View-Driven Optimization of Database-Backed Web Applications

Cong Yan
University of Washington
congyan@cs.washington.edu

Alvin Cheung
UC Berkeley
akcheung@cs.berkeley.edu

Junwen Yang Shan Lu
University of Chicago
{junwen, shanlu}@uchicago.edu

ABSTRACT

This paper describes HYPERLOOP, a system for optimizing database-backed web applications (DBWAs). Current approaches in optimizing DBWAs focus on partitioning the application among the browser, application server, and the database, and rely on each component to optimize their portion individually without developer intervention. We argue that this approach misses the goal of DBWAs in optimizing for end-user experience, and fails to leverage domain-specific knowledge that DBWA developers have. For instance, a news website might prioritize loading of the news headlines, even at the expense of slowing down loading of other visual elements on the page. HYPERLOOP illustrates the idea of *view-driven optimization* by allowing developers to specify *priorities* for each of the elements on the webpage, and uses such information to drive optimization of the entire webpage. HYPERLOOP currently focus on optimizing for render time of webpage components, and our preliminary results show that this view-driven approach can substantially improve DBWA performance by leveraging developer provided application knowledge.

1. INTRODUCTION

From banking to social networking, we interact with database-backed web applications (DBWAs) on a daily basis. Unlike transactional or analytical applications, DBWAs are structured in a three-tier manner: a presentation tier that is executed by the web browser called the *view*, an application tier that resides on the application server, along with a storage tier consisting of queries and persistent data managed by the database. Such application executes when an end user visits a website. The web server, upon receiving the request, runs the corresponding hosted application that interacts with the storage tier to manipulate persistent data. The query results are returned to the hosted application on the web server. The view tier then assembles the results and renders them into a webpage to be displayed on the browser.

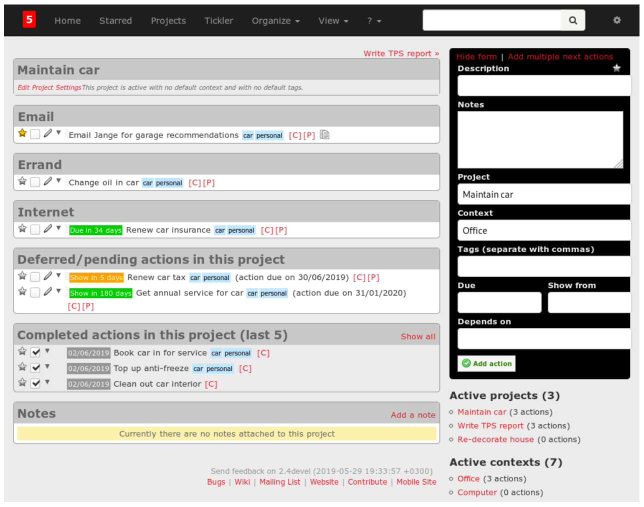
In principle, this three-tier architecture eases web application development: the DBWA components are partitioned and can be optimized by their respective hosts (i.e., the browser, web server, and the database). In practice, however, optimizing such applications is extremely difficult. Unlike transactional applications that focus on optimizing for throughput, DBWAs instead focus on end user experience (e.g., interactive websites), which often translates to the time taken to render the resulting webpage. Recent studies have shown that every 0.5s of latency in website rendering reduces website traffic by 20% [18], and that users will abandon a site if it takes longer than 3s to load [9]. In such web applications, load time is not only caused by the view tier, but also dependent on the amount of time taken for the application and storage tiers to execute application logic and queries. The three-tier architecture makes it difficult for developers to optimize their DBWAs: while end users only interact with the view tier, developers need to reason about how the view is generated through a complex myriad of code that spans the software stack.

To make things worse, webpages typically consist of multiple view components (e.g., tables, buttons, text blocks, etc.), with each component rendered using different code paths. In view design literature [5, 1], it is well-known that not all view components are created equal: a user might perceive a news website to have loaded already once the news headlines have appeared on the screen, even though the rest of the page has not been fully loaded yet. This has led to the development of asynchronous loading libraries allowing developers to modify their DBWA code to load page elements at different times, even at the expense of slowing down the rest of the page. Such tradeoffs are prevalent in DBWA design: dividing a long list of items into multiple, shorter lists and rendering each of them across multiple pages (pagination), pre-loading data that is likely to be used in subsequent pages that the user will visit (caching), etc. Today developers make such tradeoffs manually by trying to change the code across layers and see how that impacts each page element, and repeating the process until the best design is reached. However, we are unaware of any system that would systematically capture such "view-specific" knowledge from developers, and exploit them for optimization of DBWAs.

We argue that this page element-wise "trial-and-error" optimization of DBWAs is wrong. Instead, we believe the optimization of DBWAs should be *view-driven* by the domain-specific knowledge that developers possess. In this paper, we describe HYPERLOOP, a new system we are designing with that purpose. HYPERLOOP concretizes view-driven optimization by allowing DBWA developers to provide domain-specific knowledge as *priority labels* for each webpage element to indicate those that should be rendered first.[1] Given priorities and a resource budget (HYPERLOOP currently supports specifying total memory available to store data in memory), we envision HYPERLOOP to automatically analyze the

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020, Amsterdam, Netherlands.

[1]Other notions of domain-specific knowledge are certainly possible, e.g., impact on user experience, interactivity, etc. We currently use the time to render as it is an easily quantifiable measure.


5 Home Starred Projects Tickler Organize View ?

Write TPS report »

Maintain car

Edit Project Settings This project is active with no default context and with no default tags.

Email

Email Jange for garage recommendations car personal [C] [P]

Errand

Change oil in car car personal [C] [P]

Internet

Due in 34 days Renew car insurance car personal [C] [P]

Deferred/pending actions in this project

Show in 5 days Renew car tax car personal (action due on 30/06/2019) [C] [P]

Show in 180 days Get annual service for car car personal (action due on 31/01/2020) [C] [P]

Completed actions in this project (last 5) Show all

02/06/2019 Book car in for service car personal [C]

02/06/2019 Top up anti-freeze car personal [C]

02/06/2019 Clean out car interior [C]

Notes Add a note

Currently there are no notes attached to this project

Send feedback on 2.4devel (2019-05-29 19:33:57 +0300)

Bugs | Wiki | Mailing List | Website | Contribute | Mobile Site

Hide form | Add multiple next actions

Description:

Notes:

Project: Maintain car

Context: Office

Tags (separate with commas):

Due:

Show from:

Depends on:

Add action

Active projects (3)

- Maintain car (3 actions)
- Write TPS report (3 actions)
- Re-decorate house (0 actions)

Active contexts (7)

- Office (3 actions)
- Computer (0 actions)

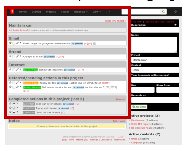
Figure 1: An example webpage from Tracks.

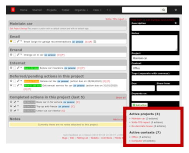
```
1: class User:
2:    has_many: projects => Project
3: class Project:
4:    has_many: todos => Todo
5: class Todo:
6:    has_one: note => Note
```

```
7:  @user = User.where(:id=?)
8:  @projects = @user.projects
9:  @left_projects = @projects.select
    {|p| p.todos.where(done=>false).count>0}
10: @right_projects = @projects.select
    {|p| p.status='active'}
```

```
11: <% for p in @left_projects %>
12:     <a href=..%=p.id%..><%=p.name%></a>
13:     <% for t in p.todos %>:
14:         <li> t.name, t.context </li>
15:         <li> t.note </li>
16:         <span> t.tags.join{} </span>
```

```
16: <% for p in @right_projects>
17:     <li> <% p.name> </li>
18:     <span><% p.todos.count></span>
```





```
 L7   Q1: SELECT * FROM users WHERE id = ?
 L8   Q2: SELECT * FROM projects WHERE user_id = ?
 L9   Q3: SELECT COUNT(*) FROM todos WHERE done=false AND project_id = ?
  …   …
L13   Q4: SELECT * FROM todos WHERE project_id = ?
  …   ….
L15   Q5: SELECT * FROM notes WHERE todos.note_id = note.id.
  …   …
L18   Q6: SELECT COUNT(*) FROM todos WHERE project_id = ?
  …   …
```

Figure 2: Abridged code to render the webpage shown in Figure 1, with blue numbers indicating which line of Ruby code at the top generated the query.

DBWA code to devise a plan to render each of the pages in the application, with the goal to reduce the render time of the high priority elements as much as possible. HYPERLOOP achieves this by applying different optimization across all three tiers, from changing the layout of each page to customizing data structures to store persistent data in memory. To help developers assign priorities, HYPERLOOP comes with a static analyzer that estimates render times and presents the results via HYPERLOOP's user interface.

While we are still in the early implementation phase of HYPERLOOP, our initial experiments have shown promising results: we can improve the start render time (i.e., time taken for the first element to be displayed on the browser's screen after initiating the HTTP request) of high priority webpage elements in real-world DBWAs by 27×. We believe this illustrates the potential of view-driven optimization of DBWAs, with HYPERLOOP presenting an initial prototype that implements this concept.

2. HYPERLOOP OVERVIEW

We now discuss how DBWA developers can use HYPERLOOP to improve their applications. Figure 2 shows a code fragment from Tracks [3], a popular Ruby on Rails DBWA for task management. Figure 1 shows a page from the Tracks listing of projects created by a user, where each project contains a list of todo actions. This page has three panels. The left panel shows a list of undone projects (we call a project "undone" if it contains undone todos), with the detail of each todo shown when clicked. The upper right panel shows a form where user can add a new todo, and the bottom right panel shows a list of active projects and its todo count.

Figure 1 shows the abridged DBWA code used to render this page. Lines 1-6 show how persistent data is organized into the `User`, `Project` and `Todo` classes. It also specifies the relationship between the classes, for instance, a project has many todos, as implemented as foreign key constraint in the database. Line 8 retrieves the list of projects that belongs to the current user from the database and into the Ruby array variable `@projects`. Line 9 then filters `@projects` to return those to be rendered on the left panels based on the number of undone todos. The filter for the right panel selects the active projects in Line 10. The code uses the `where` API provided by the Rails library which translates the object query to SQL queries as shown in the bottom of Figure 2.

Lines 11-18 show the view file written in HTML with embedded Ruby code. The bottom of Figure 2 shows the SQL queries translated by the Rails library to generate this page. Q1 retrieves the current user, followed by Q2 to retrieve her projects. A number of queries (e.g., Q3) are issued to get the count of undone todo for each project. Similarly, some queries are issued to get the todos for each project (Q4) and note for each todo (Q5) which are on the left panel, as well as todo count for each project (Q6) on the right.

HYPERLOOP allows developers to improve performance via its view-centric interface. Figure 3 shows the HYPERLOOP workflow. To use HYPERLOOP, the developer only needs to label the high priority elements on the webpage, and HYPERLOOP will automatically analyze the application code to suggest different ways to render the page by reducing the render time of high priority elements, while possibly increasing the render time of the low priority ones. We envision that HYPERLOOP will make different tradeoffs based on how the elements are labeled, and propose different render plans to the developer to further refine.

To help developers assign priorities, HYPERLOOP comes with an analyzer that statically estimates the load time of each page element, given the amount of data currently stored in the database. The estimates are presented to the developer as a heatmap, as shown in Figure 4. We envision other analyses will also be useful in aiding the developer to assign priorities, for instance the amount of memory used, query plans used to retrieve rendered data, etc.

For the example shown in Figure 1, suppose the developer decides to label the list of undone projects as high priority, based on the current load time estimate. Given this information, HYPERLOOP will suggest different ways to render the page and optimizes the data processing leveraging priority. For instance, loading the undone projects panel asynchronously (to be discussed in Section 6), and furthermore storing them away in a dedicated list in memory for fast retrieval (to be discussed in Section 7). If the developer instead labels the active projects as high priority, and the undone projects as low priority, then HYPERLOOP will generate

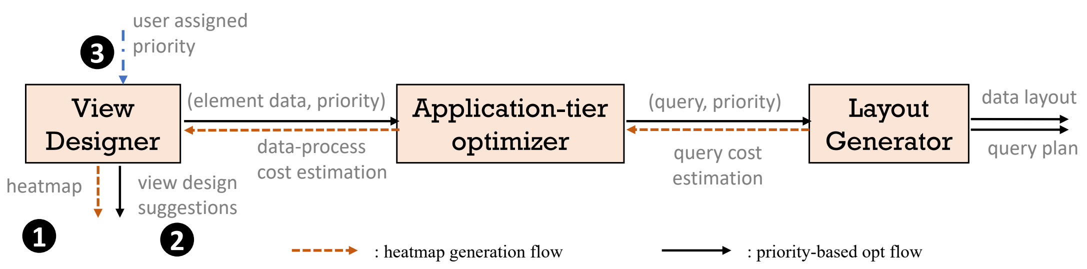


Figure 3: HYPERLOOP workflow

different rendering plans, for instance paginating the list of active projects across multiple pages (if there are a lot of active projects) while approximating the undone projects from previously cached data. HYPERLOOP is designed to be interactive: if the developer prefers not to paginate as it disrupts the user experience, she can indicate her preference using the HYPERLOOP user interface (and potentially reassigning the priorities), and HYPERLOOP will devise another render plan for the page.

After the developer picks one of the rendering plans, HYPERLOOP will automatically apply changes to the application code. As our current focus is on leveraging priorities for reducing load time, in the next sections we discuss different code changes we have implemented based on priorities, along with evaluations using real-world DBWA benchmarks. Our current prototype is designed for applications built using the Model-View-Controller (MVC) architecture [20] where the application is hosted on a single web server with source code available to be modified by HYPERLOOP.

3. PRIORITY-DRIVEN OPTIMIZATION

HYPERLOOP applies various optimization to different webpage elements depending on their priorities provided by the developer. These optimization often provides speedup to certain web-page elements (i.e., high-priority ones) at the cost of the loading time or the rendering quality of other elements (i.e., low-priority ones), and hence is not explored by traditional optimization techniques. We present a few optimization of this type below. We will then discuss how we implement these optimization in the next few sections.

Asynchronous loading. Asynchronously loading a view element e allows web users to see e before other potentially slow elements get loaded. The downside is that the total amount of computation or the total number of queries issued to the database may increase, because previously shared computation across asynchronously loaded components can no longer be shared. This optimization can be applied to high-priority elements, and will require view changes (Section 5) and application-tier changes (Section 6).

Pre-computing. While generating one web-page p_1, one can pre-compute contents needed to generate the next page p_2, which the web user is likely to visit next through a link on p_1. This will speedup the loading time of p_2 at the cost of the loading time of p_1. HYPERLOOP supports this optimization only when the developer provides high priority to the link on p_1 that points to p_2. It is implemented through our app-tier optimizer (Section 6) .

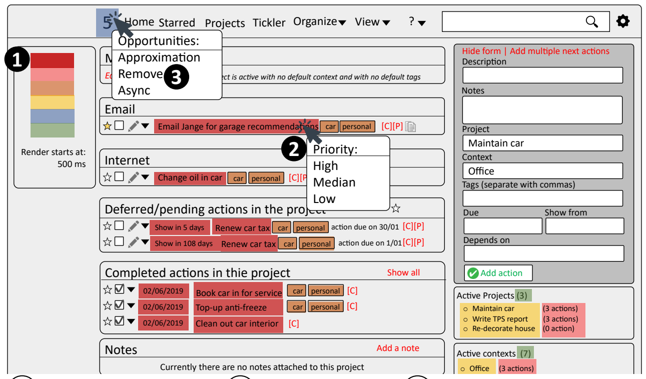
Optimizing for heavy reads or writes. There are often both read and write accesses to the same database table. Our database layout generator (Section 7) can optimize for either heavy-write workload, at the cost of read performance, or heavy-read workload, at the cost of write performance, based on the priority information provided by the developer.

Pagination. It often takes long time to retrieve and display a long list of items. One way to improve performance is to only show the first K items in the list and allow users to navigate to subsequent pages to view the remaining items. This change can greatly improve the loading time of the list, but at the cost of taking users longer time to see later part of the list. It can be applied to a list that contains both high and low priority items and items, or an overall low priority list whose content-viewing experience is less important than its loading speed. This is implemented in HYPERLOOP's view designer (Section 5) and app-tier optimizer (Section 6).

Approximation. Approximation can be applied to many aggregation queries, such as showing "you have more than 100 TODOs" instead of "you have 321 TODOs." Like pagination, approximation presents a tradeoff between loading speed and the quality (accuracy) of the content, and is suitable for low priority elements. This is implemented in HYPERLOOP's view designer (Section 5) and the app-tier optimizer (Section 6).

Using stale data. Caching data in memory and updating only periodically can improve performance at the cost of data quality and freshness. Priorities provided by developers can help HYPERLOOP determine which page element to cache. This is implemented in HYPERLOOP's app-tier optimizer (Section 6) and layout generator (Section 7).

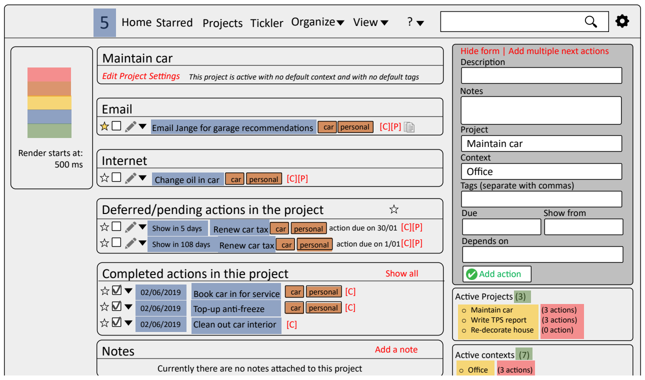
4. HYPERLOOP'S USER INTERFACE


5
Home Starred Projects Tickler Organize View ?
Opportunities:
Approximation
Remove
Async
3
1
Render starts at: 500 ms
Email
Email Jange for garage recommendations car personal [C][P]
2
Priority:
High
Median
Low
Internet
Change oil in car car personal [C][P]
Deferred/pending actions in the project
Show in 5 days Renew car tax car personal action due on 30/01 [C][P]
Show in 108 days Renew car tax car personal action due on 1/01 [C][P]
Completed actions in thie project
Show all
02/06/2019 Book car in for service car personal [C]
02/06/2019 Top-up anti-freeze car personal [C]
02/06/2019 Clean out car interior [C]
Notes
Add a note
Currently there are no notes attached to this project
Hide form | Add multiple next actions
Description
Notes
Project
Maintain car
Context
Office
Tags (separate with commas)
Due
Show from
Depends on
Add action
Active Projects (3)
Maintain car (3 actions)
Write TPS report (3 actions)
Re-decorate house (0 action)
Active contexts (7)
Office (3 actions)

(1) renders the heatmap; (2) assigns priority; (3) shows the opportunities

Figure 4: Heatmap showing the estimated loading cost of each webpage element, along with priority assignment and rendering recommendations generated by HYPERLOOP.

HYPERLOOP provides a unique interface for developers to understand the performance of their application and provide priority information. First, it presents the statically estimated cost (to be discussed in Section 6) to render each HTML element as a heat map



Maintain car
Edit Project Settings This project is active with no default context and with no default tags

Email
Email Jange for garage recommendations car personal [C][P]

Internet
Change oil in car car personal [C][P]

Deferred/pending actions in the project
Show in 5 days Renew car tax car personal action due on 30/01 [C][P]
Show in 108 days Renew car tax car personal action due on 1/01 [C][P]

Completed actions in thie project
02/06/2019 Book car in for service car personal [C]
02/06/2019 Top-up anti-freeze car personal [C]
02/06/2019 Clean out car interior [C]

Notes
Currently there are no notes attached to this project

Description:
Notes:
Project: Maintain car
Context: Office
Tags (separate with commas):
Due:
Show from:
Depends on:

Active Projects (3)
- Maintain car (3 actions)
- Write TPS report (3 actions)
- Re-decorate house (0 action)

Active contexts (7)
- Office (3 actions)

Figure 5: Heatmap after optimizing for undone projects on the left panel shown in Figure 1.

in the browser. This cost includes the time to retrieve the data from the database and process it in the application server. Figure 4(a) shows an example heat map of the webpage shown in Figure 1, where darker color means higher cost. For example, the left and bottom right panels have a high cost because of the large number of projects stored in the database, making the queries that involve them (e.g., Q1 in Figure 2) slow.

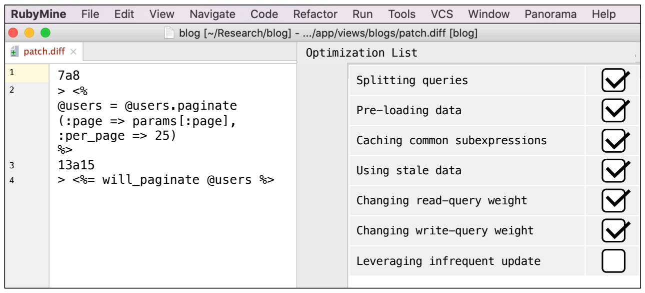
As discussed in Section 2, after examining the estimates, the developer can click on an HTML element on the page to indicate priority. We intend the interface to support applying the same priority to a group of elements after highlighting them. HyperLoop supports different priority levels as shown in Figure 4.

After assigning priorities, the developer would click on the "analyze" button on the right. HyperLoop then analyzes each element together with its priority, and provides a list of suggestions as shown in Figure 4. Some of these suggestions require further user input, for instance, how many objects to show on each paginated page. All of the suggestions are only related to webpage look and functionality designs, and the developer needs no database knowledge to choose a suggestion. We describe the list of suggested changes in Section 5.

The developer can right-click each element to view the rendering plan generated by HyperLoop. After choosing one of the plans, HyperLoop will change the application, re-estimates the cost, and renders the new webpage with a new heatmap, like the one shown in Figure 5, where the left panel is now loaded first.

HyperLoop not only suggests the rendering plan but also optimizes query processing based on priority assignment, as described in Section 6.2 and Section 7.2. These optimization strategies often involve tradeoffs, for instance, accelerating a query that retrieves data for high-priority HTML tags by slowing down queries for low-priority tags slightly. HyperLoop renders a list of such optimizations in the IDE and lets developers enable and disable them individually (by default HyperLoop applies all optimizations), as shown in Figure 6. The developer can then ask HyperLoop to regenerate the heatmap to see the effect of certain optimization(s). Doing so allows the developer to do A/B testing and understand how these optimizations interact with each other. Furthermore, HyperLoop can show the refactored code (after choosing rendering recommendations and a set of optimizations) if the developer wants to know the change in more detail, as shown in Figure 6.

HyperLoop supports assigning priority to both HTML tags as well as a form or a hyperlink to another webpage. If a form is assigned high priority, HyperLoop will attempt to reduce the time taken to process the form by changing the in-memory data layout



```
7a8
> <%
@users = @users.paginate
(:page => params[:page],
:per_page => 25)
%>
13a15
> <%= will_paginate @users %>
```

Optimization List	
Splitting queries	✓
Pre-loading data	✓
Caching common subexpressions	✓
Using stale data	✓
Changing read-query weight	✓
Changing write-query weight	✓
Leveraging infrequent update	☐

Figure 6: Left: refactored code. Right: a list of optimizations that developer can enable/disable individually.

of persistent data (to be discussed in Section 7). If the hyperlink is assigned high priority, HyperLoop will optimize the render time of the linked page, possibly by increasing the time taken to load the current page. As mentioned in Section 2, the developer can visualize such tradeoffs and reassign priorities using the HyperLoop interface as needed.

We next discuss the design of HyperLoop as shown in Figure 3 and how different tradeoffs are made given priority information.

5. VIEW DESIGNER

The *View designer* analyzes and transforms view files that define webpages' look and functionality. Its purpose is to identify which application tier object is rendered by which HTML element, and passes this information to the *Application-tier optimizer*. It also carries out priority-driven optimization as described below.

Asynchronous loading. To asynchronously load an HTML element e, the *View designer* splits the original HTML file into two files, one rendering e and the other rendering the rest of the page. To do so, the *View designer* first creates a new view file v to render e, and then creates new code to reside on the application server to compute the contents needed by e to render the view file v, and finally replaces e in the original view file with an AJAX request.

Pagination. The *View designer* detects pagination opportunities by checking whether an HTML element is rendering a list of Ruby objects in a loop. After the developer decides to paginate an element, the *View designer* rewrites the view file to render a constant number of elements first and adds a page navigation bar, as described in prior work [26]. It passes the design decision to the *Application-tier optimizer* who will change the query to return limited results (by adding `LIMIT` and `OFFSET`). Pagination itself can greatly accelerate the start render time. For example, paginating the left panel of Figure 1 to show 20 projects per page (out of 2K projects altogether) accelerates the panel rendering by 27×.

Approximation. The *View designer* detects approximation opportunities by checking if an HTML element is displaying a value that is returned by an aggregation query. Once the developer accepts an approximation optimization opportunity, the *View designer* changes the view file to add "at least" or "at most" before the aggregation value and passes it to the *Application-tier optimizer* to change the query to count only N values by adding a `LIMIT` clause.

6. APPLICATION-TIER OPTIMIZER

We now describe the static analysis framework in HyperLoop's *Application-tier optimizer* that enables a wide variety of optimization, including basic optimization that can be applied without priority information. Then we give examples on how it supports priority-driven optimization.

6.1 Analysis framework

The *Application-tier optimizer* statically analyzes the application code to understand 1) how the application computes and generates data that is to be rendered at each view component; 2) the flow of actions across consecutive pages.

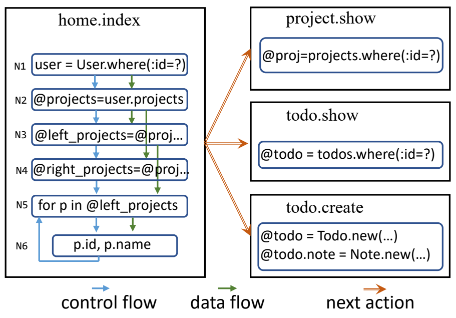


Figure 7: An Action Flow Graph (AFG) example.

To enable such analysis, the *Application-tier optimizer* constructs Action Flow Graph (AFG). An example is shown in Figure 7. An AFG is a flow graph consists of a set of hypernodes and *next-action* edges. Each hypernode represents a controller action, i.e., the complete code path used to generate a webpage. The *next-action* edge links pairs of actions (a_1, a_2) if a_2 can be invoked from a_1 as a result of a user interaction, for instance, by clicking a webpage. To identify such interactions, *Application-tier optimizer* identifies the HTML elements that contain URLs or forms and determines the subsequent actions that may be triggered as a result of an user interaction, for instance clicking on an URL or submitting a form.

Inside each hypernode, the *Application-tier optimizer* builds an action dependency graph (ADG). Every node n in the ADG represents a statement in the corresponding action. Every edge e represents either control dependency or data dependency. Nodes in the ADG are tagged as query nodes if they issue queries, with their data dependency edges labeled with database table and column names.

Using the ADG, the *Application-tier optimizer* can trace back from nodes that render data (e.g., N7 in Figure 7) to all the queries which the data-rendering node has control or data dependence upon (e.g., N1 and N2). These queries are considered as contributing queries.

This analysis enables the *Application-tier optimizer* to perform many types of optimization as introduced in earlier work [23]. Some optimization always improves performance, e.g., adding projection to load only fields being used. Others, however, requires making tradeoffs, which we discuss next.

6.2 Priority-driven optimization

We now discuss a few examples on how the *Application-tier optimizer* supports priority-driven optimization.

Example 1: Splitting queries. Very often DBWAs would issue a query to retrieve one set of data that will be filtered/processed in multiple ways to render multiple view components, as doing so can reduce duplicate work in rendering related view components. For example, a web page may show both a list of projects and a total count of these projects. The application can issue a single query to retrieve all projects while counting them in memory. Another example is the query to retrieve all projects (Q1 in Figure 2) into a Ruby array `@projects` that is filtered separately in memory to obtain `@left_projects` and `@right_projects`.

Although the shared query helps to reduce the total number of queries issued and the overall computation required to render the page, it could be sub-optimal if the multiple view components supported by it have different priorities. Specifically, to carry out an asynchronous loading optimization discussed in Section 3, the *Application-tier optimizer* splits a shared query if the result is used in asynchronously loaded elements. For example, if the left panel has high priority and is decided to be asynchronously loaded from other parts, the optimizer splits Q1 into Ql and Qr as shown below:

Listing 1: Example application code illustrating query splitting
```
Ql: @left_projects =
      user.projects.where(undone.count>0).include(todos,
      include(note))
Qr: @right_projects =
      user.projects.where(status='active').include(todos.count)
```

After the split, each query retrieves the data shown on the corresponding panel. Doing so causes the projects shown in the two panels to be retrieved in separate queries, but allows separate optimization of Ql, such as eager-loading of todos and notes for the left panel query (`where` and `include` are query functions to filter data using predicate and to eager-load the associated objects). Besides, the *Application-tier optimizer* will pass the design decision to the *Layout generator* and the splitting will allow generator to customize a layout for Ql (described in Section 7). As an illustration, with 4K total projects and 50% of them to show on the left panel, splitting the query and optimizing Ql as mentioned above reduces the query time of Ql from 5.1s to 0.5s, and the overall start render time from 13.7s to 6.1s.

Example 2: Pre-loading data. By default, each page is computed from scratch upon receiving an HTTP request. However, developers might know the next page(s) the user will likely visit and wish to pre-load data to accelerate loading of the next page, even at the expense of increasing the load time of the current page slightly.

An example is when a user visits the home page of a forum and then visits different posts by clicking on the hyperlink. As the home page shows only the title of each post, generating it once and caching it on the client slide would be optimal, but subsequent pages of individual posts might be impractical to cache as each may contain large images and contents. Yet, the developer might want the posts to load fast and is willing to trade off the performance of the home page. In that case, she can indicate priorities on the current page and HYPERLOOP will pre-load data accordingly. We use the Sugar forum application [2] as an illustration, where the database query retrieving the posts on its homepage selects not only the title but also the contents of each post. With a forum of 500 posts on the home page, doing so shortens each of the post page rendering time by 82% while increasing the home page load time by 12%.

Example 3: Caching common subexpressions. Common subexpressions are often shared among queries across consecutive pages [23]. Subsequent pages can reuse the results of these common subexpressions with a slight overhead for the current page due to caching. The *Application-tier optimizer* applies such caching if the developer chooses to pre-load data for subsequent pages after labeling with high priority. For example, for a page that shows the first 40 recent posts, the developer can assign the subsequent pages with high priority, as users will likely explore beyond the most recent 40 posts. The queries for the first and second pages are shown in Listing 2. They share the same subexpression that sorts the projects. In this case, the *Application-tier optimizer* will rewrite the queries to sort the posts, store the sorted results in a list and cache them such that the queries for all subsequent pages can simply return from this sorted list, as shown in Listing 3. With 10K posts, doing so slightly sacrifices the render time of the first page (an increase by 5%) but speedup the other pages by 2.3×.

Listing 2: Two queries sharing a common sub-expression

```
P1 : @posts = post.order(:created).limit(40).offset(0)
P2 : @posts = post.order(:created).limit(40).offset(40)
```

Listing 3: Common sub-expression result is cached and reused

```
P1 : @posts_all = post.order(:created)
     @posts = @posts_all.limit(40).offset(0)
P2 : @posts = @posts_all.limit(40).offset(40)
```

Example 4: Using stale data. It may be worthwhile to show stale data in a low-priority HTML element for better performance. The *Application-tier optimizer* implements this by changing the application code to cache data rendered in labeled HTML elements, and reuse it when the same element is rendered subsequently. For example, a developer may think the right bottom panel in Figure 1 occupies only a small and unimportant part of a webpage and thus labels it as low priority. The *Application-tier optimizer* will then suggest to cache the list of active projects, the total count, and the count of todos for each project. Doing so eliminates most of the queries to the database when rendering the page. To evaluate this, we use 2K active projects shown on the right panel of Figure 1, and the total rendering time is reduced by 65% after data for the right panel is cached.

7. LAYOUT GENERATOR

In this section we first introduce the basic optimization that the *Layout generator* can do without user interaction. Then we give examples on how it performs priority-driven optimization.

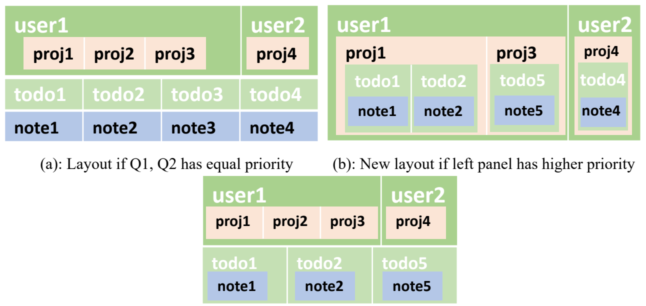
7.1 Basic optimizations

HYPERLOOP's *Layout generator* generates customized data layout for an application. It takes all the queries that can potentially be issued by the application and finds the best in-memory data layout to store the application data in order to improve the overall query performance. It also generates query plans and estimates the cost of each query.

The *Layout generator*'s data layout and query plan search space are specifically designed for object-oriented database backed applications like DBWAs. The layout design space is inspired by the object query interface. For instance, because queries often returns objects and nested objects, it is expensive to frequently join multiple tables and furthermore convert the tabular join results to nested objects. So the layout space incorporates not only the traditional tabular layout and indexes, but also deeply nested layouts.

The *Layout generator* first enumerates the possible data layout to store the data for each individual query as well as query plans that use particular layouts. Then it finds out the optimal layout for the entire workload by formulating an integer linear programming (ILP) problem. In this formulation, each data structure in every possible layout is assigned a binary variable to indicate whether it is included in the final layout; similarly for each query plan. It also estimates the memory cost for each data structure and the time for each query plan. For write queries, it generates one plan to update one data structure. The optimization constraints state that the overall cost of all included data structures are within the memory bound provided by the user, while the optimization goal is to minimize the overall runtime of all queries. It uses state-of-the-art solvers to solve the ILP problem and constructs the final layout accordingly. Finally, it generates an implementation of both data layouts and query plans.

7.2 Priority-driven optimization



(a): Layout if Q1, Q2 has equal priority (b): New layout if left panel has higher priority

(c): New layout if todo-form has higher priority

Figure 8: Priority-driven layout design.

We next discuss a few examples on how *Layout generator* supports priority-driven optimization. All examples below reduce the load time for the selected elements labeled with high priority at the expense of possibly slowing down other elements in the same page.

Example 1: Changing read-query weight. The *Layout generator* can design a data layout that better optimizes queries with high priority. Since it formulates the search of data layout into an optimization problem where the optimization goal is the sum of runtime cost of all queries, it can simply assign higher weights to those queries whose results are needed to render higher-priority HTML elements, as identified by HYPERLOOP's *Application-tier optimizer*.

For example, consider Q1 and Q2 shown in Listing 1 before assigning priorities. The *Layout generator* may produce a layout as shown in Figure 8(a). If the developer assigns higher priority to the left panel, Q1 will receive a higher weight, which results in the layout shown in Figure 8(b). In this layout, the `projects` belonging to a user are stored as a nested list within `user`; the `todos` are stored as nested objects in each `project`; and similarly for the `notes`. This layout is highly optimized for Q1: retrieving the `@left_projects` does not need to perform a join on `project`, `todo`, and `note` compared to using layout (a). This layout also avoids the expensive deserialization from denormalized table to nested objects. With this layout, the query time for the left panel is further reduced to 0.5s (compared to 5.1s using the tabular layout).

Example 2: Changing write query weight. An HTML form might be assigned high priority if it is frequently used, such as the form shown in upper right of Figure 1 if new todos are frequently added. The queries used for new todo submission are shown below:

```
@todo = Todos.new(name=param['name'], ...)
@todo.note = Notes.new(content=param['note_content'],...)
@todo.save
```

In this case, the *Layout generator* adds larger weight to the write query creating new todo with embedded notes. It would then generate data layout in Figure 8(c). Compared to (b) where inserting a new todo needs an extra query to locate the project that this todo belongs to, (c) is better optimized for adding new todos because the `todos` are stored as a top-level array and an insertion only appends to the array without the extra read query.

Example 3: Leveraging infrequent update (stale data). If an HTML element is assigned low priority, the query that retrieves data for this element can potentially read from stale data. The *Layout generator* can generate a more efficient data layout for this type of read queries. For example, if the count of all active projects (as shown in the right bottom panel of Figure 1) is assigned the lowest priority, then the generator will assign a very low weight to any plan that updates the data structure only used to compute this count. The generated data layout will pre-compute the count and store it in memory, which reduces the end-to-end webpage time when the count is rendered. For instance, using stale data for the active project and context count in the right bottom panel can accelerate rendering the panel by 26% (after the list is paginated). Without knowing the priority, it is unlikely to pre-compute the count because any delete query triggers a re-computation of this count, which greatly increases the total query cost.

8. RELATED WORK

Priority in Software Development. The concept of "priority" has been widely used in software engineering. For example, in agile programming, one common practice is to list user stories (i.e., user-facing software features) and give them priority in the development cycle [4, 6]. We use the same concept in making the view-performance tradeoffs. However, rather than using priorities to assist in ordering which software feature to implement first, HYPERLOOP instead leverages priorities to improve the page-viewing experience of end-users, with the assumption that high-priorities are assigned to webpage elements that are intended to catch viewer's attention.

Database Optimizations. The database community has proposed query optimizations similar to those described in the paper. For example, identifying and caching shared subexpressions in the context of multi-query optimization [16, 17], leveraging stale data like using lower consistency level in the context of transaction processing [21], automatic design of materialized views that different query weight leads to different views in the context of physical design [7, 15]. etc. Although many optimizations are not new, when to implement them and how to make the tradeoff in DBWAs require developer's knowledge and preference. We propose the design of an easy-to-use interface to leverage developer's preference via priority and automate the optimization implementation.

Optimizing Database-backed Applications. Much work has been done on discovering of performance issues of database-backed web applications, such as identifying performance problems in DBWAs, such as retrieving unneeded data [11], issuing long query chains that are difficult to optimize [10], and other API misuses [25]. Prior approaches also include solving these issues [13, 12, 14, 8, 19, 22, 24], but focus on performing semantic-preserving low-level code changes on the application automatically similar to an optimizing compiler, and they all assumed the goal is to reduce the latency in loading the *entire* page. HYPERLOOP is not designed to be an optimizing compiler, but instead focuses on aiding the developer prioritize page elements to optimize directly from the view, and suggests various kinds of code and view changes by leveraging the priorities provided by the developer.

9. CONCLUSION

We presented HYPERLOOP, a new system that helps developers optimize DBWAs. Unlike prior approaches, HYPERLOOP recognizes that developers often make tradeoffs when designing DBWAs, and leverages developers' knowledge to optimize DBWAs in a view-driven manner. Given priority information provided by the developer, HYPERLOOP automatically analyzes the application and suggests various design and code changes to improve the render time of different elements on the page. While still under implementation, preliminary results have shown that our view-driven approach is effective in improving end user experience of real-world DBWAs.

10. ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1546543, IIS-1651489; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center, and gifts from Adobe and Google.

11. REFERENCES

[1] *A Designer's Guide to Fast Websites and Perceived Performance*. https://www.sitepoint.com/a-designers-guide-to-fast-websites-and-perceived-performance/.

[2] sugar, a forum application. https://github.com/elektronaut/sugar.

[3] tracks, a task management application. https://github.com/TracksApp/tracks/.

[4] *User stories: an agile instroduction*. http://www.agilemodeling.com/artifacts/userStory.htm.

[5] *Website Response Times*. https://www.nngroup.com/articles/website-response-times/.

[6] *What is agile methodology?* https://resources.collab.net/agile-101/agile-methodologies.

[7] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[8] M. B. S. Ahmad and A. Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *SIGMOD*, pages 1205–1220, 2018.

[9] Akamai and Gomez.com. *How Loading Time Affects Your Bottom Line*. https://blog.kissmetrics.com/loading-time/.

[10] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1001–1012, 2014.

[11] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. In *ICSE*, pages 1148–1161, 2016.

[12] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

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

[14] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 3–14, 2013.

[15] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *Proc. VLDB Endow.*, pages 362–372, 2011.

[16] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, pages 526–537, 2012.

[17] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[18] G. Linden. *Marissa Mayer at Web 2.0*. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html/.

[19] K. Ramachandra, C. Mahendra, G. Ravindra, and S. Sudarshan. Program transformations for asynchronous and batched query submission. In *TKDE*, pages 531–544, 2015.

[20] T. Reenskaug and J. Coplien. More deeply, the framework exists to separate the representation of information from user interaction. *The DCI Architecture: A New Vision of Object-Oriented Programming*, 2013.

[21] S. Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *SIGMOD*, pages 729–730, 2012.

[22] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. In *PVLDB*, pages 444–455, 2016.

[23] C. Yan, J. Yang, A. Cheung, and S. Lu. Understanding database performance inefficiencies in real-world web applications. In *CIKM*, 2017.

[24] J. Yang, U. Sethi, C. Yan, S. Lu, and A. Cheung. Managing data constraints in database-backed web applications. In *Proceedings of the International Conference on Software Engineering*, 2020.

[25] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung. Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *FSE*, 2018.

[26] J. Yang, C. Yan, C. Wan, S. Lu, and A. Cheung. View-centric performance optimization for database-backed web applications. In *Proceedings of the 41st International Conference on Software Engineering*, pages 994–1004. IEEE Press, 2019.