Application View Maintenance: Optimizing Change Propagation in Mobile Applications

Konstantinos Zarifis
University of California, San Diego
zarifis@cs.ucsd.edu

Yannis Katsis
University of California, San Diego
ikatsis@cs.ucsd.edu

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

Abstract

Web Frameworks that adopt the Model-View-ViewModel (MVVM) design pattern have been extensively used in the web community for the development of fully-fledged applications. Such frameworks, typically, provide algorithms that automate the maintenance of the application's view when mutations occur to the underlying data (also known as model). The automation of this process, commonly referred to as Application View Maintenance (AVM), significantly improves developer productivity, since it alleviates the developer from manually performing this task. Such algorithms are also capable of mutating individual parts of the view when the underlying data mutate, thus avoiding a full reload and rerendering of the entire application view, (a very expensive operation for HTML content, especially in the mobile setting).

However, as we show in this work, AVM algorithms of existing MVVM frameworks are still suboptimal performance-wise. By continuously exploring the model for mutations, they have a complexity that is proportional to the size of the model and not to the size of mutations. This suboptimality combined with the low computational power of mobile devices, can lead to severely inefficient mobile apps, which can also impact the user experience. To address this issue, we propose a novel AVM algorithm which uses existing incremental view maintenance techniques, to directly identify the mutated parts of the model and infer the respective parts of the view that need to be updated, while avoiding a blowup in complexity proportional to the size of the model of the application. The complexity and memory consumption of the proposed algorithm are shown to be typically significantly lower than existing approaches.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks

General Terms Languages, Performance, Algorithms

Keywords MVVM, change propagation, incremental view maintenance

1. Introduction

Model-View-Controller (MVC) frameworks were, until very recently, the state-of-the-art for implementing robust client-side web applications. When using such frameworks, the application developer is solely responsible for propagating mutations to the state of the application (model) and subsequently to the respective visual layer (view). This change propagation is handled manually by employing imperative logic for every single event that might cause changes at any part of the application state. This extended use of imperative code makes application development very laborious and error-prone, especially in bigger, more complex applications, and results in a code-base that is very difficult to debug and maintain.

MVVM web application frameworks have managed to absolve developers from such low-level handling of change propagation by providing constructs that allow them to work at a higher level of abstraction. Such frameworks, typically, allow developers to describe the application using a declarative specification (template). Intuitively the template contains rules that describe how the model of the application can be transformed into its view. One of the important benefits of a declarative template is that frameworks can reason about it and automate the process of change propagation. Popu-

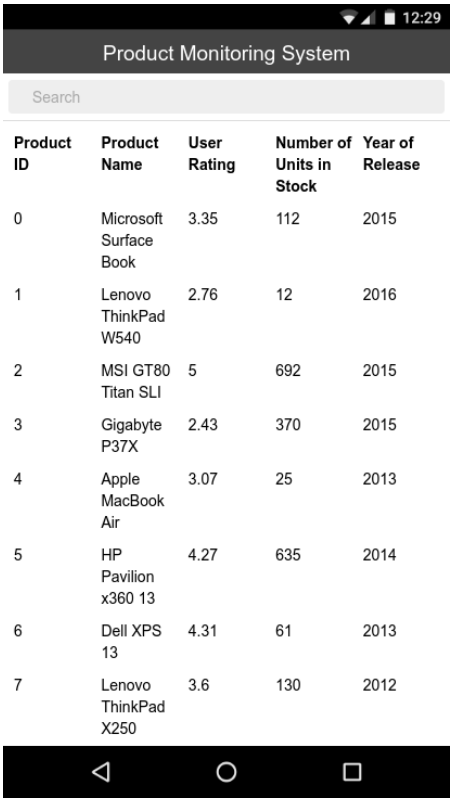
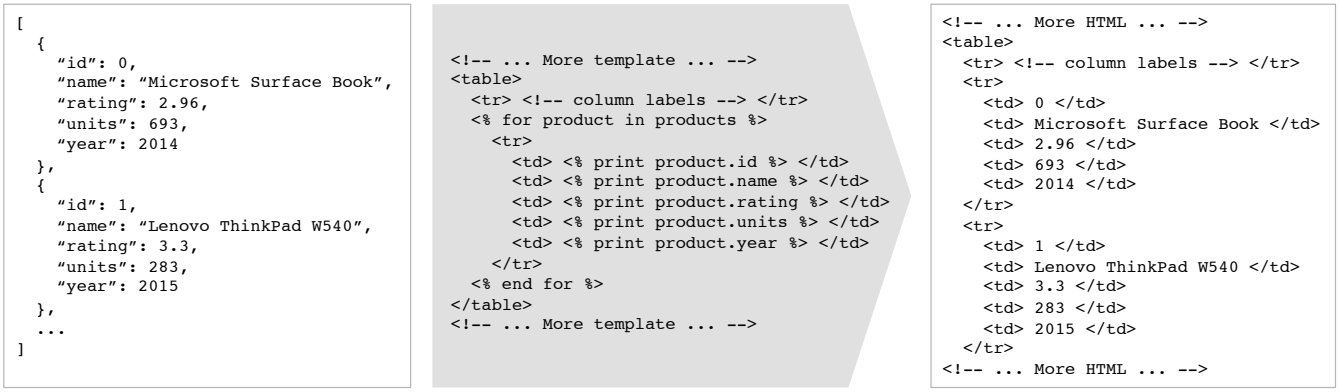


Figure 1. Running Example: Product Monitoring System

World Wide Web Conference '17 April 3– 7, 2017, Perth, Western Australia

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Model (b) Template (c) View

Figure 2. MVVM Components for Running Example

lar MVVM frameworks automatically transform mutations on the model to mutations on the view through mechanisms that are invisible to the application developer. This notably decreases the amount of code that has to be written and results in easier-to-develop and maintain applications while at the same time offering more efficient rendering since only the parts of the view that depend on mutated parts of the model are rerendered.

Unfortunately this automation does not come without drawbacks. As we explain in this paper change propagation algorithms in existing MVVM frameworks are suboptimal, performing much worse than the carefully crafted change propagation code that an application developer would write in an MVC framework. This suboptimality comes mainly from the fact that existing MVVM frameworks do not know how the model was mutated since the last invocation of the change propagation algorithm. In their attempt to identify the updated parts of the model, they continuously compare the previous state with the current state of it, resulting in a complexity that is proportional to the size of the model. In other words, existing change propagation algorithms may end up exploring the entire model for changes even if a very small part of it actually changed.

In this work, we explain this inefficiency and present a novel change propagation approach that avoids computations proportional to the size of the model. Our approach has two important components: The first component ensures that the change propagation algorithm knows directly which parts of the model changed. This is accomplished by observing that most web applications derive their model from databases, which can - using existing algorithms, known as *incremental view maintenance (IVM) algorithms* - provide information on what has been modified. The second component makes sure that these changes to the model are efficiently translated to changes on the view. We present an algorithm that performs this transformation, which, although inspired by traditional IVM approaches, significantly differs from them due to a different data model and language. The resulting change propagation algorithm has complexity that is proportional to the maximum of (a) the (typically small) size of the specification of the application and (b) the changes that need to be applied to the view (i.e., the output of the algorithm). By not depending on parts of the model that have not changed, our algorithm can exhibit significant savings over existing approaches.

Contributions. To summarize, this work makes the following contributions:

- A description of the change propagation algorithms of existing MVVM frameworks, with an analysis of their complexity, showing that they are at least proportional to the size of the model.
- A novel architecture that leverages traditional IVM techniques to directly inform the framework of specific model mutations.
- A novel change propagation algorithm that can efficiently translate model mutations to view mutations with a corresponding complexity analysis showing its complexity is typically significantly lower than that of existing approaches.
- An experimental evaluation of our change propagation algorithm against the respective change propagation algorithms employed by the most popular MVVM frameworks currently available.

Paper Outline. The paper is structured as follows: In Section 2 we explain the general architecture of MVVM frameworks. In Section 3 we describe the algorithms used by existing MVVM frameworks, showcasing their suboptimal nature. This inefficiency is addressed in Sections 4 and 5 where we introduce a novel change propagation algorithm inspired by works in incremental view maintenance. In Section 6 we provide an experimental evaluation of our propagation algorithm against other MVVM frameworks. Finally, in Sections 7 and 8 we discuss related work, future work and conclude the paper.

2. Background - MVVM Framework Architecture

We next present the basic architecture of MVVM frameworks. Although individual frameworks differ in their details, most widely used MVVM frameworks, including AngularJS [5] and ReactJS [20] follow the same basic architecture outlined below. To describe the MVVM framework architecture, we will be showing how such frameworks can be utilized for the implementation of a mobile (hybrid) application that can be used to monitor information about products offered by an e-commerce company. We next describe the details of this application.

EXAMPLE 2.1. *The screen, shown in Figure 1, displays live information about the product availability at the company's warehouse. Technically, this screen is rendered as an HTML table, with each row corresponding to an individual product and each column to a particular attribute of a product, such as product ID, name, user rating, stock availability and year of release. The screen also con-*

tains a search box that simplifies the process of obtaining information about individual products. As the user types the name of a particular product the application automatically scrolls up or down in order to show the row that contains information about the selected product while at the same time it automatically highlights the product name (similarly to how the search functionality works on a desktop browser). It is important to note that, as with most modern monitoring applications, this system is live, therefore the screen is updated whenever the underlying data, such as the user rating or the stock availability of a product, change.

An MVVM framework typically consists of three components: the *model*, containing the data that should be displayed on the page, the *view*, which is the visual page the application user finally sees and interacts with, and the *template*, which is a declarative specification file describing how to transform the data (i.e., the model) to the visual page (i.e., the view).

EXAMPLE 2.2. *Figure 2 shows the model, view, and template for our running example. The model (shown on the left) consists of a JSON array containing the product information as it is typically retrieved from a database. The view (shown on the right) is the HTML code corresponding to the product table as it is rendered by the browser.*[1] *Finally, the template (shown in the middle) is a declarative specification describing how the model is used to create the view.*

Template languages differ across frameworks. However, abstracting out their differences, all template languages offer at least the following features: (a) an *iteration* construct, similar to a for loop, allowing iteration over the members of a collection in the model, (b) a *print* construct allowing values of the model to be printed in the view, and (c) an *HTML generation* construct allowing the creation of HTML elements, such as tables, cells, etc. To abstract out from notations and implementation details of particular frameworks we will be using in this work a simple template language that offers the above features.[2] The language allows users to specify templates by writing HTML code that may have two types of embedded directives: a *for* directive, corresponding to the iteration construct and a *print* directive, corresponding to the print construct. We say that each of these directives *binds* an element of the model (being a collection or a scalar) to the view. Due to space limitations we do not give a formal definition of the template language but explain it instead through an example.

EXAMPLE 2.3. *Figure 2b shows the template of our running example. It contains HTML table tags to generate the table object, a* for *directive to iterate over the product array and create a row for each product and print directives to print in table cells each of the product attributes.*

When a page is first loaded, MVVM frameworks evaluate the template over the model and create the view, which is then rendered by the browser. What is more interesting is what happens when the model is updated (for instance when the stock availability of a product changes). To avoid recomputing the entire template and re-rendering the entire view whenever a small part of the model changes, MVVM frameworks typically contain an automatic

[1] Technically, the HTML code displayed in Figure 2c is a *description* of the view, in contrast to the actual view, which is the visual page produced as the result of rendering the code. However, this distinction is immaterial to this work and therefore we will be referring to the HTML code as the view.

[2] Although some template languages offer additional constructs, in this work we focus on the above constructs, which also represent the core constructs of web applications. Support for a more expressive template language will be the focus of our future work.

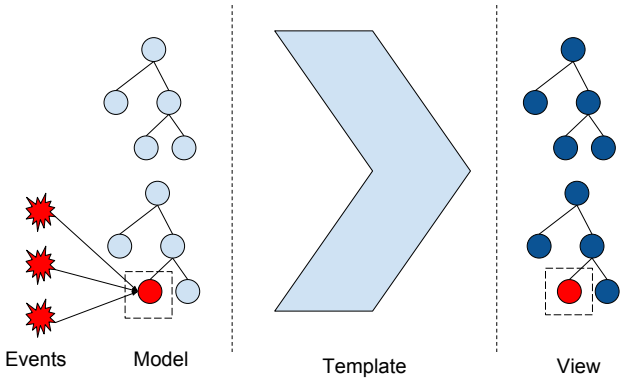


Figure 3. Life-Cycle of an MVVM framework

change propagation mechanism for modifying and re-rendering only the affected parts of the view. This mechanism is described in the next section.

3. Change Propagation in Existing MVVM Frameworks

During the lifetime of an application, the underlying model is subject to mutations. Upon such events, MVVM frameworks invoke a change propagation algorithm that identifies the model changes and infers the corresponding view changes that have to be applied. Figure 3 graphically depicts this process. In this Section we look at such change propagation algorithms and show that they are suboptimal both in efficiency and memory consumption. We deliver the argument using as an example the change propagation algorithm of AngularJS [5], one of the most popular MVVM frameworks, supported and maintained by Google and by a rich community of third-party contributors. As we briefly discuss at the end of the Section though, similar arguments can be delivered for every other popular MVVM framework.

In a nutshell, change propagation in Angular takes place in the following manner: Starting from the fully evaluated view, Angular identifies all visual elements that are derived from the model (i.e., were created through template bindings) and marks for each of them, the respective part of the model they were derived from.

EXAMPLE 3.1. *In our running example, Angular marks that the value "Microsoft Surface Book" in the view was generated from attribute "name" of the first element of the model array. Similarly, for all the other values of the view that come for the model.*

For each derived part of the view, Angular places a *watcher* that observes the corresponding part of the model that affected that particular part of the view. When this watcher is invoked, it checks whether the observed object was mutated since its last invocation by comparing the current value of it (also referred to as post-state) to its old cached value (also known as pre-state). Angular utilizes two types of watchers, *shallow* and *deep*. Shallow watchers compare changes to the reference of the object, while deep watchers compare changes to the entire value of the object (which may entail recursively comparing nested objects). If this process, called *dirty checking*, identifies that the observed part of the model was mutated, the framework invokes a renderer that is capable of appropriately mutating the respective part of the view.

EXAMPLE 3.2. *For instance, in our running example, a watcher will be placed to observe (among others) the* `units` *attribute of the first element of the model array to check whether the stock availability of the first product changes. If it does (e.g., from 693*

to 692), the watcher will call the appropriate renderer to change the text shown on the corresponding cell of the HTML table from 693 to 692.

When change propagation is initiated, AngularJS carries out a process called *digest-cycle*, which iteratively invokes *all* watchers that have been placed on the model to perform dirty checking. Since all watchers are invoked when the digest-cycle is triggered, Angular's propagation algorithm has a complexity of $O(wd)$, with w, the number of watchers and d, the size of the observed object (with $d = 1$ if all watchers are shallow). Since each part of the model bound to the view has a corresponding watcher, the complexity of the propagation algorithm is essentially proportional to the total number of model attributes that are bound to the view. This number can be substantial especially for applications that display a big part of the model. As a result, even if only a minor mutation is applied to the model, Angular will end up comparing the pre-state to the post-state of a very large portion of it (only to find that most of the checked parts were not even modified).

This procedure is not only computationally expensive, but also memory-intensive, as it requires storing both the pre-state and the post-state of the entire model in the main memory, so that watchers can perform the comparison between the two. For that reason applications that contain large views (which are derived from equally large models) often have very high memory requirements that may not be easily satisfiable by mobile devices. Furthermore, given the limited computational resources of such devices and the single-threaded nature of JavaScript (which is the language these algorithms are written in), this procedure can also lead to unresponsive applications since the user-interface often appears to be frozen while this propagation algorithm operates, which significantly impacts the user experience of the application.

EXAMPLE 3.3. *Continuing our running example, assume that the model consists of 1,000 products. Since each product consists of 5 attributes, each shown on a cell in the view, Angular will place 5,000 watchers. Thus, every time a single value of the model is changed (e.g., the stock availability of the first product), Angular's change propagation algorithm will invoke 5,000 watchers to compare the pre-state and post-state of all 5,000 values in the model (even though all of them but one have remained stable).*

Discussion. While one may think that this substantial number of comparisons is an inefficiency particular to Angular, it is in fact a common property of most popular MVVM frameworks (such as ReactJS [20], Mithril [17] and others). Although the specific details and the resulting complexity differs slightly between frameworks (e.g., ReactJS compares the pre-state and post-state of the view instead of the model), all frameworks end up performing a large number of comparisons. The reason is that they lack a way of tracking the mutations that happen to the model and thus have to perform a significant number of comparisons to reverse engineer the changes. However, as we show next this can be avoided by appropriately tracking the mutations that happen to the model using techniques borrowed from the database incremental view maintenance literature.

4. IVM-inspired Change Propagation

Existing MVVM frameworks treat the model as a black box object that may be mutated in any way between invocations of the change propagation algorithm. This is the reason why during change propagation they spend significant processing and memory resources in identifying the mutations that have happened to the model. In this section we show how this can be avoided. In particular, we show that for data-driven web applications that retrieve their model from a database, the framework can know directly which changes have happened to the model by adopting existing incremental view maintenance (IVM) techniques proposed in the database literature. Given this knowledge of the model mutations, the framework can compute the corresponding view mutations by reusing, and in this instance, also extending existing IVM techniques. Given the importance of IVM in our approach, we next briefly outline the IVM problem in databases before showing how it can be used for change propagation in web applications.

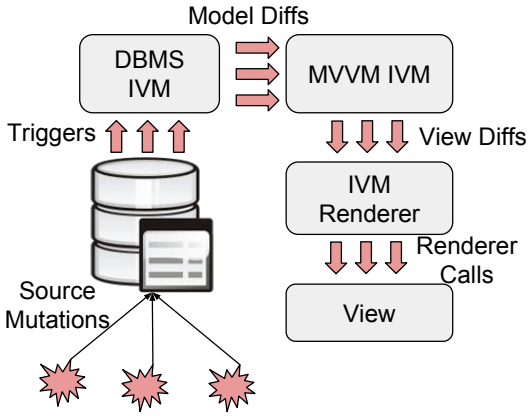


Figure 4. Architecture

4.1 IVM in Databases

A common technique used in database management systems (DBMSs) to accelerate query processing is to store in the DBMS the results of commonly asked queries. These stored query results are referred to as *materialized views*. While materialized views speed up query answering, they also introduce a complication: Since materialized views are results of queries over the data, they can get out of sync when the underlying data change and need to be maintained. A straightforward solution for view maintenance is to recompute the entire view (i.e., query) whenever the underlying data change. However, this can be very costly, especially when the dataset is large or the query performs some expensive computation (such as aggregations) or contains calls to complex user defined functions.

To avoid this problem, database researchers have worked on *incremental view maintenance (IVM)* algorithms, which given as input the changes that happened to the base tables, compute the changes that have to happen to the view. By focusing only on propagating changes (and not recomputing the query over the entire dataset) IVM algorithms are typically significantly faster than query recomputation approaches. IVM has been a widely studied technique in the database literature, with a very large number of publications ranging from the early years of the database field [6–8, 13, 18] to very recently [4, 15]. For comprehensive surveys on incremental view maintenance, the reader is referred to [11, 12].

An important concept in IVM works is the notion of *diffs*. Diffs describe changes that happened to a table and are therefore used to represent both the input of an IVM algorithm (i.e., the changes that have happened to the database tables), as well as its output (i.e., the changes that should be applied to the table corresponding to the materialized view). For the purpose of this discussion, diffs can be thought of as descriptions of which tuples have been inserted, deleted or updated in a table.

4.2 IVM-inspired Change Propagation

Having described IVM in general, we are now ready to describe how it can be used to enable efficient change propagation in

MVVM frameworks. In this work we focus on applications, whose model is created by querying a database. This is very common in today's era of data-driven web applications.

EXAMPLE 4.1. *The model of our running example could be created by posing the following query over a database containing two tables,* `Product` *and* `Stock` *containing general information about products and stock availability, respectively:*

```
SELECT p.id, p.name, p.rating, s.units, p.year
FROM Product p, Stock s
WHERE p.id = s.id
```

Since the model is the result of a query over the database, we can employ existing IVM approaches to provide the framework directly with the changes that took place on the application model. In this case, we treat the application model as a materialized view and ask the DBMS IVM to simply provide the diffs that represent the changes. In this way, we avoid expensive computations done by existing frameworks that reverse-engineer the mutations that occurred (by comparing the pre-state of the model or view with its post-state) while at the same time limiting the network utilization of the application since only the information that is needed to update parts of the model is transmitted to the client instead of the entire model.

However, this is only the first step. Given the changes to the model, the framework has to identify the parts of the view that have to be modified. In existing MVVM frameworks, such as Angular, this is straightforward, since for each part of the view that can be modified the framework assigns a watcher over the model. This watcher is attached to the particular part of the view and thus whenever the watched object changes, the framework knows which part of the view to modify. In our proposed framework though, where watchers are not preemptively built for all parts of the view that can be modified, there is no immediate connection between model diffs and corresponding view changes. The translation has to be done by an algorithm. This algorithm (which we call MVVM-IVM, as it is inspired by IVM approaches) takes as input model mutations (represented as model diffs) and translates them to mutations on the view description (represented as view description diffs). Finally, the view description diffs are translated to appropriate renderer invocations that perform the actual view mutations.

Figure 4 shows the resulting architecture. Mutations on the database tables trigger a traditional DBMS IVM algorithm, which computes the diffs to the application's model. These model diffs are eagerly propagated to the client with the use of WebSockets [2]. Upon arrival the MVVM-IVM algorithm translates these model diffs into diffs on the view description. Finally, the IVM renderer utilizes the view description diff to access the DOM element, targeted by it and invoke renderers that are capable of causing the apropriate mutations to the respective part of the view.

Out of the above modules, the main novelty lies in the MVVM-IVM algorithm. It is important to note that, in contrast to the DBMS IVM algorithm, which we simply borrow from existing work, the MVVM-IVM algorithm, although inspired from IVM works, is not a direct adaptation of any existing algorithm. It may have the same signature as traditional IVM algorithms (which input diffs and output diffs) but it operates on a different data model (JSON/HTML instead of the relational model of DBMSs), as well as on a different language (the template language in contrast to the SQL language targeted by DBMS IVM techniques). We next describe the algorithm and discuss its complexity.

5. The MVVM-IVM Algorithm

The MVVM-IVM algorithm takes as input a set of mutations on the model and produces a set of mutations on the view description.

Signature	*Semantics*
$\Delta^{\text{insert}}_{\text{array}}(\hat{p}[k]; v)$	Insert into array at path $\hat{p}$, at position k the value v
$\Delta^{\text{update}}(\hat{p}; v)$	Replace element at path $\hat{p}$ with value v
$\triangle^{\text{delete}}(\hat{p})$	Delete element at path $\hat{p}$

Table 1. Possible Diff Signatures

As in traditional IVM approaches, both input and output mutations are represented as diffs. We next describe the structure of a diff.

Diffs. A diff is of the form $\Delta^{\text{type}}(path; payload)$, where $type$ is the type of the modification (insert-array, update, or delete), $path$ is the path to the element that has changed and $payload$ is the new value of the element (unless the diff is of type delete in which case the $payload$ is not applicable). Table 5 summarizes the possible diff signatures and their semantics.

EXAMPLE 5.1. *Continuing our example, assume that the stock availability of the first product changes from 693 to 692. In that case, the incoming diff to the MVVM-IVM algorithm will be* $\Delta^{\text{update}}(products[0].units; 692)$*, which denotes that the value of the attribute* `units` *of the first products tuple changed to 692.*

Our diffs use a simple path language, where $p[i]$ represents the i-th element of the array at path p and $p.n$ represents the value of the attribute named n in the tuple at path p. We also assume that an empty path represents the root of the model. Note that for uniformity the same path language is also used by the output diffs, which refer to the view description, which is an HTML (and not a JSON) document. In this case to interpret the paths one has to look at the HTML document as a JSON document, where each node of the HTML document is represented as a tuple with a single attribute/value pair with the following structure: The attribute of the tuple is the name of the node, while the value is an array containing the JSON representation of the node's children. For instance, using this representation, an HTML document of the form `<tr><td>1</td><td>2</td></tr>` is represented by the JSON value `{"tr": [{"td": 1}, {"td": 2}]}`.

The MVVM-IVM algorithm. Given a set of diffs on the model, the MVVM-IVM algorithm (shown as Algorithm 1) translates them into diffs on the view description. To do the translation, it scans the template from top to bottom, performing work only for the directives (since they are the ones that can bind elements to the view description). In each invocation it visits only top-level directives (i.e., directives that do not appear nested inside other for directives), as nested directives are acted upon during recursive calls. Since our template language supports two types of directives - *for* directives and *print* directives - we distinguish two cases.

In the case of a *print* directive of the form $<\%$ print $\hat{p}$ $\%>$, where $\hat{p}$ is a path expression (lines 5-7), the algorithm works in two steps. It first computes how the input diff transforms the element targeted by the path (i.e., it computes a new diff for the element targeted by the path). At this point the diff describes how the path expression was changed but does not yet take into account where this value will appear in the view description (i.e the diff does not have the appropriate path yet). To add this information, the algorithm in the second step prefixes the path of the diff with the path where the directive appears in the template. The first step is performed by the `IVMPath` procedure (shown as Algorithm 2) and the second step is performed by the `prefixDiffs` procedure (omitted due to lack of space).

In the case of a *for* directive of the form $<\%$ for var in $\hat{p}$ $\%>$, where $\hat{p}$ is a path expression (lines 8-29), the algorithm proceeds

as follows. It firsts computes the diff on the path expression by calling the `IVMPath` procedure as explained above. The next step depends on the computed diff. There are three mutually exclusive cases (lines 8-24) and one case that may occur together with one of the others (lines 25-29). We start by explaining the first three cases:

- If the diff updates the entire element targeted by $\hat{p}$ (in other words if it changes the entire element over which the for loop iterates) (lines 10-14), then the algorithm evaluates the for loop on the diff payload (i.e., on the new value of $\hat{p}$) and creates a diff that replaces the result of the original for loop evaluation with the newly computed result. In order to evaluate the for loop on the diff payload, the algorithm instantiates the sub-template rooted at the for directive by calling the `instantiateTemplate` procedure.
- If the diff inserts a new element to the element targeted by $\hat{p}$ (i.e., if it inserts a new element to the collection over which the for loop iterates) (lines 15-18), then the algorithm evaluates the body of the for loop on the value of the new element and creates a diff that inserts the result of this computation at the right position of the view description.
- If the diff mutates an element of $\hat{p}$ (i.e., if it mutates an element of the collection over which the for loop iterates) or part thereof (lines 19-24), then the algorithm first constructs a diff that mutates the loop variable *var*. Then the algorithm recursively calls itself for the sub-template rooted at the body of the for-directive and the loop variable diff.
- In addition to these three mutually exclusive cases, there is also a fourth possibility, which applies in scenarios where directives that appear nested inside the for loop depend on variables other than the loop variable *var*. For instance, this would happen in our running example if the for directive contained in its body the line `<td><% print products[0].units %></td>` (i.e., if for some reason the template was printing the stock availability of the first product in the entry of *every* product produced by the for directive). In this case (lines 25-29), the algorithm first recursively calls itself for the sub-template rooted at the body of the for-directive (without binding the variable of the for loop). Then it extends the resulting diffs so that they apply to *every* element produced by the for directive.

Discussion. The complexity of three out of the five cases (i.e., the case of the print directive and the second and third case of the for directive), depends only on the size of the template. The only cases that depend on the size of parts of the model are the first and fourth cases of the for directive, when the algorithm evaluates a for directive and creates diffs for every element produced by a for directive, respectively. However, in both cases the work is proportional to the output of the algorithm, i.e., to the changes that have to be applied to the view description. Therefore, the MVVM-IVM algorithm has a complexity proportional to the maximum of the size of the template (which is usually small) and the size of the changes to the view description. This comes in stark contrast to change propagation in existing MVVM frameworks, which have complexity at least proportional to the size of the model that is bound to the view. Note that the difference in complexity can be substantial, as the complexity of the MVVM-IVM algorithm depends on the size of the changes, while the complexity of existing change propagation algorithms depends also on parts of the model that do not lead to any changes. The complexity of our algorithm is thus similar to the complexity of manually-crafted change propagation code written by application developers in MVC frameworks.

Interestingly, the MVVM-IVM algorithm is also superior to existing algorithms when it comes to memory consumption. Since it does not need to perform comparisons to figure out what has changed, it does not need to keep both the pre-state and the post-state of the model (something that happens for instance in the case of Angular). It suffices to keep only the post-state of a subset of the model (in particular, the collections that appear in for directives).

Algorithm 1: MVVM-IVM

```
1  function MVVMIVM(Template T, Model M, Set of Input Diffs D_in)
2      Set of output diffs D_out ← empty bag;
3      for top-level directive d ∈ T do
4          d.path ← path to d in template T
5          if d is <% print Path ê %> then
6              Δ ← IVMPath(ê, M, D_in);
7              D_out = D_out ∪ {prefixDiffs(d.path, {Δ})}
8          else if d is <% for Var y in Path ê %>B<% end for%> then
9              Δ^type(t̂; p) ← IVMPath(ê, M, D_in);
10             if t̂ = empty then
11                 if type = update then
12                     Let T' be the template rooted at d,
                       where ê is replaced by p;
13                     p' ← instantiateTemplate(T', M);
14                     D_out = D_out ∪ {Δ^update(d.path; p')};
15             else if t̂ = [k] and type = insert array then
16                 M' ← M#{y ← p};
17                 p' ← instantiateTemplate(B, M');
18                 D_out = D_out ∪ {Δ^insert_array(d.path[k]; p')};
19             else if t̂ = [k]ŝ then
20                 D'_in ← D_in ∪ {Δ^type(y ŝ; p)};
21                 D_nested = MVVMIVM (B, M, D'_in);
22                 remove first path step of each diff in
                   D_nested;
23                 D_out ← D_out ∪ prefixDiffs(d.path[k], D_nested);
24             D_nested = MVVMIVM (B, M, D_in);
25             if D_nested not an empty bag then
26                 l ← length of array e targeted by Path ê
27                 for i in [0 ... l] do
28                     D_out ← D_out ∪ prefixDiffs(d.path[i], D_nested);
29                 end
30     end
31     return D_out;
```

6. Experimental Evaluation

To evaluate our propagation algorithm, we compared it to Angular, Angular 2 and ReactJS in terms of performance. More specifically, we measured the time needed by each framework, to propagate a simple model mutation all the way to the view. All experiments were executed on a mobile device (LG G4-H811) running Android 6.0 (Marshmallow), with 1.8 GHz 64-bit Hexa-Core CPU (Chipset: Qualcomm Snapdragon 808) and 3 GiB of RAM. The application runs inside a native wrapper that internally uses Android's WebView [3] which is based on Google Chrome (version: 53.0.2785.124). Google Chrome's Remote Debugging tool was used to connect the mobile device to a desktop computer in

Algorithm 2: IVMPath

```
1  function IVMPath(Path Expression ê, Model M, Set of Input Diffs D_in)
2      for each Δ^type(t̂; p) ∈ D_in do
3          if t̂ is êŝ then
4              return Δ^type(ŝ; p)
5          else if ê is t̂ŝ then
6              if type = update then
7                  p' ← navigate(p, ŝ) ;
8                  return Δ^update(empty; p')
9              if type = delete then
10                 return Δ^delete(empty)
11             else
12                 p' ← navigate(M, ê) ;
13                 return Δ^update(empty; p')
14     end
15     return null;
```

order to enable us to collect the measurements presented in this section. Specifically, Google Chrome's profiling tools were used to retrieve measurements for each tested operation.

Performance of change propagation. To see how these frameworks perform with regard to change propagation, we used them to implement the product monitoring system of our running example. Since all competing frameworks operate strictly on the client, we used the part of our architecture that operates on the server-side (shown in Figure 4 on the left side) to instantiate the model and propagate it to the client. We then used the template language and API provided by each framework to generate the HTML table appearing in Figure 1.

After this step was completed we began modifying the product attributes in the underlying database tables which in turn triggered our IVM algorithm. The IVM algorithm generated a model diff for each mutation that was applied to the database tables. More specifically, we opted for modifying the User Rating field for each product in order to ensure a consistent diff size that also leads to a consistent series of steps on the client, causing the mutation of approximately equal portions of the application view. The model diff was eagerly transmitted to the client with the use of WebSockets. Upon retrieval, we applied the incoming diffs to the model of the application and we triggered the change propagation algorithm for each framework. Since all the steps prior to the invocation of the change propagation algorithm were common across frameworks, the respective time needed for their completion was also the same. What was not the same however, was the time required for each change propagation algorithm to locate the model mutation and call the appropriate renderer that mutates the view.

In order to explore how the change propagation algorithm is affected by the size of the model (and the view) of the application we first constructed a screen with a number of products x and we measured the time required for the change propagation algorithm and the renderer to execute. We then began increasing the total number of products appearing on the screen while still modifying the User Rating for each product. Figure 5 graphically depicts the time required by each framework to apply a mutation to the view. The X axis shows the number of products that appear in the view and the Y axis shows the time in milliseconds that was needed for the completion of each task, in a logarithmic scale. Each column corresponds to the total time needed by each framework to fully propagate the changes and it is broken down to the time required by the change propagation algorithm and the actual renderer to execute.

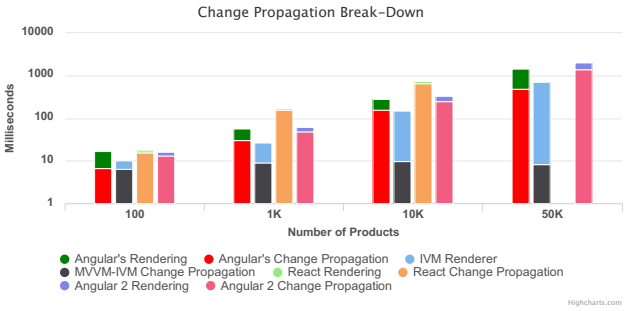


Figure 5. Experimental Evaluation

Experimental Evaluation. In Table 2 we provide detailed information about the time that was needed for each stage to terminate. As we observe, there are no measurements for ReactJS when the total number of products appearing in screen is equal to 50K products, the reason is that the ReactJS framework was unable to generate the view and the application became unresponsive for an extend amount of time, which forbid us from collecting the respective measurements. As we observe both from Figure 5 and the table 2 the rendering stage performs fairly similar for all frameworks, the slight edge that Angular2 and ReactJS appear to have in rendering performance is appointed to the fact that they invoke slightly more precise renderers that update smaller parts of the view. What is more interesting however is the fact that the MVVM-IVM's propagation algorithm is consistently more efficient than the respective propagation algorithms offered by the other frameworks. One of the biggest advantages of the MVVM-IVM algorithm is that it is not affected by the total time of products that appear in the view (or the model from which the view was derived). Additionally the fact that all the competing propagation algorithms perform similarly or much worse than the respective rendering step, proves that they introduce a significant overhead, since rendering is a notoriously expensive computation when the view consists of HTML content.

Discussion. It is worth mentioning that 30 fps is the minimum frame-rate that needs to be supported by an application in order to ensure a seamless user experience. What that means is that, each stage involved in mutating the view should be completed in about 33.3 ms, anything more than that results in a deteriorated user experience. Despite the fact that propagation algorithms appear to be independent of the rendering process, that is not the case. In fact any operation that takes more than 33.3 ms to complete can negatively affect the user experience due to the single-threaded nature of JavaScript. For that reason the minimal overhead that is added by MVVM-IVM is desired because not only it ensures the immediate propagation of mutations to the view without delays but at the same time it does not interfere with the user experience of the application.

7. Related Work

We mentioned the most closely related work, namely MVVM frameworks, in the previous sections of the paper. In section 3, we describe the fundamental limitations that prohibit such frameworks from operating in an efficient manner on mobile devices. Additionally, in section 4.1 we describe the main focus of IVM techniques employed by database systems which is what powers our propagation algorithms. In this section, we briefly describe work that is more peripherally related to the concept of change propagation.

	MVVM IVM		AngularJS		Angular 2		ReactJS	
Products	Change Propagation	Rendering	Change Propagation	Rendering	Change Propagation	Rendering	Change Propagation	Rendering
100	6.49	3.588	6.61	9.79	13.044	3.118	15.306	2.492
1K	9.052	17.746	29.736	28.05	46.972	15.088	154.542	14.238
10K	9.792	137.462	152.568	124.124	247.956	85.156	646.682	79.078
50K	8.214	696.334	493.576	961.644	1390	607.826	-	-

Table 2. Change propagation breakdown for each framework

This work has been conducted by researchers mainly in the fields of programming languages and algorithms.

In the algorithms community, researchers have worked on designing dynamic algorithms that are capable of efficiently updating their output given dynamic changes on their input. Surveys that have been conducted in this area [10] describe numerous approaches that have offered significant speedups in functions that resolve individual computation problems, especially when such functions operate on big data. These surveys also describe the significant effort that is usually associated with the development and implementation of such algorithms. Some of these algorithms took years of research to be developed and they are mostly oriented towards solving expensive domain specific problems (such as problems in computational geometry), while many such problems remain open. Since these algorithms were carefully designed for solving individual computational problems they cannot be easily extended in order to solve a bigger space of problems using the same techniques. Therefore, they are not compatible with performing change propagation in applications, mainly due to the diversity that such applications exhibit.

In the programming languages community, researchers have developed techniques that achieve automatic incrementalization of programs. The main focus of these techniques [19] is the automatic translation of conventional programs into respective programs that can respond to dynamically changed data. Since these techniques provide tools or compiler techniques that automatically perform this translation, they manage to minimize the effort required for the implementation of such functions. Recent work on self adjusting computations also includes the use of specially designed high level languages (or the extension of existing languages with annotations [9, 14]) used to express incremental computations that when combined with specifically developed compilers, can generate executables capable of efficiently handling mutations of input data.

A common denominator of the majority of these techniques is the fact that the underlying languages utilize a strong type system that enables the automatic distinction, (or in many cases the explicit manual distinction) between mutable and immutable input data. Leaving aside the fact that applications developers cannot necessarily predict which input data will be modified during the lifetime of an application, the fact that JavaScript is an untyped language can also be an obstacle, in using such techniques. It is unclear if these techniques could be used without forcefully introducing a type-system that application developers would have to adapt to. While, such a type-system might potentially assist in using some of these techniques to power incrementally maintained web applications, it would also steepen the learning curve of the respective framework since developers would have to familiarize themselves with the new type system. Lastly, it is unclear how such techniques can be used in a distributed architecture like the one described in Section 4

8. Conclusion and Future Work

We have shown that change propagation in current MVVM frameworks is in general inefficient both in terms of complexity and memory consumption. The reason is that they do not track the changes that happen to the model and thus spend significant effort (in terms of comparisons) to identify which parts of the model have changed. To address this problem, we have developed a novel and efficient change propagation algorithm, which works in the common case where the model of the application is retrieved by querying a database. In such cases, the algorithm can receive the model changes directly from the database and can avoid the comparisons of existing frameworks. The novel algorithm brings significant benefits in both complexity and memory consumption over existing approaches.

Having developed the foundations of this new change propagation approach, we will be working in the future on extensions that include among others support for more complex template languages (with additional constructs, such as if-then-else statements) and more complex views (containing in addition to HTML code also JavaScript visualization components, such as Google Maps [1] and Highcharts [16]). Particular interesting would be components that utilize the HTML5 Canvas instead of plain DOM elements for their visualizations due to the fast rendering time that it enables

Acknowledgments

We thank Panagiotis Vekris for his detailed comments on earlier versions of this paper.

References

[1] Google maps. `https://developers.google.com/maps/web/`.

[2] Websocket. `https://en.wikipedia.org/wiki/WebSocket`, .

[3] Webview. `http://tinyurl.com/pguw3sk`, .

[4] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5 (10):968–979, 2012.

[5] AngularJS. `https://angularjs.org/`.

[6] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

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

[8] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991. URL `http://ilpubs.stanford.edu:8090/8/`.

[9] Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. *ACM SIGPLAN Notices*, 47(6):299–310, 2012.

[10] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[11] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[12] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 1995.

[13] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166. ACM Press, 1993.

[14] M. A. Hammer, U. A. Acar, and Y. Chen. Ceal: a c-based language for self-adjusting computation. In *ACM Sigplan Notices*, volume 44, pages 25–37. ACM, 2009.

[15] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing ids to accelerate incremental view maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1985–2000. ACM, 2015.

[16] J. Kuan. *Learning Highcharts*. Packt Publishing Ltd, 2012.

[17] MithrilJS. `https://lhorie.github.io/mithril/index.html`.

[18] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[19] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510. ACM, 1993.

[20] ReactJS. `https://facebook.github.io/react/`.