgan Kaufmann, 1993.

[8] A. Manjhi. *Increasing the Scalability of Dynamic Web Applications*. PhD thesis, Carnegie Mellon University, Computer Science Department, March 2008.

[9] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *SIGMOD Conference*, pages 461–472. ACM, 2007.

[10] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *CIDR*, pages 56–69, 2005.

[11] Oracle. Oracle Fusion Middleware: Oracle TopLink. `http://www.oracle.com/technology/products/ias/toplink/`.

[12] Red Hat Middleware, LLC. Hibernate: Relational Persistence for Java and .NET. `http://www.hibernate.org/`.