Sync Kit: A Persistent Client-Side Database Caching Toolkit for Data Intensive Websites

Edward Benson, Adam Marcus, David Karger, Samuel Madden
{eob,marcua,karger,madden}@csail.mit.edu
MIT CSAIL

ABSTRACT

We introduce a client-server toolkit called Sync Kit that demonstrates how client-side database storage can improve the performance of data intensive websites. Sync Kit is designed to make use of the embedded relational database defined in the upcoming HTML5 standard to offload some data storage and processing from a web server onto the web browsers to which it serves content. Our toolkit provides various strategies for synchronizing relational database tables between the browser and the web server, along with a client-side template library so that portions web applications may be executed client-side. Unlike prior work in this area, Sync Kit persists both templates and data in the browser across web sessions, increasing the number of concurrent connections a server can handle by up to a factor of four versus that of a traditional server-only web stack and a factor of three versus a recent template caching approach.

Categories and Subject Descriptors: H.2 [**Information Systems**]: Database Management

General Terms: Design, Measurement, Performance.

Keywords: Cache, Client-side, Web, Browser.

1. INTRODUCTION

To support the increasingly sophisticated and feature-rich applications "hosted" on the web today, programmers who deploy them must run complex and sophisticated server infrastructures. Unlike desktop applications, where all of the processing and computation is done locally, most web applications rely on the server to fetch and process data and, often, to render that data into HTML for presentation on the client. Building servers that can scale is a tremendous challenge, of which a significant component is managing load against back-end database systems. Indeed, many companies (most famously Twitter) have experienced widely publicized database-system failures due to their tremendous growth, and have invested great effort into sophisticated database partitioning and caching infrastructures to reduce and spread database load.

The tight coupling of applications to a web server can also generate significant latency in user interactions. Web applications are located "far" from the data they present to users, and each data item may take many milliseconds to retrieve as requests are sent to servers in different parts of the world. When many data items are retrieved, latencies can grow to be seconds long. Asynchronous technologies such as AJAX allow web applications to remain responsive during these requests, but designing highly responsive web applications in the face of these latencies, especially on bandwidth-impaired devices such as phones, remains a challenge.

One way to address the database load problem and latency problem would be to offload data processing to web browsers. If browsers could access some or all of the data needed to satisfy a request from a local cache, then pages would load faster and server-side databases would do less work. This is especially true in applications such as Facebook or Twitter where the same data items (e.g., posts on a user's news feed) are sent again and again as users reload pages in search of new messages. In such scenarios, traditional web caching methodologies are too coarse-grained—web pages may share a significant amount of data across repeated accesses, but page content is still dynamic.

Fortunately, the upcoming HTML5 standard includes a client-side persistent database API that makes it possible to instantiate and store databases inside the web-browser. The standard specifies that this store is to be a SQL-compliant database accessible through a JavaScript API. Though it was initially designed to support offline access to web applications via cached data, the same technology can be used to offload data processing and reduce server-communication latency even when the client is online. To exploit this technology, however, application developers have to manually manage the cache, modifying their queries to take explicit advantage of the client-side data, and writing code to determine if cached results are still valid, and to merge cached results with updates from the server.

To address this complexity, we built a toolkit, Sync Kit, that allows application developers to easily and efficiently take advantage of client-databases. Sync Kit provides a library of data structures for commonly used synchronization patterns (e.g., a queue of objects, each of which corresponds to one result in a database query) that programmers use to express their application. When these structures are used, Sync Kit generates code for the client that populates a client-side cache of recently accessed items and reuses the contents of the cache to produce web pages, fetching new or uncacheable results from the backend server automatically. Because database results are cached client-side, Sync Kit provides a templating library that allows programmers to describe how to generate a web page from the cached query results.

Furthermore, Sync Kit's caches can be shared across a user's sessions; for example, if a user quits her browser and

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

then restarts it hours later, the cache from the previous session can still be used.

In summary, the contributions of this work are to:

- Identify data access patterns that are widely used on the web, and build corresponding data structures that programmers can use to build their applications while benefiting from client-side database caching that can achieve the same level of data consistency as their original applications.
- Demonstrate that these data structures, when used properly, can reduce server load. On two realistic benchmarks based on a blog and a wiki, Sync Kit reduces server load by a factor of three versus our implementation of a recently published template-caching approach and a factor of four versus the traditional web hosting stack. The data structures also significantly reduce data transfer to the client to up to 5% of its amount in the traditional stack.
- Show that these ideas can be integrated into Sync Kit, a practical and easy to use web-programming toolkit that requires no browser modifications and can be implemented via a simple app-server framework that runs on the server.

2. ARCHITECTURE AND OVERVIEW

In this section, we describe the basic operation of Sync Kit, including the programming model it presents to developers and the flow of data as it presents a page to users.

2.1 Programming Model

Programmers using Sync Kit are required to design two types of structures: *data endpoints*, which describe the data that will be used to render a web page, and *templates* that describe how to generate HTML from data endpoints. In our examples, we use the SQL language to describe data served by data endpoints, but in practice different web programmings frameworks (e.g., Django, Ruby) may provide their own query language. The queries and data structures are written in terms of database tables to which the server has access.

In the remainder of this section, we introduce our programming model through a series of examples.

2.1.1 Data Endpoints

An example data endpoint for a simple blogging application is shown in Figure 1; it defines two queries, `entry_data` and `tag_data`. `entry_data` defines a list of the ten most recent blog entries that will be presented to the user, and `tag_data` defines a set of tags describing those entries. Note the use of the `%entry_data%` syntax that makes it possible for one data endpoint to reference another as a nested query. In this example, we say that `entry_data`, whose contents decide the results of the `tag_data` query, is the *parent* of `tag_data`.

The resulting relations from these endpoint queries can then be used in templates. For example, the template shown in Figure 2 constructs an HTML page that contains one block on the page for each blog entry and its tags. Note that templates also include SQL, where the tables that are accessible to these queries correspond to the endpoints defined in the endpoint definition.

```
def blogDataEndpoint(clientRequestData):
  entry_data = QUERY("SELECT id, author,
                        title, body, lastModified
                      FROM entries
                      WHERE published = True
                      ORDER BY lastModified DESC LIMIT 10")
  tag_data = QUERY("SELECT entryid, tagid, tagstring
                 FROM tags
                 WHERE entryid IN (
                   SELECT id
                   FROM %entry_data%)")
  return HTTPResponse(to_json([entry_data, tag_data]))
```

Figure 1: An example data endpoint containing two data sets, `entry_data` and `tag_data`. The endpoint returns JSON data to the client to fill in a template.

2.1.2 Templates

Data endpoints are made visible to the web user by combining them with templates. These templates work much like the templates used in any popular server-side web programming environment, except they are executed on the client-side, using a JavaScript template parser, instead of being executed on the server.

Though not as popular as server-side solutions, a number of client-side template libraries are currently in use on the web [14, 5, 3], all of which operate around the similar principle that some data object, expressed in JSON, is to be combined with a declarative template to produce the final HTML for a page. XSLT [11] also provides a standard way to transform structured data results into web pages.

Any of these existing client-side template languages could be adapted to work with Sync Kit's manner of operation. Sync Kit simply needs some way to combine data results and template code once the client is in possession of both.

Our current template library is based off of the HTML5 Microdata specification. This specification provides a way to use HTML attributes to relate data entities and their properties to regions of the web document, much like RDFa. Other HTML5 additions allow for the creation of custom tag attributes that we use to supplement this semantic markup with the types of simple control structures that all template languages must provide. Figure 2 provides an example of what a Sync Kit template looks like for a table of blog entries. We make use of both upcoming HTML5 tags and attributes in this example.

```
<section id="blog_posts"
  data-query="SELECT title, author, body FROM entry_data
              ORDER BY lastModified DESC LIMIT 10"
  data-as="Entry">
    <article itemscope itemtype="Entry">
       <h2 itemprop="title"></h2>
       By <span class="author" itemprop="author"></span>
       <div class="contents" itemprop="body"></div>
    </article>
</section>
```

Figure 2: A Sync Kit template for a list of blog entries. The template uses data endpoints from Figure 1 as tables referenced from SQL queries.

In Figure 2, we see an ordinary HTML fragment decorated with tag attributes that define the way in which data from the client-side database (having been synchronized using the data endpoints previously specified) should fill the template.

On the `section` element, the `data-query` attribute specifies the SQL query to perform and the `data-as` property provides a name with which to reference a row from the result (`Entry`). Later, an item is defined (using the `itemscope` attribute) that has the same class type as the query result rows – this lets the template library know that this portion of HTML is to be repeated for each row of the result. The template library then iterates over the rows of the `Entry` result, filling in the template for each row, resulting in the output shown in Figure 3, shown truncated to only one blog post. This output is both a properly rendered HTML fragment and a machine-readable HTML5 Microdata document.

```
<section id="blog_posts">
   <article itemscope itemtype="Entry">
       <h2 itemprop="title">My Thoughts on HTML5</h2>
       By <span class="author" itemprop="author">
          Tim Berners-Lee
       </span>
       <div class="contents" itemprop="body">
          The HTML5 working group has been...
       </div>
    </article>
</section>
```

Figure 3: The output of the template in Figure 2.

2.1.3 Putting the Pieces Together

Using the Sync Kit framework, web developers do not have to veer far from their current mental model of operation. Web developers are used to thinking about their programs as being run on the server and delivered to the client. Under this new model, the same remains true, but instead of delivering a rendered page, the server delivers a tuple that specifies both the template to be rendered and the data endpoints that provide the data to fill the template. Using the blog example, this tuple is `["blog.html", [entry_data, tag_data]]`.

The Sync Kit framework handles the rest, ensuring data synchronization between client and server databases, template operations on the client, and caching using both HTTP and HTML5 Manifest caching schemes. We now describe how this synchronization is actually performed, resulting in a substantial reduction in load and bandwidth on the server.

2.2 Execution Model

At a high level, Sync Kit execution is handled by the *Sync Kit server*, which runs inside the web server (as a collection of scripts for the Django [1] web framework, in our implementation), and the *Sync Kit client library* (written in JavaScript), which is run by the client browser.

We now describe Sync Kit's operation with no template or data caching. When the browser requests a Sync Kit-enabled page, the Sync Kit server sends it the template, as in Figure 2. Complex pages are stored across multiple templates, which are stitched together. This template also references Sync Kit's client library, `synckit.js`, and contains all data endpoint definitions used by the template. The Sync Kit client library registers a callback into the browser's `onLoad` event, and takes control once the page template has loaded in the browser. The Sync Kit client library then sends an asynchronous HTTP request to the server and requests the current data for all endpoints in the page template. The server sends back a JSON object containing this data, which the client library uses to populate the template it received.

So far, we have described a mode of operation that is essentially the way most AJAX applications work. From a bandwidth and server load perspective, this approach is only marginally more efficient than a "traditional" architecture where the server is responsible for populating the template. The AJAX approach reduces bandwidth consumption slightly if the size of a repeated template is large compared to the size of the data being used to populate each instance of the template, for example.

There are two opportunities for performance gains through caching, however, which can dramatically improve the situation. First, if a template has been fetched before, it does not need to be fetched again for some time. In some cases, this can result in significant bandwidth savings, as was noted by Tatsubori and Suzumura [28].

The second and more interesting opportunity for caching arises with data endpoints. HTML5 provides facilities for the browser to store relational data in a persistent client-side database, which can be used to store the data endpoint results it fetches. Rather than re-fetching the entire data endpoint, it can request from the Sync Kit server only the contents of the endpoint that have changed (for example, the new blog posts that have been written since it last contacted the server.) It can then combine these new results with the records cached in the client-side database and use these combined results to populate the template. We describe how this caching works in more detail in Section 3.

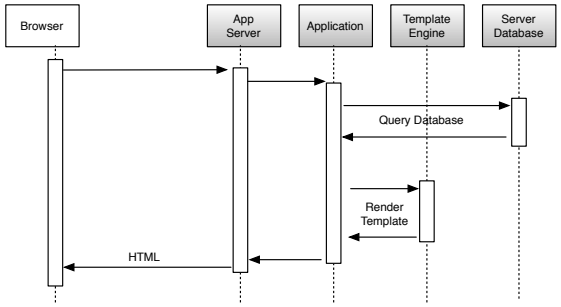
Figure 4 compares the traditional server-side form of web hosting (Figure 4a) with a template-caching approach due to Tatsubori and Suzumura [28] (Figure 4b) and with Sync Kit (Figure 4c), which caches both data and templates.

3. SYNCHRONIZATION STRUCTURES

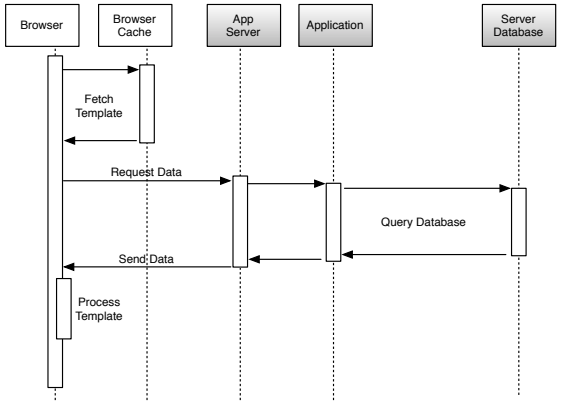
The previous section described our approach to caching. The browser caches results and endpoints issue queries to fetch results since the last cache update. Realizing this approach is difficult for two reasons. First, identifying the cached portion of a query requires semantic analysis of schemas and queries to determine the portion of a new query that intersects with previously cached results. Second, rewriting queries to use caches in way that actually reduces load on the server is a known hard problem in the database community. The main challenge is that the simplest way to reuse a cached result is to rewrite the query to include a complex set of WHERE predicates that exclude all of the tuples in the cache; such predicates slow query execution as they have to be evaluated on each tuple, and often don't reduce the data the database has to read from disk unless appropriate indices happen to be available.

The problem is usually called *semantic caching* and has been well-studied (e.g., [27, 18, 26, 16, 19], and many others.) Jónsson [26] shows that existing database systems do not perform well when presented with arbitrary, complex queries to retrieve cached results. As a result, most high performance semantic caching systems require modifications to the backend database to keep track of changes since the last access, something we wanted to avoid in Sync Kit.

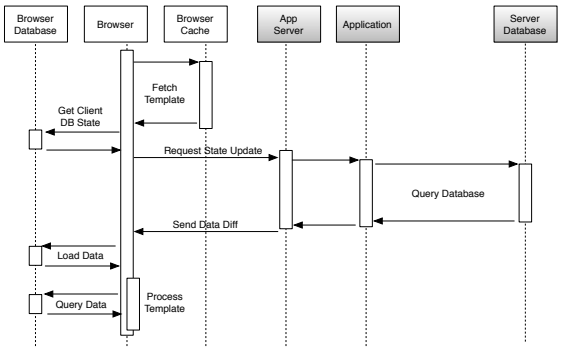
In Sync Kit, we take a simpler approach that requires no modifications to the database: programmers write their data endpoints in terms of data structures that make it easy to determine what has changed since they were last loaded. Sync Kit currently provides two such *synchronization structures*: queues and sets. We discuss other possible synchronization



(a) Traditional Control Flow: Template Rendering and Database Access are on Server



(b) Template Caching: Template Rendering is on Client



(c) Sync Kit: Template Rendering is on Client, and Database Accesses are Cached on Client

Figure 4: Web page rendering architectures. White boxes are client-side and gray boxes are server-side.

structures and improvements on the ones described in this section in Section 6.

3.1 Queues

Queues capture the idea that results are ordered on some attribute (say, time) and that this ordering reflects the way the client will access the data. This makes it easy for the client library to fetch new data items that have been created or changed since the last page load by simply sending the maximum (or minimum) value currently in the client-side cache of the queue. This abstraction is particularly useful for synchronizing information that fits the 'feed' format, which characterizes a number of websites, including any news website (e.g., `nytimes.com` or `slashdot.org`), email sites (e.g., `gmail.com`), and social networking sites that list updates to friends' statuses (e.g., `facebook.com` or `twitter.com`).

To see how queues are programmed in our model, consider the blog endpoint from Figure 1. Suppose the programmer knows that blog entries are always accessed in time order by the template in Figure 2. She can then declare that the blog entry endpoint is a queue, as follows:

```
entry_data = QUEUE(on = "lastModified"
             table   = "entries"
             order   = "DESC"
             include = "id, author, title, body, lastModified"
             filter  = "published = True"
             limit   = 10)
```

Here we see a queue synchronization object built around the entries table in reverse order by the date field. The queue is limited to 10 items and contains further information, such as which projected fields to include and how to filter the queue inputs (in this case, only messages that have a `published` variable set to `True`). The synchronization specification is similar to the SQL APIs offered by web toolkits such as Ruby on Rails [6] and Django [1], but rather than creating a query result set, it defines data endpoint capable of synchronizing the result set over multiple web sessions with a minimal amount of information exchange.

The first time the client library loads this endpoint, a SQL query identical to the one shown in Figure 1 will be run. The Sync Kit server will also send the timestamp when these results were fetched from the database, the table schema that the client-side database should maintain, and the parameters which define the queue synchronization structure.

Subsequently, whenever the client reloads this endpoint, it provides the maximum `lastModified` value in its version of the synchronized `entry_data` endpoint. The server will add a predicate to the WHERE clause of the query to only retrieve data that is more recent than the client's `lastModified` value. If few entries have been added to the blog since the last client visit, the server will fetch fewer results, requiring less work if the `entries` table is properly indexed. The server will also send less data to the client.

Upon receipt of new data, the client-side library merges the new results with those already in the browser-side database. Now that the client-side database state has been refreshed with the current data, the template library will run. It will fetch the top 10 blog entries from the client-side database and fill out the cached template with them. Finally, the client can discard entries outside the new top 10 from the browser's database, as will be discussed in Section 3.5.

3.2 Sets

We now turn to Sets, which are another abstraction provided by Sync Kit to perform client-server data synchronization. Sets capture a basket of data that is unordered from the perspective of the client. Each data item in the basket is identified by some key, a role usually served by a primary key in a relational database. Examples of such sets of items are products in a web-based store, movies and actors on the website IMDB, pages on wikis, and the tags used to describe blog posts in the previous example.

Sync Kit maintains a notion of two different types of sets. *Complete Sets* are sets that are actively transferred in their entirety to the client. After synchronizing with a complete set endpoint, the client is guaranteed to have the entirety

of the set described. Attributes of members of the set may change, and items can leave or enter a complete set over time. One example of a complete set is the tags in the previous example—the number of tags on a blog is small enough that it can be sent to the client in its entirety. On the other hand, one would not want to transfer the entire set of pages in a large site such as Wikipedia to the client the first time a user requests a single page on the site. *Partial Sets* contain members that are lazily transferred to the client on a primary key lookup.

3.3 Complete Sets

Complete sets are a useful abstraction for relatively small collections of data that see frequent client use but do not fit the access pattern defined by the queue structure. Because access is random and frequent, and the cost of transferring the entire set is low, the client and server coordinate to ensure that the client has a fresh copy of all set items.

For example, suppose our programmer from the Queue example would like to add tags to the blog entries view, as in Figure 1. Remember that the `tag_data` endpoint requires a nested query to its parent endpoint, `entry_data`. Our complete set of tags is defined by the tags table as well as the entries that the user will see which will require tags:

```
tag_data = SET(type = "complete"
               table = "tags"
               parent = [entry_data,
                         "entryid = entry_data.id"]
               key = "tagid"
               include = "entryid, tagid, tagstring")
```

This definition follows the one of `tag_data` in the blog data endpoint of Figure 1. We define the set to be a complete replica of the table `tags` for any tag involved in an equality join on the `entry_data` result set. We will request tags by the key `tagid`, and include the fields `entryid`, `tagid`, and `tagstring` from the table.

When a client first queries the data without having synchronized `entry_data` and `tag_data` before, the server will construct the query in Figure 1. For subsequent visits, the client will send the `entry_data` and `tag_data` requests with a list named *tag_data.tagids*, containing the `tagid` values already on the client. The server will construct the query in Figure 1, with an additional WHERE clause predicate indicating that only tags not on the client should be sent:

```
AND tagid NOT IN (tagid1, tagid2, ...)
```

3.4 Partial Sets

Partial sets represent a set of items for which the server does not attempt to maintain a fully synchronized copy on the client. This type of data structure may be used in cases where the set of items is too large to reasonably store on the client-side. Wiki articles are a good example—we would model the corpus articles on Wikipedia as a partial set. A table of pages could be synchronized in the following way:

```
wiki_data = SET(type = "partial"
                table = "wiki_pages"
                key = "id"
                include = "id, title, contents")
```

This definition indicates that the endpoint `wiki_data` can be maintained by performing `id` lookups on the server as the client needs them, and that whenever a desired `id` is requested, the `id`, `title`, and `contents` of the wiki page should be delivered.

Whenever client-side logic requires a given wiki page, the `wiki_data` synchronization set will first look for the page id in the client's local database. If the id is found, it can be returned to the user. If it is not found, the client can issue an AJAX request for the page to be delivered to the client.

3.5 Eviction and Consistency

There are three remaining concerns to make synchronization structures realistic in paralleling the current web experience: keeping the client-side database small, ensuring that the client data mirrors the most up-to-date data on the server, and enforcing endpoint constraints on templates.

Because the client can not support a database of the magnitude that the server can, we define an eviction policy for data in the client-side database. A simple LRU eviction policy works reasonably well, with some caveats. For queues, evicted entries can only be those outside of a distance of `limit` from the maximum `lastModified` date of items in those queues. For complete sets, any evictions must mark the entire set stale, to ensure that future queries are directed to the server to re-synchronize the data structure. Finally, for partial sets, no special precautions are required—evicted elements will be re-requested from the server.

To ensure that the client remains up-to-date with the server, any modifiable entries in the synchronization structures must be noted as such, and a last-modified time or version must be attached to each entry in the underlying server-side table. All requests to an endpoint which contains modifiable items must send a last-access time, which denotes when a client last accessed this endpoint. For results returned by an endpoint during synchronization, only those with modification time or version larger than the client's last-access time will be returned to the client.

Sync Kit currently provides no guarantee that the query executed in a template will be consistent with the endpoint defition. For example, in Figure 2, if the programmer modifies the query to "*LIMIT 100*," the client-side cache of the `entry_data` table is only defined and guaranteed to contain the 10 latest records. The programmer must take care to define the view using the same SQL expression on the server and client. As we discuss in Section 6, we are investigating techniques to automatically generate client-side code from a server-side SQL query, partly to eliminate the possibility of mismatch in these view definitions.

4. PERFORMANCE EVALUATION

In this section we compare the performance Sync Kit to our implementation of the Flying Templates [28] approach and to traditional server-side web hosting. In our consideration of the benefits of various approaches, we look at connection throughput, data transferred per request, and the client-side latency of each approach. Our experiments consider two websites we built with Sync Kit: a blog site, in which users revisit the page throughout the day looking for news updates, and a Wiki, where users browse a connected graph of web pages, potentially revisiting some pages over time. We built these websites using content update and hyperlink models from real websites.

4.1 Experimental Environment

The server-side programming environment is the Python-based Django [1] 1.1 web framework. We use nginx as the webserver to serve both static content directly and dynamic

content over FastCGI to running Django instances. The webserver runs Ubuntu 9.10 (kernel 2.6.31), and has an Intel Core 2 processor with four 2.4GHz cores, 8 MB of L2 cache, and 4 GB RAM. The database, which is on the same machine, is Postgres 8.4.2. For the throughput and data transfer tests, our client machine has a 3.2 Ghz Intel Pentium 4 and 2 GB RAM. Both machines are connected to each other over a local network with link bandwidth 112 MB/s reported by netperf and round trip time 1.516ms to transfer 1 byte of data over HTTP. We also ran in-browser timing tests on a netbook running Microsoft Windows XP and Mozilla Firefox 3.5 over a local network with a round trip time of 3ms to transfer 1 byte of data. For throughput tests, the client gradually increased its request rate using httperf until it identified the point at which the server stopped responding to all requests with HTTP 200/OK.

4.2 Benchmarked Systems

In assessing Sync Kit, we compare three systems:

Traditional. All template and data processing is performed on the server. Controller logic on the server performs queries against a server-side database, and the results are filled in on a server-side template, which delivers HTML to the client. The process is implemented with standard components in the Django web development framework.

Flying Templates. When a user first visits a site, they retrieve a template which is subsequently cached. The template issues AJAX requests to the server, which queries the server-side database and returns results to the client as JSON. The client-side JavaScript then fills in the template with the returned data. Django is used to generate the result set as JSON, and we wrote a custom JavaScript library for filling in the static template. This system is similar to the one described in the work of Tatsubori and Suzumura [28], although the implementation is our own.

Sync Kit. When a user first visits a site, they retrieve a template which is subsequently cached. Like Flying Templates, HTML generation from the template is performed on the client-side and data is retrieved from the server. Unlike Flying Templates, the JavaScript library initializes a client-side database using Google Gears [2] in which all data is stored and which is synchronized with the server using the managed data structures described in Section 3. We selected Gears because the HTML5 standard is still in flux, and as of this writing no browser implements both the HTML5 data and caching proposals completely.

4.3 Benchmarks

We implemented our blog and wiki websites for the three systems listed above. For both sites, we built a benchmark based on a sample dataset and sample workload using data from real websites. httperf was used to determine the performance of each workload on each of the three systems. Overall, the total number of lines of code written to implement the blog and wiki sites was the same across all three approaches (typically within a few lines of code) outside of the included Sync Kit libraries. This is significant because it suggests that the Sync Kit approach can be made practical from a programming standpoint.

4.3.1 Blog Benchmark

Blogs are representative of a queue-heavy workload—when a user visits a blog's front page, around ten of the most recent stories on the blog are displayed to the user. A user who visits frequently will see some new stories and some older repeated ones. Such experiences occur on sites other than blogs—web search, web-based email, and social networking sites such as Facebook or Twitter are all similar.

In order to generate a representative workload, we modeled our benchmark on popular blogs in the wild. We requested the latest RSS feed from several popular blogs, and report their time between posts and post sizes in Table 1. From these, we selected TechCrunch to parameterize a script which loaded a server-side database with three years of randomly generated content, based on a normal distribution of post length ($\mu = 5487$, $\sigma = 4349$) and an exponential distribution of time between posts ($\lambda = .53 posts/hour$).

We re-use the same template of size $100KB$ for all three serving strategies. This template consists of basic HTML, CSS, and stanard JavaScript libraries, of which SyncKit is a small fraction. All CSS and JavaScript is inlined.

We constructed several client workloads for this site to examine its performance for clients who re-visit the site at varying frequencies with respect to the update frequency. For each i^{th} client workload generated, we modeled users visiting the site over seven days at a rate of λ_i relative to the mean time between posts for the blog. We vary λ_i between .008 visits per new post (infrequent visits) and 3.8 visits per new post (frequent visits). For each visit-per-post frequency, we added users until we had generated 10,000 requests. In all cases this resulted in more than 100 users per workload.

Testing with a variety of user visit frequencies is useful because it frees our analysis from dependence on the content update frequency that parameterized our blog test data. It is also useful because user visit patterns to a blog tend to be independent of the blog's popularity [20], so a variety of visit frequencies better reflects real-world workloads.

The first time a user visits the site, both Sync Kit and Flying Templates request and cache the template for the site. To model this, we made half of the users new users to the site, causing their first request to include both data and template requests. Varying the fraction of new users to the site did not significantly affect the performance differences between systems.

On each visit, the client requests the latest 10 articles. To simulate time, each client sends a *currenttime* to the server, indicating the time at which the page is requested.

For the traditional and Flying Templates approaches, a SQL query of this form is issued on the server-side:

```
SELECT id, author, title, contents, lastModified
FROM articles
WHERE lastModified < CLIENT_PARAMS["currenttime"]
ORDER BY lastModified DESC
LIMIT 10;
```

The following Sync Kit queue manages the client cache:

```
QUEUE(on = "lastModified"
      table = "articles"
      order = "DESC"
      include = "id, author, title, contents, lastModified"
      limit = 10)
```

In addition to the *currenttime* argument, the Sync Kit client also sends a *maxclienttime* parameter to the server, to indicate the results up to the point which they have synchronized the dataset. The SQL query issued on the server-side is the same as the one above with the following predicate to fetch only results newer than the currently cached ones:

```
AND lastModified > CLIENT_PARAMS["maxclienttime"]
```

Site	Articles Examined	Article Length (kB)	Template Size (kB)	Time Between Articles (min)
Tech Crunch	25	$\mu = 5.5, \sigma = 4.3$	23.1	$\mu = 114, \sigma = 117$
ReadWriteWeb	15	$\mu = 7.5, \sigma = 6.9$	50.1	$\mu = 263, \sigma = 343$
Slashdot	15	$\mu = 3.4, \sigma = 0.6$	64.1	$\mu = 116, \sigma = 54$
Consumerist	40	$\mu = 3, \sigma = 1.6$	22.6	$\mu = 111, \sigma = 226$
Gizmodo	39	$\mu = 2.8, \sigma = 2.5$	49.7	$\mu = 73, \sigma = 114$
Engadget	40	$\mu = 7.2, \sigma = 5$	46.0	$\mu = 85, \sigma = 64$

Table 1: A sample of several popular blogs. Article length is generally an order of magnitude smaller than template size. New articles come out every one to two hours. If a user visits the front page, which displays multiple articles, several times per day, they may see the same article more than once.

4.3.2 Wiki Benchmark

If blogs are a prototypical representatives of the queue synchronization structure, wikis are good representatives of a set. A wiki (e.g. Wikipedia) can be thought of as a connected graph of primary-keyed data that is too large to send in its entirety to the client. Because of its size, a wiki is synchronized lazily, and thus represents an partial set synchronization pattern. Note that we do not evaluate complete-set synchronization in this paper—these sets are usually small enough to by synchronized in their entirely, or at least as a nested subquery on queues, and we find their performance characteristics less interesting than larger partial sets.

To generate the wiki data set, we combined previous studies of content length and link structure, and suplemented these numbers with a random sample of the pages accessed on Wikipedia from publicly available web proxy logs [10]. We then generated 10,000 articles of random content length ($\mu = 3276B, \sigma = 100B$) [9] and title length ($\mu = 22B, \sigma = 12B$) [10] with an average of 23 [8] links per page. To model article popularity, we assigned a probability of $\frac{1}{\text{article no.}+10}$ to every article, normalized to generate a proper distribution after assignment, which creates a Zipfian distribution to represent hotspots on Wikipedia. Here, lower article numbers are are more popular than higher ones. The +10 prevents the first few articles from dominating all others. This distribution is a good approximation of the actual usage patterns for web resources [15]. Article links were generated such that the source page was selected uniformly and randomly from the page set, and the target page was selected proportional to its assigned popularity probability.

Finally, to generate a workload over the wiki, we modeled 40 users visiting the site over the course of 15 days, once per day. Within a visit, each user picks an initial page i according to the pages' access probabilities, and navigates to linked pages by choosing randomly from the normalized probability distribution of the pages linked from i. We assigned the user an exit probability of .5 after each view, which would end that day's visit for the user. Because users visit the site 15 times, we can see how repeated accesses to the same page affect the performance of Sync Kit. The resulting repeated access rate in three generated workloads was 13.7%–14.9%.

4.4 Results

In this section, we describe the performance results on our two benchmarks. In both benchmarks, we generated three random workloads with the parameters in Section 4.3 and report the average performance of these three.

4.4.1 Blog Benchmark

We tested the traditional, Flying Templates, and Sync Kit approaches on the blog dataset described in Section 4.3.1. We now present the performance of each system under varying user visit frequencies, which are controlled by the λ_i parameter described in the benchmark.

The charts in Figures 5 and 6 display measurements of the maximum throughput and average data transferred per request as λ_i increases, causing the user visit-per-post frequency to increase.

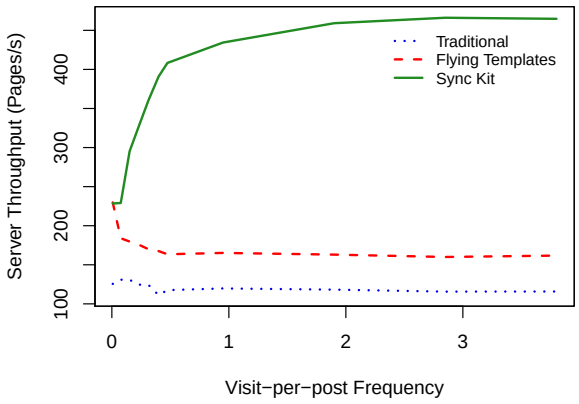


Figure 5: User visit frequency vs throughput.

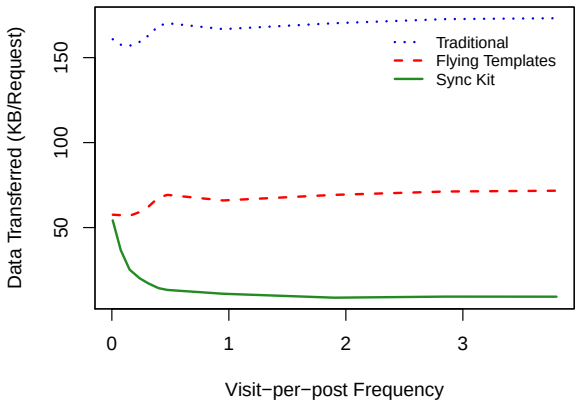


Figure 6: User visit frequency vs KB per request.

In all experiments, the Flying Templates approach provides slightly less than twice the request throughput of the Traditional approach while transferring 100KB less data per

request as it avoids transferring the template. This result is similar to that shown in [28] on a different query workload. Flying Templates sees a drop in throughput between $\lambda_i = 0$ and $\lambda_i = .5$. This is an artifact of our experiment design, as less frequent visitors see a larger share of their traffic come from static templates which are faster for the web server to serve. For infrequently visiting clients, Flying Templates and Sync Kit perform about the same, as Sync Kit is able to cache the template from one visit to the next, but there is a very low probability of any article still being in Sync Kit's data cache on the next page load. For clients who revisit more frequently, however, Sync Kit's cache helps dramatically. At the extreme, Sync Kit is able to serve a factor of four (464 vs. 116) more requests per second than the traditional approach, and nearly a factor of three more than the Flying Templates approach. It also requires around 13.2% the data transfer of Flying Templates, and around 5.4% the data transfer of Traditional.

We now look at the latency from the standpoint of a client of the system. We ran client-side requests from a netbook on the same network as the server with connection properties described above and $\lambda_i = .31$. The results are shown in Figure 7; on the X axis are the three systems with the total height of each bar representing the average latency to load a page in each system. All three approaches have approximately the same client latency (around 400 ms/request). Note that Sync Kit improves server-side performance without hurting the client's experience.

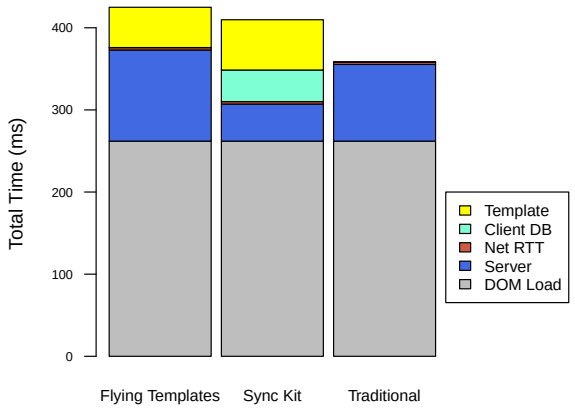


Figure 7: Client latency for blog workload ($\lambda_i = .31$).

To understand how this latency breaks down, we now look at the components of the bars in a bit more detail. Looking at the "server" component, it is clear that Sync Kit does substantially reduce the total time spent waiting for data from the server—from 93 ms in the traditional case to 45 ms in Sync Kit. However, Sync Kit spends an additional 38 ms loading data into the client database, and 61 ms populating the template. Note that in all three scenarios, "DOM Load," which represents the time to load the DOM of a page into the browser, dominates the client's latency. To measure DOM Load time, we loaded the page into the browser cache and measured the time until the "onLoad" JavaScript event. All three systems also incur a negligible 3 ms network round trip overhead. Flying Templates also performs similarly; it sends more time waiting for data from the server than Sync Kit, but does not have to populate the client-side database.

4.4.2 *Wiki Benchmark*

We ran the same experiments from the blog benchmark with our set-based wiki benchmark. Figure 8 shows the throughput (top) and mean kilobytes per request (bottom) for the wiki experiment. Since there is no varying visitation rate, we didn't vary the time parameter in this experiment, and so these are bar charts rather than line charts. From these results, it is evident that Sync Kit still has a large benefit over the Traditional approach both in terms of a severe data transfer reduction and in terms of increased throughput. Sync Kit offers slightly better throughput and slightly decreased bandwidth relative to Flying Templates due to the 14% cache hit rate for revisited wiki pages per user. This signals that improved performance through prefetching might result in better performance, but the ultimate performance benefit will depend on the predictability of client browsing.

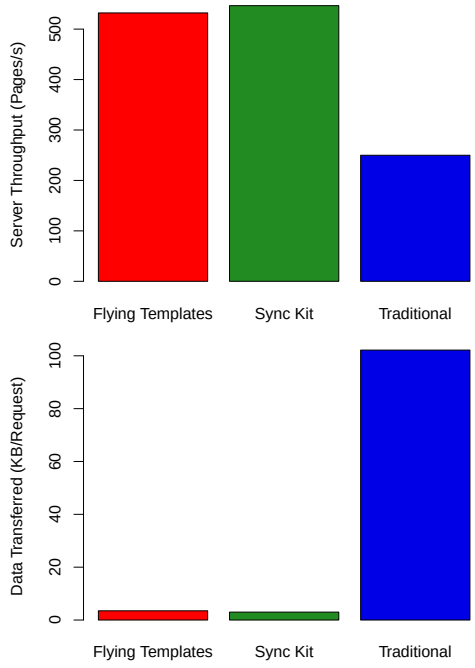


Figure 8: Server throughput (top) and data transfer per request (bottom) for the wiki benchmark.

Figure 9 shows the latency results from a client perspective (again measured from a home computer) for the three systems. The results are similar to those shown in Figure 7: overall, the differences in latencies between the three systems are small; Sync Kit spends a little more time than Flying Templates on the more complex queries it runs to send its state to the server, but the difference is is negligible. Again, the total time is dominated by DOM Load.

5. RELATED WORK

In the FlyingTemplates [28] system, HTML templates are cached on and populated locally by the client. Templates are sent to the client, where they are cached using the browser's native mechanisms. On page load, a JavaScript library attached to each web page queries the server for the appropriate data and combines the data with the page template. The authors show that this techniques yields up to a 2x

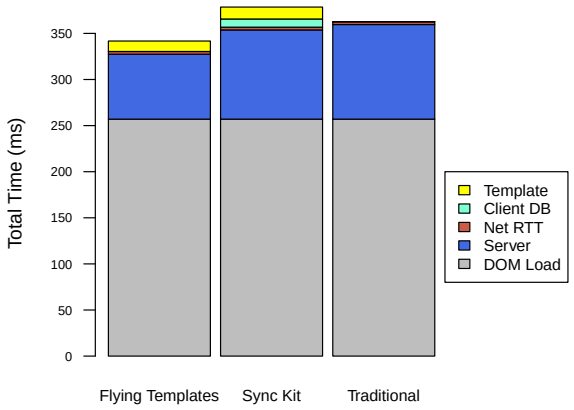


Figure 9: Client latency for wiki workload.

throughput improvement in applications where the HTML is substantially larger than the raw data used to populate a web page. Sync Kit offers the same benefits as FlyingTemplates, but also is able to cache the *data* behind a template, in addition to just caching the template. We compared explicitly to this approach in Section 4.

The Hilda [32, 31, 24] system executes both data operations and template operations on the client site within the context of a single web browsing session. It samples server log files to determine which templates and data elements are most likely to be accessed within a user's session, and then preemptively sends those portions of the web application to the client. When browsing the site, pre-fetched portions of the web application can be accessed without contacting the server. Unlike Sync Kit, Hilda requires developers to build their entire web application in an unfamiliar declarative language; this is what enables the system to move computation and data from client to server. Hilda does not consider data persistence on the client. Orchestra [17] performs similar partitioning of applications written in Java into a server-side and a client side component.

Ganesh [29] is a caching system for dynamic database data that, rather than exploiting specific properties of application data structures, uses cryptographic hashing to identify portions of query results similar to results that have already been returned and reuses them. This approach has the advantage that it is transparent to the application developer, but does not exploit in-client caches as we do.

There has been considerable work on client-side caching in database systems [30, 22, 21], but this work has typically assumed that there is a stand-alone database application running on the client, rather than pushing caching into the browser. In this work, it is assumed that these client-side applications can interface with the backend database below the SQL layer, specifying exactly the tuples or ranges of records that they have in their cache to ensure consistency.

Other database caching approaches include the Ferdinand system [23], which uses a proxy to cache database results. A proxy-based approach has the advantage that it can cache data for many users, but introduces privacy concerns and still requires users to go to a server on the Internet to retrieve cached data. Other database caching techniques – such as DBCache [12], DBProxy [13], and memcached [4] – typically also focus on server-side caches that reduce load on the database but do not have substantial effects on bandwidth and do not push rendering to the client.

Conventional web caching systems (e.g., proxy caches [7]) are similar: they offer the advantage that they work for many clients, but they still require the server to expend bandwidth to transfer data to clients. They are also tricky to get to work for dynamic content. Similarly, browsers locally cache static content, but such caches are not effective for highly dynamic web pages of the sort we consider.

As discussed in Section 3, the technique we employ for determining if a result is in the client-side cache is loosely inspired by work in the database community on semantic caching (e.g., [27, 18, 26, 16, 19]). The primary difference is that we constrain the programmer to access data through a set of synchronization data structures that we have devised, allowing us to efficiently determine if a result from the database is available in the cache. Most relevantly, Chidlovskii and Borghoff [16] observe that web applications are characterized by simple queries and as such are amenable to semantic caching, similar to our observation that a few simple data structures are sufficient to allow caching for many web-backed data intensive applications.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced Sync Kit, a toolkit for making it easy for developers to take advantage of client-side relational databases that will be introduced with HTML5-compliant browsers. Sync Kit uses a simple programming model where users define *data endpoints* that cache database objects on the server and *templates* that describe how to render web pages in terms of these endpoints. This approach requires no browser modifications and is implemented as a simple Python- and JavaScript-based library.

When an endpoint is accessed, its contents are cached in the server-side database and can be reused the next time a template that accesses the endpoint is loaded. Templates are also cached on the client. To ensure that endpoints are kept consistent with the backend database, endpoints can be declared to be *sets* or *queues*, which enables Sync Kit to run efficient SQL queries that identify changes in endpoints since they were added to the cache. Our experiments show that when cache hit rates are high (as with our blog benchmark), the Sync Kit approach performs well—approximately a factor of four better than the traditional approach and a factor of three better than the Flying Templates [28] approach. We also showed that client-side rendering does not negatively impact client-side performance, despite extensive use of JavaScript and the overheads of client-side database access. In short, Sync Kit offers significant performance benefits for data intensive web sites.

Looking forward, there are several ways to extend our work. One direction is to increase the breadth of the synchronization patterns Sync Kit supports. For example, aggregation is an expensive server-side operation that may in some cases be offloaded to the client—one can imagine delivering a compressed data cube [25] to the client in cases where aggregation is frequent. We are also exploring ways to improve the performance of our current synchronization structures. While partial sets can not be completely replicated on the client-side, some prefetching techniques can be employed to return frequently co-accessed results that may satisfy future queries and reduce client-side latency. Instead of forcing programmers to define their own synchronization structures, we are also working on a query workload analyzer to generate or recommend such structures.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their improvements. We also thank David Huynh for his thoughts on improving the performance of client-side database workloads. Our work was supported by NSF and NDSEG fellowships, and the NSF under grant number IIS-0448124.

8. REFERENCES

[1] Django web framework. http://www.djangoproject.com/.
[2] Google gears framework. http://gears.google.com/.
[3] JSON-Template Template Rendering Engine. http://code.google.com/p/json-template/.
[4] Memcached distributed key-value caching system. http://www.danga.com/memcached/.
[5] PURE JavaScript Template Engine. http://beebole.com/pure/.
[6] Ruby on rails web framework. http://www.rubyonrails.org/.
[7] Squid: Optimising Web Delivery. http://www.squid-cache.org/.
[8] Using the wikipedia page-to-page link database. http://users.on.net/~henry/home/wikipedia.htm.
[9] Wikipedia bytes per article, accessed Feb 10, 2010. http://stats.wikimedia.org/EN/TablesArticlesBytesPerArticle.htm.
[10] Wikipedia page counters. http://mituzas.lt/2007/12/10/wikipedia-page-counters/.
[11] Xslt specification. http://www.w3.org/TR/xslt.
[12] M. Altinel, C. Bornh ovd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *VLDB*, pages 718–729, 2003.
[13] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: A dynamic data cache for web applications. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 821–831. IEEE Computer Society, 2003.
[14] E. Benson, J. Meyer, and B. Moschel. Embedded JavaScript. http://embeddedjs.com/.
[15] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *In INFOCOM*, pages 126–134, 1999.
[16] B. Chidlovskii and U. M. Borghoff. Semantic caching of web queries. *The VLDB Journal*, 9(1):2–17, 2000.
[17] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6):31–44, 2007.
[18] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, San Francisco, CA, USA, 1996.
[19] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 259–270, New York, NY, USA, 1998. ACM.
[20] F. Duarte, B. Mattos, A. Bestavros, V. Almeida, and J. Almeida. Traffic characteristics and communication patterns in blogosphere. In *1st International Conference on Weblogs and Social Media (ICWSMâĂŹ06)*, Boulder, Colorado, USA, March 2007. IEEE Computer Society.
[21] M. J. Franklin, M. J. Carey, and M. Livny. Local disk caching for client-server database systems. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 641–655, San Francisco, CA, USA, 1993.
[22] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Trans. Database Syst.*, 22(3):315–363, 1997.
[23] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. *Proc. VLDB Endow.*, 1(1):550–561, 2008.
[24] N. Gerner, F. Yang, A. Demers, J. Gehrke, M. Riedewald, and J. Shanmugasundaram. Automatic client-server partitioning of data-driven web applications. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 760–762, New York, NY, USA, 2006. ACM.
[25] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
[26] B. T. Jónsson. *Application-oriented buffering and caching techniques*. PhD thesis, University of Maryland, College Park, MD, 1999.
[27] B. T. Jónsson, M. Arinbjarnar, B. Thórsson, M. J. Franklin, and D. Srivastava. *ACM Trans. Internet Technol.*, 6(3):302–331, 2006.
[28] M. Tatsubori and T. Suzumura. Html templates that fly: a template engine approach to automated offloading from server to client. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 951–960, New York, NY, USA, 2009. ACM.
[29] N. Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 311–320, New York, NY, USA, 2007. ACM.
[30] Y. Wang and L. A. Rowe. Cache consistency and concurrency control in a client/server dbms architecture. *SIGMOD Rec.*, 20(2):367–376, 1991.
[31] F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 341–350, New York, NY, USA, 2007. ACM.
[32] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-drivenweb applications. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 32, Washington, DC, USA, 2006. IEEE Computer Society.