A Program Optimization for Automatic Database Result Caching

Ziv Scully
Carnegie Mellon University, Pittsburgh, PA, USA
zscully@cs.cmu.edu

Adam Chlipala
Massachusetts Institute of Technology CSAIL, Cambridge, MA, USA
adamc@csail.mit.edu

Abstract

Most popular Web applications rely on persistent databases based on languages like SQL for declarative specification of data models and the operations that read and modify them. As applications scale up in user base, they often face challenges responding quickly enough to the high volume of requests. A common aid is *caching* of database results in the application's memory space, taking advantage of program-specific knowledge of which caching schemes are sound and useful, embodied in handwritten modifications that make the program less maintainable. These modifications also require nontrivial reasoning about the read-write dependencies across operations. In this paper, we present a compiler optimization that automatically adds sound SQL caching to Web applications coded in the Ur/Web domain-specific functional language, with no modifications required to source code. We use a custom cache implementation that supports concurrent operations without compromising the transactional semantics of the database abstraction. Through experiments with microbenchmarks and production Ur/Web applications, we show that our optimization in many cases enables an easy doubling or more of an application's throughput, requiring nothing more than passing an extra command-line flag to the compiler.

1. Introduction

Most of today's most popular Web applications are built on top of database systems that provide persistent storage of state. Databases present a high-level view of data, allowing a wide variety of read and write operations through an expressive query language like SQL. Many databases maintain *transactional* semantics, meaning that a database client can ensure that a sequence of queries appears to execute in isolation from any other client's queries, which makes it simple to maintain invariants in concurrent applications. In short, databases abstract away tricky low-level details of data structures, concurrency, and fault tolerance, freeing programmers to focus on application logic without sacrificing much performance.

Database engines have evolved to fit their nontrivial specification extremely well. Nevertheless, in practice, programmers do not implement entire applications using just a database's query language. Instead, applications are generally clients of external databases. For some of these applications, interacting with the database is a performance bottleneck, even with the maturity of today's database systems. Several pathologies are common. Database queries may incur significant latency by themselves, on top of the cost of interprocess communication and network round trips, exacerbated by high load. Applications may also perform complicated postprocessing of query results.

One principled mitigation is *caching of query results*. Today's most-used Web applications often include handwritten code to maintain data structures that cache computations on query results, keyed off of the inputs to those queries. Many frameworks are used widely in this space, including Memcached[1] and Redis[2]. Frameworks handle the data-structure management and persistence, but they leave some thorny challenges for programmers. Certain modifications to the database should *invalidate* some cache entries, but the mapping from modifications to cache keys is entirely application-specific. In mainstream practice, programmers are forced to insert manual invalidations at every update, reasoning through which cache keys should be invalidated. This analysis is difficult enough for a single-threaded server, and it gets even more onerous for a multithreaded server meant to expose transactional semantics. An alternative approach is to present a restricted database interface to most of the program, using caches as part of that interface's implementation. This limits how much manual invalidation analysis is needed, but it also limits the expressiveness of database operations throughout the program.

Past work (Ports et al. 2010) has shown how to implement caching for SQL-based applications *automatically*, using dynamic analysis in a modified database engine, based on trusted annotations that programmers write to clarify the semantics of their database operations. One might prefer to avoid both the runtime overhead and the annotation burden by using static analysis and compiler optimization to instrument a program to use caches in the battle-tested standard way, inferring the code that programmers are asked to write today. To that end, we present **Sqlcache, the first automatic compiler optimization that introduces caching soundly, preserving transactional semantics and requiring no program annotations**.

That specification would be a tall order in most widely used Web-application frameworks, where SQL queries are written as strings that can be constructed in arbitrary ways. To do sound caching, it becomes necessary to employ program analysis to understand the ways in which applications manipulate strings. The general problem is clearly undecidable, as programs may build query strings with sophisticated loops and recursion that need not even respect the natural nesting structure of the SQL grammar. One framework alternative is to use object-relational database interfaces, where the database is presented in a more conventional object-oriented way. In

[1] http://memcached.org/

[2] http://redis.io/

this case, we run squarely into the usual challenges of aliasing for pointers and references to mutable data. Furthermore, to reconstruct the higher-level understanding that underlies sound caching, we need to analyze the loops that connect together object-relational operations, in effect reconstructing (Cheung et al. 2013) high-level concepts like joins that relational databases support natively.

For these reasons, we chose to implement our optimization for Ur/Web (Chlipala 2015b), a domain-specific functional programming language for Web applications. A key distinguishing feature of Ur/Web is that its compiler does parsing and type checking of SQL syntax, guaranteeing that the queries and updates generated by the application follow the database schema. Moreover, queries are constructed as first-class abstract syntax trees, so that even programmatically constructed queries are forced to follow the grammatical structure of SQL. As a result, we need no sophisticated program analysis to find the structure of queries within a program. The most we need is to inline function definitions and partially evaluate query-producing expressions until we arrive at query templates with program variables spliced into positions for SQL literals. We have found that programs in this form are an excellent starting point for a surprisingly simple yet effective program analysis and transformation to support automatic caching. We expect that our approach could also be applied to other languages and frameworks sharing this criterion, like SML# (Ohori and Ueno 2011), Links (Cooper et al. 2006), and LINQ (Meijer et al. 2006).

The core of Sqlcache is a program analysis, demonstrated by example in Section 2, that checks compatibility between queries and updates, but that analysis is joined by a crucial supporting cast of other components. In summary, Sqlcache

- computes conditions for *irrelevance* (Blakeley et al. 1989) between queries and updates, using the analysis to instrument updates with cache invalidations automatically (Section 3);
- employs a cache implementation that supports the required operations in time constant in the number of entries (Section 4.1)...
- ...while maintaining transactional semantics in the face of concurrent accesses (Section 4.2);
- monitors cache activity at runtime, automatically deactivating caches that are invalidated too frequently (Section 4.3);
- heuristically selects program fragments to cache, often realizing opportunities to cache spans of HTML code based on multiple queries, yielding faster responses on cache hits (Section 5);
- and, with no changes to application source code required, improves the throughput of existing Ur/Web applications by factors of two or more (Section 6), even without accounting for network latency.

The current implementation is sound when the optimized application is run on a single server. While this is certainly a limitation (albeit one shared with certain features of Ur/Web itself), our benchmarks demonstrate that a single server with automatic caching is enough to run all but the very busiest Web applications. Our analysis currently handles a substantial but incomplete subset of the SQL features supported by Ur/Web (Section 3.3).

The rest of the paper explains and evaluates our approach, which we try to highlight as appealingly straightforward to understand and implement, thanks in large part to the choice of a source language that presents SQL access at just the right level of abstraction.

Sqlcache is now packaged with the main open-source distribution of Ur/Web[3], and instructions for running our benchmarks are available in the artifact accompanying this paper[4].

[3] `http://www.impredicative.com/ur/`

[4] `https://github.com/vizziv/sqlcache-bench`

```
table paper : { Title : string, FirstAuthor : string,
                Year : int }

fun allPapers () =
    theList <- queryX1 (SELECT *
                        FROM paper
                        ORDER BY paper.Year, paper.Title)
                       (fn r => <xml><tr>
                         <td>{[r.Year]}</td>
                         <td>{[r.Title]}</td>
                         <td>{[r.FirstAuthor]}</td>
                       </tr></xml>);
    return <xml><body>
      <table>
        <tr> <th>Year</th> <th>Title</th>
          <th>First Author</th> </tr>
        {theList}
      </table>
    </body></xml>

fun fromYear year =
    theList <- queryX1 (SELECT paper.Title, paper.FirstAuthor
                        FROM paper
                        WHERE paper.Year = {[year]}
                        ORDER BY paper.Title)
                       (fn r => <xml><tr>
                         <td>{[r.Title]}</td>
                         <td>{[r.FirstAuthor]}</td>
                       </tr></xml>);
    return <xml><body>
      <h2>From the year {[year]}</h2>
      <table>
        <tr> <th>Title</th> <th>First Author</th> </tr>
        {theList}
      </table>
    </body></xml>

fun addPaper title author year =
    dml (INSERT INTO paper(Title, FirstAuthor, Year)
         VALUES ({[title]}, {[author]}, {[year]}));
    return <xml><body>
      OK, I inserted it.
    </body></xml>

fun changeYear title oldYear newYear =
    dml (UPDATE paper
         SET Year = {[newYear]}
         WHERE Title = {[title]} AND Year = {[oldYear]});
    return <xml><body>
      OK, I changed it.
    </body></xml>
```

Figure 1. An example Ur/Web program to optimize

2. Analysis and Transformation by Example

Figure 1 shows an example Ur/Web program that we would like to optimize by inserting caching. The underlying database contains one table, declared in the first line of the program: a table of all papers published to date at POPL, with their titles, first authors, and years of publication. Each of the four functions that follows is callable via Web URLs, and each generates an HTML page to return to the user. The four functions are:

- `allPapers` returns a rendering of the full database table as an HTML table, sorted by year and title.
- `fromYear` builds a similar HTML table just for those papers published in a particular year, which is passed as a parameter to the function.
- `addPaper` inserts a new row into the table.
- `changeYear` adjusts the year of an existing paper, given its title and old year.

The code uses two library functions for accessing the SQL database. Function `queryX1` is for running a query over *one* table (hence the "1") to generate XML (hence the "X"): we run a provided function on every result row, concatenating together the resulting XML fragments. Function `dml` is the primitive for executing an update command for its side effects, and the function is named after SQL's Data Manipulation Language.

Though we do not have space for a full tutorial introduction to the Ur/Web language, we will highlight a property essential to enabling Sqlcache. The program contains *quoted fragments of HTML and SQL code*, including antiquoting (written with curly braces) to inject values from the host programming language. The compiler parses and type-checks these embedded fragments statically and adds appropriate escaping automatically, precluding the possibility of code-injection attacks. The compiler can analyze such fragments nicely as abstract syntax trees instead of unstructured code that produces strings, which has benefits beyond just blocking code injections: it also helps the compiler do static analysis of semantic properties of queries. We need no pointer or string analysis to reconstruct which SQL queries appear in a program. While Ur/Web does allow the use of arbitrary programmatic query generation, most queries appear as simply as the ones in the figure, and our analysis soundly overapproximates the ones whose programmatic generation is too complex.

Ur/Web programs contain two fundamental kinds of SQL operations. First, we have *queries*. These appear in the example program as arguments to the `queryX1` function, which computes an HTML fragment by folding over the rows returned by the query. In general, query results in Ur/Web are always processed using some sort of folding function. We also have *updates*, all of the SQL commands that modify the database. These appear as arguments to the `dml` function, which runs its given SQL command. (Ur/Web distinguishes between the SQL fragment specifying the update and the actual database-updating action.) We want to add **caching** of query results, not just remembering which lists of rows the database returned to us, but also remembering everything computed based on those results. Each cache is associated with one or more queries, and the *keys* of a cache are *the free Ur/Web variables that appear in the code it caches*, whether those variables appear antiquoted in the SQL queries or in the later Ur/Web looping code. Updates can invalidate some but not all cache entries, and we are after a precise characterization of which updates invalidate which cache entries. The central program-analysis task is to compute **for each update operation, a sound mapping from *its* Ur/Web variables to the Ur/Web variables of *queries whose caches it invalidates*.** We now explain by working through analysis and transformation of Figure 1's program.

First, let us consider which caches should exist. We would like to have one cache for the entire body of `allPapers`, and that expression contains no free Ur/Web variables. Therefore, the unit type is suitable as the cache key, and we effectively use a dedicated global variable for the output of `allPapers`. The case for `fromYear` is more complex. The function body has parameter variable `year` appearing free, so a whole-function cache should be keyed on the year. It is not always wise or even possible to achieve whole-function caching, such as when a function contains multiple queries of different tables or both a query and an update. As we discuss in Section 5, Sqlcache starts with query-level caching—in this example, this means caching the output of each `queryX1` invocation—then attempts to cache progressively larger subexpressions.

We have learned that our compiled program should effectively have two caches of these high-level types:

```
cache_allPapers : ( )   -> string
cache_fromYear  : (int) -> string
```

(Note that it is just a coincidence here that the caches match up with the functions, as our optimization supports finer-grained caches.) Now we must figure out how to invalidate those caches on the different update operations.

When `addPaper` runs, we need to *clear* `cache_allPapers`, as we must regenerate the HTML table to include the new paper. In contrast, since `addPaper` is passed the year of the paper, we only need to *clear the* `cache_fromYear` *entry corresponding to that year*. The `changeYear` function has the same broad effect on `cache_allPapers`, but a more interesting effect on `cache_fromYear`: *that cache should be cleared for both the old and new year values*, but it is safe to leave entries for all other years in place.

2.1 Invalidation Justified Formally

How can a compiler discover these invalidation rules on its own, proving to itself that they are sound?

For each pair of a query and an update, we solve a satisfiability problem to find if the latter should invalidate the former's cache. We think of the query as having run in the past, and we think of the update as happening in the present. Each query or update runs in some context with certain values of free Ur/Web variables. For instance, the body of `fromYear` runs with some value of `year` in scope, and we will denote that value more simply as $\mathtt{y_Q}$ below. When `addPaper` runs, it also has some local value of a variable `year`, which we will write $\mathtt{y_U}$, to indicate that this is a variable at the update site. We examine each query-update pair in turn to determine which conflicts may exist.

The easiest case is `allPapers` vs. `addPaper`. When can an `INSERT` operation affect the result of a query? The query results change only when the new row meets the filter condition given by its `WHERE` clause. This filter condition F is a predicate over the column values. The query in `allPapers` has a trivial filter condition: $F(t,a,y) \triangleq \top$. The update in `addPaper` inserts the row $(\mathtt{t_U}, \mathtt{a_U}, \mathtt{y_U})$, whose elements refer respectively to the free Ur/Web variables `title`, `author`, and `year` at the update site. Therefore, the conflict check is for satisfiability of $F(\mathtt{t_U}, \mathtt{a_U}, \mathtt{y_U})$. Of course that formula simplifies down to $\top$, which is trivially satisfiable, so *the insertion must clear the* `allPapers` *cache*.

Now, for a harder case, consider `fromYear` vs. `addPaper`. The filter condition of `fromYear` is $F(t,a,y) \triangleq y = \mathtt{y_Q}$, where $\mathtt{y_Q}$ refers to the free Ur/Web variable `year` at the query site. The conflict check is against $F(\mathtt{t_U}, \mathtt{a_U}, \mathtt{y_U})$, which simplifies to $\mathtt{y_U} = \mathtt{y_Q}$, which is trivially satisfiable when local variable `year` is equal between the query and update sites. In every such match, we need to derive the values of the query-site local variables from the values of the update-site locals. For this simple example, we just recover $\mathtt{y_Q}$ as literally equal to $\mathtt{y_U}$. In other words, *this update, run with* `year` *value* $\mathtt{y_U}$*, needs to invalidate the query cache with key* $\mathtt{y_U}$.

Next we analyze interactions with `changeYear`, which uses an SQL `UPDATE` statement and takes a bit more work to model. Under what conditions does an `UPDATE` affect the results of a query? There are three cases: (1) changing a row so that *the filter condition applies when it did not before*; (2) changing a row so that *the filter condition stops applying when it did before*; or (3) *modifying the column values of a row that is selected both before and after*. For interaction between `changeYear` and `allPapers`, this analysis still reduces to the trivial conclusion that the full cache of `allPapers` must be invalidated.

The analysis is more interesting for `changeYear` vs. `fromYear`. The filter condition of the query is still $F(t,a,y) \triangleq y = \mathtt{y_Q}$. The update has filter condition $G(t,a,y) \triangleq t = \mathtt{t_U} \wedge y = \mathtt{o_U}$, where $\mathtt{t_U}$ is the value of Ur/Web variable `title` and $\mathtt{o_U}$ the value of `oldYear`, at the update point. The effect of the update is to modify an old row (t,a,y) to a new row V, which is a function of the old row: $V(t,a,y) \triangleq (t,a,\mathtt{n_U})$, where $\mathtt{n_U}$ is the value of `newYear`.

With those preliminaries out of the way, we formalize the three scenarios for how the update might change the query result. Instead of predicates on the column values of one row, these are predicates on the column values for a *pair of rows*. We consider a row (t,a,y) that is modified to (t',a',y') by the update. We know that such a modification will occur when the update filter condition holds, and we know the relationship between the old and new rows, so every case below will include the conjunct $G(t,a,y) \wedge (t',a',y') = V(t,a,y)$.

Modifying a row so that the query selects it when it was not selected before. We start with the general formula, plug in the conditions for our particular query and update, and then simplify further, choosing a shorter formula that is implied, for reasons that we will get to shortly.

$$\begin{aligned}&\neg F(t,a,y) \wedge F(t',a',y')\\ &\qquad \wedge\ G(t,a,y)\ \wedge\ (t',a',y') = V(t,a,y)\\ &\Leftrightarrow\ y \neq \mathtt{y_Q}\ \wedge\ y' = \mathtt{y_Q}\\ &\qquad \wedge\ t = \mathtt{t_U}\ \wedge\ y = \mathtt{o_U}\ \wedge\ t' = t\ \wedge\ a' = a\ \wedge\ y' = \mathtt{n_U}\\ &\Rightarrow\ \mathtt{y_Q} = \mathtt{n_U}\end{aligned}$$

Modifying a row so that the the query stops selecting it when it did before.

$$\begin{aligned}&F(t,a,y)\ \wedge\ \neg F(t',a',y')\\ &\qquad \wedge\ G(t,a,y)\ \wedge\ (t',a',y') = V(t,a,y)\\ &\Leftrightarrow\ y = \mathtt{y_Q}\ \wedge\ y' \neq \mathtt{y_Q}\\ &\qquad \wedge\ t = \mathtt{t_U}\ \wedge\ y = \mathtt{o_U}\ \wedge\ t' = t\ \wedge\ a' = a\ \wedge\ y' = \mathtt{n_U}\\ &\Rightarrow\ \mathtt{y_Q} = \mathtt{o_U}\end{aligned}$$

Modifying a column selected by the query. In addition to checking that both the old and new row satisfy the filter condition of the query, we need to check that columns selected by the query are modified, which amounts to checking $(t',a') \neq (t,a)$. The resulting formula turns out to be unsatisfiable, reflecting the fact that the update does not modify any columns selected by the query.

$$\begin{aligned}&F(t,a,y)\ \wedge\ F(t',a',y')\ \wedge\ (t',a') \neq (t,a)\\ &\qquad \wedge\ G(t,a,y)\ \wedge\ (t',a',y') = V(t,a,y)\\ &\Leftrightarrow\ y = \mathtt{y_Q}\ \wedge\ y' = \mathtt{y_Q}\ \wedge\ (t' \neq t\ \vee\ a' \neq a)\\ &\qquad \wedge\ t = \mathtt{t_U}\ \wedge\ y = \mathtt{o_U}\ \wedge\ t' = t\ \wedge\ a' = a\ \wedge\ y' = \mathtt{n_U}\\ &\Rightarrow\ (t \neq t\ \vee\ a \neq a)\ \Rightarrow\ \bot\end{aligned}$$

These three cases cover all possible invalidations of the query by the update, so we may express the full space of possibilities as a disjunction of the three formulas. Applying the implications we calculated above, we find that, in case of an invalidation, the formula $\mathtt{y_Q} = \mathtt{n_U} \vee \mathtt{y_Q} = \mathtt{o_U} \vee \bot$ must be true. Notice how this formula has reverse-engineered the affected key values $\mathtt{y_Q}$ for the query cache! As a result, it is sound to invalidate the cache entries for the keys $\mathtt{n_U}$ and $\mathtt{o_U}$. These values can be computed at the update site, since they refer only to its Ur/Web variables.

2.2 The High-Level Recipe for Invalidation Analysis

To summarize the informal analysis we just carried out, here is a procedure to determine what invalidations must be performed with some update, to the cache of some other query.

1. Apply a general recipe (based on the kind of update) to generate a formula of quantifier-free first-order logic, containing variables x for column values before the update, x' for column values after the update, $\mathtt{y_Q}$ for Ur/Web variables at the query, and $\mathtt{y_U}$ for Ur/Web variables at the update. In general, there may be many variables in each category.
2. Apply logical simplification, similarly to how SMT solvers work, to derive an implied formula in *disjunctive normal form (DNF)*, where each literal equates some $\mathtt{y_Q}$ variable with an expression whose only Ur/Web variables are the $\mathtt{y_U}$ variables. (In some cases, these expressions might even be simpler than those in our example, e.g. constants, or more complex than those in our example, e.g. combining multiple $\mathtt{y_U}$ variables.)
3. Interpret each clause of the DNF formula as a cache-invalidation recipe. Namely, each clause invalidates all cache entries whose keys match the equalities from that clause. By construction, when executing the update, we have in scope all variables needed to compute those keys.

This recipe leaves open many questions.

- What are the general recipes for the first step that work for broad classes of queries and updates? (See Section 3.)
- Algorithmically, how do we go from the first step to the second, in reasonable time? (See Section 3.2.)
- How do we represent caches in memory so that the high-level invalidation logic is efficient to execute? (See Section 4.1.)
- How do we make all this work when several cache-enabled transactions may run at once while preserving transactional semantics? (See Section 4.2.)

The following sections go into these questions in detail.

3. SQL Analysis

Throughout the rest of this paper, we use the term *query* to refer to an SQL `SELECT` statement and *update* to refer to an SQL `UPDATE`, `INSERT`, or `DELETE` statement.

Both queries and updates may contain injected Ur/Web variables. As mentioned previously, Ur/Web variables in a query are the keys of that query's cache: at runtime, if the result of running the query with the same variable values is stored in the cache, we can reuse the result. The central program-analysis problem is the following: given a query and an update, at which keys does the query's cache need to be invalidated? The answer is given in terms of the update's Ur/Web variables, reflecting the fact that which entries are invalidated may depend on those variables' values.

Our approach to this question is to reduce the problem to finding solutions to a quantifier-free first-order logic formula, which we do by applying a well-known database technique. We first describe this process for a class of simple queries, namely those without `UNION`s or `JOIN`s. After demonstrating the process on some examples, we conclude the section by addressing generalization to arbitrary queries.

3.1 Analyzing Simple Queries

We consider the following query Q and update U, which are assumed to operate on the same table and written in abbreviated notation, explained below.

$$\begin{aligned}&Q\colon \mathtt{SELECT}\ \tilde{x}\ \mathtt{WHERE}\ F(x)\\ &U\colon \mathtt{UPDATE\ SET}\ x := V(x)\ \mathtt{WHERE}\ G(x).\end{aligned}$$

In the abbreviated notation,

- x is a vector of all SQL fields in the table,
- $\tilde{x}$ is a vector of the fields selected by the query (and is therefore a subset of x),
- F and G are SQL predicates in terms of x,
- and V is a vector of SQL expressions in terms of x describing the effect of the update.

There are two subtleties to the notation. First, though an `UPDATE` statement need not set every field in a table, we can always rewrite one such that unchanged fields are explicitly set to their old values, which we assume is done here in V. Second, each of F, G, and V

may depend not just on fields in x but also on Ur/Web variables $\mathtt{y_Q}$ (at the query site) and $\mathtt{y_U}$ (at the update site), which will be crucial for the final step of the analysis that extracts the set of invalidations.

A set of invalidations happens when the update executes. For every row x in the table, the update may change it to a row x', which in turn may require invalidating caches that contain query results based on either x or x'. There are three scenarios in which invalidation is necessary, outlined initially in Section 2.1: an unselected (by the query) x can become a selected x', a selected x can become an unselected x', or a selected x can become a selected x' with different values for the selected fields. We can write this condition as a first-order logic formula φ_Q:

$$\begin{aligned}\varphi_Q \triangleq\ & (F(x) \wedge \neg F(x')) \\ & \vee (\neg F(x) \wedge F(x')) \\ & \vee (F(x) \wedge F(x') \wedge \tilde{x} \neq \tilde{x}').\end{aligned} \qquad (1)$$

(Notation: $\tilde{x}'$ is to x' as $\tilde{x}$ is to x.) We can also express the update effect as a formula φ_U. In order for x to be updated to x', x must satisfy the update's filter, and x' must be defined as specified by the update:

$$\varphi_U \triangleq G(x) \wedge x' = V(x). \qquad (2)$$

The conjunction of formulas (1) and (2),

$$\varphi(Q, U) \triangleq \varphi_Q \wedge \varphi_U,$$

is satisfied if and only if x is updated to x' such that the query's cache requires invalidation.

There is another perspective we can take on this analysis: the formula $\psi(Q, U) \triangleq \forall x, x' \colon \neg\varphi(Q, U)$ is true if and only if U is an *irrelevant update* (Blakeley et al. 1989) for Q, meaning that U's action does not change Q's result. Modulo details in the atoms of the Boolean expressions, $\psi(Q, U)$ is equivalent to the formula in Theorem 3.3 of Blakeley et al. (1989); the theorem proves that $\psi(Q, U)$ is a necessary and sufficient condition for the irrelevance of U to Q. For our purposes, because it is sound to unnecessarily invalidate caches, a strengthening of $\psi(Q, U)$ that detected some but not all irrelevancies would suffice. This means, conversely, that we may soundly weaken $\varphi(Q, U)$.

The next step is to extract from $\varphi(Q, U)$ a concrete list of key sets at which to invalidate the query's cache. We start by rewriting $\varphi(Q, U)$ in disjunctive normal form (DNF). After dropping clauses with contradictions, each clause in the DNF formula implies a (possibly empty) conjunction of equalities, each with an Ur/Web variable of the query on one side and either a constant or an Ur/Web variable of the update on the other. (As discussed above, replacing a clause with an implied conjunction—a weakening—is sound.) The invalidations occur with the update, so both constants and Ur/Web variables in the update are known at invalidation time. This means that, at invalidation time, such a conjunction of equalities gives concrete values for a subset of the query's cache's keys, which is meant to be interpreted as "invalidate all cache entries matching these values on this subset of keys." As we will see in Section 4, our cache implementation supports a conservative approximation of this operation in time constant in the number of cache entries.

How do we automate the simplification of formulas to conjunctions with properly shaped literals? First, it is easy to flatten each initial formula to DNF in the standard way. We then separately simplify each clause, which is a conjunction of literals. Each literal is an equality or inequality, each side of which is either an Ur/Web variable (from either the query or update site), SQL field, or constant. (Any literals that are not equalities or inequalities can be approximated soundly as $\top$.) We apply a small subset of the technology that has become standard in SMT solvers, starting with Simplify (Detlefs et al. 2005). Specifically, we perform a *congruence-closure* computation, building a data structure summarizing all of the known equalities between terms in the clause. For every inequality in the clause, between two terms, we query this data structure to see if we have learned that the terms are actually equal. In that case, the clause is contradictory, and we output $\bot$ as the simplification. Otherwise, we need to find equations for the $\mathtt{y_Q}$ variables, standing for cache keys. For each such variable that appears in the clause, we query the congruence-closure data structure, looking for some equated term that only uses $\mathtt{y_U}$ variables, which are in scope at the update. The simplified formula includes one equality for each $\mathtt{y_Q}$ variable where we find a matching $\mathtt{y_U}$ term. The next subsection includes a more concrete example of this reasoning.

The derivation of φ when the update is an `INSERT` or `DELETE` is similar but much simpler. Consider the updates

$$\begin{aligned} D &\colon \mathtt{DELETE\ WHERE}\ H(x) \\ I &\colon \mathtt{INSERT\ VALUES}\ x := W, \end{aligned}$$

where H is a predicate and W is a vector of values. (Because there is no previous row to refer to, W cannot depend on any fields.) For the `DELETE`, invalidation is needed if and only if a row selected by the query is deleted, so

$$\varphi(Q, D) \triangleq F(x) \wedge H(x).$$

For the `INSERT`, invalidation is needed if and only if the inserted row would be selected by the query, so

$$\varphi(Q, I) \triangleq F(x) \wedge x = W.$$

Like the previous `UPDATE` case, we extract cache invalidations using congruence closure. Versions of these formulas also appear in Blakeley et al. (1989).

3.2 Extracting Invalidations from Analysis by Example

Consider the following queries and updates, where a and b are fields and $\mathtt{b_Q}$, $\mathtt{k_U}$, and $\mathtt{v_U}$ are Ur/Web variables injected into the queries and updates:

$$\begin{aligned} Q_a &\colon \mathtt{SELECT}\ a\ \mathtt{WHERE}\ b = \mathtt{b_Q} \\ Q_b &\colon \mathtt{SELECT}\ b\ \mathtt{WHERE}\ b = \mathtt{b_Q} \\ U_a &\colon \mathtt{UPDATE\ SET}\ a := \mathtt{v_U}\ \mathtt{WHERE}\ b = \mathtt{k_U} \\ U_b &\colon \mathtt{UPDATE\ SET}\ b := \mathtt{v_U}\ \mathtt{WHERE}\ b = \mathtt{k_U}. \end{aligned}$$

We will investigate how the analysis in the previous subsection handles each of the four possible query-update pairs.

Starting with the formula for when U_a invalidates Q_a, we compute

$$\begin{aligned}\varphi(Q_a, U_a) \triangleq\ & ((b = \mathtt{b_Q} \wedge b' \neq \mathtt{b_Q}) \\ & \vee (b \neq \mathtt{b_Q} \wedge b' = \mathtt{b_Q}) \\ & \vee (b = \mathtt{b_Q} \wedge b' = \mathtt{b_Q} \wedge a \neq a')) \\ & \wedge (b = \mathtt{k_U} \wedge a' = \mathtt{v_U} \wedge b' = b)\end{aligned}$$

(We have expanded the vector equality involving (a', b') into a conjunction of equalities.) To determine the invalidations, we put the above formula in disjunctive normal form, obtaining the following clauses:

- $b = \mathtt{b_Q} \wedge b' \neq \mathtt{b_Q} \wedge b = \mathtt{k_U} \wedge a' = \mathtt{v_U} \wedge b' = b,$
- $b \neq \mathtt{b_Q} \wedge b' = \mathtt{b_Q} \wedge b = \mathtt{k_U} \wedge a' = \mathtt{v_U} \wedge b' = b,$
- $b = \mathtt{b_Q} \wedge b' = \mathtt{b_Q} \wedge a \neq a' \wedge b = \mathtt{k_U} \wedge a' = \mathtt{v_U} \wedge b' = b.$

The first two clauses are contradictory: the first two terms of each imply $b \neq b'$, but the update does not affect b, as reflected in the $b' = b$ term of every clause. We should expect as much: these clauses correspond to the cases where the update changes whether or not a row is selected, which never happens for Q_a and U_a. The

$$\begin{tikzcd} b & \mathtt{b_Q} \arrow[l] & b' \arrow[ll, bend left] & a & a' & \mathtt{k_U} \arrow[lllll, bend right] & \mathtt{v_U} \arrow[ll, bend left] \end{tikzcd}$$

Figure 2. Congruence-closure data structure for example

third clause implies $\mathtt{b_Q} = \mathtt{k_U}$, meaning we should invalidate the cache entry with key $\mathtt{k_U}$.

To see how we infer $\mathtt{b_Q} = \mathtt{k_U}$ as the answer, consider Figure 2, which shows the congruence-closure data structure built during this simplification. We use graphs where nodes are labeled with expressions that appeared as operands of equalities or inequalities in the original formula. In general, we follow the union-find approach: we draw a directed edge from a node u to a node v to indicate that expressions u and v must be equal. Each node u has a *representative* node w, which is the end of the maximal path starting at u. Two nodes have the same representative if and only if they are in the same equivalence class.

Iterating through the equalities in the clause, for each one we look up the representatives of its two operands and draw an edge from one to the other. In Figure 2, we only need paths of length 1. Expressions $\mathtt{b_Q}$, b', and $\mathtt{k_U}$ all have length-1 paths to b, the representative of their equivalence class. Expression $\mathtt{v_U}$ also has a path to its representative a'. Altogether, we know that the different variables can between them contain at most 3 different values, since the graph is partitioned into 3 connected components.

We also iterate through the inequalities to check for contradictions. The formula includes a clause $a \neq a'$. We look up the representatives of a and a' by path-following, and we find a and a', which are not the same node, so there is no contradiction.

Since the formula is not contradictory, we must produce a simplified version of it. The only key variable that appears in the formula is $\mathtt{b_Q}$, so we look for a suitable equality with $\mathtt{b_Q}$. To do that, we search $\mathtt{b_Q}$'s equivalence class for either a constant or some $\mathtt{y_U}$ variable (an Ur/Web variable in scope at the update). The unique candidate we find here is $\mathtt{k_U}$, so we output $\mathtt{b_Q} = \mathtt{k_U}$ as the result.

The calculation for other query-update pairs is nearly identical, the only difference being how each of the clauses is interpreted as a validation, so we briefly summarize the results.

- Q_b and U_a: As with Q_a above, U_a does not affect whether Q_b selects a row, so the first two DNF clauses are contradictions. The last clause contains both $b \neq b'$, because it checks that the field selected by Q_b changes, and $b = b'$, because b is not changed by U_a, meaning it is also a contradiction, so no invalidations are required for this pair.
- Q_a and U_b: Unlike the previous cases, U_b can alter whether Q_a selects a row, so the first two clauses are not contradictions. The first clause is

  $$b = \mathtt{b_Q} \wedge b' \neq \mathtt{b_Q} \wedge b = \mathtt{k_U} \wedge a' = a \wedge b' = \mathtt{v_U},$$

  which implies $\mathtt{b_Q} = b = \mathtt{k_U}$. Similarly, the second clause (which is the same with the first two terms negated) implies $\mathtt{b_Q} = b' = \mathtt{v_U}$. The last clause contains contradictory terms $a \neq a'$ and $a' = a$, so we must invalidate the entries at keys $\mathtt{k_U}$ and $\mathtt{v_U}$.
- Q_b and U_b: The first two clauses are the same as in the previous case, and the last clause contains contradictory terms $b = \mathtt{b_Q}$, $b' = \mathtt{b_Q}$, and $b \neq b'$, so we must invalidate the entries at keys $\mathtt{k_U}$ and $\mathtt{v_U}$.

3.3 Generalization to Compound Queries

Sqlcache handles queries with three additional constructs: `JOIN`, `UNION`, and nested queries in `FROM` (but not `SELECT` or `WHERE`) clauses. The problem of "flattening" such a compound query into a union of simple queries is that of reducing from an SPJRU algebra (select, project, join, rename, union) to a normal-form SPCU algebra (select, project, cross product, union). The procedure to do so is straightforward; we refer the reader to Abiteboul et al. (1995) for details.

Sqlcache also handles a variety of other SQL features whose subtleties are orthogonal to the static analysis. For instance, grouping (`GROUP BY`) and aggregation (e.g., `SUM`) are safe to view as queries without grouping, with some postprocessing that could just as well happen in the Ur/Web code, whose precise behavior we already do not model statically. The same is true for sorting query results (`ORDER BY`) and choosing to return only certain subsequences of the results (`OFFSET` and `LIMIT`). A finer-grained analysis could in theory avoid some unnecessary invalidations by understanding those constructions (e.g., a cache computing a sum may remain valid when a new row is added with a zero value), but it is sound to overapproximate them.

The most important feature not yet modeled by Sqlcache is cascading triggers. We expect we can add clauses to $\varphi(Q, U)$ that model intertable constraints and their associated cascading effects.

4. Cache Implementation

Each cache C is parameterized by a vector of n keys and needs to support the following operations. (Unlike previous sections, here we view a vector-valued key as a vector of keys.) We write t^n for the type of tuples with n components of type t, key for the type of cache keys, and val for the type of stored values.

- $\mathsf{check}_C : \mathsf{key}^n \to \mathsf{option\ val}$, which retrieves the value with the given keys if it exists.
- $\mathsf{store}_C : \mathsf{key}^n \times \mathsf{val} \to \mathsf{unit}$, which stores a value at the given keys.
- $\mathsf{invalidate}_C : (\mathsf{option\ key})^n \to \mathsf{unit}$, which invalidates values at all keys that match the keys specified in the argument by Some, without regard for the values of keys corresponding to None. For example, $\mathsf{invalidate}\ (\mathsf{Some}\ k_1, \mathsf{None}, \mathsf{Some}\ k_3)$ invalidates values with key vectors in the set $\{k_1\} \times K \times \{k_3\}$, where K is the set of all possible keys.

We leave out the subscript C when discussing a single cache, and we disregard the simple special case of $n = 0$.

The cache also needs to support concurrent operations while maintaining Ur/Web's transactional semantics and having an approximate least-recently-used (LRU) replacement policy. Our implementation supports check and store as described and supports a conservative approximation of $\mathsf{invalidate}$, meaning that it may invalidate more values than necessary.

The implementation is written in C, which is the last intermediate language of the Ur/Web compiler (Chlipala 2015a).

4.1 Basic Implementation

Each cache is logically arranged as a prefix tree. However, to allow random access and garbage collection of individual nodes of the tree, we store the prefix-tree nodes in a hash table with entries connected in a doubly linked list. There are no direct pointers between nodes.

A single cache item is associated with a leaf of the prefix tree. The index of a leaf in the hash table is the concatenation of its keys, and the leaf holds both a cached value and a *storage timestamp*, which records when the cached value was inserted. (We intersperse a special separating character between concatenated keys.) Intermediate nodes, hereafter called simply "nodes," correspond to possible

```
fun fromYearAndTitle year title =
    firstAuthor <- oneRowE1 (SELECT (paper.FirstAuthor)
                             FROM paper
                             WHERE paper.Title = {[title]}
                               AND paper.Year = {[year]});
    return <xml><body>
      The first author is {[firstAuthor]}.
    </body></xml>
```

Figure 3. Another function that gets a more interesting cache

prefixes of the vector of cache keys. The index of a node in the hash table is the concatenation of this prefix of keys, and the node holds an *invalidation timestamp*. The cached value at a leaf is valid if the leaf's storage timestamp is greater than all of its ancestors' invalidation timestamps. Both timestamps are implemented using atomically updated global counters instead of actual clock times.

This scheme helps implement both lookups and invalidations efficiently. An invalidation needs only one hash-table lookup to mark all affected children as invalid, so it happens in constant time. A check must look up a leaf and all of its ancestor nodes to see whether the data at the leaf is valid. This requires a number of hash-table lookups linear in the number of cache keys. Crucially, the time is constant in the number of cache entries, which can grow large at runtime. In contrast, the number of cache keys is set at compile time and small in practice. We will soon see that stores require the same number of lookups as checks.

The prefix-tree arrangement works well for invalidations as long as a prefix of the cache keys is known, but when other subsets of keys are known, we have to make a tradeoff. We prioritize efficiency over granularity and interpret invalidations as if all keys after the known prefix are unknown, which invalidates more cache entries than necessary. Our compiler pass uses (quite simple, for now) heuristics to infer a key ordering that reduces the frequency of invalidations that invalidate more keys than necessary.

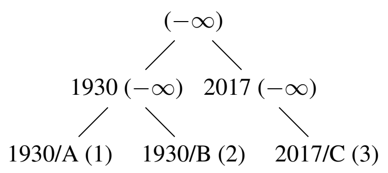
Disregarding concurrency and LRU replacement, the operations follow straightforwardly from this structure. Recall that there are no pointers between leaves or nodes in the prefix tree, with all accesses done by lookups in the hash table.

- check ks looks up each of the n indices formed by concatenating the initial segments of ks. It returns Some v if the nth index has a node with value v and storage timestamp later than all invalidation timestamps encountered among nodes, and it returns None otherwise.
- store ks v stores v in a leaf at the index formed by concatenating ks, and it records the current time as the leaf's storage timestamp. If a value exists at that leaf already, it is freed.
- invalidate oks looks up the index formed by concatenating the keys of the longest all-Some initial segment of oks, creates a node at the index if one does not already exist, and records the current time as the node's invalidation timestamp. If all the keys match Some k, we delete from the table and free the leaf rather than set its timestamp.

Supporting LRU replacement requires small additions to check and store. Both operations "bump" all leaves and nodes found to the head of the doubly linked list of hash-table entries and delete the tail of the list if the hash table exceeds its size threshold. The only complication is that nodes holding invalidation timestamps relevant to a leaf may be removed before the leaf in some corner cases. Therefore, store creates any missing nodes and gives them invalidation timestamp $-\infty$, and check conservatively returns None if any node it needs an invalidation timestamp from is missing.

To make those details concrete, consider the additional function from Figure 3, which we could add to the original example of Figure 1. We use another standard-library function `oneRowE1` to run a query meant to return a single row (hence the "1") containing a single computed expression (hence the "E"). This function's query has two free Ur/Web variables, `year` and `title`, which we use to look up a paper uniquely. Our program analysis assigns it a cache with those two Ur/Web variables as keys. In the associated in-memory hash table, we look up string keys computed by serializing the underlying keys and concatenating them with a separator. For instance, the paper with `year` 1930 and `title` "Lambda Calculus" would be mapped to a hash-table key like "1930/Lambda Calculus."

The cache is structured as a tree, using one shared hash table for efficient lookup of children from internal nodes. After several store operations, we might wind up with this tree.

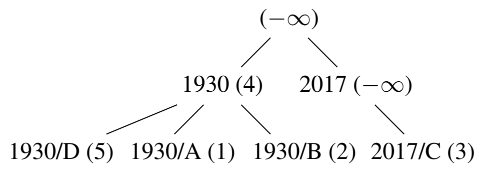


The three keys at the leaves of the tree were inserted respectively at times 1, 2, and 3, and those nodes are tagged with their *storage timestamps*. Each leaf has attached to it the cached value of running `fromYearAndTitle` on its key. Interior nodes (including the root) are tagged with *invalidation timestamps*, which here are all $-\infty$ because no invalidate calls have happened yet.

From this state, every concrete check operation will traverse the tree from root to leaf, following the edge for each successive prefix of the key sequence. If the process ends at an extant leaf, we return the associated result. Otherwise, we return None. We see that lookups are constant-time in the number of cache entries.

Now imagine an update runs, where our analysis said that it must invalidate all cache entries associated with 1930 and "B." The result is simply to delete the associated leaf of the tree.

Alternatively, a more interesting invalidation would apply only to the year 1930, as might `changeYear` from Figure 1. Imagine that we run this invalidate operation at time 4, followed by store for a new 1930 paper called "D" at time 5.



Now any check for one of the original 1930 papers will fail, because, on its path to a leaf with some storage timestamp, it encounters node 1930, with a *higher* invalidation timestamp. However, a check on new paper D will succeed, because its storage timestamp is greater than all of its ancestors' invalidation timestamps. Note again that invalidate runs in constant time, while setting the stage for the constant-time (in the size of the cache) behavior of check.

Some query-update pairs may be analyzed coarsely enough that their invalidation formulas are just $\top$, in which case, when running that update, we replace the invalidation timestamp of the root with the current time, implicitly invalidating the whole tree. In scenarios at these different levels of coarseness, the tree can end up with entries that will not be used again. No dedicated garbage collection is necessary; such entries naturally rotate out through the LRU policy as new entries are stored.

4.2 Concurrency

Request handlers in an Ur/Web application are atomic transactions with respect to its database. That is, given a handler involving several database queries and updates, it appears as if its queries and updates occur in sequence without in-between updates from concurrently executing handlers. This means there are two concurrency challenges our caches must address: preserving transactional semantics and, as usual, preventing data races. We address these concerns separately with two readers-writer locks per cache: a *transaction lock* ensures transactional semantics, and a *data lock* prevents data races. We take advantage of the transactional semantics and storage timestamps to reduce contention on the data lock.

Transaction locks are held by request handlers for the entire duration of their execution. A handler takes the transaction lock with write permissions for every cache it may invalidate and the transaction lock with read permissions for every other cache it may check and store to. This is determined by a syntactic static analysis that checks each handler (and, transitively, functions it may call) for instances of check_C, store_C, and $\mathsf{invalidate}_C$. All such transaction locks are released when the transaction terminates, whether or not it successfully commits. (If the transaction is restarted, it will take the locks again.) Because invalidations need write permissions, only one transaction that may invalidate any particular cache executes at a time. It is clearly sound for checks to execute concurrently with each other, but, less obviously, it is also sound for stores to execute concurrently with checks. This is because any concurrently executing stores are storing values that are a pure function of the keys and the database state, so in the absence of relevant database modifications, concurrent stores with identical keys will be storing the same value. Asking whether a handler changes the database state in a way that might affect cache C is a question we have already answered with the analysis in Section 3: those that *do* make C-relevant changes must call $\mathsf{invalidate}_C$. Thanks to the transaction lock, no such handlers execute concurrently with stores and invalidations to C.

It is not safe for store to modify the cache immediately, because any Ur/Web transaction must be prepared to abort and restart, say because the database reports a deadlock. Instead, store maintains a queue of insertions to perform if the transaction terminates successfully. The insertions happen before the transaction locks are released.

It remains only to prevent data races. This is done by simply having each of the cache operations take read or write permissions on the cache's data lock whenever they read or write, respectively, the hash table. Unlike transaction locks, no thread holds more than one data lock at once. There are two subtleties. First, LRU "bumps" require write permissions, so to avoid excessive read serialization, check does the LRU "bump" with low but nonzero probability. Second, to reduce contention on the data lock, check releases it before returning its result. However, because store frees any value it overwrites in the cache, this enables the following three-thread race condition:

1. Thread 1 calls check ks, which returns None, so it begins computing the value to store in the cache;
2. Thread 2 calls check ks, which also returns None, so it also begins computing the value to store in the cache;
3. Thread 1 computes v and calls store $ks\ v$;
4. Thread 2 computes w and calls store $ks\ w$;
5. Thread 1 commits, putting v in the cache;
6. Thread 3 calls check, which returns Some v;
7. Thread 2 commits, putting w in the cache and freeing v;
8. Thread 3 tries to access v and segfaults.

Fortunately, v and w are computed by the same pure function of ks and the database state, so they are equivalent. We therefore use timestamps to prevent the insertion and freeing in step 7. Specifically, when we push v and w onto store queues in steps 3 and 4, we mark the insertions with timestamps. When v is inserted into the cache in step 6, its storage timestamp is recorded. In step 7, we observe that the insertion timestamp for w is earlier than the storage time for v, which means they are equivalent and the insertion is unnecessary.

4.3 Runtime Monitoring

It is common for request handlers to perform both queries and updates involving the same tables, which often results in caches that are always invalidated immediately before or after they are checked, making it impossible to get a cache hit. In such cases, Sqlcache causes a performance regression, somewhat because of data-copying overhead but mostly due to contention on the transaction locks described in the previous section. It is in theory possible to determine such cases statically, but it is simpler to catch badly performing caches at runtime, deactivating those with hit-to-invalidation ratios that are too low.

Using runtime information has the added benefit of deactivating caches that perform badly due to reasons that would be impossible to determine statically. For example, an admin page for a busy interactive website might be accessed infrequently enough relative to user actions that, in practice, all of its caches get invalidated between page loads. A purely compile-time solution would require either past runtime statistics or manual annotations on queries to reach the same domain-specific conclusion.

In more detail, the runtime monitoring works as follows. Each cache maintains a hit-to-invalidation ratio using exponential smoothing on the series of hits (value 1) and invalidations (value 0). When the smoothed value crosses below a threshold, the cache deactivates: checks automatically miss, stores and invalidations become no-ops, and, most importantly, the transaction locks for the cache do not need to be taken out. Cache deactivation always occurs during an invalidation, which means the deactivating thread holds that cache's transaction lock, so there are never concurrently executing requests relying on data from the cache being deactivated.

In the event that a cache has a temporary unsuccessful spell but is in general useful, the runtime monitoring provides a glimmer of hope for cache reactivation. A small fraction ($< 0.1\%$) of checks, stores, and invalidations are simulated to keep track of a hit ratio estimate. If the ratio becomes high enough, the cache is reactivated. To preserve transactional semantics without requiring a lock for checking cache deactivation, checks and stores treat the cache as deactivated for a few wall-clock seconds following reactivation. This potential compromise of transactional semantics, which can be made arbitrarily improbable by tuning the time delay, is well worth it for the performance of transaction-lock-free operation while the cache is deactivated.

5. Program Instrumentation

Rather than caching query results directly, Sqlcache infers which query-dependent computations are possible to cache and inserts unified caching code around their expressions. This step often identifies large computations, such as entire pages, as cacheable, which can be a great boon for performance.

When caching an expression E with cache C, where ks are the values of the cache keys, Sqlcache replaces E with

```
case check_C ks of
    Some v ⇒ v
  | None ⇒ let val v = E in store_C (ks, v); v end
```

Although the Ur/Web source language is purely functional, the intermediate representation used during Sqlcache is an effectful functional language. If we read the result of evaluating an expression from a cache, we must also reproduce any effects the expression may have had. That is, we have to cache both values and effects. Currently, we support caching Ur/Web's primary effect: writing response text. This applies to both HTML pages and other responses, such as those used for remote procedure calls (Ur/Web's interface to AJAX). To capture output, check_C sets a pointer to the tail of the current output, and store_C finds the output to be cached by copying the string from that pointer to the new output tail. We do not currently support caching effects other than writing output; our analysis conservatively refuses to cache computations that might have other effects, which notably include SQL updates.

As a baseline, Sqlcache attempts to cache every SQL query in an Ur/Web program aside from those that use certain advanced SQL features that are not yet supported. Ur/Web's primitive for queries is to loop over their result rows, maintaining an accumulator and possibly causing other side effects. When the effects go beyond writing to the page, we refrain from inserting a cache. Those with no effects other than writing output, however, are guaranteed to be cached in the instrumented program. In an effort to reduce the number of cache lookups that need to occur when handling a request, Sqlcache consolidates multiple caches into one when possible. For instance, two caches for two queries parameterized by the same Ur/Web variable have the same key, so we can store both results in the same cache.

Sqlcache performs this cache consolidation by analyzing nodes of the intermediate language's AST for effectfulness and caching the maximal subexpressions such that

1. there are no effects other than SQL queries and writing output;
2. at least one query is in the subexpression;
3. Sqlcache can "easily compute" from the free Ur/Web variables in the subexpression all Ur/Web variables that appear in *query strings*, some of which may be bound by inner let expressions;
4. and all free Ur/Web variables in the subexpression appear in the *query subexpression*, either directly or through an intermediate bound variable.

We distinguish between two similar but different notions. The *query string* is an expression of type string that represents the actual query to be sent to the database. The *query subexpression* includes both the query string and an iterator that is applied to each row of the result, accumulating a value and emitting response output along the way. Variables in the subexpression are, in some sense, the "most entangled" with the query results; the current implementation has no way to cache queries without including such variables as cache keys.

In the current implementation, "easy computation" includes record projection and Ur/Web's several SQL-injection functions. It is in principle possible to support any injective function, but these two are the most common in Web applications that provide relatively shallow interfaces atop databases. The last condition is a heuristic that reduces the key space of caches from what they might otherwise be, which results in fewer cache misses after an invalidation. This is because no invalidation formula will specify anything about keys that do not appear in any query, so the corresponding invalidation component will always be None, likely invalidating many entries. One can imagine tweaking these conditions to get a variety of cache-consolidation heuristics, which may be a direction for future research in a context more general than Sqlcache's.

Once this cache consolidation is complete, instrumentation of update sites is straightforward: we simply execute any cache invalidations needed for that update immediately after the update. (Transactional semantics preclude the race condition of a cache hit between an update and its associated invalidations.) The invalidation analysis works on a single query-update pair, but cache consolidation can cause a cache to depend on multiple queries. In these cases, we turn invalidations of the hypothetical single-query cache into invalidations of the multiple-query cache by adding None as the argument for any keys not present in the analysis for the single query.

6. Performance-Benchmark Results

To evaluate the effectiveness of Sqlcache, we measured its impact on several applications. All experiments were run an Intel Xeon E5620 workstation, with a 2.4 GHz processor providing 8 hardware threads. The machine had 12 GB of RAM, which did not come near to being the bottleneck here. We use version 9.3.1 of the PostgreSQL database server and C client libraries.

6.1 Dyncache: a Dynamic Sqlcache Alternative

To provide a fair comparison with other caching techniques, we also measure an Ur/Web strategy called *Dyncache*. Dyncache is a straightforward cache mapping query text to the list of rows returned directly by the database engine as the query result. Each update invalidates cache entries of queries that reference the table it modifies. Dyncache prioritizes speed of the cache operations over maintaining transactional semantics, with minimal locking to maintain concurrent memory safety. The most important differences between Dyncache and Sqlcache are lack of fine-grained invalidation and lack of caching for values computed from the raw database results, both features that depend on Sqlcache's compile-time analysis.

Dyncache works using two global hash tables: *Stores* and *Flushes*. Each entry in Stores has a *storage timestamp*, and each entry in Flushes has an *invalidation timestamp*. Each time a cacheable query runs, Dyncache puts its results in the Stores table, keyed by the query text, and sets its entry's storage timestamp to the current time. Each time an update runs, its table name is looked up in Flushes and given the current time as an invalidation timestamp. To check if a query has a cached value, we look up the query text in Stores and look up each of the tables the query references in Flushes; the check is a hit if the retrieved storage timestamp is more recent than all retrieved invalidation timestamps.

There are two details elided in the previous paragraph. First, we avoid doing many hash-table lookups when checking the cache by doing lookups in Flushes only once per new entry in Stores, keeping pointers to the relevant Flushes entries. Second, to maintain memory safety without using locks, we delay freeing the memory of replaced query results for a few seconds to give in-flight requests a chance to finish using them. This requires a simple garbage-collection system. As we will show, Dyncache's performance is generally robust to the rate of cache invalidations encountered in our benchmarks, and no garbage collection is required in the read-only benchmarks, so we will not elaborate further on this detail.

Despite its suggestive name, Dyncache is not completely dynamic. It uses compile-time analysis to determine which queries are cacheable (namely, which queries do not contain `RANDOM` or `CURRENT_TIMESTAMP`), which tables are read by each cacheable query, and which tables are modified by each update. This gives Dyncache a mild edge over completely dynamic proxies that must parse and analyze query syntax at runtime, so using it to stand in for more dynamic methods is a conservative choice.

6.2 TechEmpower Web Framework Benchmarks

We started by measuring the effect of Sqlcache on the Ur/Web implementation of the TechEmpower Web Framework Benchmarks[5], which had previously been used to evaluate the effectiveness of

[5] `https://www.techempower.com/benchmarks/`

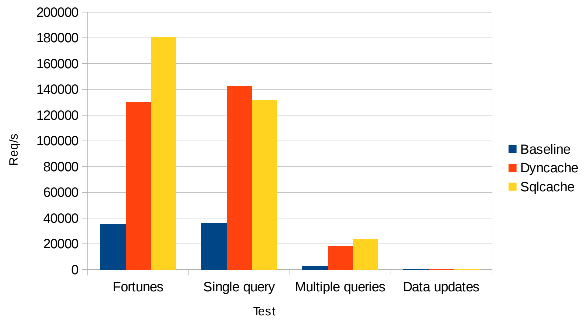


Figure 4. Effect of Sqlcache on TechEmpower benchmarks

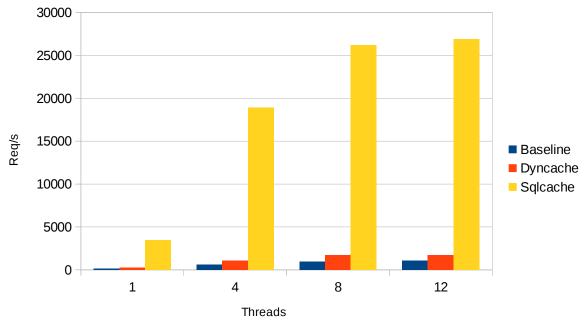


Figure 5. Concurrency scaling for Dinners

Ur/Web's baseline compiler (Chlipala 2015a). These benchmarks are attractive as a way of grounding our baseline in comparison to other frameworks. For instance, in the most recent official benchmarks run, Round 12, on a high-capacity server with 40 hardware threads, Ur/Web dominates the results for the default test called Fortunes, achieving almost 300,000 requests per second, with the second-place finisher achieving only a little more than half as much throughput. 145 different Web-framework configurations are represented, including all of the most popular real-world options. As a result, in experimenting with optimizing this baseline implementation, we need not worry about seeing improvements only because the original implementation had missed some standard optimization trick.

Each benchmarking run uses the `wrk` program[6], opening 12 connections to the server and saturating them for 10 seconds, where the server has also spawned 12 worker threads. We also do a 2-second warm-up run for each test, before starting to measure timing. This is roughly the official benchmark methodology, but with fewer threads. Our experiments also only run SQL servers on the same physical machines as the Web servers; Sqlcache would bring additional benefits if the two were physically separated.

Figure 4 shows the results for the four TechEmpower test cases that use the database. They are:

- **Fortunes**, which runs a parameter-free query and renders the results as an HTML table. Throughput increases to over 5X the original for Sqlcache and almost 4X for Dyncache. (Note that the baseline is lower here than in the official TechEmpower results because we benchmark with a lower-capacity machine.)
- **Single query**, where each Web request looks up a random database row by key and returns the result as JSON. Sqlcache and Dyncache score similarly, increasing throughput by almost 4X. It makes sense that Sqlcache's gain would be lower here than in the last case, because different requests access different cache keys at random.
- **Multiple queries**, which makes 20 different random lookups in the same table, again returning the full set of results as JSON. Throughput increases to more than 8X for Sqlcache and 6X for Dyncache.
- **Data updates**, the only test with database write operations, and one where Ur/Web's original performance was poor, since, unlike almost all other frameworks compared in the benchmarks, Ur/Web enforces transactional semantics and incurs significant synchronization overhead. Either caching strategy performs poorly in this case where caches are constantly being invalidated, but runtime monitoring deactivates the caches and leaves both Sqlcache and Dyncache performing about as well as the baseline.

[6] `https://github.com/wg/wrk`

6.3 Macrobenchmarks

We also evaluated Sqlcache's effectiveness on two production applications built with the Ur/Web People Organizer[7] library. Applications in that library wind up with surprisingly write-heavy page-handler code, because of the machinery for live updating of data in browsers as the server receives changes. The server needs to store information on which clients are waiting for which sorts of updates, which means that every time a new page is loaded, several SQL `INSERT` commands run to record the new session's subscriptions to data. Moreover, the Ur/Web runtime system spends significant time garbage-collecting sessions that have timed out, which adds up to significant overhead with hundreds of requests per second or more, each starting a new session. For that reason, we disabled the live-updating machinery for these experiments. We also removed use of cookies for authentication, instead hard-coding a username for each experiment, because our analysis does not yet track cookies as an external program input.

In the performance experiments that follow, for each set of server parameters, we average throughput over a 10-second trial, using as many client threads as server threads, after a 2-second warm-up period in the same mode.

The first application we looked at, Dinners, helps groups of people coordinate to vote on times and places to go to dinner. Users can add and remove restaurants and times, vote on their favorites, and maintain the list of past dinners, with comments annotated on them. We modified this application to be something of a best case for Sqlcache, by removing a few places where the code accesses the current time, which tends to thwart caching. The result retains complex and varied functionality while still offering many opportunities for sound caching. We benchmarked the performance of the main page of this application, which is the only page that most users see directly, containing different tabs for all of the above activities. None of the caches here depend on any particular keys.

We have preserved the code we benchmarked in the main UPO repository, branch `sqlcache−experiments`, in the example program in `examples/jfac.urp`.

We measured how performance of each server scales with number of server threads, in each experiment running N benchmarking threads against a server with N worker threads. Figure 5 shows the results, which confirm that this is indeed a very compelling use case for Sqlcache, as it consistently increases throughout by about 30X over the baseline. Dyncache also improves performance significantly, but its 2X improvement is much less than Sqlcache's.

We also experimented with another, more complex application, Course Management, which is used in production to manage a university course. It supports distributing assignments and collecting student solutions, collecting grades for assignments, summarizing

[7] `https://github.com/achlipala/upo`

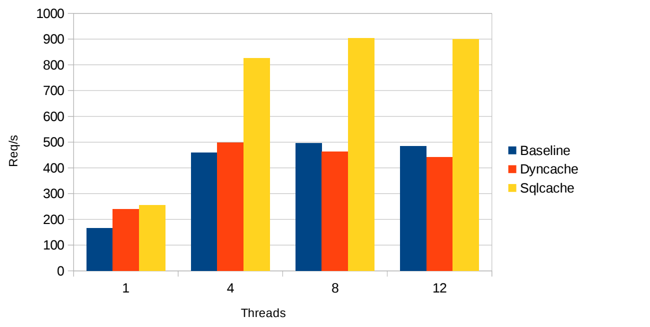


Figure 6. Concurrency scaling for Course Management

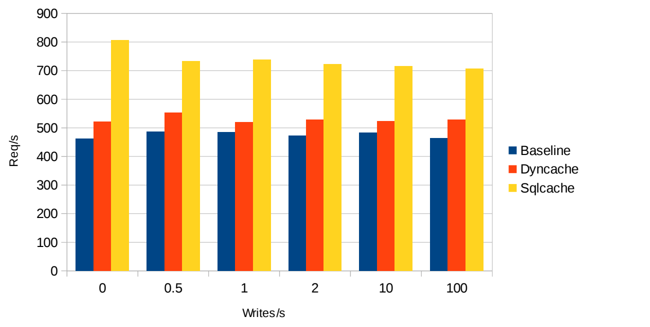


Figure 7. Write scaling for Course Management

grades for students, announcing news items, maintaining a calendar, taking a poll on preferences for office-hours times, and, most importantly for us, one global message forum and one dedicated forum per lecture, assignment, etc. Sqlcache infers that it should create a cache keyed on the lecture number, to record current forum contents. This cache coexists with many others, as well as several queries that are not cached, for instance because they use SQL subquery expressions, which Sqlcache does not yet model in detail. Another important source of uncacheability is fundamental: some queries use the current time, for instance to look up which lecture happened most recently and dedicate a tab to it, including a listing of the associated forum. However, all is not lost in those parts of the code, which is structured as queries to look up most-recent entities using the current time, followed by further lookups directly on the entity names. The second category is very well-suited to caching by entity name.

This application displays a few main pages, including those for students and instructors. We found that our analysis gets overwhelmed when run against the full application; it seems clear that a future extension of the system should use dynamic profiling data to limit which queries are cached, unless authors of large applications do not mind running the optimization for an hour. We decided to limit our experiment to the student page only, which still leaves us with about 150 distinct SQL queries in the application, and which takes a few seconds to compile with Sqlcache activated.

We preserved the reduced version of the application in branch `sqlcache−experiments` of its main repository[8].

Figure 6 shows the results of a concurrency-scaling experiment for the Course Management app. Dyncache never manages to diverge very far from the baseline. Sqlcache starts out with only a slight throughput advantage over the baseline, but its advantage grows to about 2X as we increase concurrency. Examining the database-server logs, we confirmed that, in the cache-enabled version, only two sorts of queries are executed in steady state: those that use SQL features not yet modeled in detail by Sqlcache, and those that depend on the current time and thus cannot be cached.

Those last results consider a read-only workload. We only expect caching to pay off for read-dominated workloads, but some writes should be expected in a real deployment. Figure 7 shows how performance degrades as we increase the number of write operations in parallel from 0 per second up to 100 per second, for a 4-thread server. Each operation picks a random lecture and posts a message in its forum. Since the front page only depends on the forum contents for the *most recent* lecture, so long as write requests only post to other lectures, the Sqlcache application is able to reuse all its caches. The baseline and Dyncache are nearly unfazed by this level of write activity, while Sqlcache gradually degrades down to about 90% of its peak performance, maintaining its solid lead over the other variants. Higher write ratios could significantly hurt the performance of all variants, but only very popular Web applications receive more than 100 write requests per second in practice.

We were also pleasantly surprised to find another place that Sqlcache brings a big win for this application. Many different modules of the application call a common access-control library, to determine if the user is authorized to perform particular actions. In the baseline application, the student page runs about 20 SQL queries to answer questions about access control. These queries are all redundant with each other, following from one access level that could be read from the database once, but there are software-engineering advantages to not forcing the application code to maintain its own explicit cache of the result. Indeed, Sqlcache correctly adds caches for each of the forms of queries, so that no access-control queries run in steady state.

7. Related Work

Our optimization caches the output of database-backed Web applications. There is a conceptual connection to the classic idea of *materialized views* in databases (Larson and Yang 1985), where extra logical tables are maintained as caches of the results of particular queries over primary tables. Much past work has studied how to use materialized views to answer queries efficiently (Goldstein and Larson 2001) and how to update materialized views as the underlying tables are modified (Blakeley et al. 1986). Here views are generally used to cache the direct results of SQL queries, rather than full application outputs that combine query results with computation in a Turing-complete language. An advantage of the classic materialized-views approach is that sophisticated analysis finds ways to use a single set of views to answer many different queries efficiently. On the other hand, each of our caches can be more efficient for its specific purpose, as it saves all results needed to respond to part of a client request. We also generate the caching structure automatically from unannotated application code, rather than requiring careful crafting of an explicit database schema with views.

Past work has considered how to do mostly automatic caching of database-dependent application results. One established database-level technique is semantic caching (Dar et al. 1996), which replaces client-side caching schemes that work just at the level of, e.g., numbered pages in the on-disk database (much as an OS file-system cache generally works). Instead, semantic caching associates each saved set of rows with a semantic condition, based on which sorts of filtering conditions appear in queries. MtCache (Guo et al. 2004) uses runtime analysis of queries to decide whether a local replica of portions of a database can answer a query. Application servers must run their own database-server replicas to employ either scheme.

The TxCache (Ports et al. 2010) system is much closer to our goals and techniques in this work. Through modification of an off-the-shelf relational database server, they support caching of ap-

[8] `https://github.com/achlipala/frapapp`

plication outputs while preserving a transactional semantics for database accesses. While TxCache depends on the presence of trusted semantic program annotations to drive caching, our optimization guarantees soundness despite analyzing and transforming programs automatically. We also maintain compatibility with widely used, unmodified database servers. However, the more dynamic approach of TxCache may detect optimization opportunities that our program analysis is not sophisticated enough to notice; and, in making caching choices, they may take advantage of runtime information like statistical properties of current table contents.

CachePortal (Li et al. 2001), another dynamic caching system, can be fully automatic but has design goals very different from Sqlcache's. Like Sqlcache, CachePortal caches dynamic page output, saving time not just on database queries but also on page generation. A CachePortal cache is automatically invalidated when relevant updates occur, with analysis occurring at runtime, and extensive runtime monitoring drives decisions about which pages to cache. All of CachePortal's components live entirely outside the application and database servers, inferring information at runtime by examining server logs and occasionally sending queries of their own to the database. While this separation fulfills CachePortal's goal of compatibility with off-the-shelf software, it means a lot of work in CachePortal goes into reconstructing information that is easily available at compile time, such as which queries execute to respond to a given URL. Relative to CachePortal, one might view Sqlcache as a demonstration of how much simpler automatic invalidation can be when compile-time analysis is an option!

Sync Kit (Benson et al. 2010) is a Web application toolkit employing client-side caching to reduce server load. It dodges the issue of supporting full semantic caching by restricting supported queries to a few efficiently cacheable patterns, such as key-value lookups, and automatically maintaining fresh client-side caches for those queries. Sqlcache places no such restriction on queries. Sqlcache's SQL analysis is already well-suited to fine-grained invalidation of key-value-like queries; intelligent invalidation of other common patterns, such as ordered lists with filters (Sync Kit's other wheelhouse), is a possible direction for future work.

Our optimization moves computation work from the database server to the application server via caching. A complementary approach moves work from application to database through the *stored procedures* support offered by most database engines. Here the motivation is to avoid the cost of extra round trips between application and database, by pushing into the database the logic that glues together several dependent queries. Pyxis (Cheung et al. 2012) automatically moves application code into stored procedures, combining static and dynamic analysis in an attempt to minimize round trips. Automatic compiler-based partitioning has also been used to support security policies in applications. For instance, Swift (Chong et al. 2007) guarantees that information-flow policies are respected in an automatic partitioning, and SELinks (Corcoran et al. 2009) generates stored procedures that help enforce policies associated with a broad range of custom executable checkers.

Another approach to optimizing database-backed applications is to schedule work more efficiently, cutting down on dead time while the application is waiting for the database server to respond to a query. Query prefetching (Ramachandra and Sudarshan 2012) relies on program analysis and transformation to take a program written in a natural style and replace it with one that executes several queries at once, reducing the number of database round trips. The Sloth system (Cheung et al. 2014) adopts a more dynamic approach. It treats query statements *lazily* in the sense of functional languages like Haskell, only executing queries when their results are needed to generate output. Several suspended queries may then be sent to the server simultaneously, realizing benefits similar to prefetching without requiring sophisticated static analysis.

Query synthesis (Cheung et al. 2013) is a further compiler-based approach to automatic, sound optimization of database-backed application code. Popular high-level database libraries in languages like Java encourage coding in an idiomatic object-oriented style, as if the database were a collection of heap-allocated objects. The library mediates between this perspective and standard SQL, sending queries and updates to the server as needed. It is easy for programmers, especially unsophisticated ones, to write rather inefficient code in this style, leading to surprisingly many database round trips or surprisingly inefficient filtering code within the application. Query synthesis analyzes code patterns and deduces opportunities to replace pieces of Java code with equivalent SQL syntax, which is then executed more efficiently at the database, using index structures and the usual optimizations.

Compared to our work, one distinguishing feature of the last few techniques is that, while they reduce the number of database round trips, they always contact the database when running any application code containing database operations. Database servers are often physically distinct from application servers, so even a single round trip may impose a significant latency cost. Our compiler optimization avoids contacting the database server altogether for a wide variety of read-only transactions and imposes reasonable overhead in workloads with moderate shares of write operations.

Recent work has used the Scala Lightweight Modular Staging system to generate efficient in-memory database servers automatically from high-level code, using compile-time metaprogramming (Klonatos et al. 2014; Rompf and Amin 2015). Our work is complementary to theirs, which provides compilation of query and update operations in a novel way. We expect that our caching optimization would be fruitful to apply in their setting, where, as in Ur/Web, we have the advantage of a high-level, non-string-based representation of query code amenable to automated analysis.

The database-caching problem is closely related to the classic programming-languages area of incremental computation (Ramalingam and Reps 1993), which finds opportunities to reuse computational work across different executions of an algorithm. For instance, the Ditto system (Shankar and Bodík 2007) improves the efficiency of Java data-structure invariant checks through sound and automatic caching of method-call results across invocations. Self-adjusting computation (Acar et al. 2008; Hammer et al. 2009) is another approach that saves computation histories at runtime, as suitably annotated dependency trees, and looks for opportunities to reuse subtrees in new computations. These systems work with traditional functional and object-oriented programs, whose structure is relatively challenging for static analysis. By instead analyzing high-level SQL operations, we are able to generate more specialized, efficient caching that avoids the broad runtime instrumentation of general-purpose incremental-computation systems.

We implemented our optimization for the Ur/Web language (Chlipala 2015b), where a key feature is integrated parsing and static type checking of SQL syntax, so that we do not need to do string or reference analysis to recover the database operations within a program. Ur/Web's whole-program compiler (Chlipala 2015a) already applied some SQL-specific optimizations, but it previously maintained the one-to-one mapping between SQL calls in source code and runtime round trips with the database server. The high-level treatment of SQL code had previously been exploited to statically check declarative access-control and information-flow policies (Chlipala 2010).

8. Conclusion

Database access is a great example of the classic tradeoff between performance and programmer effort. By writing code that queries the database directly, programmers avoid cluttering that code with explicit caching of results. However, caching can bring dramatic performance benefits. The most difficult aspect of its implementation is *invalidation*, where cache entries need to be removed as data updates modify their dependencies. We presented Sqlcache, the first static analysis and transformation that *automatically* modifies applications to use sound caching.

Our analysis applies to the Ur/Web programming language, in which the structure of SQL queries is exposed quite explicitly, thanks to explicit parsing and type checking of SQL code by the Ur/Web compiler. Each eligible SQL query is assigned a cache keyed off of the free Ur/Web variables that appear in the query. We compare each pair of an SQL query and an SQL update in a whole program, reducing the question "which keys in the query's caches are invalidated by this update?" to a problem of simplifying a formula of first-order logic. The simplified formula has a computational reading as a set of cache-invalidation operations. Each cache operation is performed in constant time against a simple in-memory data structure, and a further stage of our transformation adds automatic locking, to preserve ACID transaction semantics for multithreaded workloads.

We plan several future-work directions to improve Sqlcache's effectiveness. First, we would like to extend our modeling of SQL features to include subquery expressions, data constraints that trigger cascading effects when changes happen, and other syntactic constructs that are currently approximated very conservatively. Second, we would like to investigate other cache-concurrency-control strategies beyond the lock-based one described in Section 4.2. We might consider using optimistic concurrency control, logging cache-relevant operations without locking, being prepared to roll back a transaction if we find inconsistencies at the transaction commit point. It would likely also help to use more in-depth dynamic profiling data to control which queries are worth caching, and, as for improving compile-time performance, it could be fruitful to investigate better data structures for quickly winnowing down the full set of query-update pairs to those that might conflict. We would also like to extend our analysis to facilitate other kinds of optimization on SQL-based code, for instance to infer opportunities for transaction chopping (Shasha et al. 1995), which soundly breaks one transaction into several that can execute in parallel.

Another natural direction for future work is applying the techniques used in Sqlcache to other programming languages and system architectures. It is likely that with a sufficiently disciplined database interface, compile-time SQL analysis can be made simple in a wide array of languages. Relaxing Sqlcache's single-server restriction would involve implementing a more sophisticated distributed system. One approach might be to use a distributed cache instead of an in-memory hash table to store the nodes of our cache data structure.

Acknowledgments

For their feedback on earlier drafts of this paper, we thank Peter Bailis, Clément Pit-Claudel, Paul Steckler, and Murali Vijayaraghavan. This research was supported in part by NSF award CCF-1217501. The first author was supported by an ARCS Foundation scholarship. Most of the reported work was done while the first author was a student at MIT.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1252522. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. ISBN 0201537710.

U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proc. POPL*, pages 309–322. ACM, 2008.

E. Benson, A. Marcus, D. Karger, and S. Madden. Sync Kit: a persistent client-side database caching toolkit for data intensive websites. In *Proceedings of the 19th International Conference on World Wide Web*, pages 121–130. ACM, 2010.

J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. SIGMOD*, pages 61–71. ACM, 1986.

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

A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, July 2012.

A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *Proc. PLDI*, pages 3–14. ACM, 2013.

A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proc. SIGMOD*, pages 931–942. ACM, 2014.

A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. OSDI*, pages 105–118, 2010.

A. Chlipala. An optimizing compiler for a purely functional Web-application language. In *Proc. ICFP*, pages 10–21. ACM, 2015a.

A. Chlipala. Ur/Web: A simple model for programming the Web. In *Proc. POPL*, pages 153–165. ACM, 2015b.

S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. SOSP*, pages 31–44. ACM, 2007.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. FMCO*, pages 266–296, 2006.

B. J. Corcoran, N. Swamy, and M. Hicks. Cross-tier, label-based security enforcement for web applications. In *Proc. SIGMOD*, pages 931–942. ACM, 2009.

S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. VLDB*, pages 330–341. Morgan Kaufmann Publishers Inc., 1996.

D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proc. SIGMOD*, pages 331–342. ACM, 2001.

H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein. Support for relaxed currency and consistency constraints in MTCache. In *Proc. SIGMOD*, pages 937–938. ACM, 2004.

M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: A C-based language for self-adjusting computation. In *Proc. PLDI*, pages 25–37. ACM, 2009.

Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, June 2014.

P.-A. Larson and H. Z. Yang. Computing queries from derived relations. In *Proc. VLDB*, pages 259–269. VLDB Endowment, 1985.

W.-S. Li, K. S. Candan, W.-P. Hsiung, O. Po, D. Agrawal, Q. Luo, W.-K. W. Huang, Y. Akca, and C. Yilmaz. Cache Portal: Technology for accelerating database-driven e-commerce Web sites. In *VLDB*, pages 699–700, 2001.

E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. SIGMOD*, pages 706–706. ACM, 2006.

A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *Proc. ICFP*, pages 307–319. ACM, 2011.

D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proc. OSDI*, pages 1–15. USENIX Association, 2010.

K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proc. SIGMOD*, pages 133–144. ACM, 2012.

G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proc. POPL*, pages 502–510. ACM, 1993.

T. Rompf and N. Amin. Functional pearl: A SQL to C compiler in 500 lines of code. In *Proc. ICFP*, pages 2–9. ACM, 2015.

A. Shankar and R. Bodík. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Proc. PLDI*, pages 310–319. ACM, 2007.

D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3): 325–363, Sept. 1995.