A Data Aware Web Architecture

by

Edward Benson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

© Massachusetts Institute of Technology 2010. All rights reserved.
A Data Aware Web Architecture

by

Edward Benson

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis describes 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.

Sync Kit 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.

Thesis Supervisor: David R. Karger
Title: Professor
Acknowledgments

Many thanks to my parents for pushing me in the right directions and encouraging me to pursue what I found interesting.

Thank you to Adam Marcus, my co-conspirator on the Sync Kit project and the person with whom most of this work was performed. Also the Haystack and Database groups at MIT CSAIL, the members of which gave helpful feedback along the way.

Thank you to my advisor David Karger for many reasons, but most of all for constantly reminding me, in various ways, not to forget that the reason we ultimately do this type of research is because we believe it will help someone, and that aspect of the problem should not get lost amidst the technical jargon.

Samuel Madden provided invaluable advice and guidance throughout the project, most especially helping to design and analyze the performance tests, but also helping to craft the synchronization methods that Sync Kit employs.

Finally, thank you to my wife Grace, for her unconditional support and encouragement throughout this whole process.
Chapter 1

Introduction

Take a look at your web browser window, and odds are most of its tabs are opened to web-based applications. Web-based email, social networking, online auctions, storefronts: what began as a web of interconnected documents is today a thriving medium for application delivery. Even many of the documents on the web are now applications. Blogs and breaking news on the New York Times website appear to the user in document format, but they are actually dynamically-generated pages from content management systems running on the server-side.

This change to the way in which information is prepared for delivery to users has enormous consequences. But despite the fact that today’s web is driven by dynamic, structured data on the server-side, the majority of the web still appears as a collection of monolithic documents to clients: giant blobs of HTML markup. This is both a problem and an opportunity, for if we could change the web to take advantage of the separation between data and documents end-to-end, many benefits would result:

**Better caching.** The dynamism that makes the modern web so flexible as an application medium also makes it difficult to cache when delivered as pre-rendered HTML text. If anything changes in a pre-rendered HTML page, no matter how small the change, caches are invalidated and browsers must re-request the entire page. As a result, web application servers are overloaded while web clients sit waiting idly for their responses. But if web browsers are aware of underlying
data that makes up a web page, caching, and data reuse, becomes possible at a much finer-grained level. When the New York Times posts a new article on their homepage, for example, the server may need only to send this new headline and link, not an entirely new copy of the homepage as would happen today.

**Reusable data.** Providing a data API for web applications is currently an extra burden on developers: it costs time and money to develop a mechanism to give users access to the structured data that powers a website. But for every dynamic web site, this data API already exists – it is just hidden by the fact that its output is interpolated amidst HTML text before delivery to the user! If this separation of data and HTML is preserved all the way into the client browser, then access to raw data becomes the default, not the costly exception. Innovative mashups, data reuse, structured query, and new applications that make use of this data are sure to follow.

**Reusable templates.** If data and templates are transmitted separately to web clients, then clients have the ability to mix and match templates at will. Merge the news from three web sites into one interface, change the look of someone’s blog, or quickly preview how a new site would look with a variety of styles. This certainly raises questions about branding and control on the web, but it is nevertheless an interesting opportunity. If a user has data but no way to display it, they could even perform template query on the web to find a template whose fields matches the data they have.

**A better web design process.** We believe that a web application architecture built around the idea that data and documents should be separate resources presents a better development model for web applications. Just as CSS provided a way to extract and consolidate styling information on a web page, we believe that similarly extracting dynamic data out of HTML documents will result in an improved design workflow for both designers and developers.

This thesis presents a web framework called Sync Kit that enables a web application’s data and documents to be hosted and cached as separate resources. Sync
Kit works with applications built in the popular Django framework [1] and does not require developers to deviate far from their existing practices as Django developers, but its use enables the true structure of the web application to be exposed to the client-side, allowing far more sophisticated caching and even offline access to dynamic web resources. Sync Kit relies on upcoming features from the HTML 5 specification, such as client-side databases and offline caching, to achieve its results, showing that its proposed changes to web architecture can be made without leaving current (and upcoming) web standards.

Sync Kit represents an architectural argument about what the post-HTML 5 web should look like. Just as CSS split apart a web page’s styling from its structure, this work suggests that HTML 5 browsers will be sufficiently capable to allow splitting web content (data) apart from structure as well. So Sync Kit is a proposal, in the form of a framework, to divide the conversation that takes place between web clients and web servers into three separate tracks: styling track (CSS), a document track (HTML), and a data track. A web response in this architecture is not HTML, but rather a [template, data] pair. This fundamental change is the basis on top of which a range of performance enhancements and side-benefits may be realized, many of which will be discussed in this thesis.
Chapter 2

Background

Sync Kit is a framework that explores a change to the way that web application development and hosting is performed. The techniques employed to do this build upon much related work from the web and database communities. This chapter provides an overview of that related work so that the remainder of the thesis may be read with knowledge both of current development practice and of recent research results relevant to this project.

2.1 Web Architecture

Traditional web application architecture is centered around the idea of dynamic document generation on the server side. This architectural style is likely a result of the web’s origins as a distributed hypertext document repository. Documents could be accessed from anywhere on the internet as long as they resided on a server that supported HTTP, and users could automatically download and view new documents by clicking on hyperlinks embedded within hypertext.

In such an architecture, it is easy to see how a web document could be turned into a web “application” via what essentially amounts to a clever trick on top of the system. By permitting URLs to reference executable files on the server, web requests could trigger remote code execution. If the output of these applications was formatted as HTML, developers could craft applications that ran on the server but were displayed
in the client browser. State could be passed in from the user via HTTP arguments, and hyperlinks and forms on the page provided callbacks into further functionality residing on the server. This type of web application – the Common Gateway Interface (CGI) application – fueled the growth of the web throughout the 1990s.

The complexity of web applications has grown tremendously since the early days of CGI scripts, however. Today web applications rival their desktop counterparts for functionality. Because of the complexity of desktop applications, desktop development environments have long incorporated risk-mitigating practices such as object orientation, design patterns, encapsulation, and architectural decomposition. With the migration of desktop applications onto the web, developers were challenged to reproduce for the web a style of development that could cope with complexity of large-scale projects.

Sync Kit focuses on the MVC style of web development, which is one of the more popular modern web development methodologies used to organize large and complex codebases. Unlike CGI scripting environments such as PHP\(^1\), MVC-style web applications are not composed of URL-addressable scripts in the web server’s file system. Instead, these web applications are programs, running on the server, that take HTTP requests as input and produce some output.

The design pattern these architectures follow is the “Model-View-Controller” pattern, in which the applications functionality is divided into three main modules:

- The **Model** contains code that represents and manages the application’s data. For the web, this usually means the model is responsible for encapsulating database tasks and providing an object oriented representation of the resources in the application’s domain.

- The **View** contains code that specifies the user interface. In web development, the view consists of *templates*, documents whose only purpose is to specify how

\(^1\)MVC frameworks exist for PHP as well, but because of PHP’s origins as a CGI scripting language, even these can not fully enforce this model on to the developer, who at any point can break out of the model by simply echoing raw text to standard out, which will be returned to the web client.
some dynamically generated data (specified in a manner specific to the web framework) should be interpolated into a pre-written HTML document, Mad-Libs style.

- The **Controller** contains the application logic responsible for scripting the conversion of an HTTP request into an HTTP response. It performs any authentication necessary, uses the model to query and possibly alter data in the database, selects a template to use for the response, and then combines dynamic data with the template to produce and HTML document for return to the user.

In MVC web applications, URLs serve as abstract addresses into the application’s functionality rather than actual references to filesystem objects. An application listening on port 80 catches all HTTP requests, and for those whose URL does not correspond to static resources such as an image or JavaScript file, it “routes” the request into the MVC application’s functionality using a set of rules, usually provided in regular expression format. Figure 2-1 depicts this architecture.

![Diagram of MVC-style Web Applications](image-url)

**Figure 2-1: Simplified View of MVC-style Web Applications**

The code that the router selects to respond to a web request is a method on some controller class. Because of the rigidity of the MVC architecture, a common and useful pattern emerges in this controller code: fetch data, fetch template, render HTML. This rendered HTML is then returned to the client as the web response. In pseudocode, this appears as the following:
In the above code, the `BlogController` contains a `recent_blog_posts` method which is capable of responding to a HTTP request. This method uses a model, `BlogArticle`, to fetch recent blog posts from a database encapsulated by that class. It loads a view, the template, from the filesystem, which describes how to convert a list of blog articles into HTML. It then renders the template to produce HTML and returns this HTML to the client.

This thesis assumes this MVC style of web development as a base from which to implement. When we claim that Sync Kit requires little change to existing web development practice, what we mean is that it requires little change to the common interpretation (Django, Ruby on Rails, etc) of MVC web development practice.

We select the MVC web application style to build off of because it provides a particularly clean way to define and separate both the components of a web application and the sequence of operations that take place in the pipeline of serving a web application. In a traditional CGI scripting environment such as PHP, anything goes, which would make it difficult to precisely characterize how Sync Kit operates; the constraints of the MVC application style provide a useful scaffolding on which to build.

### 2.2 Caching on the Web

Sync Kit achieves significant throughput performance gains because it is able to cache both data and templates in the client browser. Templates change infrequently and
may be cached as static resources. Data changes in ways that can be described by the developer to achieve efficient cache updates upon repeat visits. Previous application hosting approaches do not employ these techniques because they do not store both data and templates in the client. This section provides an overview of existing approaches to caching on the web today.

2.2.1 HTTP Caching

Conventional caching on the web (e.g., proxy caches) is very common and offers the advantage that it is transparent to the client. In these systems, HTML documents are cached on the server or in a content distribution network and can be served multiple times without going through the application and data logic to re-produce the HTML. Such systems still require clients and servers to expend bandwidth transferring data on each request, however, and they can be difficult to configure for sites with highly dynamic content. Similarly, browsers have long employed local caching for static content, but such local caches do not work for sites whose content changes over time: web mail, blogs, messages boards, etc.

Conventional caching is generally activated by the developer using the headers of an HTTP response. The Expires header permits the developer to list an absolute time at which the resource being served is to be invalidated. After this point the cache should check back with the origin server for a new version. Alternatively, HTTP 1.1 contains a Cache-Control header that provides an elaboration of the Expires field. The Cache-Control header allows developers to specify a time-to-live field instead, which is convenient because unlike an absolute time, it does not have to be regularly updated by the developer. It further contains several other arguments related to secure connections and revalidation mechanisms.

Once an item’s lease has expired, either because an absolute time has been reached or the time-to-live is up, the browser must check with the server for a new version. HTTP provides ways to do this which prevent retransfer of the document if nothing has changed. The first such way is the If-Modified-Since request header, which enables the web browser to conditionally request a web page only if it has been
HTTP 1.1 introduced a second form of conditional resource request called the ETag. ETags can be thought of as checksums of a resource that are transmitted in the HTTP header information. When an ETag-aware web server sends a file, its header contains this checksum. Clients requesting updates to the resource can refer to this ETag value in their request using the If-None-Match header. If the ETag specified in the If-None-Match field matches the current resource checksum, the server responds with a 304 Not Modified response code, indicating that the client’s cached version remains the most recent. Otherwise, it sends the resource in normal fashion.

While this conventional form of HTML caching is very effective for resources that are static over time, it has limited returns for dynamic resources. In actual practice, developers of dynamic web sites set short, or no, time-to-live values on their pages to ensure that users receive the very latest information. ETags can still help in this situation: the user must re-request the resource each time, the server can avoid sending back a full response if nothing has changed. But ETags only help for this situation if the user is visiting the page more frequently than the page is changing.

ETags still leave room for improvement, however. Caching is performed at the page level, which means that if even only a small portion of a web page is dynamically generated, the ETag for that entire page must change. Because of this, current HTTP caching represents an all-or-nothing approach. If anything changes, everything has changed, and a page re-transfer is required. Sync Kit will improve upon that by providing a caching mechanism that scales in resource requirement with the amount of data that changed on a web page.

2.2.2 Query Caching

Because highly dynamic web pages are so difficult to cache, web developers go to great lengths to speed up the pipeline that generates them. The most common trick applied to this purpose is query caching, which is performed by a software component that sits in between the application’s database driver and the database server. Query
caching works on a fundamentally simple principle: keep a hash-based cache of queries performed and their results [5]. This allows fast lookup of frequent query results without having to recompute them in the database.

While query caching is a good performance optimization for sites with many users and few distinct queries, it has less benefit for sites with great query variability between users. Personalized sites, of which there are a great number, are difficult to accelerate with this model of caching, for example. Consider web-based email: the queries issued for each distinct user are parameterized on their user id, and thus there is no overlap between the queries (and thus their results) from user to user, making server-side query caching essentially a wasted effort.

The Sync Kit library is useful in exactly this scenario because it is essentially performing a form of query caching on the client side. While queries across users have no overlap on such sites from the server’s perspective, from the perspective of any one particular user, queries have nearly perfect overlap. And because these caches are decentralized across the client machines, Sync Kit can scale the cache size with the number of users without burdening the centralized server. Chapter 5 will develop this idea more.

Ganesh [31] 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 [32, 20, 19], 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.
2.2.3 View Maintenance

Once a query’s results have been cached, how can the results be efficiently updated without re-running the entire query? In the database community, this question is known as the view maintenance problem: given a pre-defined query and the relation $R$ that it defines over a database, how can subsequent changes to the database be mapped into changes to $R$. In Sync Kit this is an important problem because browsers who visit a page multiple times have already seen the data involved in producing the page, so Sync Kit seeks to calculate and transmit only updates to that data instead of the full data set.

Existing approaches to solving the view maintenance problem use a combination of database triggers and state storage on the server-side. For any one query $q_i$, it is is possible to derive the list of tables for which create, delete, and update operations have the potential to modify the query’s results. A trigger placed upon these tables then cause the database to re-evaluate the view at the appropriate times. This strategy places the burden of maintaining a view on modification operations (insert, update, delete), which may not always be desirable. For this reason, other approaches propose keeping a log of view maintenance operations that need to be performed in a worker queue which may operate asynchronously in the background or may execute in blocking fashion when a to-be-updated view is queried [35]. This turns view maintenance into a background process in the best case, and places the burden of maintenance on the database querier in the worst case.

Sync Kit performs a form of view maintenance on each client to the web site, but it does so in a way that does not place the burden of maintaining the view on the server. For personalized sites in which the number of unique queries scales with the number of unique users, this results in significant work that can be avoided on the web server. As will be seen, the way in which Sync Kit performs this low-overhead view maintenance will only work for a certain class of queries and will come with it a set of conditions under which client-side caches must be invalidated.
2.2.4 Dynamic CDNs

While Content Distribution Networks (CDNs) have traditionally only served static resources, new methods have been devised to enable them to serve dynamic web sites as well. The Ferdinand project uses the traditional understanding of view maintenance to bring the benefits of edge caching from the static world into the dynamic [21]. In Ferdinand, a web application is simultaneously hosted in several locations around the world, each location possibly offering a different portion of the web application’s functionality. A master is responsible for mutable data operations. In this environment, each slave server can be seen as containing several data views on the master database, as each only contains the regions of the database necessary to perform their part of the web application. When a change occurs in the master database, it determines which views might be affected by that change and then sends a multicast message to the slaves that contain those views.

The Ferdinand project successfully pushes dynamic data caching into a CDN, but its techniques for doing so would not work to push it all the way into the browser as Sync Kit does. Trying to cache data in the client browser with Ferdinand’s approach would require the web server to maintain multicast relationships with every client that has ever used the web application. The resource burden this would place on the server outweighs the benefits such a setup would bring.

2.3 Rich Client Involvement

Several projects have experimented with giving the web browser a larger role in the web application hosting process. This helps offset the load on the server. In particular, the Sync Kit hosting approach stands on the shoulders of the Templates that Fly design [30], which advocated the benefits of placing clients in charge of template rendering. Another system, Hilda, is also notable for its approach to delivering large portions of a web application to the client in bulk in anticipation of a user’s browsing choices. Finally, Google Gears paved the way for systems like Sync Kit to exist within the realm of web standards.
In the Templates that Fly system, HTML templates are cached on the client by the browser’s native caching mechanisms. On subsequent page loads, servers can re-use this cached template and only need to request the data that is to fill it from the server. The authors show that this technique yields up to a 2x server-side throughput improvement in applications where the HTML is substantially larger than the raw data used to populate a web page. Sync Kit utilizes the Flying Templates approach as its base case, but also is able to cache, and incrementally update, the data behind a template, allowing for both offline access and increased throughput gains on the server side. Because Sync Kit builds on the Templates that Fly approach to web hosting, we compare our system explicitly to this approach in Section 6.

The Hilda [34, 33, 22] caches both templates and data on the client side, but it does so only within the context of a single browsing session and requires that both the web application and its data be encoded in a special tree-structured language. Once an application has been running for a sufficiently long time, Hilda contains an optimizer that uses statistics gleaned from server log files to calculate a cut separating the tree into two portions: one high-traffic portion that should be bulk-loaded on the client, and one low-traffic portion that should remain on the server. Once this cut is identified, web users visiting the site receive the high-traffic portion of the site up front, such that no further network traffic is necessary to host it, while the low-traffic portion is hosted on the server for as-needed access. The Hilda system is interesting in two regards: first, it caches data on the client side based on measurements of user activity rather than decisions by the developers. Second, it supports a “mixed hosting” model where some portions the web application are hosted entirely by the client and other portions are hosted by the server. Orchestra [14] performs similar partitioning of applications written in Java into a server-side and a client-side component.

Sync Kit differs from Hilda and Orchestra in several respects. First, its caching mechanisms are persistent across multiple browsing sessions and are designed specifically to cope with data that changes over time. Second, Sync Kit is an enhancement on top of the existing programming model for MVC web applications rather than a replacement for it. Finally, Sync Kit does not automatically decide what portions
of a web application to host on the client side, rather leaving that choice up to the application developer.

Finally, it is impossible to talk about rich client involvement on the web without bringing up Google Gears [3]. The next section of this chapter discusses the importance of HTML 5 to Sync Kit, but many of these HTML 5 features are standardizations of functionality originally provided to browsers through Google’s now “retired” Gears plugin to browsers. In particular, Gears provided a client-side relational database and an offline resource cache, both accessible from a JavaScript API. These two tools provide the raw capabilities needed to achieve, on the client side, the functionality that Sync Kit needs to perform. Some web sites, such as GMail, relied on these tools to accomplish much of what Sync Kit does – client-side data storage, offline resource access, and efficient data synchronization between client and server – but they did so using custom code written with many engineer-hours of effort. Sync Kit provides a standard way to incorporate these types of optimizations into any website with minimal effort and a perspective on how this web architecture fits into the larger picture of web standards and web application design.

2.4 Current Developments

In addition to past work from the web and database communities, we note two important ongoing trends that have influenced this work: HTML5 and the rise of commodity computing.

2.4.1 HTML5

At the time of writing, the HTML5 specification is still very much in flux. Despite that, several key features have emerged in the HTML5 working drafts that have already seen preliminary adoption in web browsers and generated excitement from the larger community of web developers. Because of this, we believe it is reasonable to expect that, while the HTML5 draft may still change before its final publication, the major additions to HTML that it defines, from a high-level standpoint, will remain.
Of these additions, two stand out as particularly relevant to Sync Kit: in-browser databases and an improved browser cache. An additional feature related to the extensions to the HTML attribute space will be discussed in Chapter 4 on Templates.

**In-browser Databases.** HTML5 states that web browsers will have a fully transactional relational database accessible through a JavaScript API. While this feature has been debated, owing to the enormous size of the SQL specification and the consequent ambiguity of which subset of SQL browsers should be responsible for, the move on a general level is much needed. For years web developers have been devising clever ways to persistently cache structured data in the client browser using cookies and Adobe Flash APIs, but these techniques have always been limited in their capability. In-browser databases will finally allow web applications to persistently store relational data across web sessions and perform queries and processing at the client side. Sync Kit will make extensive use of this feature.

**Improved Browser Cache.** A second feature of HTML5 relevant to the application hosting model proposed in this thesis is an improved client cache. HTML5 browsers will have access to an offline cache similar to the one found in Google Gears [3], that serves as a proxy between the browser and the internet. Developers are able to interact with this cache both via manifest files hosted on the server and via a JavaScript API on the client. This cache enables Sync Kit to store page templates fully offline, requiring no initial trip to the web server when a page loads. It also provides a mechanism through which to perform client-initiated template invalidation.

### 2.4.2 Commodity Computing

The recent development in web hosting of interest is the ongoing migration to commodity computing as a hosting architecture. Now famous for its decision, Google was one of the pioneers in forging this trend. Rather than pay high sticker prices for specialized high-powered servers, they opted instead to purchase off-the-shelf commodity hardware. Management of these large clusters required the development of new software platforms, such as GFS [24], MapReduce [23], and BigTable [12], but
with the software abstractions to handle such large clusters in place, scaling became cheap (buy commodity hardware) and easy (plug in a new box).

We note this trend not for its cost-savings to web hosts, but rather for its validation that commodity hardware is perfectly adequate for serving a web application. Therefore there is no hardware-based reason for why client computers – those surfing the web – aren’t perfectly capable themselves of taking on data processing and template rendering tasks of hosting a web application.

In fact, in many respects the client machine is actually better suited to take on these tasks. It has far less latency to the “eyeballs” viewing the screen, for one. When a template is rendered or a data result is updated, it can be displayed immediately on the screen instead of first having to traverse the internet. Second, the client machine is likely far less loaded than the centralized web host. While the web host acts as a bottleneck for all concurrent viewers of the web site, each particular client machine’s CPU is idling while it waits for a response from the network. As will be shown, Sync Kit takes advantage of this imbalance in resource utilization by shifting work from the server onto the client.
Chapter 3
Sync Kit

Sync Kit is a software library that enables web applications to move data storage, data processing, and template rendering tasks onto the client side. It is able to accomplish this by maintaining the separation of data and templates all the way into the web browser. Sync Kit templates come in the form of HTML and are cached in the browser for later re-use. Along with these templates, Sync Kit sends relational data, which is cached in a database in the client-side browser. In the client, the Sync Kit JavaScript library is responsible for querying this database and rendering the results into the cached HTML template.

In addition to decentralizing the work required to serve a web application, Sync Kit takes advantage of the fact that many web users repeatedly visit a site over time. When a user re-visits a Sync Kit-enabled web page, the client browser “synchronizes” its cached copy of the page’s data with the server so that only a minimal amount of work need be performed in order to display the new version of the web site.

Sync Kit delivers four main contributions to the web community:

The Sync Kit Architecture. Sync Kit provides a generalization of a collection of client-side performance enhancements that previously only existed in a handful of custom-written web applications, such as GMail Offline [2]. You might call Sync Kit a library to “Gears-ify” web applications, referencing the fact that the now-retired Google Gears provided an offline static cache and client-side
database, even if only a few mainstream web application were ever written to take full advantage of those capabilities. Sync Kit integrates easily into existing MVC-style web application code, demonstrating that large changes to web hosting style can be achieved with only incremental changes to the way in which web application code is authored.

**Easy Data Synchronization.** Data changes over time, and Sync Kit introduces a method of synchronizing the structured data associated with a web page that only requires developers to specify simple metaphors such as Queues and Sets (see Section 5.1 or even simply raw SQL queries (see Section 5.2).

**Performance Study.** This thesis includes a performance study which indicates that the Sync Kit method of hosting web applications provides significant benefits to server throughput when compared to existing practice.

This chapter discusses the fundamental idea behind Sync Kit, how it works, and the programming model that it requires developers to use. Section 3.1 first examines the pseudocode of a typical web request handler and walks the reader through modifying it so that it uses the Sync Kit API. It then explains why this modification is important and what it allows the library to do. Section 3.2 then illustrates the behavior of Sync Kit as a hosting platform – what conversation takes place between client and server. Finally, remaining questions about server-side code execution are addressed.

Any written document faces a challenge imposed by the necessity of ordering information linearly. In this document, the challenge rears its head with the question of when to talk about the details of data synchronization in Sync Kit. In this chapter, synchronization is referenced in the abstract; it just “happens.” Chapter 5 will focus on this part of the process and will discuss in detail how developers specify the type of synchronization to be performed and how Sync Kit handles the resulting synchronization.
3.1 Programming Model

Sync Kit’s programming model is designed to integrate easily with existing MVC-style web frameworks. Achieving relatively “natural” feeling integration with these environments was a chief design constraint influencing the framework. Previous client-side acceleration frameworks have required large changes to the web programming model, such as writing the application as a Swing-like Java program that is compiled to web format [4] or encoding the application as a hierarchical structure containing data and view templates [34]. This framework demonstrates that rich client-side participation in the application hosting process can be achieved via adjustment, rather than replacement, of the existing model of web development.

Sync Kit is implemented as a plugin to Django, a Python-based web framework. As such, it is meant to augment Django applications rather than provide a complete web framework itself. Django follows the MVC model displayed graphically in Figure 2-1: each request is mapped onto a method on some Python class that acts as a controller. This method then uses model objects to load data and provide it to a template view for rendering. Repeating the code example from Section 2.1, this results in code that matches the pattern of the following pseudocode.

```python
# /blog/recent
class BlogController
    def recent_blog_posts
        data = BlogArticle.find(order = 'date', limit = 10)
        template = Template.load('blog.template')
        html = render(data, template)
        return html
    end
end
```

The Sync Kit programming model introduces one conceptual change to this code. Whereas Django developers today perform queries and render templates, Sync kit instead asks the developer merely to specify the query to be performed and the template to be rendered, without actually doing anything with those objects. This allows
the library to take control over the way in which these two actions are performed, both moving them into client side and performing query rewriting to send incremental data updates to the client.

This change is manifested in two lines of the above code:

```python
# /blog/recent
class BlogController
def recent_blog_posts
    data = Synchronize('BlogArticle', order='date', limit=10)
template = Template.load('blog.template')
bootstrapper = SyncKit(data, template)
    return bootstrapper
end
```

First, where a Django developer would have normally performed a query, the Sync Kit developer instead declares a Synchronization Primitive – a class that represents both a query that takes place on the client side and how to update the results from that query. For now we will just use the abstract method call Synchronize to represent this synchronization primitive, but more detail will be given in Chapter 5.

The second change is the rendering step. Where the Django pseudocode returns the rendered HTML output of some template engine, the Sync Kit developer instead passes both the synchronization primitives and the template to the Sync Kit library. Sync Kit crafts a custom HTML response that consists of a modified version of the HTML template with the Sync Kit JavaScript bootstrapper attached to the head element of the page. This bootstrapper will execute when the page is loaded in the browser and initiate Sync Kit’s synchronization process and template rendering process. Bindings for the particular synchronization primitives for this endpoint are also encoded into the page so the client browser knows what database schema to build and how to request updates for those particular queries.

Note that this bootstrapping page is the web template, it has just been augmented by the Sync Kit library to contain additional information and code to handle data synchronization, storage, and template rendering tasks. Figure 3-1 shows a diagram
of the template after Sync Kit has augmented it.

![Diagram of Sync Kit HTML Response]

Figure 3-1: Contents of a Sync Kit HTML Response

Sync Kit could alternatively function by simply linking to much of the data it currently embeds in the `<head>` element. Some of this information is shared across all Sync Kit pages (such as the Sync Kit JavaScript library), while some of it is particular to each page (such as the schema and synchronization structures). In our prototype, we have chosen to simply embed all of this information inline to make each page self-contained. This simplifies the performance testing process prevents situations where multiple round-trips need to occur to the server.

As a final clarification, the above code represents an idealized depiction of what takes place in a request handler: query data, load a template, and then respond to the client. Sync Kit’s operation does not prevent many other actions from taking place inside this request handler, from other database operations to authentication procedures, for example. Additionally, while the template in the above example is loaded statically from disk, there is no reason why this template couldn’t be dynamically generated before being handed to the Sync Kit library.

### 3.2 Sync Kit Serving Strategy

This section will describe the sequence of interactions that take place once the client browser loads the bootstrapping page prepared by the server. Each time the page
loads, the Sync Kit JavaScript library executes and begins the process of synchronizing, storing, querying, and then rendering data. Figure 3-2 shows a flow chart which describes the decision process involved.

This bootstrapper first checks whether or not this page has been loaded before from the client browser. It does this by examining the contents of the browser database to check for the presence of tables that correspond to the particular synchronization structures that have been defined (and inserted into the head element) of this page.

**If it does not find these tables,** then this is the first time the page has been loaded, so there is no synchronization to perform: all the data necessary for the page load has been pre-packaged into the page itself. Instead, Sync Kit prepares the caching mechanisms for subsequent page requests. It uses data injected into the head element of the page to build a database schema and populate it with the initial set of data necessary to display the page. It additionally stores information about each synchronization primitive used on this page.

**If it does find these tables,** then Sync Kit already has a data cache for this page. It requests each synchronization primitive to load its state from the database. This will be explained in Chapter 5. The client then sends this state to the server to request a data update. It adds the response to the local database.

**In both cases,** the final step is to query the database and perform template operations. Several methods for performing templating are possible here. Chapter 4 discusses issues related to templating in a system like Sync Kit and provides details about the Sync Kit template engine.

This workflow on the client side results in the sequence of client-server interactions depicted in Figures 3-3 and 3-4. Figure 3-3 shows the sequence of interactions that occur the first time a web client visits a Sync Kit-enabled website.

Notice that, from the standpoint of work performed, Sync Kit's first page load essentially acts as Tatsubori and Suzumura's Web Templates that Fly approach except
with extra data persistence operations added on the client side. Figure 3-4 shows the interaction that takes place on this same page when visited a subsequent time by the same user. This subsequent load involves more work on the part of the client than both traditional hosting and the Templates that Fly approach, but that is exactly the point: by shifting work from the server to the equally-capable client, clients suffer little degradation in performance but server resources are freed to handle far more concurrent users.

On subsequent visits to the web site, caching begins to take effect both for static and dynamic resources. The initial HTTP request that would normally be made for a given URL never occurs, as this URL has been associated with an offline resource in the browser's cache. Instead, the page is loaded and parsed from the cache without blocking on the network. When the head element of the page is parsed, the Sync Kit bootstrapper begins executing and first checks whether or not the page has been loaded before by inspecting the client-side database.

In this case, the page has been loaded before, and there is data stored away in the database that Sync Kit will extract to send to the server so that it may calculate an appropriate update. For each data query necessary to populate a template's contents, Sync Kit has maintained, in the client-side database, information about the associated synchronization primitive that defines how to gather enough client-side state such that the server may know how to update the client. The particular strategies employed will be discussed in Chapter 5, but a brief example can be given here for an email inbox. If the server knows the last time a user checked (and cached) their inbox contents, the server can calculate only the new emails that have arrived since that time to send as an update. Therefore the client, in this case, collects the time of the most recent cached message as the state to send to the server. This is an example of the Queue synchronization pattern that will be introduced later.

Sync Kit then sends a request to the server containing all the relevant data cache state. For each synchronization primitive associated with the page, the server calculates the database tuples that need to be sent back to the client so that its local database, when queried during template rendering, will respond with the correct
results for the current time. There are also conditions under which the server must instead invalidate the client’s relational cache and sent it a full update instead. These, too, will be discussed in Chapter 5.

Once this Ajax request for synchronization has completed, Sync Kit takes the results from the server and bulkloads them into the client side database. After bulkloading, the rendering process proceeds just as it did on the first page load: Sync Kit queries the client-side database with the queries necessary to render the page and then combines these query results with the template contained in the page body.
Is a data cache already present in the browser database?

No

Load the page's schema into the browser database

Make AJAX request to server to request data updates

Load updates into the local database

Query the browser database to determine its local version

Query the browser database to retrieve data for page rendering

Render the page template in the body of the page using the query result data

Yes

First page load

$n^{th}$ page load

Figure 3-2: Sync Kit Client-side Bootstrapping Process
Figure 3-3: Sync Kit on the first page load

Figure 3-4: Sync Kit on the $N^{th}$ page load
Chapter 4

Templates

Looking at the design of both Sync Kit and Templates that Fly, it is clear that for either of these two application serving strategies to work, the template system used to craft HTML output must be able to work within a browser. Template languages are a critical but often overlooked part of web framework design. They define the relationship between the data and the presentation, and as such they shape much of the relationship between the developer and the code.

An underlying theme in this thesis is the many benefits that come when data and display are separated in a web application’s design. While a lot of machinery goes into making this possible, templates play a critical role. Sync Kit requires that templates be able to be both cached and processed in the browser. In addition to these requirements, many other features are desirable, such as a system that can co-exist inside of syntactically valid HTML and a template system whose output retains the markup that defines the relationship between the document and the data that fills it.

The nature of the hosting model that this thesis proposes requires a more careful look at template systems from an abstract point of view. This chapter explores this topic and identifies properties of various approaches to web templating that can be used to characterize a template system for comparison to its peers. We define several such properties – procedural, declarative, annotational, and idempotent – and provide examples of each as well as their pros and cons with respect to the decoupled web
that we envision with the Sync Kit project. Finally, this chapter presents the Sync Kit template engine, which is built to exhibit the properties that are ideal for this hosting environment.

4.1 Forms of Templating

Throughout the section, we will provide pseudocode examples of the same scenario in order to illustrate the differences between template approaches. This scenario centers around the address page for a university and involves iteration, output, and a conditional. In pure Python pseudocode, the scenario is this:

```python
print "\"<ul>\""
for contact in contacts:
    print "\"<li>\"" + contact.name
    if contact.position == "\"prof\"":
        print "\"(Professor)\""
    print "\"</li>\""
print "\"</ul>\""
```

The above code is what a web developer might have written back in the days of CGI scripting: a purely procedural program that constructs a web page by writing to the standard output stream. Web applications are sufficiently complex nowadays that few develop this way anymore, but the above code is useful to define the scenario we will use for comparison: create an unordered list (<ul>) of names, and append "(Professor)" if the title is appropriate.

4.1.1 Procedural Templates

Procedural templates are the natural evolution of the traditional CGI script-style of web coding shown in the chapter introduction. This type of template is procedural code that has been written in a format optimized for outputting strings. In fact, the nature of the code has been inverted. In typical coding, a fragment of text is considered code unless specially wrapped with tokens that designate it as program
output. In procedural templates, the opposite is true: a fragment of text is considered output unless it has been wrapped with special tokens that designate it as code.

The prototypical example of a procedural templating environment is PHP, one of the most widely used web programming languages today. Microsoft's ASP and the ERB template engine used by Ruby on Rails are also procedural. In such languages, the contact list above would be implemented as follows:

```
<ul>
    <? foreach $contacts as $contact { ?>
        <li>
            <? echo $contact.name ?>
            <? echo (($contact.position == 'prof') ? '(Professor)' : '') ?>
        </li>
    <? } ?>
</ul>
```

Notice how the inversion of code and program output gives the appearance of code embedded in text, which is a comfortable and very useful way to work for many developers. This code is actually just a syntactic shorthand the traditional, procedural version of the code that is represents, however, displayed below:

```
echo '<ul>';
foreach $contacts as $contact {
    echo '<li>';
    echo $contact.name;
    echo (($contact.position == 'prof') ? '(Professor)' : '');
    echo '</li>';
}
echo '</ul>';
```

Procedural template systems are attractive because they provide developers a way to author web output the same way they author code, intermingling text output and code however necessary. This results in a highly effective way to write small-scale projects, but also has two problems:

First, the lack of separation between business logic, data logic, and program output in procedural template environments tends to create code that is difficult to
maintain in the absence of some strict development framework imposed on top of the system. Ruby on Rails uses a procedural template language, for example, but also imposes a strict MVC organization to application code.

Second, and more importantly, these frameworks prevent a decoupling of data and display because the data output and the display output are simultaneously generated by the same process: they are generated procedurally alongside each other as output of the same underlying code. This presents a problem for a system like Sync Kit, which requires templates to be statically cached on the client side. To some degree, this problem could be avoided if clients and servers both executed a common scripting language – imagine if one were to write a JavaScript interpreter in the style of PHP, for instance. But still, the co-generation of template and data output limits the ability to cache resources, and despite a common languages environmental differences between client and server would still introduce problems. Certain APIs and libraries that may be on the client may not be on the server, or vice-versa. The developer in this case would have to delicately remember which portions of the procedural language were common to both and which were not.

4.1.2 Declaratives Templates

The second type of template language is declarative. Declarative templates are different than procedural in that the template itself exists separately from any executable code that may make use of the template. These template systems generally include control structures, such as conditionals and loops, but they do not allow for arbitrary code execution. Rather, a limited set of commands exist that can be used within the template, and these commands themselves are not executable code but are instead interpreted by some template processor.

Declarative templates are generally processed by providing them, along with some bit of data, to a template engine that knows how to use directives in the template file to perform lookup and replacement based on the data. XSLT [10] and Django
templates are two examples. With XSLT, the data file is an XML document. With Django templates, the data file is a Python dictionary. Other examples of declarative templates include Velocity and Google’s CTemplate system.

This style of templates has two large benefits. First, the restricted set of possible commands makes it feasible to develop template parsers for a variety of diverse platforms – both on the server and in the web browser, for instance – which isn’t possible for procedural templates that permit arbitrary executable code is allowed. Second, declarative template enforce a division of code modularization in which all application logic must take place outside of the template, as the template only contains control structures sufficient to affect the rendering of the template. This is beneficial to a system like Sync Kit because it means that it is reasonable to perform application logic in one location (such as the server) but template rendering in another (such as the client) because the two tasks have been discretized and separated.

Below are two examples of declarative templates, the first is a Django template, which places template directives in the text of a web page, and the second is an XSLT template, which specifies the template directives in an XML document. Notice in the Django example two different types of template “tags”: those with percent signs, indicating a control structure, and those with double-braces, indicating that a piece of data from the current scope should be inserted into the page. While this code may look similar to the PHP code above, it is actually very different, because whereas PHP permits arbitrary code execution, the Django template code below permits little more than the operations shown in the below example.

```html
<ul>
{% for contact in contacts %}
<li>
    {{ contact.name }}
    {% ifequal contact.position "prof" %}
        (Professor)
    {% endifequal %}
</li>
{% endfor %}
</ul>
```
Because Django templates exist inside an HTML page as text content, it is possible for the designer to work with the template language, stubbing out pieces of a design mockup with placeholders, like this: “The conclusion of my thesis is {{profound_thought}}”. But not all of the directives are easy to insert using a WYSIWYG editor, such as the line between the <ul> and <li> elements. Further, once this template is rendered, the directives that comprised the template are removed, erasing the instructions used to build the final document.

XSLT is another declarative template language popular for transforming XML documents from one form into another. XSLT is less often used in web development, likely both because web data sources tend to be database iterators (not XML) and because the XSLT syntax is complicated compared to its peers. The contact list implemented as an XSLT transform might appear as follows:

```xml
<xsl:template match="/">
  <ul>
    <xsl:for-each select="contacts/person">
      <li>
        <xsl:value-of select="name"/>
        <xsl:if test="position.=.prof">
          <xsl:value-of select="position"/>
        </xsl:if>
      </li>
    </xsl:for-each>
  </ul>
</xsl:template>
```

Again, we see a declarative specification for data rendering independent of the data or the code that will ultimately execute it. It doesn’t matter if a JavaScript engine or a FORTRAN engine processes the template, so long as it understands how to parse the declarative language.
4.1.3 Endomorphic Templates

In mathematics, a transformation $f$ is endomorphic if $f : X \rightarrow X$. We will call a template language endomorphic if the unprocessed template is syntactically valid according to the rules of its output format. In other words, the syntax used to express the template is the same as the syntax of the document produced by the template. Note that this classification must be made within the scope of a particular output format: A PHP script used to generate PHP scripts is endomorphic, a PHP script used to generate HTML is not.

Endomorphism with regard to web templates means that the template language is, itself, valid HTML. This is important to a web architecture such as Sync Kit because it is easier for the browser to cache and manipulate web templates if they are contained within HTML.

Examples of template languages that are endomorphic with respect to the web are Django templates, XSLT, and the Sync Kit template language. Examples of non-endomorphic languages include PHP, ERB, and ASP in their default configurations. The division happens to occur along the lines of declarative and procedural here, but it does not have to. If Django templates represented variables for replacement with angular brackets `<? like.this ?>` instead of with curly braces `{{ like.this }}`, then they would not be endomorphic.

4.1.4 Annotational Templates

Annotational templates are endomorphic templates in which the template directives can be authored entirely within the non-visible “annotational” space of the document. Like endomorphism, templates can only be considered annotational with respect to some output format. A template system that is annotational with respect to the web uses template directives that exist only within the tag- and attribute-space of the DOM and not in the text-space. In other words, the template directives decorate the

---

1 using an XHTML interpretation that allows for extended namespaces

2 ERB can be customized, making it endomorphic, but its common configuration in Ruby on Rails is non-endomorphic.
structure of the HTML document, but they don’t appear between tags as text. When rendered using a web browser, these annotations are thus invisible to the user.

XSLT templates, for example, are annotational with respect to the web because they are defined in the tag-space of an XHTML document, making them invisible when rendered in an XHTML browser. Django templates, on the other hand, are not, as its template commands will show up as text visible to the user when rendered as HTML.

The advantage of annotational templates is that they enable the template file to appear as the finished product with regard to design. Because the commands that dictate how the template should be processed are annotations placed behind the human-visible rendering, designers can operate entirely within the realms of design-specific WYSIWYG tools, handing their creations off to developers for annotation without having to factor the coding aspect of web programming into the design process.

It should be noted that non-annotational templates have their own advantages. By pushing the template directives into the visible space, it makes it possible to use existing WYSIWYG tools to perform both design and coding. This works well for variable substitution (e.g.: the double curly-brace tags in Django templates), but it tends to fall apart for control structures. The problem is that control structures often must be placed in regions of the page where text does not often go – such as the loop declaration between the `<ul>` and `<li>` tag, in the Django template example above. Because text nodes are not generally placed in such spaces, WYSIWYG editors will not allow their users to target a text addition to this specific location in the DOM. Consequently, a nonspecialized WYSIWYG editor used for both code and template directives often ends up only able to perform half of the duties necessary to author a full template.

4.1.5 Idempotent Templates

Idempotent templates are endomorphic templates in which the template input can be perfectly recovered from its output. Because of this, the template engine can be
fed its own output n-many times and, along with the same data to pair it with, will continue producing the same output. Figure 4-1 illustrates this process graphically.

None of the template systems discussed thus far in this chapter are idempotent, as the rendering process of each of them involves the removal of the templating information. Consider the Python template, for example, in which the template string `<p>Hello {% name %}</p>` would be replaced with something like `<p>Hello Ted</p>`, destroying the original template information. In fact, for a template system to be both idempotent and usable on the web, it must necessarily also be annotational. Otherwise the web user’s view of the page would be polluted with stray template tags everywhere.

Idempotent templates are useful on the web because they introduce the ability copy and paste the structure of a web page for reuse as a template itself. For years, web users have been able to copy content, but even today copying a web page’s underlying template structure is a manual and time-consuming process. To make a blog hosted on blogger.com look like a blog hosted on wordpress.com, for example, would be a time consuming process requiring the manual deconstruction of the HTML output of one page and then a reconstruction in terms of the dynamic language used by the other system. With idempotent templates, the rendered HTML is the template. To make one site look like another, only the data input need be changed.

![Figure 4-1: Template Idempotency](image)
4.2 HTML 5 and Template Languages

The in-progress HTML5 specification contains two components that are highly relevant to web templates. The first addition, data attributes, allows the HTML developer to arbitrarily add new tag attributes as long as they begin with data-. This permits arbitrary data annotations within HTML, such as

```html
<div id=spaceship data-x=10 data-y=100 data-velocity=99></div>
```

The second addition, dubbed Microdata, might be interpreted as an adaption of the RDFa [26] standard. This addition provides a standard vocabulary for relating semantic entities to the content of a web page. A person’s contact information, for example, might be displayed as:

```html
<div itemprop="Contact">
  <span itemprop="name">Edward Benson</span>
  <span itemprop="nickname">Ted</span>
</div>
```

These annotations state that there exists an item of type “Contact” with two properties: a name of “Edward Benson” and a nickname of “Ted.”

Both of these extensions to the HTML attribute space are important to a system such as Sync Kit because they make it possible to build a endomorphic, annotational template language for HTML. Such a template language can be processed by a JavaScript template engine that is actually invoked from within the template itself, since the template can be loaded as an ordinary web page. This is important because popular HTML template engines today tend to use non-HTML document formats as input.

4.3 Templates in Sync Kit

Sync Kit uses a template engine that is declarative, endomorphic, annotational, and idempotent. It is declarative because this makes it easy to write a rendering engine that runs equally well on both client and server; it is endomorphic so that Sync Kit
can take advantage of the browser cache to store templates and use the templates as vehicle through which to execute our JavaScript bootloader; and it is both annota-
tional and idempotent so that the output of templates may be copied and pasted for re-use across the web.

The Sync Kit template engine is a relatively simple in features, and was con-
structed for the purpose of studying the performance of the Sync Kit hosting method. It is based on the idea that SQL queries can be embedded within the tag-space of HTML, associating a result set with that node in the DOM. This node is duplicated, once per result row, and for each row, text data is injected into the DOM based on HTML5 Microdata annotations.

Below is an example from the front page of a blog that has been implemented using Sync Kit.

```html
<div data-query='"SELECT id, title, contents
    FROM articles
    ORDER BY date DESC
    LIMIT 10;"
    data-as='article'>
    <div itemscope itemtype='article'>
      <h2 itemprop='title'></h2>
      <div itemprop='contents'></div>
    </div>
</div>
```

This blog has a client-side database cable called articles, from which the template is extracting the id, title and contents from the ten latest rows. The query is issued using the data-query attribute and the result set is named “article” by the data-as attribute on the same tag.

Inside the subtree of this query-defining div element, we see another div with an itemscope attribute. This is an HTML5 attribute that declares the existence of a semantic entity that spans the subtree of the current tag. Another HTML5 tag, itemprop, specifies the class of this semantic entity: article. Because the item type is the same as the name of a result set, the Sync Kit template engine will duplicate
this subtree once per result row.

Within this duplicated subtree, we see two itemprop attributes. This attribute is another HTML5 addition that binds the semantic item currently in scope (via the itemscope attribute) to some property-value. The property is listed as the itemprop attribute value (in this case, title and contents), and the value is the contents of the respective tags that the itemprop attributes occur in. The Sync Kit template engine will look in the result row for a column whose name matches the item property and will fill in the text content of the HTML tag using data from the result row.

Fully rendered, the above code will result in output that looks like the following code. Notice in the code below that all of the template directives are preserved. Although some of these directives have been duplicated, the duplication has occurred in a way that can be reliably detected by the template engine if this output is used as a template itself.

```html
<div id='main'
    data-query='SELECT *
        FROM articles
        ORDER BY date DESC
        LIMIT 10;'
    data-as='article '>
    <div itemscope itemtype=article />
    <h2 itemprop='title'>Title of article 1</h2>
    <div itemprop='contents'>Contents of article 1</div>
    </div>
    <div itemscope itemtype=article />
    <h2 itemprop='title'>Title of article 2</h2>
    <div itemprop='contents'>Contents of article 2</div>
    </div>
    ...
    <div itemscope itemtype=article />
    <h2 itemprop='title'>Title of article 10</h2>
    <div itemprop='contents'>Contents of article 10</div>
    </div>
</div>
```
The Sync Kit template engine allows for arbitrarily-many queries in an HTML template. If a query occurs within a block that is repeated, the query will occur once per repetition of the block. The following code, for example, executes the query for `daily_meals` once per day:

```html
<div data-query='SELECT name FROM days;'></div>

<div itemscope itemtype="day">
  <div data-query='SELECT name FROM daily_meals;'></div>
  <ul itemscope itemtype="meal">
    <li itemprop="meal.name"></li>
  </ul>
</div>
```

Supporting multiple query scopes at the same time introduces namespace ambiguity. In the above code, both the `day` result and the `meal` result have a field named `name`. Sync Kit resolves this ambiguity by allowing developers to prefix property names with the result name followed by a period, resulting in the names `day.name` and `meal.name` to refer to the two different `name` properties. If no prefix is given, Sync Kit resolves the ambiguity by attempting to select the most direct ancestor, in this case `meal`.

The rendering engine also supports referencing a result row from within a SQL query using double curly-brace wrappers. In the example of the blog main page above, the developer could query for the author name of the blog from within the context of the blog item result set using the following query:

```
SELECT name
FROM authors, articles
WHERE author.id = article.author_id
  AND article.id = {{id}}
```
This is an otherwise ordinary SQL query, except the \{\{id\}\} token, which the Sync Kit template engine will replace with the corresponding result value before passing to the database.

The Sync Kit template engine is relatively simple in operation, and lacks some features needed to be considered industrially viable, but it demonstrates a powerful approach to templating that offers many advantages over standard template practice today, namely endomorphic, annotational, idempotent HTML templates. As valid HTML5, templates can be crafted using professional design software and immediately used without any post-processing. They can also be parsed and cached, in template form, by web browsers and thus provide flexibility: either the server or the client can choose to render them. And their rendered output can be re-used as templates themselves, making template-reuse easier than with non-idempotent templates.
Chapter 5

Synchronization

The Sync Kit architecture’s performance gain over previous architectures is due to its reliance on persistent client-side caching of data. Rather than retransferring the data that populates a web template with each page request, the server only sends a changelist of updates that bring the client’s state up to date with the latest version. This chapter will develop methods of synchronizing remote views that meet two objectives:

- Minimize (to zero) the amount of per-view state that must be kept on the server
- Make the synchronization process easy for developers to incorporate into their code

This chapter will describe two different strategies for synchronization that meet the above criteria. The first, “Supervised Synchronization” is a way of synchronizing data based on common patterns of data usage and change in a web site. By explicitly specifying these patterns, the web server can efficiently send updates to the client caches. The second, “Unsupervised Synchronization” is a generalized way of synchronizing data that works for arbitrary SQL queries of a certain class common to web applications. These techniques are loosely inspired by work in the database community on semantic caching (e.g., [28, 15, 27, 13, 16]).

Both of these synchronization methods are designed to be useful for queries that remain fixed over time from the standpoint of a particular user. For example, the
webmail query for President Obama’s email inbox remains the same every time he re-visits his webmail inbox page, but the results of that query will change over time. For each such unique query to be synchronized to the client, Sync Kit requires a new synchronization process to be set up.

This means the most effective way to perform faceted browsing on the client is to synchronize the entire data set with Sync Kit’s synchronization methods, and then use the client-side database API to dynamically issue facet queries to that synchronized data set. This way, the query for the entire data set is fixed from the standpoint of Sync Kit, and the rapidly-changing, dynamic queries are confined to the client.

Personalized queries, like the inbox query above, are pain points for traditional query caching and view maintenance solutions because they are parameterized on the user’s ID. In a traditional system which performs server-side query caching and “active” view maintenance on the server-side, caching is ineffective because there is no overlap from user to user: with potentially millions users filling the cache with personalized queries, cache hits are a rarity. View maintenance in these systems is also largely wasted effort because the server must actively maintain a large number of views (at least one per user) even though it does not know when the users will return to need that data.

Sync Kit caches data on the client side, relieving the problem of cache misses for personalized queries. And by requiring that the synchronization methods store view maintenance-related state only on the client, Sync Kit is limited to a lazy style of view maintenance with attractive characteristics for this particular environment. As will be seen, it is not as powerful as traditional, active view maintenance because it only works for a certain class of queries and client-side views will occasionally have to be invalidated. But the tradeoff is that the server has no additional per-view burden and is guaranteed to only update views immediately before they are needed. Even the worst case is an improvement over traditional hosting methods: in the worst case, when client-side views must be invalidated, Sync Kit performs equivalently to the Templates that Fly design, which achieves a doubling of server throughput.
This chapter presents both Supervised and Unsupervised synchronization methods. Each method is targeted for a different class of queries and use cases, the assumptions of which be introduced at the beginning of the appropriate sections.

5.1 Supervised Synchronization

Supervised Synchronization is the first method of synchronization that Sync Kit provides, and it centers around the idea of “Synchronization Primitives” as programming constructs. Synchronization Primitives are analogous to queries, but while a query only specifies a relation upon a database, a Synchronization Primitive specifies both a relation and how to update that relation. They are essentially patterns that encapsulate a common scenario of data, and the way it changes, on the web. These patterns are simple, but we believe broadly applicable. As Chidlovskii and Borghoff [13] observe, web applications are characterized by simple queries and as such are amenable to semantic caching; the same simplicity that makes them amenable to semantic caching should indicate a broad range of scenarios where synchronization primitives can be applied.

Synchronization primitives are exposed to the programmer in a way similar to how Object-Relational Mappers (ORM) expose database data. Many ORM mappers provide “finder” methods that serve as an API to translates developer-provided parameters into a SQL query. For example, finding the ten most recent articles in a database might be accomplished using a traditional ORM with a call that looks like the following:

```ruby
articles = Article.find("limit" = 10,
    "order" = "publish_date",
    "direction" = "DESC")
```

In a traditional web hosting environment, the result of such method call would be rows from the database that would subsequently be provided to the template engine to produce HTML output.

Synchronization primitives aim to provide a similar interface while additionally
specifying how the results of the query can be updated in the future. Sync Kit provides two primitive types that can be used in this way:

- **Queues** represent feed-like data on a page that is added and removed according to some monotonic ordering on a field

- **Sets** represent collections of items that appear on the page but do not have queue-like semantics over time

These primitives become wrappers around the query that the developer wishes to perform. Since the front page on a blog behaves like a queue with respect to the articles shown, the above `Article.find` example can be expressed using the Queue primitive similarly:

```ruby
articles = QUEUE(table = "articles",
                 "limit" = 10,
                 "on" = "publish_date",
                 "direction" = "DESC")
```

Note that the synchronization primitive call looks essentially the same as what the traditional query would have been, except the table name, represented in the traditional example by the class `Article`, has become just another parameter to the primitive call. The result of this call can be treated as a proxy for the data that an ordinary query would have produced and can be provided to the Sync Kit library or some Sync Kit-compatible template library that operates on the server side.

Two key differences should be noted about the way in which this API functions. First, while the ORM method call specifies only a query to perform upon the database, the synchronization primitive method call also implies that a particular strategy can be used to update the results of that query at a later time. Second, the ORM method call causes an immediate, blocking query to be issued the database, while the synchronization primitive merely creates the specification for such a query. In the synchronization primitive case, the developer no longer controls when the work is performed; that is left up to the framework. In the existing implementation of Sync Kit, this work is always deferred to the client browser, but one could easily imagine
how a server-side optimization process could dynamically decide which machine was better suited to perform a particular query given a set of objectives to optimize.

5.1.1 Queues

The Queue synchronization primitive captures the idea that results are ordered on some attribute (say, time) and that this ordering reflects the way the client will access the data. Queues are ordered, finite lists of items that change in a regular manner. When new items are appended to the head of the queue, the oldest items are removed from the tail of the queue. Queues are prevalent on the web and are 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).

Assumptions

The queue structure assumes that items will behave like a queue with respect to the field on which the queue is built. That is, updates to the queue field will be monotonically increasing or decreasing. A prototypical example of this would be the publish_on date of a blog. Each successive setting of this field for a tuple in the scope of the queue should be greater than the maximum value of that before it. If this property is not held, the data is not actually behaving as a queue, and the tuples synced to the client can not be guaranteed to be correct.

Defining a Queue

To see how queues are programmed in our model, consider the blog article query from Section 5.1. Suppose the programmer knows that blog articles are always accessed in the same time order for display on a web page, and new articles are always added with a more recent publication time than existing articles. We can now declare a queue primitive as follows:
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 first time the client library loads this endpoint, a SQL query that fetches the full result set of ten blog articles will 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.

### 5.1.2 Synchronization

Because of the fixed way in which a queue is allowed to change, only a small amount of information is necessary to describe the state of the queue at the client: the maximum (or minimum) value currently in the client-side cache of the queue. Each time clients revisit a web page containing a queue structure, they send this value to the server and receive any new items that have been added to the queue since the last visit. Since clients are aware of the size of the queue, they may perform their own eviction for items that have “fallen off” the bottom.

On its end, the server will append an additional filter on the query for queue items based on the maximum (or minimum) value sent by the client. It will add a predicate to the query’s WHERE clause such that only tuples with a queue field greater than (or less than) the client’s watermark will be returned. If few entries have been added to the queue since the last client visit, the server will fetch, and transmit back, fewer
results.

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, performing the SQL queries embedded in the template and merging their results into the cached template. Finally, the client can discard entries outside the new top 10 from the browser’s database.

Cache Invalidation

To ensure correctness on the client side, the application developer is responsible for instructing Sync Kit to invalidate the client cache in three conditions:

An item is removed from the queue

A value for the field on which the queue is built changes non-monotonically

An item in the queue changes on some field other than the queued field

It is useful to describe these conditions in the concrete terms of the main page of a blog:

A post is deleted from the main page

A post from two years ago has its date changed, bumping it to the second item on the main page

The title is changed on the top post on the main page

Lucky, the above conditions tend to occur less often than the basic assumption of queue behavior for a blog, but when they do occur, the application developer must manually instruct sync kit that the queue has been dirtied so that clients know to pull down a fresh set of information.

The first condition is impossible for the queue to detect because if the tuple is no longer returned by the query, there is no way for the queue structure, given its query
rewrite on repeat visits, to detect that something has changed about that tuple. The second condition will also not be detected by the particular query re-write that the queue structure performs, necessitating cache invalidation.

The third case is actually solvable, but the current implementation of Sync Kit does not handle it. If every tuple is assumed to have a global version number (as the Unsupervised Synchronization will assume), then the query can be written to filter for any new items in the queue or any existing items in the queue window that have a version number more recent than the client's most recently known tuple version. Given the existence of tuple version numbers, this feature is relatively simple to implement, so handling this case should be a future feature of Sync Kit.

5.1.3 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.

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.
Assumptions

Sets in Sync Kit are assumed to be monotonically increasing; if an item is removed from a set, then the developer must manually instruct Sync Kit to invalidate client-side caches. Each tuple in a set must also contain a primary key that Sync Kit can use to uniquely identify it.

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 blog programmer from the Queue example would like to add a list of friends' blogs to the main page. This can easily be accomplished by treating this list of friends as a complete set and synchronizing it to the client for display.

```
friends = SET(type = 'complete'
    table = 'friends'
    key = 'friend.id'
    include = 'friend_id, name, link')
```

We define the set to be a complete replica of the table friends (though we could have added a filter or join; joins are discussed in Section 5.1.4). We will request friends by the key friend_id, and include the fields name and link from the table.

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 wiki pages could be synchronized in the following way:
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.

**Synchronization**

Synchronization is performed differently for complete sets and partial sets.

*Complete set synchronization*

Complete set synchronization ensures that the client has every item in the set after each synchronization. It does so by comparing the list of tuples that the client has (using primary keys) to the list of tuples that the set should return.

The first time a client visits a page containing a complete set, the server executes the database query that it derives from the set’s parameters and sends back the entire contents of the result. Subsequent requests made by the client will contain the list of primary keys that are already cached for the set in the client database. The server will then construct the set query with an additional WHERE clause predicate indicating that only items not on the client should be sent:

\[
\text{AND tuple.id NOT IN (tuple.id.1, tuple.id.2, ...)}
\]

Unlike the queue primitive, whose client state is a constant size (the value of the maximum or minimum item), the client state for the complete set scales in size with the size of the set. This means that synchronization will achieve better returns under two conditions in particular:

1. The tuples in the set are much larger than the primary key field. This means that the upstream state sent by the client is more likely to be small compared to the downstream data returned.
2. New items are regularly being added to the set. This increases the chance that the client is not sending data upstream for no return.

Despite the fact that the client state parameter grows with the size of the set, it is reasonable to expect that many complete sets used in a web application will not grow large in cardinality. Indeed, this is why two different types of set synchronization are provided.

**Partial set synchronization**

Partial set synchronization essentially acts as a lazy load of tuples that are subsequently cached on the client side. Some synchronization primitive defines a partial set on the server, for example one containing all wiki pages in WikiPedia, and then each particular wiki page queries for an item in that set. If the item is cached in the browser, no communication is necessary between the browser and the server. If the item is not present in the browser database, Sync Kit will lazily fetch it from the server. In this way, users slowly build up a cache of items that have been accessed in the past.

This trick results in an obvious performance enhancement for both the client (less latency) and server (less load), but it begs the question of lease invalidation on tuples. If the client bypasses the synchronization step to the server, how is it to learn of updates to data items that it already has? This problem is generally addressed with a concept called leasing, in which data is considered “current” for some fixed period of time, after which an update to the server is necessary. If the Sync Kit client adds a `last.synced` field to each client-side relation, for example, it would be able to check for set members that already exist on the client but exceed some maximum age. In this case, the client would contact the server for an update even though it is capable of displaying a (potentially stale) result to the user without blocking on the network. Sync Kit currently has no facilities for such leasing capability, but utilizing it should be a simple addition to the framework already in place.
Cache Invalidation

There are two conditions when the Sync Kit developer must instruct Sync Kit to invalidate client-side set caches:

- When an item is removed from the set.
- When some field on an item in the set is changed.

Both of these conditions are actually solvable without the need to invalidate, but these solutions are not currently implemented by Sync Kit. The first condition, item removal, can be determined because the client sends a list of primary keys representing the client’s set contents. By issuing a query for any tuples in the client’s list that are not in the server’s definition of the set, the server can determine a list of tuples that should be removed from the client view.

The second problem, item modification, can be solved using the same version tactic applied to the corresponding problem for queues. If a tuple version is sent along with each tuple primary key, then the server-side query can not only query for additions to the set, but also existing tuples that have newer versions on the server than they have on the client. This doubles the size of the state that the client needs to send, but eliminates the need to invalidate the cache on the client side.

5.1.4 Join Paths

The previous examples of sets and queues are missing a critical component of database queries in web applications: joins. Joins are handled as a special case for Sync Kit, using a features called Join Paths, because joins are often handled as a special case by web developers. This section will first explain why web developers tend to perform joins manually, using code, rather than from within SQL, and will then show how Join Paths mimic this practice for synchronization.

Consider again our running example of the front page of a blog. Each article is associated with several tags via some intermediate join table, shown in Figure 5-1.
The obvious query, from a database perspective, to extract articles their associated
tags would be to perform a join:

```sql
SELECT article.title, article.body, tag.name
FROM articles, articles_tags, tags
WHERE articles.id = articles_tags.article_id
AND articles_tags.tag_id = tags.id;
```

But performing a join takes the cross-product of the tuples that match the join
conditions, resulting in duplication of content. If a single article is associated with
ten different tags, for example, then ten joined tuples will be returned, each with a
re-duplication of the article paired with exactly one tag.

This is an inconvenient result format for a web developer, who would have to write
logic to manually "undo" this duplication before display to the user. In a system
like Sync Kit, sending duplicate information like this is also wasteful of valuable
bandwidth resources.

Instead, web developer prefer to think of joins differently. Rather than thinking
of the above task as a join, they think of it as two separate queries, one dependent
on (or parameterized by) the results of the other.

Q1 = SELECT id, title, body FROM articles;

Q2(X) = SELECT tag.name
FROM tags, articles_tags AS at
WHERE at.article_id = X
AND at.tag_id = tag_id;
With this decomposition of the join, the developer can work with the result sets in a way that matches the iteration loops that already exist in the code:

```python
articles = Query(Q1)
for article in articles:
    print(article)
tags = Query(Q2, article.id)
print(tags)
```

This approach makes a tradeoff: in exchange for more round-trips to the database, the developer gets a data result that is free from redundancy and that feels more natural with respect to the program flow.

Synchronization primitives provide support for joins in the style of these dependency decompositions commonly performed on the server side. Synchronization primitives use this method of decomposing joins for two reasons:

1. The redundancy that can occur in a cross product is even more costly when that cross product is shipped all the way to the browser.

2. Decomposing queries in this way permits a minimalist synchronization primitive interface that only allows one database table per primitive. This reduces the number of edge cases that have to be considered.

Just as Q2 above is dependent on the results of Q1, synchronization primitives can express their dependence on the results of other synchronization primitives using a parent parameter in their definition. The value of this parameter is a 2-tuple that specifies the synchronization primitive upon which there is a dependency and the join condition that links the child tuples to the parent. An equi-join with the parent primitive would be accomplished by setting the parent parameter to: `[parent.primitive, ‘parent.field = child.field’]`. These structures are referred to as join paths.

Just as the tag query above depended on the articles query – one tag query happened for every article, not the other way around – join paths encode a notion of directionality, too. If a synchronization primitive has a parent relationship, then only new or updated items that join with a to-be-synced item from the parent primitive
will be sent. This mirrors the loop structure of the procedural code example: it isn’t all new tags that should be synchronized, but rather all new tags among the set of tags associated with the queue of articles.

For example, the join shown above, articles \(\rightarrow\) articles\_tags \(\rightarrow\) tags, would be accomplished by defining the following join paths. First we recall the queue of articles from section Section 5.1.1.

\[
\text{articles} = \text{QUEUE( .. params ... )}
\]

Given that queue, the first step to create the join path is to create a set of all of the articles\_tags tuples that depend on the queue. We define a complete set and create a join path to the articles primitive using the parent parameter.

\[
\begin{align*}
\text{articles\_tags} &= \text{SET(type = 'complete')} \\
&\quad \text{table} = '\text{articles\_tags}' \\
&\quad \text{parent} = [\text{articles}, 'id = article\_id'] \\
&\quad \text{key} = 'id' \\
&\quad \text{include} = ['id, article\_id, tag\_id']
\end{align*}
\]

When Sync Kit synchronizes the articles\_tags primitive, it will only synchronize the tuples that join to article tuples that are contained in the window of the articles queue.

Next we create another set, this time to synchronize the tags table. Again, we’ll use the parent property to create a join path between the tags primitive and the articles\_tags primitive:

\[
\begin{align*}
\text{tags} &= \text{SET(type = 'complete')} \\
&\quad \text{table} = 'tags' \\
&\quad \text{parent} = [\text{articles\_tags}, 'tag\_id = id'] \\
&\quad \text{key} = 'id' \\
&\quad \text{include} = ['id, tag']
\end{align*}
\]

With the above definitions, Sync Kit will synchronize the following data into the browser database (and keep it up to date when the client returns):

1. The most recent ten articles
2. All tags associated with those articles
3. The join-table data necessary to encode the article-tag relationships in relational form

5.2 Unsupervised Synchronization

While synchronization primitives provide an easy way for developers to specify their web application’s data synchronization in terms of basic patterns, Sync Kit also includes the ability to synchronize the results of arbitrary queries without using primitives. This section explores how data synchronization can be performed for arbitrary “select, project, join” queries that are so common to web applications, providing that tuples are assumed to have version numbers. Such a capability has a number of benefits:

1. It requires less change, on the developer’s part, to the codebase of an existing application because it requires nothing more than the SQL queries already being used to fetch data.

2. It makes Sync Kit-style performance enhancements possible to achieve at the framework level, transparent to the developer, rather than at the application level, putting Sync Kit on par with systems like Ferdinand [21].

3. In removing the need of the developer to explicitly state synchronization primitives, an optimizer could instead decide when synchronization is appropriate. For any given web endpoint, such an optimizer could dynamically alternate between traditional hosting, Flying Templates-style hosting, or Sync Kit-style hosting based on usage load.

4. Finally, because unsupervised synchronization assumes a version number on each tuple, it is able to handle a broader range of scenarios than synchronization structures alone.

Consider the queue structure, for example. When an element is inserted into the middle of a queue, as occurs in Figure 5-2, the developer has to manually
invalidate the client cache. A fix for this was described that involves tuple versioning. The unsupervised synchronization method essentially implements this fix, but does so for general queries instead of for the queue or set specifically.

![Figure 5-2: Inserting an Article into the Middle of a Queue](image)

5.2.1 Assumptions

**Tuple versioning.** Unsupervised synchronization assumes that tuples are versioned with version numbers that increase monotonically across the entire database. We will use \( V(u, t) \) to represent the version of tuple \( u \) at time \( t \). This is a reasonable assumption in practice not just because many databases include tuple versioning as part of their internal state, but also because many web applications include a `last_modified` or some similar field on most tables anyway. For this section, we will assume that this `last_modified` field exists on each table and we will make use of it for tuple versioning.

**Join versioning.** When two tuples are joined together, the version of the joined tuple is the maximum version number between the two tuples participating in the join.

**Query result versioning.** The result set for a query \( Q \) at time \( t \) will be denoted as \( Q(t) \), and the version of \( Q(t) \) is defined as the maximum version number across its tuples, \( \arg \max_{u \in Q(t)} V(u, t) \). This value is denoted as \( V(Q, t) \).

**Query Predicates.** Only simple predicates, of the form \( A \ OP \ B \) may be used in the join conditions and filter of the query. \( A \) and \( B \) may either be primitive data
types or references to fields. $OP$ may be any standard comparison operator. Examples of predicates that are *not* allowed by this definition include:

- $A > (B-C)$
- $A$ in (SUBQUERY)

**External Dependencies.** No part of the query may depend on some source of data that is neither a tuple or a primitive value. For example, LIMIT RAND() or ORDER BY RAND() are not permitted because the random sequence returned by the RAND() function is not specified in the query.

### 5.2.2 Queries Decomposition

Unsupervised synchronization depends on a process of query decomposition, which takes an input query $Q$ and produces a set of component queries, $q_1 \ldots q_n$. One component query exists for each database table $t_i$ that is listed in the FROM clause of the query.

For example, the following query would be decomposed into two component queries, one for $A$ and one for $B$:

```sql
SELECT A.foo, B.bar FROM A,B WHERE A.baz = B.zap;
```

The goal of this decomposition is to produce a set of queries which, table per table, provide the source data that is necessary to answer the combined query. In the above example, the two component queries $q_1$ and $q_2$ should produce the table-level results, for the $A$ table and $B$ table, that are required to answer $Q$.

Query decomposition allows Sync Kit to take a large query and split it up into several simple queries, one per component table. Because we have assumed that tuples are versioned, each component query can now be thought of as a queue, queued on the tuple version. The synchronization conditions between component tables are then linked so that if a tuple in one of them changes to a new version, then every tuple that combines with it from other component queries must also be re-transmitted to the client, even if those other tuples have "old" version numbers.
Decomposition is performed as follows:

**Input** $Q$, a query

**Output** $Q'$, a vector of queries, one per component table in $Q$

**Implementation** Let $t_1 \ldots t_n$ be the component tables of $Q$ specified in the FROM clause. Let $Q'_i$ be $Q$ with one change:

- The SELECTed fields are replaced by all those fields that appear in $Q$ and which come from $t_i$

The result of this simple algorithm is a set of queries that, individually, are far more easier to reason about. The next section will discuss the circumstances under which synchronization to the client is needed, and then the following section will discuss “flood synchronization”, which is the process Sync Kit uses to determine which tuples should be sent to the client.

### 5.2.3 Synchronization

The synchronization task is to figure out the difference between $Q(t)$ and $Q(t+n)$. To make comparisons easy, version numbers will take the form of timestamps (the time at which a tuple was created or updated). Thus $V(Q, t) \leq t$ by definition.

There are three possible reasons a change could occur to the results of $Q$ between time $t$ and $t + n$. We will examine each and determine, if possible, which tuples need to be sent to the client for an update. For the below tree, the notation $u \in Q(t)$ means that tuple $u$ is a tuple in some component table of $Q(t)$ according to the decomposition process above.

1. A tuple $u$ was added to the database
   - If $u \in Q(t+n)$, it will appear in $Q(t+n)$ with $V(u, t + n) > V(Q, t)$ and must be flood-synced to the client-side database.
   - If $u \notin Q(t+n)$, the change is not relevant.
2. A tuple $u$ was modified in the database

- If $u \in Q(t + n)$, it will appear in $Q(t + n)$ with $V(u, t + n) > V(Q, t)$ and must be flood-synced to the client-side database.
- If $u \notin Q(t + n)$ and $u \in Q(t)$, the tuple must be removed from the client-side database.
- If $u \notin Q(t + n)$ and $u \notin Q(t)$, the change is not relevant.

3. A tuple was removed from the database.

- If $u \in Q(t)$, the tuple must be removed from the client-side database.
- If $u \notin Q(t)$, the change is not relevant.

Three possible scenarios occur in the above list. Either a change is not relevant, it requires a removal from the client-side database, or it requires a “flood sync” to the client-side database.

**Irrelevant changes** need not be handled by the algorithm, as they do not have an effect on the query results needed at the client side.

**Need for tuple removal** above occurs when some tuple $u$ exists such that $u \in Q(t)$, but $u \notin Q(t + n)$. Detecting this situation is possible if the state be sent from the client as a list of primary keys, as with the set synchronization primitive. Only then can the server construct a query for items that are on the client-side but are no longer on the server side. Sync Kit currently only uses queue-style state keeping for unsupervised sets, meaning that the current implementation does not have a facility to do this. As such, the developer must currently implement application logic that instructs Sync Kit to invalidate the client-side cache when a tuple is removed that plays a role in an unsupervised synchronization process.

**Need for flood-sync** above occurs only in cases where some tuple $u$ is guaranteed to have a version number $V(u, t + n)$ greater than the version known by the client $V(Q, t)$. This can always be detected by simply placing a filter on the
component query for the table from which \( u \) comes: \( \text{WHERE} \ldots (\text{version} > \text{client.version}) \). Sync Kit will then perform a “Flood sync”, as described in the next section, to ensure that this tuple is not only delivered to the client, but also any component tuple it co-occurs with in the creation of the query result is delivered to the client. This ensures that even tuples with old version numbers, undetected by this process, that happen to join with a new tuple will be transmitted to the client.

The next section will talk about query decomposition, which is how Sync Kit determines which tuples participate in a query (i.e., whether or not some \( u \in Q(t) \)) by splitting a single query into multiple queries, one for each table that plays a role, and projecting out any fields that occur in the query. It will then address flood sync, which is a manner of sending tuples to the client based on tuple additions and updates to guarantee that the client will receive all possible tuples that join with it in the query.

5.2.4 Flood Sync

When a tuple in some component table \( t_i \) changes, we can detect that change, as noted above, because the version number will become greater than the known version number at the client side. But we also will need to send tuples from the other component tables, and this must happen conservatively, possibly redundantly sending data, because this new tuple in one component table might join with tuples that appear old from the perspective of other component tables. A later example will demonstrate this scenario; here we will see how to handle it.

“Flood sync” simply refers to the way in which we choose to synchronize tuples from each component table. Recall that we can think of these component tables as queues since the tuples are versioned. But we will not use the simple filter used to sync queues in the synchronization primitive case, e.g.: \( \text{WHERE} \ldots \text{AND} (\text{version} > \text{client.version}) \). Instead, we will use a more complex filter that ORs together the synchronization conditions for all of the component tables. This will ensure that,
for any given component table, a tuple will be transmitted to the client either if it is
newer than the client’s version for that component query or it joins with some other
tuple that needs to be synced to the client.

This where filter is constructed in the following way:

**Input** $Q'$, a vector of component queries for tables $t_1 \ldots t_n$ of query $Q$

**Output** $Q'(V)$ a vector of component queries, $q_1(V) \ldots q_n(V)$, parameterized on $V$,
a vector of version numbers for each component table in $Q$

**Implementation** Append an additional filter that restricts the results of each $q_i$.
This filter takes the form:

- $OR_{i=1\ldots n}(t_i.last\_modified > V_i)$

Given this, the synchronization is not across the results of query $Q$, but rather
across the tuples that participate in creating its output. To illustrate an example,
recall the articles - articles.tags - tags example from Section 5.1.4 on Join Paths.
We will step through a series of updates over time that show how the following query
$Q3$ would be decomposed and updated. $Q3$ selects all articles associated with Tag
number 1.

$$Q3 = \text{SELECT } \text{articles}.id, \text{tags}.id$$
$$\text{FROM } \text{articles}, \text{articles}.\_tags, \text{tags}$$
$$\text{WHERE } \text{articles}.id = \text{articles}.\_tags.\_id \text{\_article\_id}$$
$$\text{AND } \text{articles}.\_tags.tag\_id = \text{tags}.id$$
$$\text{AND } \text{tags}.id = 1$$

This query is first decomposed by Sync Kit into one query for each component
part. For brevity’s sake, we’ll abbreviate the where clause that performs the 3-way
equijoin above as $\text{WHERE\_CLAUSE}$. Notice below that $Q3,1$, $Q3,2$, and $Q3,3$, share the
same structure except the projected fields differ.

$$Q3,1 = \text{SELECT } \text{articles}.id$$
$$\text{FROM } \text{articles}, \text{articles}.\_tags, \text{tags}$$
$$\text{WHERE } \text{WHERE\_CLAUSE}$$
$$\text{AND VERSION\_FILTER}(v1, v2, v3)$$
Q3.2 = SELECT articles_tags.article_id, articles_tags.tag_id
    FROM articles, articles\_tags, tags
WHERE WHERE\_CLAUSE
    AND VERSION\_FILTER(v1, v2, v3)

Q3.3 = SELECT tags.id
    FROM articles, articles_tags, tags
WHERE WHERE\_CLAUSE
    AND VERSION\_FILTER(v1, v2, v3)

The VERSION\_FILTER clause limits the results of each component query based on the version numbers (v1, v2, v3) that the client has for each component table participating in the join. For this particular query, this filter is defined as:

$$\text{VERSION}\_\text{FILTER}(v1, v2, v3) := (\left(\text{articles\_last\_modified} > v1\right) \lor \left(\text{articles\_tags\_last\_modified} > v2\right) \lor \left(\text{tags\_last\_modified} > v3\right)$$

With this decomposition, Sync Kit now has a set of queries $Q_{3,1} \ldots Q_{3,3}$, parameterized on three version numbers from the client, that it can use to compute updates for repeat visits. Figures 5-3, 5-4, and 5-5 show a walkthrough of these queries on the example blog schema.

The first time the client visits a page that includes $Q_3$, it has no cached version numbers to send, so the server assumes a version of zero and consequently sends it all tuples involved in the join between articles and Tag 1, shown as the grey rows Figure 5-3. Boldface rows represent data that is cached on the client after this synchronization. In addition to the tuples received, the client is also given $V(Q_{3,i}, t)$ for each $i$ at the time it requested.

When the client returns at time $t + n$, shown in Figure 5-4, it sends its vector of version numbers, [4, 2, 3]. The server runs its three queries and returns the three rows shaded grey. The rows from the `articles` and `articles\_tags` tables are new to the client, but the row from the `tags` table is duplicate information. In this particular
case, the server could have determined that the client already had this row because it’s tuple version is equal to the client’s version number, but in general it is impossible to always detect whether joined tuples such as this already exist on the client, so Sync Kit conservatively elects to send it. The next step of this example will include such a situation.

When the client returns again at time $t + n + k$, shown in Figure 5-5, it once again sends its vector of version numbers, which is now $[5, 6, 3]$. This time, a new articles_tags row has associated Article 1 with Tag 1, causing this articles_tags tuple to have a version number more recent than the client’s version number for that component query. Because of this, both Tag 1 and Article 1 send to the client as well. Notice that both Tag 1 and Article 1 have version numbers older than the client’s version for their respective component queries. Once again, Tag 1 is a redundant
transfer, but Article 1 is not, illustrating why Sync Kit needs to conservatively send all participants in the join in case one of them is absent on the client despite having an "old" version number, like Article 1.

![Table: Articles, Articles_Tags, Tags]

Figure 5-5: Unsupervised Synchronization - Third client visit
Chapter 6

Experimental Performance

In this section we compare the performance Sync Kit to our implementation of the Flying Templates [30] 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 a blog homepage repeatedly across several days, and a Wiki, in which users browse a connected graph of web pages, potentially revisiting some pages over time. We built these websites using template and content characteristics, update schedules, and hyperlink models characteristic of real websites that we measured.

Testing the performance of a web framework is challenging for a number of reasons, one of which is choice of benchmark. In this respect, the tester is essentially faced with three choices: take an existing web site with organic, real-world, traffic and augment that site to be a live test; use an established benchmark, which comes with a site model and a user load to use for testing; or create a custom test built specifically for the being tested.

For this work we have chosen to take the third option, a custom benchmark, so that we could carefully test the synchronization primitives in some of prototypical situations for which they were designed. Sync Kit relies heavily on caching that takes place in the browser over an extended sequence of re-visits, all while data on the server is also changing over time. This style of caching deviates enough from standard model
of web application hosting that existing web benchmarks would not have created the environment needed to appropriately test it.

Testing for Sync Kit used an end-to-end test harness that directed a Firefox browser to perform simulations of several users accessing the server over time. The client browser recorded timing, caching, and request characteristics in a database during this test, and this data was used to automatically generate a series of input files to another harness used to push the server to its limits using the same access patterns and cache-update requests recorded by the clients. Creation of this test harness involved many parameters and setup which are described in detail in Sections 6.1.1 and 6.2.

Following a description of the test setup from a modeling and programming perspective, Section 6.3 will describe the experimental environment used to perform our tests and describe the three serving strategies that we implemented for the blog and wiki approaches. Then the details and results of each test is provided with some analysis.

At a high level, the lessons to be learned from these results can be summarized as follows:

- When users visit a site more often than the data changes, Sync Kit can significantly improve server-side throughput

- When Sync Kit is unable to benefit from the client side cache, its performance improvement over traditional hosting is equivalent to the Templates that Fly approach (roughly 100% throughput improvement at the server)

- Despite promising performance benefits on the server-side, Sync Kit currently provides web clients no performance improvement, though it is reasonable to expect this result could be improved with more work

More detail about each, along with other conclusions to be drawn is provided alongside the test results for each of the two benchmarks.
6.1 Test Model

End-to-end performance testing of a dynamic web application can be a difficult process because so many systems and interactions are involved. In the case of Sync Kit, this process is even more complicated because Sync Kit's server-side behavior is dependent on state kept at the client side. This section will describe the Sync Kit test harness in terms of an abstract web application, and then the following sections will discuss the particular test workloads we tested using this harness.

6.1.1 Site Model

For testing purposes, we first construct a model of a website $S = P, Z, L, D, W$ which will be used to conduct the test. In this model:

1. $P$ is a set of $j$ pages (unique URLs) that the web application offers.
2. $Z$ is a set of static templates, one for each page $p$ in $P$.
3. $L$ is a set of $k$ 2-tuples $(l_{11}, l_{12}), (l_{21}, l_{22})..(l_{k1}, l_{k2})$, each of which corresponds to a link from page to page. Each value in this 2-tuple is an index of $P$, ranging from $1...k$.
4. $D$ is a set of data operations $d_1, \ldots d_j$, each corresponding to the data operations required by page $P$.
5. $W$ is a set of weights $w_1 \ldots w_j$, each weight $w_i$ corresponding to the relative popularity of page $p_i$.

The purpose of this model is to permit the Sync Kit test harness to simulate a randomized workload of users visiting the site. $P, L,$ and $W$ are all required as explicit inputs (from a data file or database, for example) to the test harness to generate such a simulation.

The inputs $Z$ and $D$ are provided to the test harness as code. Each page load $p_i$ corresponds to some function $d_i$ being executed, the results of which are combined
with template $z_i$ to make a rendered web page. Web hosting strategies like the traditional method, Templates that Fly, and Sync Kit differ on how and where $d_i$ and $z_i$ are handled. Because of the way in which Sync Kit operates, it is necessary to parameterize $d_i$ on both the time $t$ that the user visits the page and the state $s$ of the user at that time. So each $d_i$ may really be thought of a function, $d_i(t, s)$ that represents the data operations for page $p_i$ at time $t$ for a user with client state $s$. This is easy to state in an abstract model, but also relatively easy to implement in the code that physically implements $D$. When a client makes an HTTP request, it is assumed that the parameters $t$ and $s$ are provided as GET or POST parameters on that request, and on the server side the test harness simply extracts these parameters and then applies them to $d_i$.

6.1.2 Usage Model

The next component required by our test harness is a usage generator, $G$. $G(S, \theta)$ is a function that takes a website definition, $S$, and a set of parameters, $\theta$, and produces a set $U$ of user click trails.

Each element $u_i$ in $U$ represents a user to the website, so $|U|$ is the total number of simulated users that have been generated for the website. The element $u_i$ itself is an ordered set, representing the unique visits of user $i$ to the website. This set takes the form $u_i = (c_{i,1}, t_{i,1}), (c_{i,2}, t_{i,2}), (c_{i,n}, t_{i,n})$. Each $(c_{i,j}, t_{i,j})$ pair is a visit to the site at time $t_{i,j}$ consisting of click trail $c_{i,j}$. $c_{i,j}$ is a set of $k$ pages, each page representing an index into $P$, the set of pages provided by the website.

To put this in narrative terms, the usage generator $G$ will generate a set $U$ of users that use the site. User $i$ visits the site $|u_i|$ times. The $j^{th}$ such visit occurs at $t_{ij}$, and on this visit the user visits the list of pages stored in $c_{i,j}$.

What is important here is not how the usage generator generates this data but rather why this rather complicated scheme is necessary. Because the optimizations Sync Kit provides to web applications dependent on both a user's state and time (as with the $d_i(t, s)$ functions above), in order to demonstrate Sync Kit's effectiveness it is necessary to
create a test set that tracks multiple users visiting the site over the course of time in such a way that the test harness actually simulates the browser conditions of each user.

6.2 Test Harness

For any given web site test, defined by site $S$ and usage $U$, the Sync Kit test harness performs both client-side and server-side benchmarks for the three serving strategies Traditional Hosting, Templates that Fly, and Sync Kit. It does so in two rounds: first a client-focused round that measures performance from the client perspective and then from the server-focused round that pushes the web server to its limit to find the throughput and data requirements of the three hosting strategies.

6.2.1 Client Test Harness

The client round is performed using a test harness implemented in HTML and JavaScript. This piece of client side code allows the user to specify a site usage model, $U$, and it then proceeds to iterate through each user pattern $u_i$ specified by $U$. The user’s visits to the web site are simulated by controlling an IFRAME embedded within the test harness so that it may measure how the browser loads and renders each page.

Recall from Section 6.1.2 that each user visit $u_i$ in $U$ is paired with an access time, and the visit itself consists of a click trail through the pages on the website. The client-side test harness keeps a simulated clock which it resets for each user visit to the site. If the click trail associated with that visit is longer than one page load, the clock is incremented by a fixed amount (we used 30 seconds) for each page in the click trail. This clock value is passed as a GET parameter on each page request, and it is used by the server as a parameter of the data loading operations.

The test harness takes measurements about each page request that loads. In between these page loads it does “housekeeping” tasks and records data about the test. At the beginning of each user description $u_i$ that the test harness encounters, the
browser uses JavaScript’s access to cache and database control to clear any information stored on the client. This ensures that each separate “user” to the site incurs the effort required to load a page for the first time, after which it may have access to cached materials. In the Traditional and Templates that Fly case, these cached materials are static resources, but in the Sync Kit case, they also include relational data.

At the end of each page load, the test harness makes an HTTP request to a server-side logging application that persists the measurements associated with that page load. Each log entry includes a number of fields to characterize the nature of the request:

- A label given to this particular test by the tester
- The date the test was performed
- The application serving strategy being tested (e.g.: Traditional, FlyingTemplates, or SyncKit)
- The user number, visit number, click number, and URL associated with the request
- The simulated time from the perspective of the user

Alongside these fields are performance-related data:

- The time elapsed between the page request and the DOM Ready event
- The time required to make a data-only request to the server (applicable to the Templates that Fly and Sync Kit cases)
- The time required to perform client-side data bulk loading (applicable to Sync Kit)
- The time required to perform client-side data query (applicable to Sync Kit)
- The time required to perform client-side template rendering (applicable to Sync Kit)
6.2.2 Server Test Harness

The server-side round of tests is performed by taking logs from the client-side test harness output and transforming them into scripts for the \texttt{httpperf} tool, which can be used to simulate client load on a server. A script then performed a binary search over the parameter space of options to \texttt{httpperf} to identify the point at which the server stopped responding to all requests with a HTTP 200/OK response code.

6.3 Experiments

The server-side programming environment is the Python-based Django \cite{django} 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.

In assessing Sync Kit, we compare three systems:

\textbf{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.

\textbf{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
Table 6.1: 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.

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 [30], 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 [3] in which all data is stored and which is synchronized with the server using the managed data structures described in Section 5.1. 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.

We implemented our blog and wiki websites for the three systems listed above. 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.
6.3.1 Blog Workload

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 6.1. 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 \( \lambda = .53 \text{posts/hour} \).

We re-use the same template of size 100KB for all three serving strategies. This template consists of basic HTML, CSS, and standard 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 \( \lambda_i \) relative to the mean time between posts for the blog. We vary \( \lambda_i \) 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 [17], 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.

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 `max-clienttime` 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"]
```

The charts in Figures 6-1 and 6-2 display measurements of the maximum throughput and average data transferred per request as \( \lambda \) increases, causing the user visit-per-post frequency to increase.

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 [30] 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 6-3; 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
Figure 6-2: User visit frequency vs KB per request.

same client latency (around 400 ms/request). Note that Sync Kit improves server-side performance without hurting the client's experience.

Figure 6-3: 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 3ms 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.

6.3.2 Wiki Workload

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 be synchronized in their entirety, 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 supplemented these numbers with a random sample of the pages accessed on Wikipedia from publicly available web proxy logs [9]. We then generated 10,000 articles of random content length \( \mu = 3276B, \sigma = 100B \) [8] and title length \( \mu = 22B, \sigma = 12B \) [9] with an average of 23 [7] 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 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 [11]. 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%.

We ran the same experiments from the blog benchmark with our set-based wiki benchmark. Figure 6-4 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 6-5 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 6-3: 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 negligible. Again, the total time is dominated by DOM Load.
Figure 6-4: Server throughput (top) and data transfer per request (bottom) for the wiki benchmark.

Figure 6-5: Client latency for wiki workload.
Chapter 7

Conclusion

This thesis presented a web framework called Sync Kit that explores the possibility of separating a web application's data from its document structure so that web clients are able to process and cache each separately. Sync Kit demonstrates that such a web architecture can be achieved as an enhancement to, rather than a replacement for, existing web practices. It demonstrates this by working as a plugin on top of the popular Django framework, and also by utilizing upcoming HTML5 technologies on the client side, such as browser databases, offline caching, and Microdata-based web templates. Sync Kit also introduces the concept of synchronization primitives, which are simple metaphors for common synchronization patterns on the web that developers may use to keep data in sync on the client.

Our experiments show that when cache hit rates are high (as with our blog benchmark), performance of the Sync Kit approach is very good — approximately a factor of four better than the traditional approach and a factor of three better than the Templates that Fly approach. We also showed that client-side rendering does not negatively impact client-side performance, despite the extra work required by Sync Kit in the client-side bootstrapper and the overheads of client-side database access. In short, Sync Kit offers significant server-side performance benefits for data intensive web sites at little cost to the client.

Sync Kit represents an architectural argument about what the post-HTML 5 web should look like. Just as CSS split apart a web page's styling from its structure, this
work suggests that HTML 5 browsers will be sufficiently capable to allow splitting web content (data) apart from structure as well.

Looking forward, there are a number of ways this work can be extended. Much work still needs to be done in improving the performance at the edge cases for synchronization and expanding the library of synchronization primitives to include more patterns – aggregate structures and compressed data cubes [25] are two possibilities. For unsupervised synchronization, a more principled theoretical proof of the exact subset of SQL that can be supported for synchronization without over-taxing the server or becoming incorrect on the client needs to be completed before that synchronization method can be reliably used. Global synchronization optimization across all primitives on a site is yet another target.

Outside of the “sync” component of Sync Kit, much future work is needed to further the general goal of a web in which the division of data and template is preserved all the way to the client. Standardization of a “Data Style Sheet” format, possibly an extension to HTML, could enable a universal templating language with many beneficial side-effects to the web community. A standard way to transfer structured data – whether relational or graphical – over JSON is also sorely needed, as it appears today every developer uses a custom solution for this. Finally, integration of all or portions of Sync Kit into a production environment will help validate this model of hosting outside of the laboratory and generate a better picture of its performance benefits under a diverse workload.
Bibliography


