ORM Optimization through Automatic Prefetching in WebDSL

Christoffer Marinus Gersen

ORM Optimization through Automatic Prefetching in WebDSL

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Christoffer Marinus Gersen
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

ORM Optimization through Automatic Prefetching in WebDSL

Author: Christoffer Marinus Gersen
Student id: 1382845
Email: chris.theblackshadow@gmail.com

Abstract

Object-Relational Mapping (ORM) frameworks can be used to fetch entities from a relational database. The entities that are referenced through properties are normally not fetched initially, instead they are fetched automatically by the ORM framework, when they are used by the application. This is called lazy-fetching and can result in many queries, causing overhead. The number of queries can be reduced by prefetching multiple entities at once. There are two types of prefetching techniques, static and dynamic. Static techniques perform optimization during compilation and dynamic techniques collect information during runtime in order to perform prefetching. Multiple static prefetching techniques are implemented into WebDSL that all use the same static code analysis, however, they generate different queries. The static analysis determines the entities that are going to be used and should be prefetched. These static techniques are compared to the dynamic techniques already present inside the Hibernate ORM framework. The evaluation is performed using the OO7 benchmark and complete WebDSL applications. The results of the OO7 benchmark show a response time improvement of up to 69% over lazy-fetching. On complete web applications some of the static techniques implemented in WebDSL improve the performance on average, however, the performance may be improved further, using a more fine-grained method of choosing an optimization technique.

Thesis Committee:

Chair:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Daily supervisor:	D.M. Groenewegen MSc., Faculty EEMCS, TU Delft
Committee Member:	Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. Hidders, Faculty EEMCS, TU Delft

Preface

I would like to thank both Danny Groenewegen and Eelco Visser for their assistance with my thesis project, by giving me ideas and feedback. I am also grateful for their help with writing a paper on the same subject as this thesis project. I would also like to thank the thesis committee for taking the time to read and assess my thesis project. Next I would like to thank my brother for checking my spelling and grammar. Finally I would like to thank my parents, for their support and for giving me the opportunity to study.

Christoffer Marinus Gersen
Spijkenisse, the Netherlands
August 12, 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of web applications on the Web keeps increasing. Examples of web applications include wikis like Wikipedia, webshops like Amazon and social networking services like Facebook. Web applications can also replace traditional desktop applications, for example, Gmail and Hotmail are web applications that implement an e-mail client, like Mozilla Thunderbird or Microsoft Outlook. The main advantage of using web applications over desktop applications is that they run on a web server and do not need to be installed or updated on individual client computers. The client computer only needs to have a compatible webbrowser, like Mozilla Firefox or Google Chrome, which can be used to access the web application on the web server.

Web applications generate responses for requests that are send to the web server, often using and storing data in a relational database. Data that is or will be stored inside the database is also called persistent data, because that data persists after a request has been fulfilled and even if the web server is restarted. Other data, like generated responses or intermediate values, are discarded after fulfilling a request. Persistent data can be accessed from a relational database directly, by executing a query that will return a recordset containing the requested data. When working in an object oriented programming language, persistent data is often accessed through an Object-Relational Mapping (ORM) framework. The ORM framework will send the queries to the database and expose the persistent data through persistent objects, also known as entities. Entities are the same as regular objects, except that they represent persistent data. Using objects instead of recordsets is preferred in an object oriented language, because they can use all the features of the language, like inheritance or dynamic dispatch.

Recordsets can also be turned into objects manually, however, an ORM framework does more than just that. For example, an ORM framework can write changes made to entities back to the database automatically. Another advantage of an ORM framework is that it will automatically generate SQL queries, which can often also be generated in different vendor specific SQL dialects, making a web application easily portable to another database.

An entity can have properties that contain a reference to another entity or even a collection of entities. Loading an entity or collection from the database is called fetching and an ORM framework can have configuration options to specify when and how an entity or collection property is to be fetched. By default a property should be fetched when it is used,

by executing a query that fetches just that entity or collection. Not fetching an entity or collection until it is used is called lazy-fetching and prevents the fetching of entities and collections that are never used. The downside is that properties are accessed one by one and each access could execute its own query to fetch the referenced entity or collection. An ORM generally caches entities it has fetched before, to prevent fetching the same entity again. However, fetching only one entity or collection per query will still lead to an excessive number of queries. Sending too many queries is bad for performance, because every query is a round trip to the database and causes a little overhead. This problem is called the $1+n$ query problem, because accessing a property inside a loop over a collection of n elements, will execute 1 query for the collection and 1 query during each iteration to fetch the property.

The $1+n$ query problem can be solved by prefetching properties that are going to be accessed, using fewer queries. There are multiple methods that can be used to do so. For example, when fetching an entity, a query could be generated that uses joins to prefetch some of its properties. Without joins it is also possible to fetch more than one entity at a time, by using a different condition, which can be as simple as matching a list of entity identifiers instead of only one. These prefetch queries can be specified manually, however, manual queries are hard to maintain, because they also need to be updated manually every time the data requirement of the application changes. Therefore, this thesis project implements and evaluates multiple techniques, to generate and execute these prefetch queries automatically.

By using an ORM framework it is possible to completely remove the need to write queries manually, however, most ORM frameworks still offer interfaces through which custom queries can be specified and executed. These interfaces allow a developer to request specific persistent data, which the ORM framework will translate into a SQL query, allowing for efficient execution by the relational database. This is useful when searching for entities. For example, to get all the recent posts of a specific user on a forum. This can be implemented without custom queries, by accessing a property on the user that stores all of its posts and then checking one by one if they are recent. Accessing such property will fetch all post and not just the recent ones. Alternatively, a custom query can be used to ask the database for just the posts of the user that are recent, without fetching the older posts of the user. The implementation using a custom query is more efficient, because less data is returned from the database. This thesis project implements a static code analysis that is able to detect these kinds of conditions on collections. The detected conditions are added to the queries that fetch these collections whenever possible. This will have the same performance benefits of using custom queries, without the developer having to specify or maintain them. Custom queries can also be used to perform aggregation operations, like summing up all salaries of all employees of a specific department. This kind of operation is harder to detect using a static code analysis and is still left for the developer to optimize.

1.1 Goal and Research Questions

Websites that use an ORM framework can often be optimized by prefetching entities and collections that are used in the future. Prefetching usually fetches more than one entity or

collection using a single query, which avoids having to fetch them using individual queries. The goal of this project is to add automatic prefetching to WebDSL, which uses Hibernate as ORM framework. WebDSL is a domain specific language that targets web applications. Prefetching is added to WebDSL by changing the WebDSL compiler to generate extra code that performs the prefetching. To generate efficient prefetching code, the compiler can use a static analysis on the abstract syntax tree. The static analysis determines which properties are accessed in the future and should be prefetched. The static analysis also detects the conditions under which the elements of a collection are used. These conditions are then added to queries that fetch these collections and allow these collections to be initialized partially, whenever possible. The research questions are defined as follows.

How to best prefetch data under various circumstances?
As discussed in the introduction, there are two ways to fetch multiple entities or collections using a single query. By using joins or by using a less restrictive condition, for example, by matching a list of identifiers instead of just one. Both methods can be used to prefetch entities or collections that are going to be used, yet they perform differently. To answer this question we look at both types of queries and their resulting recordsets, to determine when it is beneficial to use one over the other.

When do the prefetching techniques included in Hibernate improve performance?
Hibernate includes its own prefetching techniques, which can easily be enabled inside its configuration. These techniques can be enabled selectively for individual classes and properties, however, since this thesis project strives for automatic optimization, they are enabled for all classes and properties. Response time is the most important measurement for the evaluation of the performance. Factors that have an influence on the response time are also measured, which includes the number of queries executed, the number of fetched entities and memory consumption.

How do the prefetching techniques implemented in WebDSL compare to the techniques provided by Hibernate?
To answer this question different prefetching techniques will be implemented in WebDSL. The implemented techniques will all use the same static analysis to predict future property accesses. The techniques will perform prefetching at different locations and use different queries, for example, by allowing joins or not. The techniques that are implemented in WebDSL are evaluated in the same way as the techniques from Hibernate and a comparison is made.

1.2 Outline

This section describes the structure of this thesis. Chapter 2 provides a brief introduction into WebDSL. The different methods to perform prefetching and their benefits are discussed in Chapter 3. Additions to the WebDSL syntax to define prefetch specifications are also explained in Chapter 3. Chapter 4 continues by introducing the prefetching tech-

niques that will later be evaluated. Chapter 5 explains how the performance of the prefetching techniques is evaluated and also presents improvements to the prefetching techniques, which were implemented after initial testing. The static analysis that has been implemented in WebDSL, to automatically generate prefetch specifications, is described in Chapter 6. Chapter 7 provides a brief introduction into the OO7 benchmark and uses this benchmark to evaluate the prefetching techniques. Chapter 8 evaluates the prefetching techniques using complete WebDSL applications. Related work will then be discussed in Chapter 9. To conclude, Chapter 10 answers the research questions and discusses possible future work.

Chapter 2

WebDSL

WebDSL [18] is a domain-specific language for developing dynamic web applications with a rich data model. The WebDSL language is a combination of multiple sub-languages, that each provide abstractions for common tasks, making it possible to perform those tasks using fewer lines of code. Ultimately making development easier and more efficient. For example, WebDSL includes sub-languages for defining the data model, pages and access control rules.

WebDSL also includes previously existing languages, like XHTML and the Hibernate Query Language (HQL), which can also be found in web applications that are developed in other languages, like Java. However, in other languages these sub-languages are often contained inside strings and are not checked for errors during compilation. Any errors may go undetected until unexpected behavior is noticed. Including these existing languages inside WebDSL allows them to be checked for errors during compilation. Their inclusion also allows them to be used as regular WebDSL expressions and statements, without having to place them inside strings or otherwise separate them from WebDSL code.

The WebDSL language is implemented by a compiler that transforms a WebDSL application into a Java Servlet, which can be deployed to a web server, like Apache Tomcat. Entity definitions from the data model and page definitions are all translated into Java classes. Generated entity classes are configured so that they can be persisted using the Hibernate framework. A Java Servlet class is also generated, that dispatches requests to the corresponding generated page classes. The WebDSL compiler is implemented using Syntax Definition Formalism (SDF) [16] and the transformation language Stratego [17], however, these languages will not be described in further detail.

2.1 Data Model

The data model in a WebDSL application is defined by a set of entity definitions. An entity definition has a name, a set of properties and a set of functions. Figure 2.1 shows an example of the entity definitions `User` and `Message` . Each property has a name and a type. A property type can be of a primitive type, like `Int` , `Bool` , `String` , or a domain-specific variation of a `String` . For example, the `Secret` type is a `String` type, which is mostly

used for passwords, because its input and output fields on pages show asterisks for its value. Similarly a `WikiText` type is a `String` that contains markup instructions, which can be converted into HTML when the property is shown on a page. Properties of these primitive types can be recognized by the `::` between the property name and the type.

Properties can also contain a reference to another entity, or a collection of entities. These properties are recognized by an `->` between the property name and the type. The type can be an entity-type, or a collection that contains elements of an entity-type. A collection type is a `List` or `Set`, with its element type appended between < and > characters, so a `List` with elements of type `Entity` becomes `List<Entity>`.

A property can also have annotations, which are added between parentheses behind the property type. These can be used to define validation (constraints), or to define an inverse property, which is a property, of the referenced entity, that contains a reference back. This creates a relation between two entities that can be accessed from both sides. An example is a `previous` and a `next` property, where accessing the `previous` property and then the `next` property on an entity, or the other way around, will return the same starting entity. Another example is shown in Figure 2.1 on lines 16 to 21, where entities are constructed and their properties are given a value. After constructing the `Message` entity, the `inbox` property of the recipient will contain the new message automatically. Since the inverse annotation ensures that setting the `recipient` property of a message automatically adds that message to the inbox of the recipient.

From the entity definitions WebDSL generates Java classes. These classes contain annotations of Hibernate and the Java Persistence API (JPA), so that they can be persisted using Hibernate[13]. The generated classes also contain getters and setters for properties and for inverse properties these setters will include code that updates the other side of the relation. Inverse properties will also create foreign keys inside the database to enforce these kinds of relations. The functions that are defined on an entity are directly translated into Java methods, so they behave mostly the same, however, function overloading is handled within the WebDSL compiler and behaves slightly different. Static functions can also be defined on entities and are translated to static Java methods. Global functions are functions defined outside of entities and are also translated to static Java methods.

```
1  entity User {
2    name :: String
3    password :: Password
4    inbox -> List<Message> (inverse=Message.recipient)
5  }
6  entity Message {
7    subject :: String
8    body :: WikiText
9    sender -> User
10   recipient -> User
11   function getSize() : Int {
12     return subject.length + body.length;
13   }
14 }
15
16 var sender : User := User{ name := "User1" password := "pass" };
17 var recipient : User := User{ name := "User2" password := "pwrd" };
18 var m : Message := Message { subject   := ""
19                              body      := ""
20                              sender    := sender
21                              recipient := recipient };
```

Figure 2.1: An example of the entity definitions named `User` and `Message`.

2.2 Pages

A page in WebDSL is directly accessible through a URL, so links can point to them. The home page of the application is called `root` and does not take arguments. Other pages can have arguments of primitive types and entity types. Collections are not allowed as page arguments. Figure 2.2 shows an example of a page that displays a link to the `inbox` page of ”User1”. The for-loop that is shown iterates over all `User` entities and all users are fetched from the database. However, the body of the for-loop is only executed for one user that has the name ”User1”, because of the `where` and `limit` clauses. The `navigate` keyword takes a call to a page and generates a link to that page, on top of the following group of elements. In this case that group contains only one call to the built-in `output` template, which writes a value onto the page. The `inbox` page calls the `showMessages` template, which is defined in the next example to explain templates.

```
1  page root() {
2    for(u : User where u.name == "User1" limit 1) {
3      navigate(inbox(u)) { output(u.name) }
4    }
5  }
6  page inbox(user : User) {
7    showMessages(user.inbox)
8  }
```

Figure 2.2: An example page showing a link to the `inbox` page of a `User`.

```
1  template showMessages(messages : List<Message>)
2    table {
3      row {
4        column { "Sender"  }
5        column { "Subject" }
6      }
7      for(m : Message in messages) {
8        row {
9          column { output(m.sender.name) }
10         column { output(m.subject)     }
11       }
12     }
13   }
14 }
15 template table(){
16   <table> elements() </table>
17 }
```

Figure 2.3: An example template showing the `inbox` of a `User`.

2.3 Templates

Pages can be made reusable by declaring them as templates. Templates are not accessible through a URL like pages, however, they can be called from pages and other templates. Templates can even take collections as arguments, which is shown by the example in figure 2.3. The example calls the built-in `table`, `row` and `column` templates. These templates take the body of a template as argument. At the bottom of the example a simplified definition of the `table` template is shown, where the provided template body is called by the call to `elements`, so that the provided template body is placed inside a `table` XHTML-tag.

2.3.1 Dynamic Scoping of Template Definitions

Another important feature of templates is the ability to dynamically redefine a template definition within the current scope. In figure 2.4 there is a definition of a `main` template. It is common for WebDSL applications to call a `main` template from every page and redefine templates as needed. For example, the `showUser` page calls the `main` template and

```
1  template main() {
2    header
3    body
4    footer
5  }
6  template header() { "Header" }
7  template body()   { "Default body" }
8  template footer() { "Footer" }
9
10 page showUser(user : User) {
11   main()
12   template body() {
13     output(user.name)
14   }
15 }
```

Figure 2.4: An example of a scoped dynamic template redefinition, where `body` is redefined within the scope of `showUser` .

```
1 template main() requires body() {
2   header body footer
3 }
4 page showUser(user : User) {
5   main() with {
6     body() {
7       output(user.name)
8 } } }
```

Figure 2.5: A required template example.

redefines the `body` template. A redefinition stays in effect until the template or page that performed the redefinition is finished, binding the redefinition to the scope of that template or page. A template can also redefine a template that has already been redefined. When such a redefinition goes out of scope, then the previous redefinition will be restored. A redefined template can also use variables and arguments from the template or page that performed the redefinition.

2.3.2 Required Templates

Templates can require its callers to define a set of templates. A template defines the templates it requires using the `requires` keyword, followed by the required template signatures. A template signature is just a template name, with argument types as argument. A caller must define all the required templates when calling a template, using the `with` keyword. Take for example Figure 2.5, where the `main` template is similar to that of Figure 2.4, except that it now requires the definition of a `body` template. As a result every call to `main` must have a `with` block that defines the `body` template for that call.

Chapter 3

Prefetch Considerations

Lazy properties are normally fetched when they are first used, using a separate query for each property. When a property is accessed inside a loop, then this causes the $1+n$ query problem (Chapter 1). One solution is to use joins inside the query, to prefetch properties that are used within the loop. This is called join-fetching and in some cases it can improve the performance. Join-fetching can also degrade the performance, because some data may be fetched multiple times. This is most noticeable when prefetching collection properties, as is explained in Section 3.1, and can be avoided by using batch-fetching. Section 3.2 explains that batch-fetching can also be used to avoid prefetching duplicate entities for non-collection properties. Section 3.3 describes additions to the WebDSL language that allow prefetching to be defined within the code.

3.1 Prefetching Collections Properties

A collection property of an entity can be prefetched, by adding a join to the query that fetches the entity. Prefetching collection properties using joins can cause Cartesian products inside the recordset. Cartesian products make join-fetching inefficient, because this causes unnecessary data duplication inside the recordset. For example, if there are `Person`, `Color`, `Cat` and `Dog` entities and each person has a favorite color and a set of cats and dogs. Fetching all person entities with their favorite color, cats and dogs join-fetched, results in the recordset shown in Table 3.1. In this example recordset every person is returned four times, because there is a Cartesian product between the cats and dogs of a person. Non-collection joins, like the favorite color, are returned as many times as a person. A Cartesian product will become a large influence on the performance of a query when the joined collections are large. The average size of a collection is hard to predict and can change over the lifetime of an application. Therefore, Cartesian products should always be avoided.

Cartesian products can be avoided by joining no more than one collection property. All other collection properties need to be fetched by additional queries. In the example, the Cartesian product can be avoided by executing two queries, returning the two recordsets shown in Table 3.2. Person entities and non-collection joins are still returned multiple times. These duplicates should also be avoided, because the data duplication per element

Person		Color		Cat		Dog	
Id	Name	Id	Name	Id	Name	Id	Name
1	Peter	1	Red	1	Oscar	1	Max
1	Peter	1	Red	1	Oscar	2	Lucky
1	Peter	1	Red	2	Felix	1	Max
1	Peter	1	Red	2	Felix	2	Lucky
2	Alice	2	Blue	3	Rosey	3	Charlie
2	Alice	2	Blue	3	Rosey	4	Sammy
2	Alice	2	Blue	4	Holly	3	Charlie
2	Alice	2	Blue	4	Holly	4	Sammy

Table 3.1: The recordset when all properties are join-fetched.

Person		Color		Cat	
Id	Name	Id	Name	Id	Name
1	Peter	1	Red	1	Oscar
1	Peter	1	Red	2	Felix
2	Alice	2	Blue	3	Rosey
2	Alice	2	Blue	4	Holly

Person	Dog	
Id	Id	Name
1	1	Max
1	2	Lucky
2	3	Charlie
2	4	Sammy

Table 3.2: The recordsets when only one collection is join-fetched.

and the number of elements in the joined collection can both be large. The amount of data duplication per element is especially large when there are many non-collection joins or long text fields. A separate query can be used for every collection property, to avoid these duplicates, which will only cost one more extra query.

The second recordset in Table 3.2 contains two dog sets. Both sets can be fetched by the same query, because they both have the same collection-role, meaning that they both represent the same property, for different instances of the same entity-type. This means that both collections are normally fetched by similar queries, where only the `Person` identifier inside the where-clause is different. To fetch both collections using the same query, only the where-clause of the query has to be changed, in order to allow for multiple `Person` identifiers. Using one query to fetch multiple collections with the same role is called batch-fetching and uses fewer queries than lazy-fetching.

A batch for batch-fetching can be generated automatically in various ways, however, it is often possible to generate a batch from a collection. For example, when prefetching a property for a loop, then the collection that the loop iterates over can be used to generate a batch, by taking all elements for which the property is uninitialized. In the previous example, the first query fetched a collection of `Person` entities. This collection is then used as a batch to prefetch the associated dog collections, which ultimately results in the second recordset of Table 3.2. From now on this method of batch-fetching is called guided batch-fetching, to distinguish it from methods that use different batch generation techniques.

3.2 Prefetching Non-Collections Properties

Prefetching non-collection properties using join-fetching will never increase the number of returned rows, like it does for collection properties that contain more than one element. However, there is still a risk of fetching duplicate entities. In the example from Section 3.1 both `Person` entities could have had the same favorite color, for example. The colors could also have been fetched earlier for another purpose and join-fetching them again will also cause duplicate entities. Avoiding these duplicates can increase performance, especially when there are many duplicates or when the duplicates are large.

Duplicate fetches for non-collection properties can also be avoided by using guided batch-fetching instead of join-fetching. This avoids fetching duplicate entities, because guided batch-fetching first checks for which entities the property is uninitialized. For uninitialized properties it remembers the identifier of the entity that should be fetched. After batch generation, guided batch-fetching executes one query, using the remembered identifiers, if there are any. This initializes all uninitialized properties and if two properties had the same entity as a value, then that entity is fetched only once[1].

Guided batch-fetching also allows for more flexibility in the placement of optimization code than join-fetching. Join-fetching should not be used after an entity is fetched, because fetching an entity again, just to join-fetch its properties, is inefficient. Guided batch-fetching can perform prefetching after entities have been fetched. Take for example the code in Figure 3.1, which shows donations to charities made in a given year. Guided batch-fetching can fetch all required entities and collections with three simple queries. The first one fetches all `Charity` entities, the second one fetches all `donations` collections and the last query fetches all `madeBy` properties. For join-fetching the `donations` and `madeBy` properties could be joined on the query fetching all `Charity` entities. The resulting query will probably fetch duplicate `Charity` entities, because of the collection join. The query can also fetch duplicate entities for the `madeBy` property, when the property has the same value for two donations. Not placing the joins on the query fetching `Charity` entities will increase the number of queries.

The disadvantage of guided batch-fetching is that it does not work for single entities, because in that case it uses at least as many queries as lazy-fetching. Take for example an entity that represents a node in a tree, where the `parent` property is the parent node. To find the root of the tree the `parent` property can be accessed recursively. In this case join-fetching can prefetch the next n `parent` properties using a single query, by selecting the first one and joining the next $n-1$ `parent` properties. This allows join-fetching to reduce the number of queries with a factor of n, while batch-fetching does not reduce the number of queries, because the identifiers of multiple parent nodes need to be known in order to prefetch them all at once, however, only one is known at a time.

[1] Guided batch-fetching cannot avoid fetching duplicates for the inverse side of bidirectional one-to-one properties, because the identifier of their associated entity is unknown if uninitialized. The absence of these identifiers is explained in subsection A.2.3

```
1  page showDonations(year : Int) {
2    for(charity : Charity) {
3      section {
4        header{ output(charity.name) }
5        showDonationsOfCharity(charity, year)
6      }
7    }
8  }
9  template showDonationsOfCharity(charity : Charity, year : Int) {
10   list {
11     for(d : Donation in charity.donations where d.year == year) {
12       listitem {
13         output(d.value)
14         if(!d.anonymous) {
15           "(" output(d.madeBy.name) ")"
16         }
17       }
18     }
19   }
20 }
```

Figure 3.1: A page showing donations to charities.

```
1  page showDonations(year : Int) {
2    for(charity : Charity) {
3      prefetch-for charity {
4        donations {
5          madeBy
6        }
7      }
8      section {
9        header{ output(charity.name) }
10       showDonationsOfCharity(charity, year)
11     }
12   }
13 }
```

Figure 3.2: A manual prefetch specification for the code in figure 3.1

3.3 Manual Prefetch Specification

New syntax has been added to the WebDSL language to define prefetch specifications. A prefetch specification specifies the properties that are accessed on a specific variable, which are also the properties that need to be prefetched. The new syntax allows prefetch specifications to be defined manually. The syntax also allows automatically generated prefetch specifications to be displayed. Take for example Figure 3.1, where the for-loop in the `showDonations` page can prefetch all the used properties, by adding the prefetch specification shown in Figure 3.2.

The prefetch specifications inside Figure 3.2 shows that properties can be nested to form a tree like structure, which is useful for both join-fetching and batch-fetching. For batch-fetching the tree structure is useful, because parent nodes need to be fetched before batches can be generated for their children. For example, the `donations` properties are prefetched before `madeBy` properties, because the `Donation` entities are used by guided batch-fetching to determine which `madeBy` properties are uninitialized. For join-fetching, the nesting is also useful, because child properties need to be joined on their parent property. For the example, this means that the `donations` property needs to be joined on the root entity of the query, which are the `Charity` entities, and that the `madeBy` property is joined on the `donations` property. Prefetch specifications can be defined for for-loop iterator variables and for the arguments of pages, templates and functions.

The provided prefetch specification fetches all `Donation` entities, while only the ones that are of the specified year are used. The `donations` collection can be prefetched partially, so that only the used elements are fetched. This can be specified by adding a where-clause on a collection, like is shown in Figure 3.3. The conditions inside the where-clause can only access properties of a primitive type, which are always defined directly on the elements of the collection. This limitation is imposed on the conditions to prevent joins inside queries. Variables of a primitive type are also allowed, however, the variables must be declared before prefetching and their values should not change after prefetching, because that changes the condition and the elements that are required. Prefetch specifications for for-loop iterator variables can also define a where-clause on the root variable. The where-clause on the root variable of a prefetch specification can describe the same conditions as the where-clause of the for-loop. The difference between both where-clauses, is that the where-clause of the for-loop can contain any valid expression, because it is never translated into a query and is always evaluated after the collection has been fetched.

Figure 3.3 also shows an if-clause on the `madeBy` property, which may be defined on any property inside the prefetch specification, even in conjunction with a where-clause. The if-clause specifies a condition under which the property should be fetched. The same limitations are opposed on an if-clause as on a where-clause, except that properties accessed within the condition should be defined on the parent property instead. This limitation is no longer imposed to prevent joins, instead it prevents lazy-fetching of properties used by the condition. The parent property has already been prefetched by guided batch-fetching, therefore, accessing primitive properties on those entities will not result in a lazy-fetch query. Properties referencing other entities may not have been fetched yet and are not allowed. In the example, the `madeBy` properties are only prefetched for donations that are not anonymous.

Currently only one prefetch specification has been added, however, the specification in Figure 3.4 can also be added to the for-loop in the `showDonationsOfCharity` template. Everything in this specification has already been prefetched if the template is called from the `showDonations` page. The prefetch specification should still be added, because the template may also have been called by another page or template that did not perform prefetching. Because it is inexpensive to check if all properties have been fetched, the addition of the prefetch specification will barely be noticeable in the performance of the `showDonations` page.

```
1 prefetch-for charity {
2   donations where(.year==year) {
3     madeBy if(!.anonymous)
4   }
5 }
```

Figure 3.3: Adding partial initialization of collections to a prefetch specification.

```
1 prefetch-for donations where (.year==year) {
2   madeBy if(!.anonymous)
3 }
```

Figure 3.4: Adding prefetch specifications.

There are still a number of syntax additions that have not been explained yet. These additions are more complicated and their explanation is left for Chapter 6, where their analysis is also explained.

Chapter 4

Prefetch Techniques

This chapter describes different techniques to prefetch multiple entities and collections at once, using only a single query. This will reduce the total number of queries, resulting in increased performance when the number of fetched entities does not increase. Some of these techniques use knowledge about the properties that will be used by the application in the future, which is determined by a static analysis that is explained in Chapter 6.

4.1 Hibernate Batch-fetching

Hibernate batch-fetching is a feature of Hibernate, which can be used to reduce the $1+n$ query problem (Chapter 1). Instead of using a query to fetch just a single entity or collection, Hibernate will try to prefetch other instances of the same entity-type or collection-role, using that same query. This technique is very similar to guided batch-fetching from Chapter 3, except that it uses a different method to generate batches.

To generate batches, Hibernate maintains a list of uninitialized entities and collections. Entities and collections get added to this list as soon as their uninitialized proxies are created. A proxy is a wrapper for an entity or collection to implement lazy-fetching, because a proxy does not fetch the entity or collection it represents until it is used. Proxies are created for lazy properties when an entity is fetched. This means that the order of the maintained list depends on the order in which entities and collections are fetched from the database. When a single entity or collection is requested from the database, Hibernate will generate a batch that includes the requested entity or collection. Additional entities or collections of the correct type or role are added to the batch, from the list of uninitialized proxies, up to a configured maximum batch size. If the requested entity or collection occurs within the list of uninitialized proxies, then Hibernate first adds proxies that follow it. The entities or collections of those proxies are likely to be requested next, because they probably represent properties of entities that were fetched around the same time.

Take for example the code in Figure 4.1, which shows a list of all `Item` entities, with their corresponding `owner`. With lazy-fetching $1+n$ queries are used, one to fetch all `Item` entities and then one per iteration to fetch the corresponding `owner`. Since the `Person` proxies are created in the order in which they are used, Hibernate batch-fetching will fetch

```
1  entity Item {
2    name  :: String
3    owner -> Person
4  }
5  entity Person {
6    name  :: String
7  }
8  page root() {
9    list {
10     for(item : Item) {
11       list-item {
12         output(item.name) " - " output(item.owner.name)
13       }
14     }
15   }
16 }
```

Figure 4.1: Batch-fetching the `owner` property.

the `owner` properties for the next b iterations using a single query, where b is the configured batch size. This means that Hibernate batch-fetching needs only $1 + \frac{n}{b}$ queries.

However, in more complex cases Hibernate batch-fetching may fetch entities and collection that are never used. For example, when the `owner` property is used for only some `Item` entities, then the unused `owner` properties are still fetched, because of their position in the uninitialized entities list. Another example is when the `Item` entity has another property of type `Person` that is never used. This new property will also get fetched by Hibernate batch-fetching, because its values have the same entity-type as the `owner` property.

Hibernate batch-fetching is not suitable for every situation, because fetching unused entities and collection may cost more time than that is saved by reducing the number of queries. However, Hibernate batch-fetching is still a useful feature, because it can improve the performance of an application with a simple configuration change.

4.2 Hibernate Subselect-fetching

Subselect-fetching is another feature of Hibernate that can be enabled for collection roles. When such a collection property is accessed on an entity, then the query that fetched the entity is used as a subquery, to prefetch the collection property for all other entities that were fetched using that query. The advantage over Hibernate batch-fetching is that all collection properties are fetched by one query, instead of being limited by a maximum batch size.

A limitation of subselect-fetching is that it cannot be enabled for single value properties. This means that after fetching a number of collections, every accessed single value property is fetched using a separate query, because lazy-fetching is used. This limitation can be overcome by combining this technique with Hibernate batch-fetching. When both subselect-fetching and Hibernate batch-fetching are enabled for a collection, then subselect-fetching is used when a subquery is available, otherwise Hibernate batch-fetching is used.

Just like with Hibernate batch-fetching, subselect-fetching can fetch more collections than necessary, because it prefetches a collection property for a set of entities, while the collection property may only be required for a number of elements. Another disadvantage is that the used subquery should be simple, because the database has to re-evaluate the query. When a subquery is too expensive, then the performance can even be reduced instead of increased. A query from Hibernate subselect-fetching can also be used as a subquery to fetch a collection property for the resulting elements. This causes subqueries to be nested, making it increasingly more important that the first query is simple, because it will be re-evaluated every time.

4.3 Join-fetching

Join-fetching is another way to fetch more than one entity using only one query and has already been discussed in Chapter 3. This prefetch technique uses the prefetch specifications for page, template and function arguments. For functions defined on entities the `this` keyword is also handled as if it was a function argument. Queries are placed at the beginning of pages, templates and functions, because they should be placed as early as possible, because at those locations there is the highest chance that entities have not been fetched yet. Even though this approach uses join-fetching, collection properties are never joined and always get their own query. Adding collection joins has been attempted and causes out-of-memory exceptions, because there were too many collection joins in a single query.

4.4 Guided Batch-fetching

For for-loops that iterate over a persistent collection, it is possible to prefetch the properties that are used inside the for-loop, using guided batch-fetching, which has been discussed in Chapter 3. Before the first iteration of the for-loop is performed, guided batch-fetching iterates over the same collection, generating batches for the properties that are specified in a prefetch specification.

Take for example the code in Figure 4.2, which shows a list of `Publication` entities. The example has a manually defined prefetch specification, telling which properties should be prefetched. Guided batch-fetching will iterate over the `pubs` collection to generate batches for the `authors` and `journal` properties. A property is only included in a batch when it has not been fetched before. After iterating over all `Publication` entities in the collection, both batches are complete and guided batch-fetching will execute queries to prefetch them, making sure that both properties are initialized for all elements in the collection. After prefetching both properties a batch will be generated for the `alias` property in the same way, by iterating over the elements of all `authors` collection properties.

Guided batch-fetching can use the same methods that Hibernate batch-fetching uses to generate and executes its queries. The methods used by Hibernate batch-fetching normally do not allow an application to specify their own batches for prefetching, however, Hibernate can be extended to allow it. Alternatively, queries can be generated using the provided HQL or Criteria APIs. However, both APIs do not offer a way to batch-fetch collection

```
1  template showPublications(pubs : List<Publication>) {
2    for(pub : Publication in pubs) {
3      prefetch-for pub {
4          authors {
5            alias
6          }
7          journal
8        }
9      }
10     section {
11       header{ output(pub.name) }
12       "Journal:" output(pub.journal.name)
13       break
14       "Authors:"
15       for(aut : Author in pub.authors) {
16         output(aut.alias.name)
17       } separated-by { "," }
18     }
19   }
20 }
```

Figure 4.2: Example for Guided batch-fetching.

properties, without join-fetching them on the entities they belong to, making those queries less suitable for batch-fetching. Another advantage of using the same methods as Hibernate batch-fetching is that it generates static queries when the application starts, for various batch sizes up to the configured maximum. Guided batch-fetching can be implemented in the same way, by splitting the larger generated batches into multiple smaller batches that fit the generated static queries. The advantage is that queries for guided batch-fetching will also have to be generated only once.

Instead of using static queries, a query can also be generated dynamically to fit the generated batch, which ensures that every batch requires only one query. Since query generation is fast and the number of required queries is reduced, this is likely to be more efficient. If queries are generated dynamically, then those queries can also contain joins, to prefetch properties for the entities being batch-fetched. However, this will increase the complexity of the queries and will also increase the risk of fetching duplicate entities. Allowing join-fetching in batch queries also makes this technique very similar to the one in section 4.3, because they mainly differ in code placement.

Chapter 5

Improving Prefetch Techniques

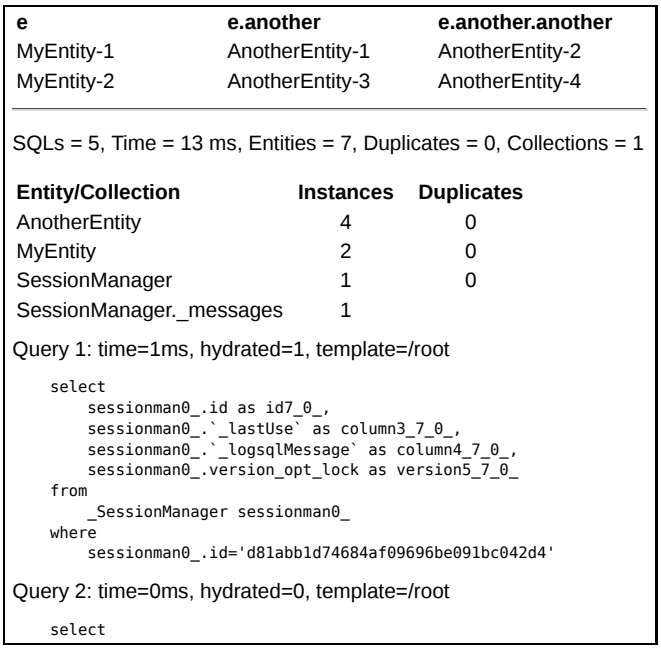
This chapter discusses subtle changes to the implementations of the prefetching techniques from Chapter 4. These changes address some of the shortcoming that were noticed during the implementation and initial testing of the prefetching techniques. The initial testing also lead to other optimizations that are unrelated to prefetching, which are therefore discussed in Appendix A. These other optimization are not irrelevant, because they do influence test results.

Section 5.1 explains the method that is used for measuring the performance of the prefetching techniques. Section 5.2 gives a short introduction of the page that was used for most of the initial testing and shows the results for that page. The following sections each discuss the preliminary results of a prefetching technique and how a technique is altered to overcome its initial shortcomings.

5.1 Measuring Performance of Prefetch Techniques

There are several performance indicators that are important when comparing prefetch techniques. The most important one is the response time of a request. The response time is related to the number of queries executed and the number of entities and collections returned. The same entity may be fetched more than once, which has a negative effect on the response time. Also, besides improving the response time, a prefetch technique should not use much more memory, which is also related to the number of entities and collections fetched.

All these performance indicators are measured by a script, which accepts a set of WebDSL applications and performs measurements for a configured page. These WebDSL applications are all the same, except that they are compiled using different prefetching techniques. The script measures the performance of the prefetching techniques individually, by deploying only one application to Tomcat at a time. Tomcat is then launched with an initial and maximum heap memory size of 4 GB. Setting the initial heap size reduces the chance that memory allocation interferes with the test results. The same is done for the PermGen memory, which has a capacity of 256 MB. The script also adds the `-Xloggc:<file>` and `-XX:+PrintGCDetails` Java arguments when launching Tomcat, so that Java logs detailed



e	e.another	e.another.another
MyEntity-1	AnotherEntity-1	AnotherEntity-2
MyEntity-2	AnotherEntity-3	AnotherEntity-4

SQLs = 5, Time = 13 ms, Entities = 7, Duplicates = 0, Collections = 1

Entity/Collection	Instances	Duplicates
AnotherEntity	4	0
MyEntity	2	0
SessionManager	1	0
SessionManager._messages	1	

Query 1: time=1ms, hydrated=1, template=/root

```
select
    sessionman0_.id as id7_0_,
    sessionman0_.`_lastUse` as column3_7_0_,
    sessionman0_.`_logsqlMessage` as column4_7_0_,
    sessionman0_.version_opt_lock as version5_7_0_
from
    _SessionManager sessionman0_
where
    sessionman0_.id='d81abb1d74684af09696be091bc042d4'
```

Query 2: time=0ms, hydrated=0, template=/root

```
select
```

Figure 5.1: A screenshot of a request with a `?logsql` suffix. The table above the horizontal line is the content of the requested page and everything below the line has been added because of the `?logsql` suffix.

information about its garbage collection to the specified file, later referred to as the gc-log.

After starting Tomcat, the test script requests the test page 500 times, with additional Hibernate logging enabled. The additional Hibernate logging is enabled by adding a `?logsql` suffix to a request and is automatically disabled when there are no more requests that require the additional logging. A unique request identifier is inserted into log entries to identify the request they belong to. The response time and memory usage are not measured for these requests, because the additional logging has a significant impact on these performance indicators. The initial requests also serve to warm-up caches and to trigger automatic optimizations from the JVM for the tested code, like just-in-time compilation or dead code elimination.

Adding a `?logsql` suffix to a request also adds information from the Hibernate log and the Hibernate session to the end of the response, as shown in Figure 5.1. This includes the SQL queries that were executed. The Hibernate log also has entries that list entity-type/entity-id pairs for the entities present in recordsets returned from the database. This information is used to count the number of duplicate entity fetches. This is not simply the sum of all duplicate entity-type/entity-id pairs, because the actual entity-type may be a sub-type of the mentioned entity-type. To ensure that all duplicates are found, the actual entity-type is always used and determined by loading the entity from the Hibernate session. The number of unique entities and collections fetched is also shown at the end of a `?logsql` response, which is collected from the Hibernate session.

Before measuring the response time and memory usage, the test script forces garbage

collection. This is done by using `jmap` with the `-histo:live` argument, that returns a list of live objects, which it gets by garbage collecting first. After garbage collection, the script sums up the memory that was collected thus far, from the gc-log. The script remembers the amount of memory that was freed, so that the memory usage of the first 500 requests can be removed, when summing up the gc-log later.

Next, the script sends 2500 requests using `ab` [7], which measures the response time for the requests it sends. After `ab` terminates, another garbage collection is forced. By summing up the gc-log again and by removing the memory that was used by the first 500 requests, the script knows the total amount of memory that was used during the last 2500 requests. This number is divided by the number of requests, to get the average amount of heap memory that is required to respond to a request. Finally the script writes all collected results to a file, terminates Tomcat and starts again for the next prefetch technique, while there are more left.

5.2 Initial Test Page

To compare the prefetch techniques during the implementation, performance was measured for an actual WebDSL page from Researchr. The version of Researchr used during the implementation is older and contains fewer manual optimizations, than the version used for the evaluation in Chapter 8. The used test page is shown in Figure 5.2. This page uses paging to show only a subset of a collection, containing `PublicationCategory` entities. The `PublicationCategory` entities represent a year and contain a set of `Publication` entities that were published in that year. The division of `PublicationCategory` entities into pages is non-trivial and hard to translate into a query, because a page does not contain a fixed number of `PublicationCategory` or `Publication` entities. The test page shows 21 of the 40 categories and 92 of the 144 publications. There is also a second page that shows the remaining categories and publications and the prefetching techniques are challenged not to prefetch entities that are only required by this second page.

Performance is measured using the script from Section 5.1. The script is executed on an Intel Core i7-860 at 2.8 Ghz with 8 GB of memory. MySQL is used as the database, running on the same machine. The results are shown in Table 5.1 and Figure 5.3. The following sections discuss the results and how the techniques were changed to deal with their shortcomings.

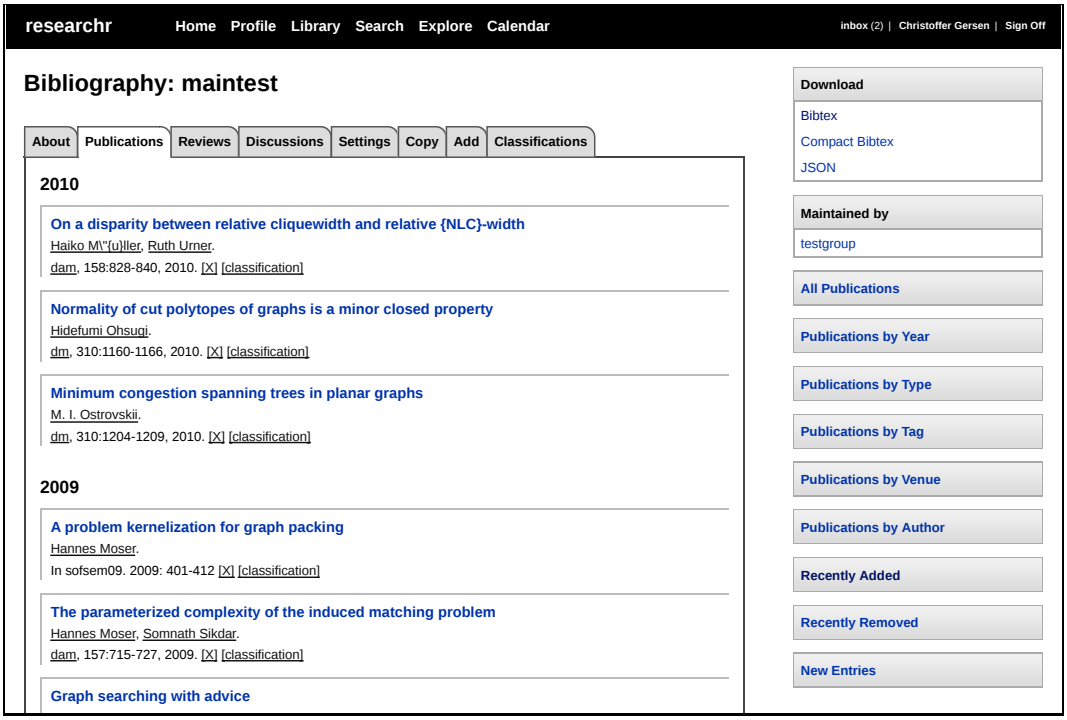
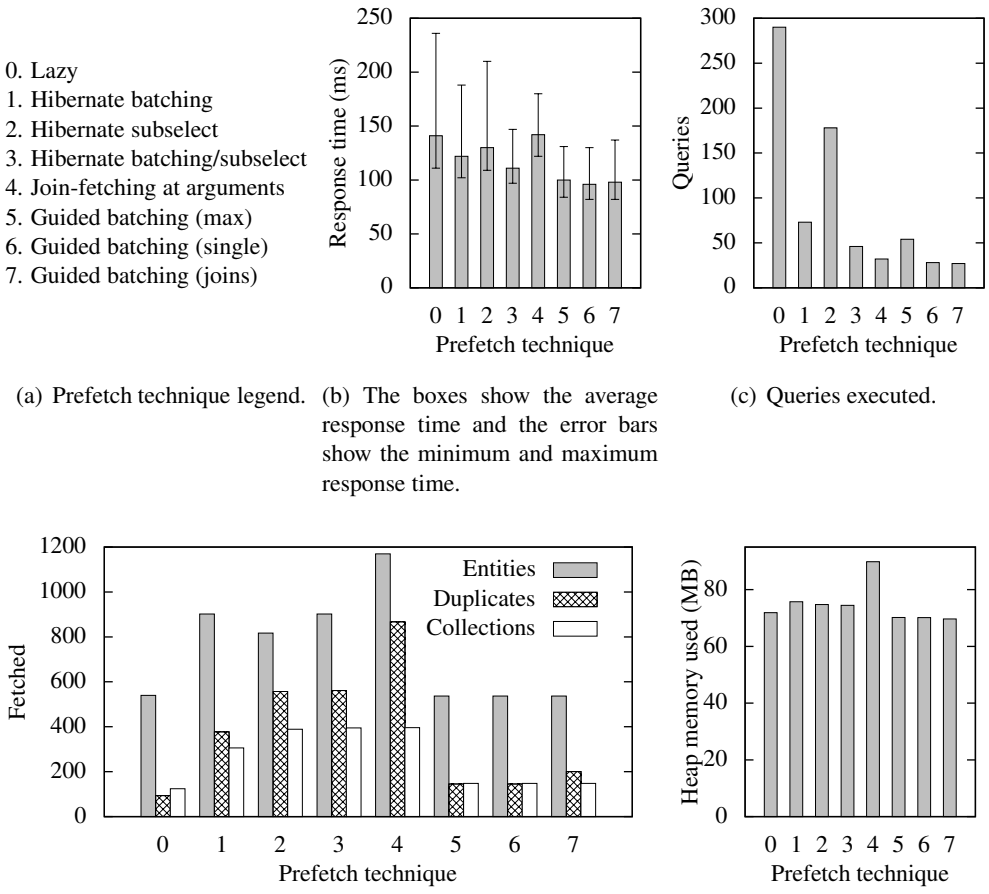


Figure 5.2: A screenshot of the top of the webpage that was used for the test

Measurement \ Prefetch technique	*Lazy* 0	*Hibernate batching* 1	*Hibernate subselect* 2	*Hibernate batching/-subselect* 3	*Join-fetching at arguments* 4	*Guided batching (max)* 5	*Guided batching (single)* 6	*Guided batching (joins)* 7
Min. response time (ms)	111	102	109	97	122	84	82	82
Mean response time (ms)	141	122	130	111	142	100	96	98
Max. response (ms)	236	188	210	147	180	131	130	137
Queries executed	290	73	178	46	32	54	28	27
Entities fetched	540	902	817	902	1170	537	537	537
Duplicates fetched	93	377	557	561	867	145	145	200
Collections fetched	124	306	389	395	396	148	148	148
Garbage collections	133	140	138	138	166	130	130	129
Garbage collection time (s)	5.16	4.94	5.01	4.80	5.99	4.76	5.15	4.84
Total heap memory (GB)	175.47	184.93	182.50	181.81	219.38	171.37	171.19	170.12
Heap per request (MB)	71.87	75.74	74.75	74.46	89.85	70.19	70.12	69.68

Table 5.1: Results for 2500 requests send to the page shown in figure 5.2.



(a) Prefetch technique legend.

(b) The boxes show the average response time and the error bars show the minimum and maximum response time.

(c) Queries executed.

(d) Number of entities, duplicate entities and collections fetched.

(e) Average memory used per request.

Figure 5.3: Visualized results from table 5.1.

5.3 Lazy

The *Lazy* prefetch technique does not perform prefetching and relies on the lazy-fetching mechanism of Hibernate. Lazy-fetching only fetches entities when they are used and that is why this technique fetches the least number of entities and collections. The downside is that this technique executes the most queries, because queries fetch only a single entity or collection. The average response time is not among the fastest as a result, because the *Lazy* technique suffers from the overhead of executing many small queries.

5.4 Hibernate batching

Hibernate batching was configured with a maximum batch size of 10, meaning that whenever an entity or collection is fetched, Hibernate will try to prefetch up to 9 similar entities or collections, using the same query. As discussed in Section 4.1, this technique does not know which entities are going to be used and as a result it may prefetch entities and collections for the second page. The batch generation process works as expected for entity properties, yet seems to provide random results for collection properties, which should not happen with a deterministic batch generation algorithm. The problem was that Hibernate copied the list with collection proxies into a set before batch generation, losing the sequence in the process. This has been fixed for WebDSL, by copying the list of properties into another list, instead of a set, maintaining the original sequence. As a result, batch-fetching for collection properties also provides reproducible results.

This technique is slightly faster than lazy-fetching, because the number of queries has been reduced. However, it could have been faster if fewer unused entities and collections were fetched. The number of duplicate entities fetched is also increased, because more unused collections are fetched and some of those collections contain elements that were already fetched. The large number of unused entities and collections is also shown by the increase in memory usage.

5.5 Hibernate subselect

Hibernate subselect attempts to use subqueries to prefetch multiple collection properties at once. Single value properties are lazy-fetched and still require individual queries. Queries are remembered for subselect-fetching, when the returned entity-type of the query has collection properties. However, a query may also return entities of a sub-type and the sub-type can contain collection properties, while the entity-type of the query does not. Hibernate does not remember these queries, because not all returned entities are guaranteed to have a collection property. When entities are cast to their correct sub-type and a collection property is accessed, then there will be no query available to use as a subquery. The Hibernate behavior is altered to also remember queries, when they return an entity-type that has sub-types with collection properties.

This technique is slower than *Hibernate batching*, because it executes more queries and fetches more collections. However, *Hibernate subselect* is still faster than lazy-fetching,

because of the reduction in the number of queries.

5.6 Hibernate batching/subselect

Hibernate batching/subselect combines the previous two prefetching techniques. A collection property is fetched by *Hibernate subselect* if there is a query that can be used as a subquery and otherwise it is fetched using *Hibernate batching*. *Hibernate batching* is also used for fetching single value properties. This combination reduces the number of queries even further than using *Hibernate subselect* or *Hibernate batching* alone and because of that this technique is faster than both. However, this approach still fetches many more entities than necessary and uses more memory.

5.7 Join-fetching at arguments

The *Join-fetching at arguments* technique prefetches entities and collections at the beginning of pages, templates and functions. It join-fetches properties to support the traversals made on the arguments, as specified by a prefetch specification. Collection properties are prefetched using a separate query, instead of using joins, in order to avoid Cartesian products. This technique also prefetches entities and collections that are only required by the second page, because the template that shows the collection performs prefetching before the elements of the second page are removed. Therefore this technique fetches the most entities and collections and uses the most memory, which makes this technique slow. However, this technique does execute fewer queries than the previous techniques.

5.8 Guided batching (max)

Guided batching (max) performs prefetching for a collection just before a for-loop starts iterating over the elements (Section 4.4). Batches are split into smaller batches of size 10, which makes it possible to use the same loaders that are used by *Hibernate batching*. Single value properties accessed on page arguments are never prefetched by *Guided batching (max)*. This can be solved by using join-fetching for non-collection properties when fetching page arguments. Since a page argument is one of the first entities that get fetched, it unlikely that join-fetching here will cause duplicate entity fetches. However, this reduces the number of queries only slightly and generating a query with joins can be slower than allowing lazy to use its prepared queries. The joins can also prefetch entities that are not used, which is unacceptable for such a small reduction in the number of queries. Therefore, *Guided batching (max)* does not perform prefetching for page arguments. For template and function arguments that are uninitialized proxies, join-fetching is only performed for recursively accessed properties, otherwise *Guided batching (max)* is unable to prefetch single-value properties inside a recursive function/template.

This technique is the first one that is able to reduce the number of queries, while only slightly increasing the number of fetched entities and collections. It is also the first tech-

nique that does not use more memory than lazy-fetching, which is also related to the number of fetched entities and collections.

5.9 Guided batching (single)

This technique is a variation of *Guided batching (max)*, that does not use static queries and instead generates new queries, so that the generated batches fit perfectly. Essentially this technique just reduces the number of queries at the cost of generating new queries. The generated queries also cost slightly more memory, however, the amount is neglectable.

5.10 Guided batching (joins)

Guided batching (joins) does the same as *Guided batching (single)*, except that it also generates joins for batch queries. These joins prefetch entities that would otherwise be fetched by another batch. On the test page this resulted in one less query, because one join prefetched all entities from another batch. This difference is not noticeable by looking at the response time. Join-fetching did fetch more duplicate entities, because some of the join-fetched properties were fetched already. Generating a query with joins is more complex and it does not provide a significant improvement and as a result *Guided batching (single)* is more promising.

Chapter 6

Static Analysis

The prefetch techniques from Chapter 4 will generate prefetch queries at the beginning of pages, templates, functions and for-loops, based on a static analysis of these code blocks. This chapter describes that static analysis. The static analysis is based on the work by Wiedermann et al. [19, 20]. Table 6.1 describes the static analysis using a rewriting semantics and shows how each language element is rewritten to its analysis result. The analysis assumes that the compiler has performed name binding and gave variables and definitions a unique name to avoid name clashes. Furthermore, it expects function and template calls to be resolved, taking overloading into account. The rewrite function is called QA. **resolve** indicates retrieving the definition a call is pointing to from the compiler and **resolve-multiple** indicates that there are multiple possible definitions, which occurs when a template has local overrides. **extract-condition** is a utility function that selects conditions that can be used for prefetching. **subst** indicates a global substitution in the element that precedes it. It is possible that not all variables are substituted, because the value of a variable may be unknown. Therefore parts of the analysis results may become invalid after substitution and are removed. Invalid variables inside conditions are removed by applying **extract-condition** again, which extracts the parts of the condition that are still valid.

Analysis blocks are pages, templates, functions, and for-loops that iterate over a persistent collection. The analysis results of these analysis blocks are stored for reuse, which prevents the same analysis block from being analyzed multiple times. The stored analysis results are also used after the analysis has been performed on the entire application, to automatically generate prefetch specifications for the persistent root variables ($\overline{v_r}$) of the analysis block. Persistent root variables are the inputs of an analysis block that can contain a persistent value. For pages, templates and functions these persistent roots are their arguments. For functions that are defined on entities, the `this` keyword is also a persistent root. For-loops occur inside other analysis blocks and they inherit the persistent roots of their parent analysis block, with the addition of the persistent iterator. For-loops that do not loop over a persistent collection are not considered analysis blocks, because they do not require prefetching.

During the analysis of an analysis block the properties accessed on a persistent root variable are recorded in the analysis results. The detection of property accesses on a persistent root is straightforward, because a property access is a node inside the abstract syntax

tree. In some cases it may be useful to detect the condition under which a property is accessed, because if the condition is false, then the property does not need to be fetched. The detection of conditions is discussed in Section 6.1. Section 6.2 walks through the analysis of an example step-by-step and describes how the analysis results can be transformed into a prefetch specification. The analysis of the different analysis blocks are discussed in more detail in sections 6.3, 6.4, and 6.5, for for-loops, functions, and templates, respectively.

signature QA: $(\overline{v_r}, e_{ast}) \quad \Rightarrow \quad (\overline{t}, \overline{e_{value}}, \overline{e_{return}})$ t = (e_{path}, *cond*, *effectful*)
$(\overline{v_r}, \text{|[} v \text{]|}) \quad \Rightarrow \quad ([(v, \textbf{true}, \textbf{false})], [v], [])$ **where** $v \in \overline{v_r}$
$(\overline{v_r}, \text{|[} e.f \text{]|}) \quad \Rightarrow \quad ([\overline{t}, \overline{t_v}], \overline{e_{value2}}, [])$ **where** $(\overline{t}, \overline{e_{value1}}, \_) = QA(\overline{v_r}, e)$ $\overline{e_{value2}} = \overline{e_{value1}}[\textbf{subst } x \rightarrow x.f]$ $\overline{t_v} = \{ (x, \textbf{true}, \textbf{false}) \mid x \in \overline{e_{value2}} \}$
$(\overline{v_r}, \text{|[} e.f(e_0 \ldots e_n) \text{]|}) \quad \Rightarrow \quad ([\overline{t}, \overline{t_{this}}, \overline{t_0} \ldots \overline{t_n}, \overline{t_v}], \overline{e_{return}}, [])$ **where** |[**function** $fd(a_0 \ldots a_n) \{ \ldots \}$]| = **resolve**(f) $(\overline{t_{this}}, \overline{e_{valuethis}}, \_) = QA(\overline{v_r}, e)$ $(\overline{t_0}, \overline{e_{value0}}, \_) = QA(\overline{v_r}, e_0)$ $\vdots$ $(\overline{t_n}, \overline{e_{valuen}}, \_) = QA(\overline{v_r}, e_n)$ $(\overline{t}, \_, \overline{e_{return}}) = QA([], fd)[\textbf{subst } this \rightarrow \overline{e_{valuethis}}, a_0 \rightarrow \overline{e_{value0}} \ldots a_n \rightarrow \overline{e_{valuen}}]$ $\overline{t_v} = \{ (x, \textbf{true}, \textbf{false}) \mid x \in \overline{e_{return}} \}$
$(\overline{v_r}, \text{|[} f(e_0 \ldots e_n) \text{]|}) \quad \Rightarrow \quad ([\overline{t}, \overline{t_0} \ldots \overline{t_n}, \overline{t_v}], \overline{e_{return}}, [])$ **where** |[**function** $fd(a_0 \ldots a_n) \{ \ldots \}$]| = **resolve**(f) $(\overline{t_0}, \overline{e_{value0}}, \_) = QA(\overline{v_r}, e_0)$ $\vdots$ $(\overline{t_n}, \overline{e_{valuen}}, \_) = QA(\overline{v_r}, e_n)$ $(\overline{t}, \_, \overline{e_{return}}) = QA([], fd)[\textbf{subst } a_0 \rightarrow \overline{e_{value0}} \ldots a_n \rightarrow \overline{e_{valuen}}]$ $\overline{t_v} = \{ (x, \textbf{true}, \textbf{false}) \mid x \in \overline{e_{return}} \}$
$(\overline{v_r}, \text{|[} \textbf{return } e \text{]|}) \quad \Rightarrow \quad (\overline{t}, [], \overline{e_{value}})$ **where** $(\overline{t}, \overline{e_{value}}, \_) = QA(\overline{v_r}, e)$
$(\overline{v_r}, \text{|[} \textbf{if} (e) \{ s_1 \} \textbf{ else } \{ s_2 \} \text{]|}) \quad \Rightarrow \quad ([\overline{t_c}, \overline{t_1}[\textbf{subst } cond \rightarrow cond \wedge c], \overline{t_2}[\textbf{subst } cond \rightarrow cond \wedge \neg c]], [], [\overline{e_{return1}}, \overline{e_{return2}}])$ **where** c = **extract-condition**(e) $(\overline{t_c}, \_, \_) = QA(\overline{v_r}, e)$ $(\overline{t_1}, \_, \overline{e_{return1}}) = QA(\overline{v_r}, s_1)$ $(\overline{t_2}, \_, \overline{e_{return2}}) = QA(\overline{v_r}, s_2)$
$(\overline{v_r}, \text{|[} \textbf{for}(v : \text{Type in } e \textit{ where order limit offset}) \{ b \} \text{]|}) \quad \Rightarrow \quad ([\overline{t_{early}}, \overline{t_0}[\textbf{subst } cond \rightarrow cond \wedge c], \overline{t_1}], [], \overline{e_{return}})[\textbf{subst } v \rightarrow \overline{e_{value}}]$ **where** $\overline{v_{rnew}} = [v, \overline{v_r}]$ $(\overline{t_{early}}, \_, \_) = QA(\overline{v_{rnew}}, [where, order, limit, offset])$ c = **extract-condition**($where$) $(\overline{t_0}, \_, \overline{e_{return}}) = QA(\overline{v_{rnew}}, b)$ $(\overline{t_1}, \overline{e_{value}}, \_) = QA(\overline{v_r}, e)$
$(\_, \text{|[} \textbf{function } fd(a_0 \ldots a_n) \{ b \} \text{]|}) \quad \Rightarrow \quad (\overline{t}, \_, \overline{e_{return}})$ **where** $\overline{v_{rnew}} = [this, a_0 \ldots a_n]$ $(\overline{t}, \_, \overline{e_{return}}) = QA(\overline{v_{rnew}}, b)$
$(\_, \text{|[} \textbf{template } td(a_0 \ldots a_n) \{ b \} \text{]|}) \quad \Rightarrow \quad (\overline{t}, \_, \_)$ **where** $\overline{v_{rnew}} = [a_0 \ldots a_n]$ $((t_0 \rightarrow do_0) \ldots (t_n \rightarrow do_n))$ = **local-overrides**(td) $(\overline{t}, \_, \_) = QA(\overline{v_{rnew}}, b)$ [**subst** $t_0 = do_0 \rightarrow \textbf{true} \ldots t_n = do_n \rightarrow \textbf{true}$, $t_0 = d_0 \rightarrow \textbf{false}$ where $d_0 ! = do_0 \ldots t_n = d_n \rightarrow \textbf{false}$ where $d_n ! = do_n$]
$(\overline{v_r}, \text{|[} t(e_0 \ldots e_n) \text{]|}) \quad \Rightarrow \quad ([\overline{t}, \overline{t_0} \ldots \overline{t_n}, \overline{t_u}], [], [])$ **where** |[**template** $td(a_0 \ldots a_n) \{ \ldots \}$]| = **resolve**(t) $(\overline{t_0}, \overline{e_{value0}}, \_) = QA(\overline{v_r}, e_0)$ $\vdots$ $(\overline{t_n}, \overline{e_{valuen}}, \_) = QA(\overline{v_r}, e_n)$ $(\overline{t}, \_, \_) = QA([], td)[\textbf{subst } a_0 \rightarrow \overline{e_{value0}} \ldots a_n \rightarrow \overline{e_{valuen}}]$
$(\overline{v_r}, \text{|[} t(e_0 \ldots e_n) \text{]|}) \quad \Rightarrow \quad ([\overline{t_0} \ldots \overline{t_n}], [], [])$ **where** $[td_0 \ldots td_n]$ = **resolve-multiple**(t) $(\overline{t_0}, \_, \_) = QA(\overline{v_r}, td_0\ (e_0 \ldots e_n))$ [**subst** $cond \rightarrow cond \wedge t = td_0$] $\vdots$ $(\overline{t_n}, \_, \_) = QA(\overline{v_r}, td_n\ (e_0 \ldots e_n))$ [**subst** $cond \rightarrow cond \wedge t = td_n$]
$(\overline{v_r}, \textbf{otherwise}) \quad \Rightarrow \quad ([\overline{t_1}, \overline{t_2}], \overline{e_{value}}, \overline{e_{return}})$ **where** $(\overline{t_1}, \overline{e_{value}}, \overline{e_{return}}) = QA(\overline{v_r}$, AST-children)[**subst** *effectful* $\rightarrow$ **true**] $\overline{t_2} = \{ (v, \textbf{true}, \textbf{true}) \mid v \in \overline{v_r} \}$

Table 6.1: Definition of the static analysis. Elements are rewritten to their analysis result.

6.1 Conditions and Effectful Statements

The conditions of if-statements can have an effect on the persistent values that are used by an analysis block. Initially traversals are given a true condition, meaning that they are accessed unconditionally. A traversal is the tuple of the path to a persistent value (a persistent root variable with property accesses to reach the value), the condition under which it is accessed, and its effectful flag. The effectful flag is initially set to false and is discussed later in this section. The extracted condition of an if-statement can be added to the traversals in the analysis results of the if-block, using the logical and-operator. The same can be done for traversals inside the analysis results of the else-block, except that a negated condition should be used. Take for example, Figure 6.1. It loops over all `Employee` entities, yet only uses them if their salary is above 65000, because the condition on line 4 is just used to filter employees. For-loops like this one, already execute a query to fetch all entities of a given type and because of that the condition can be placed inside that query, to fetch only the employees that are used. There are two requirements for a condition to be extracted and added to traversals in the analysis results.

1. The condition must be portable to the database and have the same semantics. This can be achieved by translating expressions to use basic comparators that compare primitive types.

2. The condition contains only:

    a) Persistent roots or properties accessed on them

    b) Constants of a primitive type

If only a part of a condition adheres to these requirements, then only that part will be added to the traversal inside the analysis results. The requirements on conditions do not allow conditions with local variables to be extracted. This makes the analysis simpler, because it removes the need for a data flow analysis, to determine the value of those local variables at different places inside the code. Within templates assignments are only allowed at the beginning, which would make the data flow analysis unnecessary in many cases anyway.

Statements that can affect non-local state should be handled differently by the analysis, because they determine when a persistent value is required. For example, the statements on lines 5-8 in Figure 6.1 affect non-local state and are effectful, because they write to the response stream of a HTTP request. As a result the `Employee` entities need to be fetched for those statements, yet not for the condition on line 4. Possibly effectful operations, like writing to a stream or assignments, fall into the **otherwise** rewrite rule, at the bottom of Table 6.1. The rewrite rule analyzes the AST-children and sets the effectful flag to true for all traversals in the analysis result.

The analysis also records an effectful traversal, on every persistent root, for each effectful statement, even if an effectful statement does not use any persistent data. This is to record conditions on those effectful statements, because their execution may change non-local state based on the existence of persistent data. Take for example Figure 6.2, which shows the salary of employees. The condition on line 3 should not be used, because both

```
1  page root() {
2    list {
3      for(e : Employee) {
4        if(e.salary > 65000)
5          listitem {
6            output(e.name) ", "
7            output(e.manager.name)
8          }
9        }
10     }
11   }
12 }
```

Figure 6.1: A condition in a for-loop

```
1  template showSalary() {
2    for(e : Employee) {
3      if (e.salary > 65000) {
4        output(e.salary)
5      } else {
6        "Salary too low"
7      }
8    }
9  }
```

Figure 6.2: Showing the salary of employees. The underlined template elements affect the state outside of the for-loop and depend on the existence of employees.

underlined template elements write to the response stream. The template element on line 6 does not use the `Employee` entities and simply writes a string literal to the response stream, yet it must be executed for employees with a salary of 65000 and below. Therefore, the analysis result of the for-loop will include an effectful traversal on `e`, with an `(e.salary <= 65000)` condition. The traversals for line 4 ultimately get a condition of `(e.salary > 65000)` and combining the conditions of all effectful traversals on `e`, using the or-operator, will result in the tautology `(e.salary > 65000 || e.salary <= 65000)` and is replaced with `true`. As a result all employees will be fetched, as is required to ensure the correct output.

The template call to `output` on line 4 of Figure 6.2 is effectful, however, the analysis of the call does not fall into the **otherwise** rule, because there is a rule for template calls. The template call rule knows that the called template is effectful, when its analysis result contains at least one effectful traversal. Templates without arguments have an empty analysis result and are also considered effectful. For effectful template calls, new effectful traversals are also recorded for persistent roots that do not yet have an effectful traversal inside the analysis result of the called template. Just like with other effectful statements, these effectful traversals are recorded, because the template call can depend on the existence of those persistent roots. Effectful function calls are handled in the same way as effectful template calls.

6.2 Example Analysis and Transformation

6.2.1 Example Analysis

This subsection demonstrates how Table 6.1 can be applied to a simple example, shown in Figure 6.3. The steps performed by the static analysis are shown in Table 6.2. Template elements and expressions have been given a unique identifier inside the first column for easy reference. The second column shows the code fragment that is to be rewritten to its

analysis result. Referenced code fragments are listed in column three, with the traversals ($\bar{t}$) of their analysis result listed inside the last column. Substitutions have already been applied to the last column. If the last column is empty, then no substitution was required and the traversals of the referenced code fragment are copied without changes. The table does not show $\overline{e_{value}}$ and $\overline{e_{return}}$ of the analysis results, because inside this example they are easily derived from the code fragments. For template elements (t_*) the $\overline{e_{value}}$ is always empty and for expressions (e_*) the $\overline{e_{value}}$ contains only the input expression, except for e_3 and e_7. These two expressions have an empty $\overline{e_{value}}$, because they contain operators. The $\overline{e_{return}}$ is empty for all code fragments, because the example does not contain return-statements.

We walk through the example table from the bottom up, because higher code fragments may require the analysis results of lower ones. In the example, t_9 is a built-in template of WebDSL, which writes a string value to the response stream. The single effectful traversal comes from the **otherwise** rewrite rule that is applied to the text-call. The rewrite rule adds an effectful traversal for every persistent root variable, which is just `s` , since the `output` template has only one argument. The same is done for t_6 and t_8, however, these code fragments occur inside t_1 and t_3, meaning that their persistent roots are `c` and `d` , resulting in two effectful traversals.

The code fragments t_7 and e_8 show how template calls and expressions are handled. The expression is rewritten to three traversals, using the first two rewrite rules of Table 6.1. The analysis of t_8 uses the analysis results of e_8 and t_9. For the results of t_9 the `s` variable is substituted with `d.madeBy.name` , since that is the $\overline{e_{value}}$ of e_8 and serves as the first argument of the template call. The template call is effectful, because the analysis result of t_9 contains an effectful traversal. Therefore, the results of t_7 should contain an effectful traversal for every persistent root, resulting in a second effectful traversal on `c` .

Expression e_7 uses only the first two rewrite rules, just like e_8. Expression e_6 is a special case, because it does not fall under the **otherwise** rule, since operators like these can be seen as function calls that are not effectful. The $\overline{e_{value}}$ for e_6 is empty, because function calls are not allowed inside $\overline{e_{value}}$ or $\overline{e_{return}}$. For t_5 the $\overline{e_{value}}$ of e_6 is unimportant, because the **extract-condition** utility function is used to extract the condition. The extracted condition is `!d.anonymous` and is substituted into the conditions of traversals from t_6, t_7, and t_8.

The next interesting rewrite step is that of t_3. The substitution of `d` to `c.donations` has already been performed on the analysis result. The analysis result before the substitution is also remembered, in order to generate a prefetch specification for variable `d` later on. The condition extracted from e_3 is substituted into the conditions of t_4 and t_5. Some traversals from t_5 already had a non-true condition, so not all traversals have the same condition, because the and-operator is used to combine the extracted conditions. The remaining rewrite steps are not discusses in further detail, because those steps are similar to previous ones.

Id	Code	Ref.	$\bar{t}$

t_1	**for**(c : Charity) {
	  t_2 t_3
	}
		t_2, t_3	See Table 6.3

t_2	output(e_1)	e_1 t_9	 (c , true , true)
e_1	c.name		(c , true , false), (c.name , true , false)

t_3	**for**(d : Donation
	  in e_2
	  where e_3) {
	  t_4
	  t_5
	}
		e_2
		e_3
		t_4
		t_5
			
			(c.donations , true , false),
			(c.donations.year , true , false)
			(c , $cond_2$, true), (c.donations , $cond_2$, true),
			(c.donations.value , $cond_2$, true),
			(c.donations , $cond_2$, false),
			(c.donations.anonymous, $cond_2$, false),
			(c , $cond_3$, true), (c.donations , $cond_3$, true),
			(c.donations.madeBy , $cond_3$, false),
			(c.donations.madeBy.name , $cond_3$, true)

e_2	c.donations		(c , true , false), (c.donations , true , false)
e_3	e_4 == 2013	e_4	
e_4	d.year		(d , true , false), (d.year , true , false)
t_4	output(e_5)	e_5 t_9	 (c , true , true), (d.value , true , true)
e_5	d.value		(d , true , false), (d.value , true , false)

t_5	**if**(e_6) {
	  t_6
	  t_7
	  t_8
	}
		e_6
		t_6, t_8
		t_7
			
			(c , $cond_1$, true), (d , $cond_1$, true),
			(c , $cond_1$, true), (d , $cond_1$, true),
			(d.madeBy , $cond_1$, false),
			(d.madeBy.name , $cond_1$, true)

e_6	!e_7	e_7	
e_7	d.anonymous		(d , true , false), (d.anonymous , true , false)
t_6	"("		(c , true , true), (d , true , true)
t_7	output(e_8)	e_8 t_9	 (c , true , true), (d.madeBy.name , true , true)
e_8	d.madeBy.name		(d , true , false), (d.madeBy , true , false), (d.madeBy.name , true , false)
t_8	")"		(c , true , true), (d , true , true)

t_9	**template**
	output(s:String} {
	  text(s)
	}
			(s , true , true)

Table 6.2: Example of rewriting code to their analysis result. Conditions are replaced by the following labels: $cond_1$ = !d.anonymous , $cond_2$ = c.donations == 2013 , and $cond_3$ = c.donations == 2013 && !c.donations.anonymous .

```
1 for(c : Charity) {                                                    // t1
2   output(c.name)                                                      // t2(e1)
3   for(d : Donation in c.donations where d.year == 2013) {             // t3(e2,e3,e4)
4     output(d.value)                                                   // t4(e5)
5     if(!d.anonymous) {                                                // t5(e6,e7)
6       "("                                                             // t6
7       output(d.madeBy.name)                                           // t7(e8)
8       ")"                                                             // t8
9 } } }
```

Figure 6.3: Listing donations to charities. This is a simplified version of Figure 3.1.

Path	Condition	Effectful
`c`	`true`	true
`c.name`	`true`	true
`c.donations.year`	`true`	false
`c.donations`	`c.donations.year==2013`	true
`c.donations.value`	`c.donations.year==2013`	true
`c.donations.anonymous`	`c.donations.year==2013`	false
`c`	`c.donations.year==2013 && !c.donations.anonymous`	true
`c.donations`	`c.donations.year==2013 && !c.donations.anonymous`	true
`c.donations.madeBy`	`c.donations.year==2013 && !c.donations.anonymous`	false
`c.donations.madeBy.name`	`c.donations.year==2013 && !c.donations.anonymous`	true

Table 6.3: Recorded traversals for the outer for-loop in Figure 6.3.

```
1 prefetch-for c {
2   donations where(.year==2013) {
3     madeBy if(.year==2013 && !.anonymous)
4   }
5 }
```

Figure 6.4: The automatically generated prefetch specification for the outer for-loop in Figure 6.3.

6.2.2 Transforming Traversals to a Prefetch Specification

The traversals recorded by the static analysis can be transformed into a prefetch specification. Take for example the code in Figure 6.3, and the recorded traversals shown in Table 6.3. The prefetch specification generated by the transformation of the analysis result is shown in Figure 6.4. The tree structure of the `c`, `c.donations` and `c.donations.madeBy` nodes of the prefetch specification is already visible in the table, after sorting the paths. The `name`, `year`, `value` and `anonymous` properties are not placed inside the prefetch specification, because they are of a primitive type, which will be fetched with the entity they are accessed on.

Next, it is attempted to extract a where-clause for the root variable. To do this all effectful conditions for traversals on `c` are collected from Table 6.3. Conditions that do not meet the requirements for a where-clause are replaced with a true condition. These requirements state that properties may only be accessed on `c` and that those properties should be of a primitive type. Constant values and primitive template/function arguments are also allowed. All effectful conditions that are collected for `c` are combined using the or-operator and this results in a true condition, which can be left out. The same is done for the collection property `c.donations` , where collecting all effectful conditions for traversals on `c.donations` , combined using the or-operator, yields the condition `(c.donations.year == 2013 || (c.donations.year == 2013 && !c.donations.anonymous))` , which is simplified to `(c.donations.year == 2013)` and provides the where-clause `(.year == 2013)` .

If-clauses are extracted in a similar fashion, however, also conditions for non-effectful traversals are used, because even if a property is only used in the condition of an if-statement to filter a collection, then it still needs to be prefetched. Otherwise the evaluation of the if-statement will execute a lazy-fetch query. Conditions that do not meet the requirements for an if-clause are also replaced with a true condition, just like with where-clauses. For example, collecting conditions for `c.donations.madeBy` yields the condition `(c.donations.year == 2013 && !c.donations.anonymous)` . Instead of allowing property accesses on `c.donations.madeBy` , they are only allowed on `c.donations` , because if-clauses are executed on parent nodes in the prefetch specification. This provides the condition `(.year == 2013 && !.anonymous)` for the if-clause.

6.3 For-loops

For-loops are analysis blocks that collect information about the use of the elements in a collection, by adding the iterator as a persistent root variable. The analysis result is stored without the substitution of the iterator. During the generation of the prefetch specification for the iterator, only traversals on the iterator are used. For for-loops without an in-clause or when the in-clause does not contain a traversal on a persistent root of the parent analysis block, then the substitution cannot be applied. As a result all traversals on the iterator are removed from the analysis result that is returned to the parent analysis block. An example of such an in-clause is a manual HQL query, which is never optimized further and does not need to be captured by the analysis.

6.3.1 Conditions on Collections

The collection of entities that a for-loop iterates over can be a collection property of an entity, like is shown in Figure 6.5. These for-loops do not always require a query, because the collection may have been fetched already. For example, the `d.employees` collection may or may not have been fetched before line 7, however, it has certainly been fetched on line 14, where the previously fetched collection will be reused. This will cause incorrect output when the collection is fetched on line 7, using the condition of the prefetch specification for variable `e1` , because then the collection will not contain the elements that are required by the for-loop on line 14.

```
1  template showEmployees(d : Department) {
2    prefetch-for d {
3      employees where hint (.salary > 60000 || .salary < 30000)
4    }
5    section {
6      header { "Salary above 60000" }
7      for(e1 : Employee in d.employees where e1.salary > 60000) {
8        prefetch-for e1 where (.salary > 60000) {}
9        output(e1)
10     } separated-by { break }
11   }
12   section {
13     header { "Salary below 30000" }
14     for(e2 : Employee in d.employees where e2.salary < 30000) {
15       prefetch-for e2 where (.salary < 30000) {}
16       output(e2)
17     } separated-by { break }
18   }
19 }
```

Figure 6.5: Multiple conditions on the same collection

A solution is to keep track of the condition that is used to fetch a collection and to re-fetch the collection when a different filtering is required. To do this conditions on collections are implemented by defining Hibernate filters. Hibernate filters are defined for the element type of a collection and have a name, a query string and parameters. A limitation of Hibernate filters is that conditions may not require joins inside the query, excluding properties that reference other entities from being used. A Hibernate filter is generated during compilation for every condition defined in the `where` -clauses of prefetch specifications. If a condition is a disjunction, then every part of the disjunction gets its own filter. If a filter with the same query string and parameter types already exists, then that filter is reused. After defining filters for all parts of a disjunction, the filters are combined into a new combined filter. A combined filter is similar to a regular filter, except that it also has a list with the names of its sub-filters, in the order they are combined in. Combining filters is important, because a filter can only be active once and enabling a second instance of the same filter with different parameters will disable the previous instance.

At runtime these filters can be temporarily enabled when a collection is accessed. The collection wrappers of Hibernate are replaced with new ones that also remember the filter used to fetch a collection. When an initialized collection is accessed, then the new collection wrappers will re-fetch the collection without filters, if the enabled filter is not represented by or not equal to the remembered filter. Filters are equal if they have the same name and parameter values. A filter is represented by a combined filter, when all the required elements of a collection are still present after applying the combined filter. Combined filters are always disjunctions, so a combined filter is represented by a second combined filter, when the sub-filters of the first one are a subset of the second one. For example, the combined filter of $f_1 \vee f_2$ is represented by the combined filter $f_1 \vee f_2 \vee f_3$, because $\{f_1, f_2\} \in \{f_1, f_2, f_3\}$.

Using filters this way will ensure the correct output. Unfortunately it will also use an extra query to fetch the same collection again, when a different filter is required. A better solution is to fetch the collection using a disjunction of all required filters, before the collection is used, for example at the beginning of a template. However, the guided batch-fetching technique performs prefetching before for-loops and not at the beginning of templates.This means that the `d.employees` collection could be fetched at the first for-loop. Transferring conditions between for-loops during compilation is difficult, especially if they are located in different templates or functions, because then the condition may also depend on the call stack. Since one caller may call multiple templates or functions that each require a different filter, while another caller may call only one of them and does not require multiple filters to be combined.

While guided batch-fetching does not prefetch collections at the beginning of templates, it can attach filters to uninitialized collections, to be used when it is fetched. When a collection is fetched at a for-loop, then the attached filter will be used, if that filter represents the filter enabled by the for-loop. If the attached filter does not represent the enabled filter, then the collection should be fetched without any filters, because there apparently is another for-loop that requires a conflicting filter.

Uninitialized properties may have to be accessed, in order to reach the collections for which there are filters to attach, causing lazy-fetch queries. Therefore, filters will only be attached to collections, when the analysis result contains conditions from different for-loops. When substituting the conditions in the analysis result of a for-loop, then the conditions are also annotated to reflect that they come from the for-loop. When extracting the where-clauses for collection properties of a prefetch specification from the analysis results, then the resulting condition can contain annotations from more than one for-loop. When there are annotations for more than one for-loop, then a `hint` keyword will be added to the resulting where-clause. The keyword states that parent properties may be accessed to attach a condition to the uninitialized property. Figure 6.5 shows a generated prefetch specification with the `hint` keyword on line 3.

6.3.2 Where, Order By, Limit and Offset

Besides a `where` clause a for-loop can also have `order by` , `limit` and `offset` clauses. Translating these into a query is useful, because it reduces the number of fetched entities. However, the `limit` and `offset` clauses may only be translated into a query if both the `where` and `order by` clauses are completely translated into the query, otherwise they will select the wrong entities. For batch queries these extra clauses are extremely difficult to translate, because setting a limit and an offset on a query applies to the recordset and not to the individual collections it contains. As a result the `order by` , `limit` and `offset` clauses are never translated into a query.

For guided batch-fetching, prefetching occurs after applying the `where` , `order by` , `limit` and `offset` clauses. This allows guided batch-fetching to perform prefetching for the correct elements, even when these clauses are not translated, or only partially translated, into the query that fetches the collection. However, any persistent data required by these clauses should be prefetched before their execution. Inside a prefetch specification this can

```
1 for(e : Employee where e.active order by e.projects.length desc limit 10){
2   prefetch-for e where (.active == true) {
3     projects fetch-early
4     department
5   }
6   output(e.name) ", "
7   output(e.projects.length) ", "
8   output(e.department.name)
9 } separated-by { break }
```

Figure 6.6: This example shows the prefetching of a property before applying the `where` , `order by` and `limit` clauses.

be specified by adding a `fetch-early` modifier on a property, as is shown in Figure 6.6. Such a property will be prefetched after the collection is fetched and before the filtering and ordering clauses of the for-loop are executed. Traversals coming from these clauses of the for-loop are annotated by the analysis. In the analysis described by Table 6.1, the annotated traversals are all inside $\overline{t_{early}}$. When the analysis result is transformed into a prefetch specification, then the `fetch-early` modifier is placed automatically on properties that have at least one traversal with the annotation. The annotations resulting in a `fetch-early` modifier are dropped, when the analysis results are returned to the parent analysis block, because there the properties can be prefetched in their regular order.

6.4 Functions

The static analysis follows calls to functions and analyzes them. After the function is analyzed, the persistent root variables are substituted for their persistent value inside the call. Functions that are declared on entities may be overridden by sub-entities and the called definition will depend on the subtype of the entity on which the function is called, which is not known until runtime. It is possible to check if a sub-entity overrides a function during compilation, and if no sub-entity does, then the called definition is known, under the assumption that entity-types are not changed or added after compilation. Besides entity functions, WebDSL also has static and global functions. Both function types cannot be overridden by sub-entities, because they are not defined for entity instances. As a result their called definition is always known.

When the called definition of an entity function is not known, because there are multiple definitions, then the call is not resolved and is considered an effectful operation. As a result the analysis does not include information about the properties that are accessed inside the called function. Take for example Figure 6.7, where there are two possible definitions for the call to `callee` on line 5. Inside the analysis result of `caller` there will be no traversal on property `prop` . While unimplemented, it would be possible to analyze all definitions of `callee` , and add a different condition to traversals for each definition, where the conditions check the type of `e` . This alternative approach is unimplemented, because it requires batch generation to keep track of a context. For example, the `caller` function can be called

```
1  entity Super {
2    name :: String
3    prop -> Super
4    function caller(e : Super) {
5      e.callee();
6    }
7    function callee() : String {
8      return this.name;
9    }
10 }
11 entity Sub : Super {
12   function callee() : String {
13     return this.prop.name;
14   }
15 }
```

Figure 6.7: A caller and a callee.

from within a for-loop, where the value of `e` changes every iteration. Batch generation then requires more than just a collection of entities for which it should prefetch a property, because it also needs to know the iteration that each entity comes from, so that the type of `e` can be computed.

6.4.1 Recursion

The analysis follows calls to functions and analyzes those function definitions, however, this will never terminate for recursive functions. One solution would be to capture the properties that are accessed recursively and to let guided batch-fetching prefetch for those properties until a generated batch is empty, meaning that the entire recursive data structure has been prefetched. This solution risks fetching an excessive number of unused entities, for example, when a depth-first-search is performed on a tree. The chosen alternative is to resolve the same function only a finite number of times. If the number of times is too low, then the resulting optimization may be unnoticeable, however, when it is set higher and the recursion stops early, then more unused properties are prefetched. The analysis resolves the same call up to four times by default, however, this can be customized.

Guided batch-fetching normally checks if a property has been fetched before, and if it is not, then it will be included in a batch for prefetching. This initialization check is cheap, however, for a recursive functions it can be performed many times for the same property. The check is performed more than once per property, because after a recursive function calls itself, it may check properties again, which were already checked and prefetched in a previous invocation.

A solution is to stop batch generation earlier. Figure 6.8 shows a prefetch specification, which stops batch generation at properties with a `no-empty-batch` modifier. The `no-empty-batch` modifier following a property specifies that if the batch for that property is empty, then no batches should be generated for its sub-properties. Without this modifier every collection property would be checked four times for initialization and with the

```
1  function getSize(dir : Directoy) : Int {
2    var size : Int := dir.size;
3    for(subDir : Directoy in dir.directories) {
4      prefetch-for subDir {
5        directories no-empty-batch {
6          directories no-empty-batch {
7            directories no-empty-batch {
8              directories no-empty-batch
9            }
10         }
11       }
12     }
13     size := size + getSize(subDir);
14   }
15   return size;
16 }
```

Figure 6.8: A prefetch specification that avoids checking recursive properties multiple times.

modifier only twice. Traversals that were added by resolving a recursive function call are annotated as such during the analysis. This allows the `no-empty-batch` modifier to be added automatically to properties inside a prefetch specification, when all traversals from the analysis containing the property are annotated as coming from a recursive call.

When an entity argument of a recursive function is uninitialized, then guided batch-fetching should prefetch that argument, with joins for single-value properties accessed recursively on that argument. Batch queries are unable to prefetch the joined properties, as was discussed at the end of Section 3.2. There is a small risk of fetching more duplicates. The risk is small, because the initial entity is uninitialized, which mean that is was not prefetched at a for-loop and makes it likely that the joined properties are not prefetched either. To select properties for join-fetching, guided batch-fetching also uses the `no-empty-batch` modifier from the prefetch specification of the function argument. It will join all single-value properties with a `no-empty-batch` modifier, which are reachable from the root, without having to prefetch collection properties. All single-value properties along the path are also join-fetched.

6.4.2 Return values

Properties can be accessed on persistent values that are returned by functions. These property accesses benefit from prefetching and thus should be recorded by the static analysis. The persistent value returned by a return-statement can frequently be described by an expression that performs a series of property accesses on a persistent root. These expressions are recorded inside the $\overline{e_{return}}$ part of the analysis result. These expressions can be used to describe the persistent value of a function call, after performing the substitution for the call arguments. Having expressions that describe the persistent values of function calls allows traversals to be recorded for any property accessed on their results. Take Figure 6.9 for

```
1  function getManager(e : Emplopyee) : Employee {
2    if(e.manager == null) {
3      return e;
4    }
5    return e.manager;
6  }
7  function getManagerAddress(e : Emplopyee) : Address {
8    return getManager(e).address;
9  }
10 function getManagersManager(e : Emplopyee) : Emplopyee {
11   return getManager(getManager(e));
12 }
```

Figure 6.9: A property is accessed on a persistent return value.

example, where the call to `getManager` on line 8 returns one of two possible persistent values. The analysis result for the `getManagerAddress` function will contain traversals and return expressions for both `e.address` and `e.manager.address`.

Return values are not only important for properties accessed directly on function calls, because function calls can also occur inside the arguments of function calls and template call, and also inside the in-clause of for-loops. These places all describe the persistent value that a persistent root has inside the called or containing analysis block. As a result, substitution may have to replace one persistent root with multiple expressions, when a call argument or in-clause contains a function call. For each replacement expression the traversals and return expressions inside the analysis result are copied and the persistent root is replaced with the expression. Invalid variables are removed after substitution as usual. For example, in Figure 6.9 on line 11, the inner call to `getManager` can return two possible values. However, the outer call, which is passed the result of the inner call, can return three values, which are `e`, `e.manager`, and `e.manager.manager`.

6.5 Templates

For the static code analysis, templates are similar to global function calls that do not return values, because they are both not defined on entities and cannot be overridden within sub-entities. However, WebDSL provides two other means to override the called template definition for template calls. These are dynamic scoping of template definitions, and required templates. Both features were introduced in Section 2.3 and require special attention during the analysis.

6.5.1 Dynamic Scoping of Template Definitions

Dynamic scoping of template definitions can be used to redefine a template within the scope of another, as discussed in subsection 2.3.1. This means that the called template definition depends on the call stack, because it could have been redefined by any caller on the call stack. The call stack is unavailable during compilation, so for calls to templates that

```
1  entity Collection { elements -> Set<Element> }
2  entity Element {
3    name :: String
4    default -> Element
5    alt -> Element
6  }
7  template dynamic(elem : Element) {
8    output(elem.default.name)
9  }
10 template caller1(col : Collection) {
11   template dynamic(elem : Element) {
12     output(elem.alt.name)
13   }
14   callee(col)
15 }
16 template caller2(col : Collection) {
17   callee(col)
18 }
19 template callee(col : Collection) {
20   for(elem : Element in col.elements) {
21     prefetch-for elem {
22       alt if(dynamic(Element) from caller(Collection))
23       default if(dynamic(Element))
24     }
25     dynamic(elem)
26   }
27 }
```

Figure 6.10: A prefetch specification checking the definition of a template at runtime.

have dynamic redefinitions somewhere in the code, the called definition may be unknown until runtime. The compiler constructs a list of all templates that have dynamic redefinitions. For calls to templates in that list **resolve** will fail. Instead the rewrite rule using **resolve-multiple**, from Table 6.1, is used. This rule calls the static analysis for all possible definitions, including the default definition, and adds conditions to the resulting traversals, checking the current definition of the template. These conditions can be evaluated at runtime. Take Figure 6.10, for example, where the definition for the call to `dynamic` , on line 25, is unknown during compilation. In the prefetch specification on lines 21-24, the correct property is prefetched by evaluating the if-statements. The if-statements contain template signatures, which are a combination of template name and argument types. The condition on line 22, checks if the definition for `dynamic` comes from `caller1` . Without a second template signature a condition checks if the global definition is active, like on line 23

When a template locally redefines a template, then conditions on that template can be evaluated during compilation. The rewrite rule analyzing template definitions does so by substituting the conditions with true or false, based on the now statically known definitions. This removes all conditions on the locally redefined templates and also removes traversals if they have a false condition. The substitution is also performed at for-loops inside templates,

```
1  template caller1(col : Collection) {
2    template dynamic(elem : Element) {
3      output(elem.alt.name)
4    }
5    for(elem : Element in col.elements) {
6      prefetch-for elem { alt }
7      dynamic(elem)
8    }
9  }
10 template caller2(col : Collection) {
11   for(elem : Element in col.elements) {
12     prefetch-for elem {
13       alt if(dynamic(Element) from caller(Collection))
14       default if(dynamic(Element))
15     }
16     dynamic(elem)
17   }
18 }
```

Figure 6.11: The caller templates from Figure 6.10, with `callee` inlined. The prefetch specification inside template `caller1` no longer requires the checks on the definition of `dynamic`.

because there the definitions are also statically known. Table 6.1 does not state this to keep the definition on a single page.

6.5.2 Required Templates

A template can require its caller to define certain templates, as discussed in subsection 2.3.2. Required templates can be viewed as a special case of dynamic scoping of template definition. Since required templates are always defined by the caller, conditions checking the current template definition can always be removed from the analysis result of the caller.

6.6 Implementation Differences

The actual implementation of the static analysis algorithm is slightly different than described in this chapter thus far. Recursive calls and dynamic scoping of template definitions are both handled differently in order to reduce the size of the generated prefetching code. Reducing the code size avoids running into the method size limitation of Java and can also improves performance.

6.6.1 Recursion

Resolving and analyzing recursive template or function calls directly may cause an excessive number of traversals, when multiple templates or functions are part of the same recursion. A simple example is shown in Figure 6.12, where all possible combinations of

```
1  entity Item          { name :: String   }
2  entity ItemA : Item { subItemA -> Item }
3  entity ItemB : Item { subItemB -> Item }
4  template output(i : Item) {
5    output(i.name)
6    if(i is a ItemA) { output(i as ItemA) }
7    if(i is a ItemB) { output(i as ItemB) }
8  }
9  template output(a : ItemA) {
10   output(a.subItemA) // Calls output(Item)
11 }
12 template output(b : ItemB) {
13   output(b.subItemB) // Calls output(Item)
14 }
```

Figure 6.12: A complex recursion that will generate too much prefetch code when extra entity-types are added.

`subItemA` and `subItemB` , up to a maximum depth of four, will be prefetched. This means $2^4 = 16$ different combinations, like `subItemA.subItemB.subItemB.subItemA` . While the generated code size is still acceptable for this example, it will grow very fast when more sub-types of `Item` are added to the example.

In order to avoid resolving too many recursive calls, there is an additional limit on the total number of recursive calls. Recursive calls are not resolved directly, instead they are placed inside the analysis results, alongside $\overline{t}$, $\overline{e_{value}}$ and $\overline{e_{return}}$. Variable substitutions are also performed on the recursive calls. After all analysis results are collected for all analysis blocks, then the analysis results can be visited again to resolve the recorded recursive calls. Resolving and analyzing a recursive call will result in new unresolved recursive calls, which are placed inside a new list. The calls inside the new list are not resolved until the previous list has been completely resolved. Resolving in this breath-first order is important, otherwise one recursion may be resolved completely, while another recursive call is ignored. Resolving recursive calls stops for an analysis block, when the total number of recursive calls has been reached. In the current implementation a maximum of 50 calls was chosen, because that was high enough to allow most common recursions to be resolved, while still limiting extreme cases.

6.6.2 Dynamic Scoping of Template Definitions

Using **resolve-multiple** to inline the prefetch specifications of all possible redefinitions will generate many conditions, which increases the generated code size. A simpler and faster solution would be to call the prefetch specification for possibly redefined templates at runtime. The analysis uses **resolve-multiple** to determine if the called template has multiple definitions, yet does not add conditions. Instead the template call remains unresolved and is placed inside the analysis result, just like is done with recursive calls. The unresolved template calls are used during the transformation into a prefetch specification, to generate

```
1 template callee(col : Collection) {
2   for(elem : Element in col.elements) {
3     prefetch-for elem templates [dynamic(this as Element)] {}
4     dynamic(elem)
5   }
6 }
```

Figure 6.13: Prefetch specification for `callee` inside Figure 6.10, using the actual WebDSL syntax, calling the correct prefetch specification at runtime.

calls. Take for example Figure 6.13, where a list with one template call has been added on the root of the prefetch specification. The template call is actually a template signature, where one argument type is replaced with the `this` keyword and an optional cast. These template calls can also be added to properties inside a prefetch specification. Each argument of a template has a prefetch specification and the specification for the argument with the `this` keyword will be called. The called prefetch specification is given all the values for the root variable or property on which the call has been defined, so in the example this will be the `col.elements` collection.

When transforming the analysis result into a prefetch specification, then for the root variable and each property inside the prefetch specification, the list with unresolved template calls will be searched. The search looks for calls that are passed the root variable or property in question. For the calls that are found the matching argument is replaced with the `this` keyword and the other arguments are replaced with their types, resulting in the list of template calls inside the prefetch specification.

Chapter 7

Evaluation with OO7 Benchmark

This chapter will evaluate the previously discussed prefetching techniques using the OO7 benchmark. Section 7.1 describes the evaluation method and the OO7 benchmark and Section 7.2 discusses the results.

7.1 Evaluation Method

To evaluate the prefetching techniques, a WebDSL implementation of the OO7 benchmark [3, 2, 4] was created. The OO7 benchmark is widely used to measure performance benefits of optimization techniques for Object Oriented Database Management Systems and Object Relational Mapping tools, including some of the techniques [11, 9, 10, 1, 20] discussed in Chapter 9. The OO7 benchmark does not model any specific application, however, it is intended as a benchmark for the CAD/CAM/CASE type of applications. This means that the intended domain of the OO7 benchmark is not that of web applications. Therefore Chapter 8 performs another evaluation, using complete WebDSL applications. The results of the OO7 benchmark are still discussed here, because it is so frequently used in related work.

7.1.1 Test Cases

The test cases of the OO7 benchmark are divided into the categories ”queries” and ”traversals”. The query cases are labeled Q1 to Q8, and traversals as T1 to T9. The difference between queries and traversals is that traversals perform operations from a single root entity, and queries fetch a collection of root entities on which operations are performed. Some traversals change the database, which are excluded from this evaluation, because they are variations of other traversals that perform updates. They provide no extra information, because updates are not optimized.

Every test case is implemented as a separate page in the WebDSL application, so that they can be accessed through an URL. Some of these test cases use random number generation. To make these test cases deterministic, every request gets its own random number generator that is always created using the same seed. For the traversals T1 and T6 a second variation is implemented that uses templates instead of functions and are labeled T1T and

Entity	Number of instances	Total
Modules	1	1
Manual	1 per Module	1
CompositePart	500 per Module	500
Document	1 per CompositePart	500
AtomicPart	20 per CompositePart	10000
Connection	3, 6 or 9 per AtomicPart	30k, 60k or 90k
ComplexAssembly	$\sum_{i=0}^{5} 3^i$ per Module	364
BaseAssembly	3^6 per Module	729

Table 7.1: Number of entities in the small OO7 database.

T6T respectively. The query Q8 also has two implementations. The one from OO7 is Q8, where a related entity is accessed by loading it manually using its identifier, instead of a regular property access. The implementation that is labeled Q8P uses a regular property access, because manually fetching by identifier is unusual in a WebDSL application.

Two new cases have also been added, to evaluate the effect of partial collection initialization and the traversal depth of recursive calls. The first test case is called Query Condition (QC for short), which uses only some elements from a collection, by using a condition. This condition is detected by the static analysis and is placed inside the query that fetches the collection, to perform partial initialization for the collection. The second test case is Traversal Depth (TD for short), which performs a depth first traversal up to a maximum depth. The maximum depth is changed to evaluate the prefetching techniques for low and high traversal depths.

7.1.2 Database

The OO7 database comes in the sizes small, medium and large. For this evaluation the small database is used. The small database is chosen in favor of the larger databases, because the most interesting test cases already use many more entities than that is typical for a web application. The number of entities and relations inside the database is always the same, yet some of the relations are randomly generated. The random number generator is always created using the same seed, which ensures that database generation is deterministic. The number of entities per entity-type of the small database is shown in Table 7.1. The table shows that the number of connections can be configured differently. For most test cases, 9 connections per atomic part are used. The databases with 3 or 6 connections per atomic part are only tested for the T1 and T1T test cases.

The structure of a module entity, which is the root of traversal test cases, is shown in Figure 7.1. The figure shows fewer instances and relations than that are present in the small database, for space and readability. The entity-relationship diagram, in Figure 7.2, is also useful while the module structure is explained. In the small database each `Module` has a `Manual` and a root `ComplexAssembly` . This is also the root of an `Assembly` tree of seven levels, with three children per `Assembly` . Every leaf is a `BaseAssembly` and the other entities are a `ComplexAssembly` .

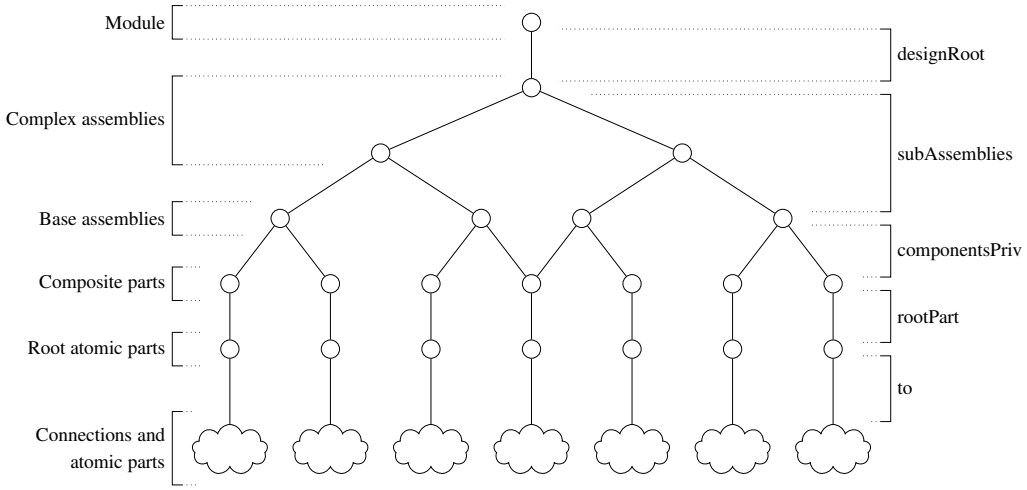


Figure 7.1: The structure of a module in the OO7 database. The labels on the left side describe what the entities on that level represent, and labels on the right side do the same for relations.

Every `BaseAssembly` has two collections, each containing three randomly selected `CompositePart` entities. Both of these collections should be implemented using a bag, however, bags are not supported by WebDSL. Instead a set is used, which disallows a `BaseAssembly` to be connected to the same `CompositePart` more than once. If a `CompositePart` is selected to be added to a set that already contains it, then the next available `CompositePart` that is not in the set is added.

Every `CompositePart` has a `Document` and a set of twenty `AtomicPart` entities. The `AtomicPart` entities are all connected through `Connection` entities. `AtomicPart` entities are connected as a ring, with extra random connections to get three, six or nine connections per `AtomicPart` . The first generated `AtomicPart` is chosen to act as a root, which is directly accessible from its `CompositePart` .

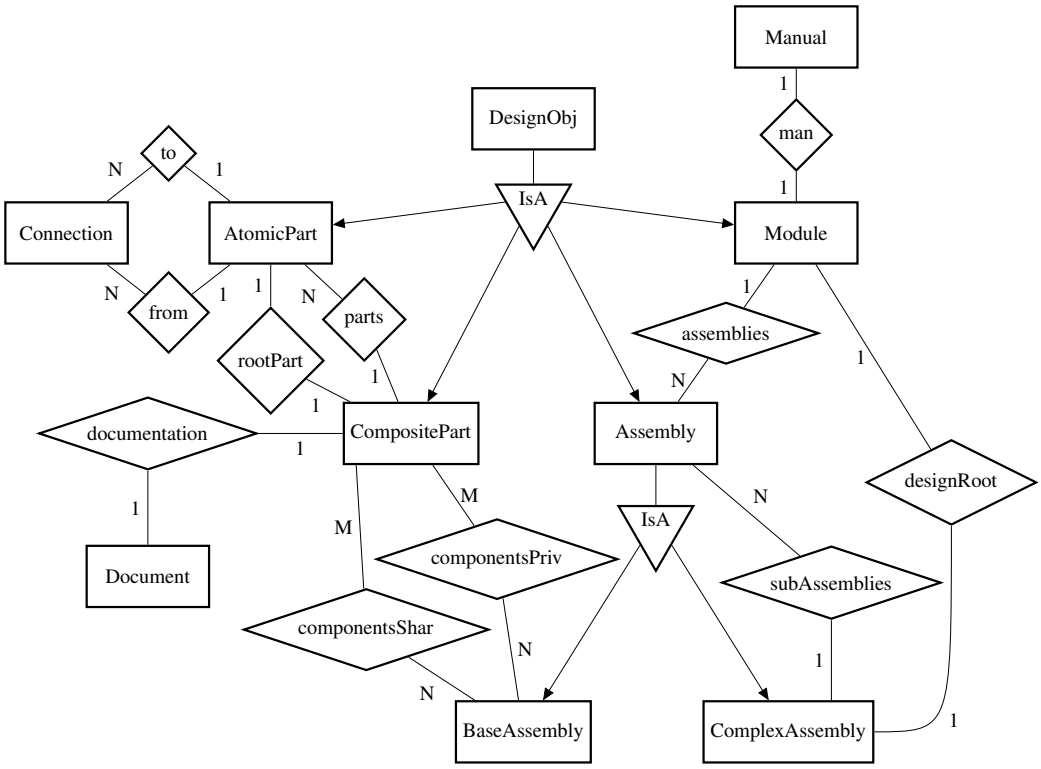


Figure 7.2: Entity-relationship diagram of the OO7 database.

7.1.3 Test Setup

The method used to measure the performance of the OO7 benchmark is very similar to the one described in Section 5.1. The test script that was used has only been changed slightly. The test script still accepts a set of WebDSL applications and additionally accepts a set of database creation scripts and a list of pages. The database creation scripts are of the three variations of the small OO7 database and are executed before deploying an application to Tomcat. And the URL of every included test case is listed in the list of pages. Some test cases require much time, so the number of initial requests with logging enabled is reduced from 500 to 5 and the number of requests without logging is reduced from 2500 to 50, in order to reduce the time required to run the test script.

7.2 Results

The results of the OO7 benchmark are shown in Table 7.2, Figure 7.3 and Figure 7.4. All test cases execute 2 queries and fetch 2 entities and 1 collection. These operations are shown in the results, however, they are unrelated to the test cases themselves. In the following subsections the results for traversals 1 and 6 and the queries 5, 6 and 8 are discussed in more detail. The query condition and traversal depth test cases are discussed in more detail as

well. The other test cases did not get optimized and as a result their performance remained about the same.

		Lazy					Guided batching (single)					
Case	DB	Time	Queries	Entities	Dupl.	Col.	Time	Queries	Entities	Dupl.	Col.	%
Q1	S9	22	12	12	0	1	22	12	12	0	1	0.0
Q2	S9	34	3	107	0	1	31	3	107	0	1	8.8
Q3	S9	39	3	947	0	1	39	3	947	0	1	0.0
Q4	S9	40	32	79	1	11	40	32	79	1	11	0.0
Q5	S9	203	733	1227	1692	731	63	5	1227	1692	731	69.0
Q6	S9	290	1097	1591	1692	1094	123	171	1591	1692	1094	57.6
Q7	S9	294	3	10002	0	1	294	3	10002	0	1	0.0
Q8	S9	387	503	10502	0	1	382	503	10502	0	1	1.3
Q8P	S9	369	503	10502	0	1	319	4	10502	0	1	13.6
T1	S3	12496	20897	41191	1692	10994	6004	4693	41191	1692	10994	52.0
	S6	14071	21057	71455	1688	11074	6942	3631	71455	1688	11074	50.7
	S9	15301	20897	100591	1692	10994	7973	3217	100591	1692	10994	47.9
T1T	S3	15112	20897	41191	1692	10994	8174	5562	41191	1692	10994	45.9
	S6	16822	21057	71455	1688	11074	8901	3981	71455	1688	11074	47.1
	S9	17870	20897	100591	1692	10994	9732	3314	100591	1692	10994	45.5
T6	S9	430	1592	2086	1692	1094	7511	3217	100591	1692	10994	-1646.7
T6T	S9	497	1592	2086	1692	1094	281	473	2086	1692	1094	43.5
T8	S9	21	4	4	0	1	20	4	4	0	1	4.8
T9	S9	18	4	4	0	1	18	4	4	0	1	0.0
QC	S9	9293	10687	77882	24092	10685	2956	6	72974	24092	10685	68.2

Table 7.2: Results from running the OO7 benchmark, using the small9 database configuration and also small3 and small6 for T1 and T1T. The improvement in response time is shown by the percentages in the last column.

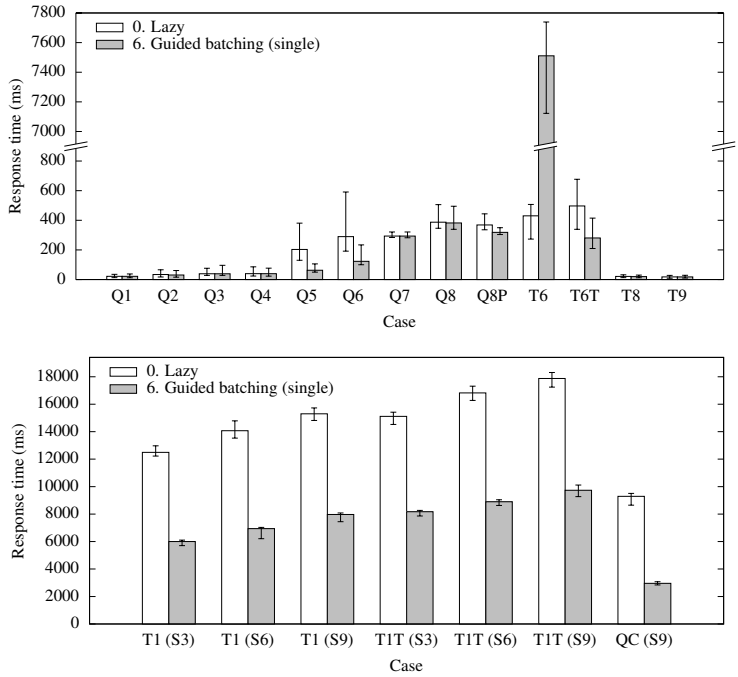


Figure 7.3: Response times for the OO7 benchmark. The boxes show the average response time and the error bars show the minimum and maximum response time.

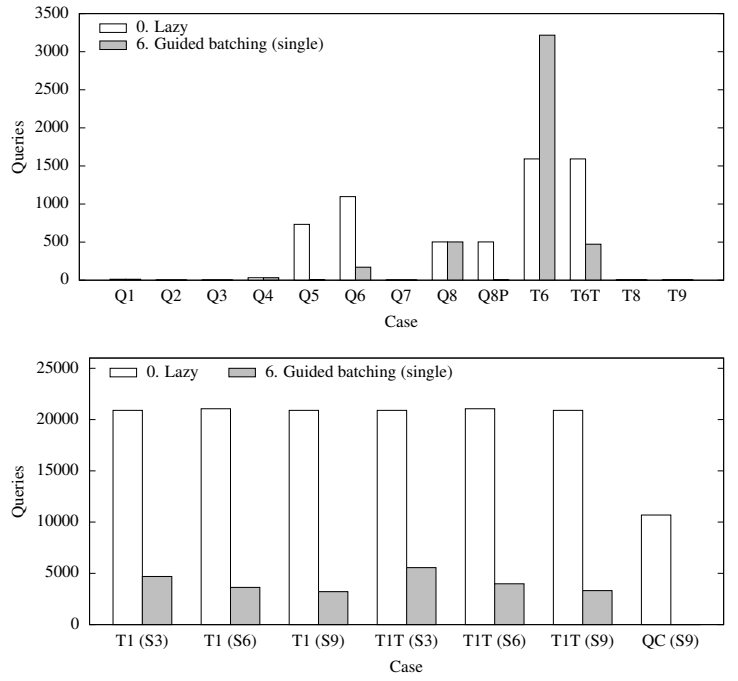


Figure 7.4: Number of queries executed for the OO7 benchmark.

7.2.1 Traversal 1

Traversal 1 traverses the entire module structure displayed in Figure 7.1. This includes all entities associated with a `Module` , with the exception of `Manual` and `Document` entities. An implementation of this test case is shown in Figure 7.5. Traversal 1 accesses all associated `Assembly` and `AtomicPart` entities. The `Assembly` entities form a tree structure, which does not contain cycles. The `AtomicPart` entities are connected through `Connection` entities, which can contain many cycles.

The traversal on a data structure with cycles makes join-fetching techniques less efficient, because the joins are likely to fetch duplicates. A join will fetch a duplicate entity, when the joined property leads back to a previously fetched entity. Figure 7.6 shows that the join-fetching techniques do indeed fetch many more duplicates and that they can still provide a small performance improvement regardless. This improvement is possible, because collection properties are never join-fetched, to avoid Cartesian products. If the implementation of the prefetching techniques would allow joins of collection properties, then the number of fetched duplicates would increase significantly and the performance would decrease.

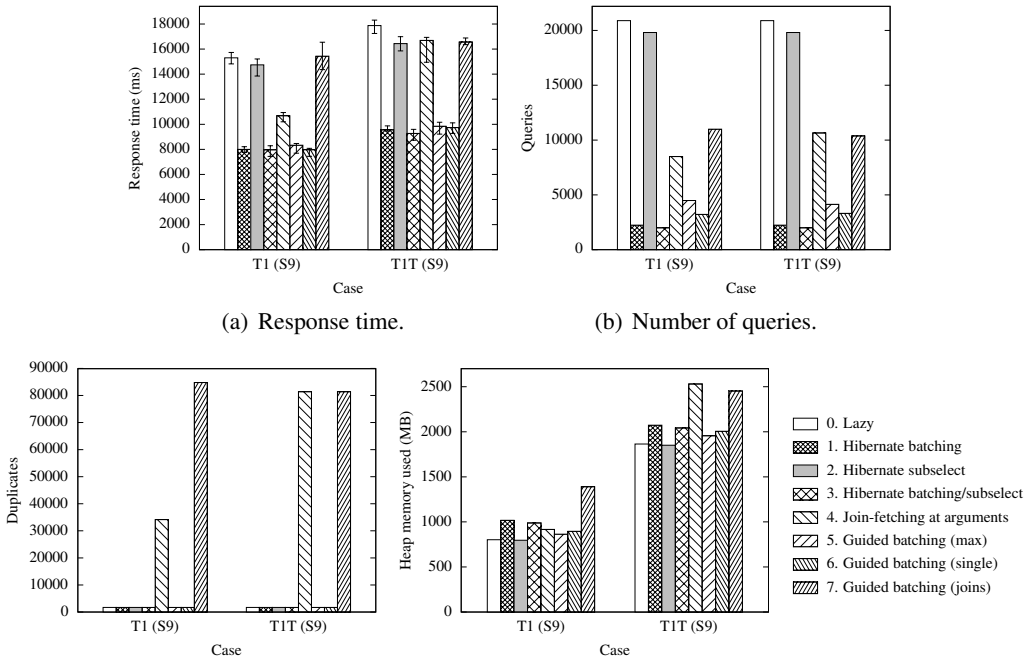
The batch-fetching techniques 1, 3, 5 and 6 all show a significant improvement in performance, which is the result of fetching the same data using fewer queries. Hibernate batch-fetching executes fewer queries than guided batch-fetching for this test case, because Hibernate batch-fetching performs well when most entities are accessed. In that case Hibernate batch-fetching does not fetch entities that are never used and most queries fetch 10 entities or collections. Guided batch-fetching is more careful with regard to fetching unused data and because of that the analysis only follows recursive calls four times, which causes many collection batch queries to fetch less than 10 collections. The guided batch-fetching techniques does use less memory than Hibernate batch-fetching, because of the way Hibernate generates batches for collections, which first copies the list of proxies it picks from. This is a problem with the Hibernate version used by WebDSL, the batch generation has been improved in the latest version of Hibernate.

Hibernate subselect-fetching does fetch the `Assembly` entities more efficiently that the other techniques, because there is only one query per level of the tree. The traversal through `AtomicPart` and `Connection` entities does not get optimized by subselect-fetching, because the `to` property on `Connection` entities is fetched by a lazy-fetch query. The query is not suitable as a subquery for subselect-fetching, because it returns only one `AtomicPart` entity.

For prefetching technique 6 the response time and number of queries can be reduced beyond that of Hibernate batch-fetching, by using the manual prefetch specification in Figure 7.7. The prefetch specification describes all accessed properties for case T1T using the small9 database configuration. Using the manual specification reduces the response time from 9732ms to 4381ms and the number of queries from 3314 to 19. This manual prefetch specification exploits prior knowledge about the database, which makes it an unfair comparison. However, it still shows the flexibility of prefetching technique 6, because the other prefetching techniques are unable to use this prefetch specification as effectively. Prefetching techniques 4 and 7 would fetch more duplicates and prefetching technique 5 is still

```
1  page traversal1template() {
2    var count : Int := 0;
3
4    traverse(Module.getRandomProxy())
5    output(count)
6
7    template traverse(part : CompositePart) { traverse(part.rootPart) }
8    template traverse(part : AtomicPart) {
9      var visited : List<AtomicPart>;
10     traverseDFS(part, visited)
11   }
12   template traversalOp(part : AtomicPart, first : Bool) {
13     init {
14       count := count + 1;
15       part.DoNothing();
16     }
17   }
18 }
19 // The following definitions are shared with other traversal cases
20 // These definitions are dynamically redefined by cases when needed
21 template traverseDFS(part : AtomicPart, visited : Ref<List<AtomicPart>>) {
22   init { visited.add(part); }
23   traversalOp(part, visited.length == 0)
24   for (connection : Connection in part.to) {
25     if (!(connection.to in visited)) {
26       traverseDFS(connection.to, visited)
27     }
28   }
29 }
30 template traverse(mod : Module) { traverse(mod.designRoot) }
31 template traverse(assembly : Assembly) {
32   if(assembly is a ComplexAssembly) {
33     traverse(assembly as ComplexAssembly)
34   } else {
35     if(assembly is a BaseAssembly) {
36       traverse(assembly as BaseAssembly)
37     }
38   }
39 }
40 template traverse(assembly : ComplexAssembly) {
41   for(subAssembly : Assembly in assembly.subAssemblies) {
42     traverse(subAssembly)
43   }
44 }
45 template traverse(assembly : BaseAssembly) {
46   for (component : CompositePart in assembly.componentsPriv) {
47     traverse(component)
48   }
49 }
50 template traverse(part : CompositePart) { }
51 template traverse(part : AtomicPart) { }
52 template traversalOp(part : AtomicPart, first : Bool) { }
```

Figure 7.5: The implementation of T1T (T1 using templates).



(a) Response time. (b) Number of queries.

(c) Number of duplicates. (d) Heap used per request. (e) Legend.

Figure 7.6: Results for the T1 and T1T test cases of the OO7 benchmark, using the small9 database configuration. The number of entities and collections fetched are similar for all prefetching techniques for both cases and these results are excluded from this figure.

limited to fetch only 10 entities or collections per query.

```
1  template traverse(mod : Module) {
2    init {
3      for(module : Module in [mod]) { // Forces batch-fetching on one value
4        prefetch-for module {
5          designRoot {
6            ComplexAssembly.subAssemblies {
7              ComplexAssembly.subAssemblies {
8                ComplexAssembly.subAssemblies {
9                  ComplexAssembly.subAssemblies {
10                   ComplexAssembly.subAssemblies {
11                     ComplexAssembly.subAssemblies {
12                       BaseAssembly.componentsPriv {
13                         rootPart {
14                           to {
15                             to {
16                               to {
17                                 to {
18                                   to {
19                                     to {
20                                       to {
21                                         to
22   } } } } } } } } } } } } } } } } } } }
23   traverse(mod.designRoot)
24 }
```

Figure 7.7: A manual prefetch specification for T1T, exploiting prior knowledge about the database.

7.2.2 Traversal 6

Traversal 6 is similar to traversal 1, except that traversal 6 stops after reaching the root atomic parts. As a result this case does not access a recursive data structure with cycles. The function implementation fetches more entities, because it also fetches some entities that are required by traversal 1. The incorrect entities are fetched, because the analysis did not handle a return-statement properly. The return-statement occurs within an if-statement and causes an early exit out of the function. Therefore a negated condition should be attached to traversals from statements following the if-statement, however, this is not implemented. The response time increased the most for prefetching techniques 5 and 6, because they fetch the most unused entities. Prefetching techniques 4 and 7 do not fetch as much unused entities, because join-fetching prefetches the entire next recursive batch and `no-empty-batch` causes prefetching to stop early. The template implementation does not have this problem and does show a performance improvement.

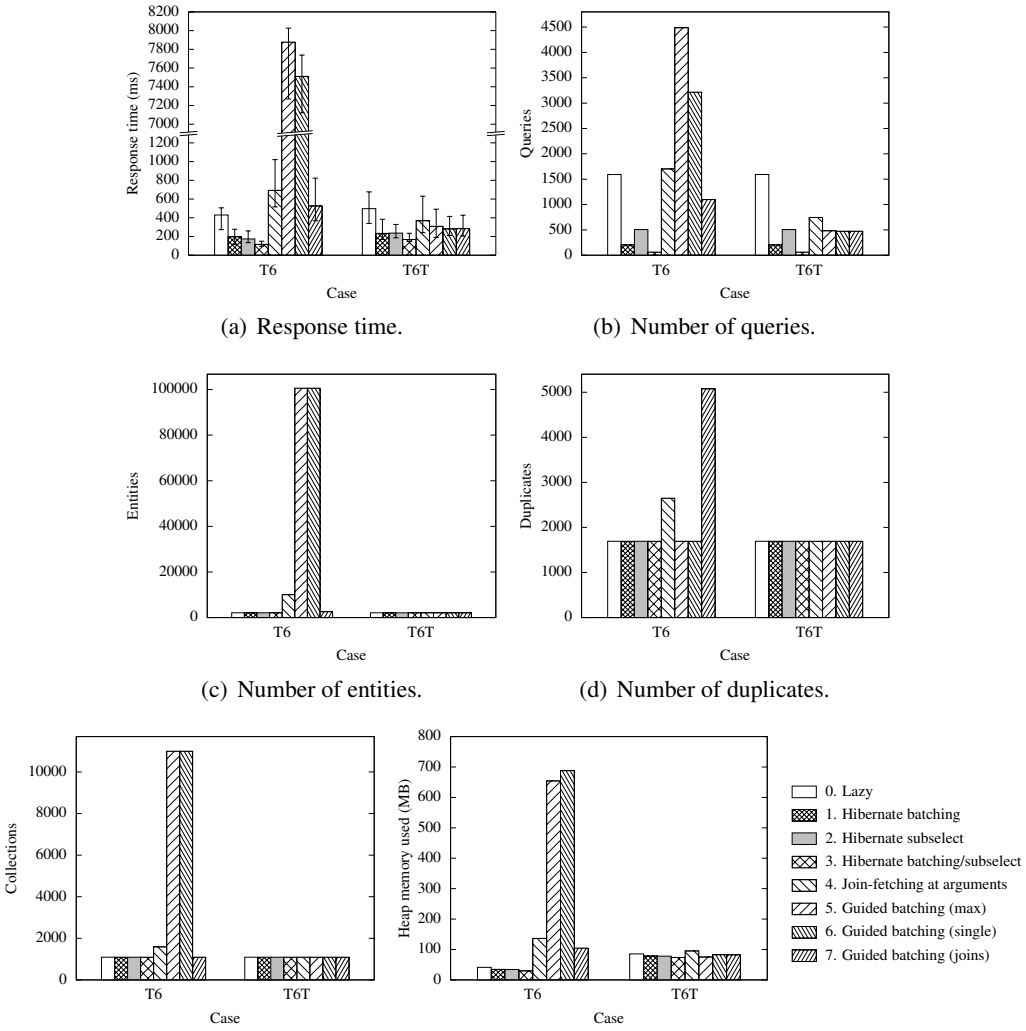
Hibernate subselect fetching with batch-fetching (technique 3) performs the best on this test case. Hibernate subselect fetching is able to fetch the tree of `Assembly` entities using only one query per level, with the query of the previous level as subquery. The query that fetched the leaves of the tree is used as a subquery to fetch all `CompositePart` entities. Finally, Hibernate batch-fetching takes over to fetch the root `AtomicPart` entities of the `CompositePart` entities. This is very efficient, because the subqueries that are used are

```
1  page traversal6template() {
2    var count : Int := 0;
3
4    traverse(Module.getRandomProxy())
5    output(count)
6
7    template traverse(part : CompositePart) { traverse(part.rootPart) }
8    template traverse(part : AtomicPart) {
9      init {
10       count := count + 1;
11       part.DoNothing();
12     }
13   }
14 }
```

Figure 7.8: The implementation of T6T (T6 using templates), using some definitions from Figure 7.5.

very simple. However, if the first query was more complex and slower, then it would also have been included in the following few queries, making those queries slower as well.

Hibernate subselect fetching alone also improves performance, yet is still slower, because every root `AtomicPart` is fetched by its own query. The other prefetching techniques also improve performance, however, they have to fetch the `Assembly` entities using more queries, which makes them slower. Traversal 6 also shows that guided batch-fetching techniques use more queries than Hibernate batch-fetching when prefetching for a complete traversal of a tree. This is because guided batch-fetching only prefetches for the next four recursive calls, using one query for each. When it finds another uninitialized node it repeats this process. This means that some collection batches contain less than 10 collections to fetch, while Hibernate batch-fetching will do its best to create batches with 10 collections.



(a) Response time. (b) Number of queries.

(c) Number of entities. (d) Number of duplicates.

(e) Number of collections. (f) Heap used per request. (g) Legend.

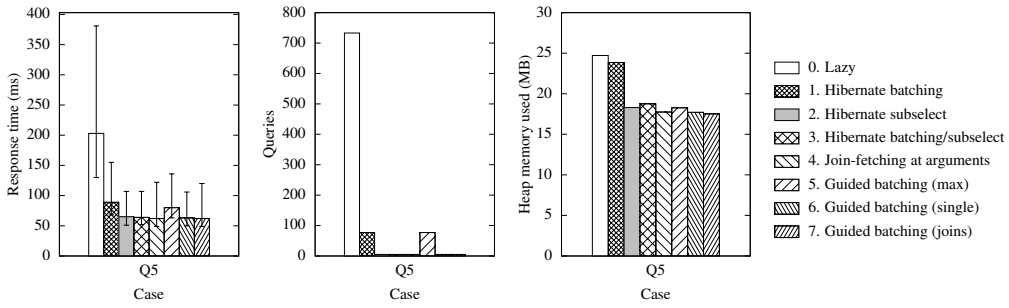
Figure 7.9: Results for the T6 and T6T test cases of the OO7 benchmark.

```
1  page query5() {
2    var count : Int := 0;
3    init {
4      for(module : Module) {
5        for(assembly : Assembly in module.assemblies) {
6          for(compositePart : CompositePart in (assembly as BaseAssembly)
7                                                          .componentsPriv) {
8            if (compositePart.buildDate > assembly.buildDate) {
9              assembly.DoNothing();
10             count := count + 1;
11           }
12         }
13       }
14     }
15   }
16   output(count)
17 }
```

Figure 7.10: The implementation of query 5.

7.2.3 Query 5

Query 5 is a test case that uses nested loops to iterate over collections, as seen in Figure 7.10. The results in Figure 7.11 show that all prefetching techniques perform well for this test case, because it is a simple example of the $n+1$ query problem, which all techniques try to avoid. However, prefetching techniques 1 and 5 execute a few more queries and are slightly slower, because their queries can only fetch ten collections with a single query.



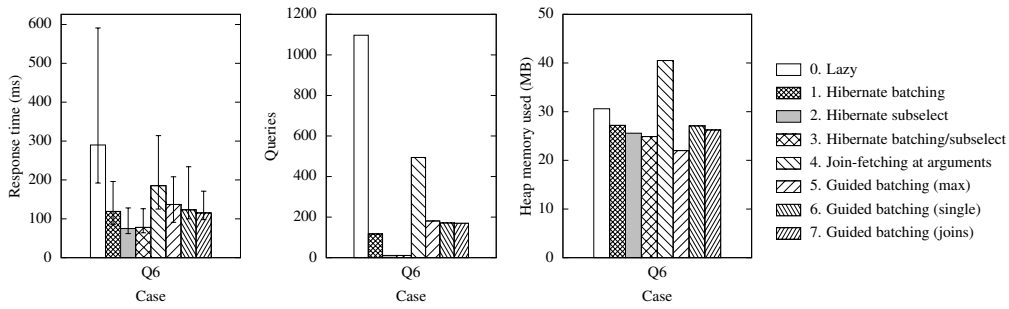
(a) Response time. (b) Number of queries. (c) Heap used per request. (d) Legend.

Figure 7.11: Results for the Q5 test case of the OO7 benchmark. The number of entities, duplicates and collections fetched are the same for all prefetching techniques and these results are excluded from this figure.

7.2.4 Query 6

Query 6 is very similar to traversal 6, because they both traverse the assembly hierarchy. However, query 6 does this for all modules, instead of a single one and query 6 does not traverse to the root part of a `CompositePart`. Unlike the function implementation of traversal 6, query 6 does not fetched unused entities because of an early exit. As a result this test case is optimized correctly and the response time and number of queries have both been reduced, by all prefetching techniques.

Hibernate subselect fetching is again the fastest prefetching technique for this test case, because of the balanced tree structure of `Assembly` entities.Prefetching technique 4 executes more queries than technique 6, because the recursive prefetching resumes at different levels of the tree of `Assembly` entities. Prefetching technique 4 resumes recursive prefetching at a lower level in the tree, where there are more entities. Every entity on that level will be a root for the prefetch specification and this determines the number of batches and queries that are generated.



(a) Response time. (b) Number of queries. (c) Heap used per request. (d) Legend.

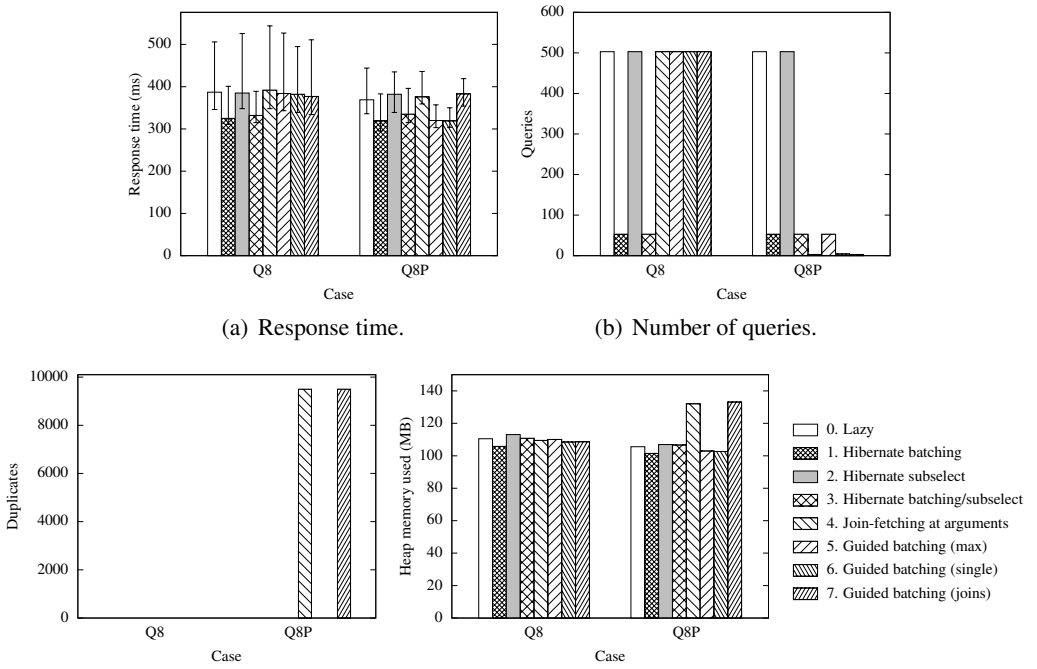
Figure 7.12: Results for the Q6 test case of the OO7 benchmark. The number of entities, duplicates and collections fetched are the same for all prefetching techniques and these results are excluded from this figure.

7.2.5 Query 8

Query 8 iterates over all atomic parts and manually loads a `Document` entity by its identifier, which has been stored inside a `UUID` property on the atomic parts. A manual load is unusual in WebDSL and because of that the related `Document` is also stored inside a `document` property of an `AtomicPart`. The Q8P test case uses this property instead of a manual load.

Hibernate batch-fetching seems to be the only technique that optimizes the manual loading test case. However, the technique uses the `document` property that was added for the Q8P test case. In the original OO7 benchmark this property does not exist and Hibernate batch-fetching would also fail to optimize the Q8 test case.

For the Q8P test case most prefetching techniques are able to reduce the number of queries, except for Hibernate subselect fetching which does nothing, because there are no collection properties involved. The prefetching techniques that use joins (4 and 7) slightly



(a) Response time. (b) Number of queries.

(c) Number of duplicates. (d) Heap used per request. (e) Legend.

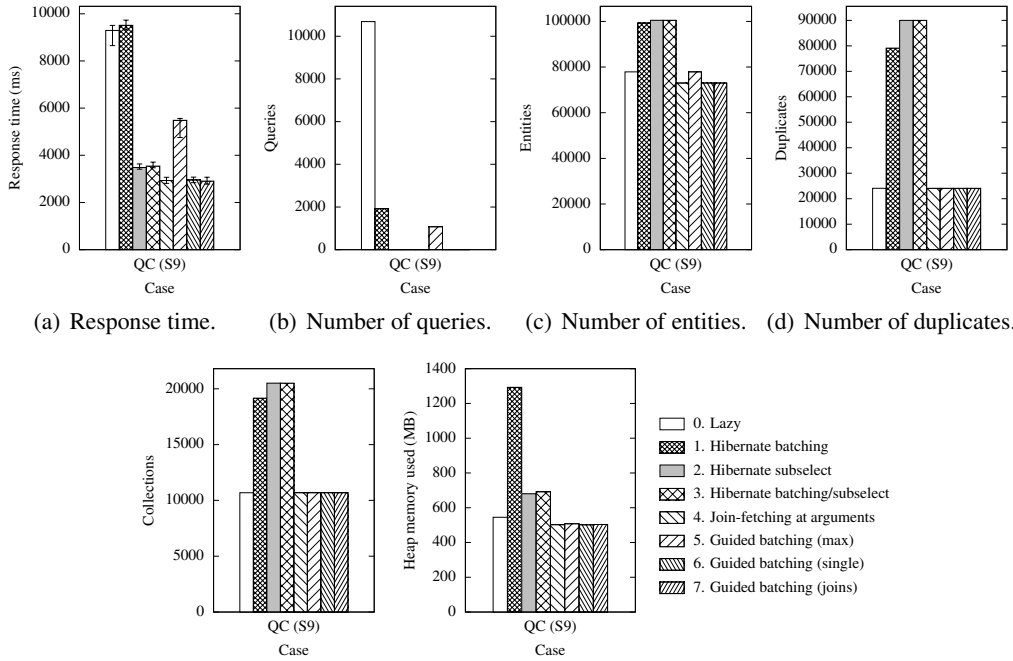
Figure 7.13: Results for the Q8 and Q8P test cases of the OO7 benchmark. The number of entities and collections fetched are similar for all prefetching techniques for both cases and these results are excluded from this figure.

reduce performance and use more memory, which is caused by the high number of duplicate entity fetches. The duplicate entities also contain a text property with 2000 bytes, which increases the time and memory cost of the data duplication. The other prefetching techniques all perform about the same, however, of those techniques number 6 reduces the number of queries the most, because it has no maximum batch size.

7.2.6 Query Condition

This test case is not included in the standard OO7 benchmark and is added to show the benefits of adding a condition to a query, for partial collection initialization. This test case has nested loops, where the outer-loop iterates over all `CompositePart` entities. The inner-loop iterates over the list of related `AtomicPart` entities. The inner-loop has a where-clause that selects about half of the `AtomicPart` elements, by checking if the `x` property has a higher value than the `y` property. Inside the inner-loop the largest connection is found by iterating over the `to` and `from` properties. The length of the largest connection is added to a variable, which is eventually shown on the page.

For this test case the Hibernate techniques all prefetch too many entities, yet some still improve the performance, because of the reduction in number of queries. Both techniques



(a) Response time. (b) Number of queries. (c) Number of entities. (d) Number of duplicates.

(e) Number of collections. (f) Heap used per request. (g) Legend.

Figure 7.14: Results for the QC test case.

with a maximum batch size (1 and 5) are slower, because they require more queries. Join-fetching has no influence on this test case, because collection properties are never join-fetched, making techniques 4, 6 and 7 all similar. Technique 5 fetches more entities than technique 6, because it uses static queries instead of dynamically generated ones, meaning that the condition to partially initialize collections is not present.

7.2.7 Traversal Depth

This traversal test case is not part of the standard OO7 benchmark, yet it is similar to traversal 1 and traversal 6. However, when reaching a `CompositePart` it does not visit all connected parts or just the root part, like in traversal 1 and 6, respectively. Instead it tries to visit previously unvisited `AtomicPart` entities from the root part, up to a traversal depth that varies from 1 to 19. A traversal depth of 0 would make this case identical to traversal 6 and a traversal depth of 19 will visit all `AtomicPart` entities for the used test database[1], like traversal 1, except that the traversed paths are different.

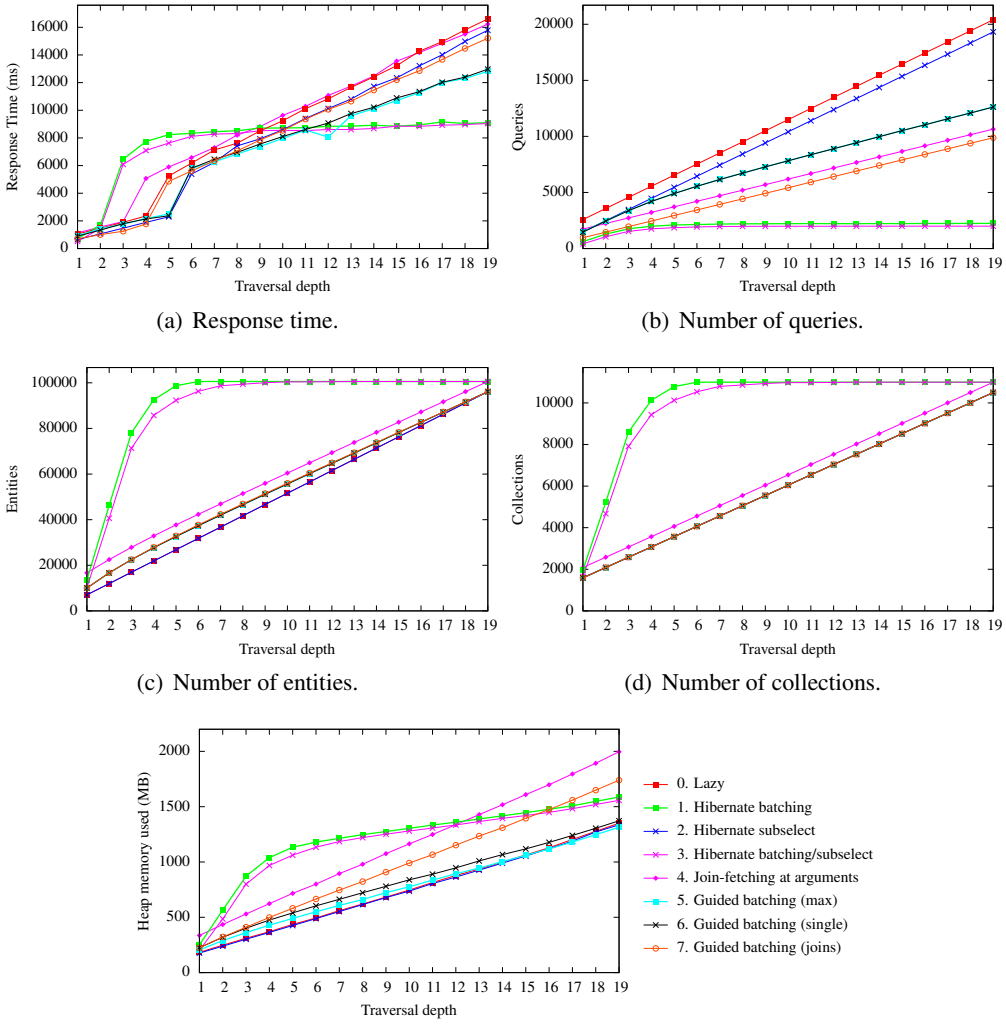
By increasing the traversal depth, the number of required entities and collection also increases. Using lazy-fetching the required number of queries, fetched entities and fetched collections will all increase linearly when the traversal depth is increased. Hibernate subs-

[1] 19 parts + 1 root part = 20 `AtomicPart` entities, which are all parts of a `CompositePart`, see Table 7.1.

elect is very similar to lazy-fetching for this test case, because only the assembly hierarchy is prefetched, however, that part does not change as the traversal depth is changed.

The prefetching techniques that use Hibernate batch-fetching are very slow for a low traversal depth. They do use the least amount of queries, at the cost of prefetching most of the `AtomicPart` entities early on, when not even half of the entities are required. Hibernate batch-fetching does become more interesting when more entities and collections are actually used, making it the best technique for a high traversal depth. However, because Hibernate batch-fetching is unable to take into account the traversal depth, it is unsuitable as a prefetching technique for a recursive traversal that does not traverse most of the recursive structure. It is hard to automatically determine if a recursive traversal will traverse most of the data structure, because that may require information about the database, which can change over time.

The other prefetching techniques are all able to take the traversal depth into account and that results in lower values for number of queries, entities and collections compared to lazy-fetching. The join-fetching techniques 4 and 7 fetch more duplicates, which results in more memory consumption making them less ideal. The performance of prefetching techniques 5 and 6 are very similar for this test case and provide the best overall performance, because they use fewer queries than lazy-fetching, without fetching many more entities, duplicates and collections.



(a) Response time. (b) Number of queries.

(c) Number of entities. (d) Number of collections.

(e) Heap used per request. (f) Legend.

Figure 7.15: Results for the TD test case.

Chapter 8

Evaluation with Applications

In this chapter we evaluate the performance of the previously discussed prefetching techniques on complete web applications. The evaluation is performed on Researchr, Yellowgrass, WebDSL.org and Weblab. These are all web applications that have been developed using WebDSL. The web applications and benchmarking method used in this chapter are better suited for the evaluation of prefetching techniques on web applications, when compared to the OO7 benchmark from Chapter 7. The version of Researchr that has been used here is more recent than the one used during initial testing, which was discussed in Chapter 5. This newer version contains more manual optimizations. Section 8.1 discusses the method used to benchmark the web applications. Section 8.2 discusses the results.

8.1 Evaluation Method

A set of URLs is required in order to benchmark the performance of a web application. This set of URLs is generated beforehand for each web application. The database is used to select entities for page arguments. For page arguments of a primitive type there exists a list of predefined test values. However, this list is not used when there is at least one suitable constant value inside the source of the page. Here a suitable constant means that there is an (in)equality comparison with the page argument. The web applications also have access control rules. However, access control errors are avoided by sending requests using the session of a user that has sufficient rights.

The applications use a copy of the database from their production environment. All these databases are stored inside one local MySQL server, which has its query cache disabled and is not restarted during benchmarking.

The applications are compiled multiple times using different prefetching techniques. The performance is measured by deploying each technique-application combination to Tomcat one at a time. Tomcat is started after a deploy and all the generated URLs are requested one by one, before stopping Tomcat again and continuing to the next technique-application combination. An URL is not requested multiple times in a row, because that is not a typical access pattern of a web application user. Successively accessing the same URL will introduce many opportunities for caching, which normally are less beneficial, because a

user typically visits an URL only a few times. Between requests Java is forced to perform garbage collection using `jmap -histo:live` .

Deploying all technique-application combinations and requesting their URLs once is called a benchmark run. The measured response time is discarded for the first benchmark run, because extra logging has been enabled with the `?logsql` URL suffix from Section 5.1. These first requests determine the number of executed queries, and the number of fetched entities, duplicate entities, and collections. After the first benchmark run, another ten runs are performed without extra logging enabled, measuring only the response time. Finally, the ten measured response times for each URL-application-technique combination are combined, by taking the average.

8.2 Results

The results are shown in Table 8.1. In this table the response times have been normalized, by dividing the response time of a technique by the response time of the lazy version, of the same URL. So 100% means that there was no improvement, everything below is faster and everything above is slower. If the response time differs only 5ms or less from lazy, then the normalized response time is also 100%, because the difference is not significant and is likely a measurement error. The table shows the total number of URLs for an application and the number of URLs that are faster and slower, followed by a percentage that shows how many URLs that are compared to the total. For both the faster and slower URLs an average is calculated, showing the average improvement for faster requests, and the average decrease in performance for slower requests.

Guided batching (single) is able to improve the performance for many URLs, when compared to the other prefetching techniques for an application, while the number of slower URLs is among the lowest. The average improvement is also more significant than the average performance decrease for slower URLs, except for Yellowgrass, however, the extreme low number of slower URLs makes up for this. The average performance decrease for slower URLs is so close to 100%, because the difference is often small and just a few milliseconds. The minor slowdown is caused by generating batches for smaller collections, making lazy slightly more efficient in these few cases. This is also the case for Weblab, where it makes the results look negative, because there are more slower than faster URLs. However, even for Weblab the faster URLs provide a more significant improvement. Compared to the other prefetching techniques guided batch-fetching also fetches the least duplicate entities. Duplicate entities can always be considered overhead. For WebDSL.org the number of duplicates is even lower than lazy, because of conditions that are added to queries.

Guided batching (max) shows that generating queries up to a maximum batch size beforehand can have a beneficial effect on the response time. This improvement becomes clear when comparing techniques 5 and 6 for Researchr. The other three applications do not show this improvement. The additional improvement of technique 5 on Researchr comes mostly from simple publication pages, showing only data that is relevant to a single publication entity. These pages already use a few number of queries and do not get optimized as much by

App	URLs	Technique	Faster # URLs (%)	Faster Avg %	Unchanged # URLs (%)	Slower # URLs (%)	Slower Avg %	Avg # SQL	Avg # Ent	Avg # Dup	Avg # Col
Researchr	705	0. Lazy						58.43	243.37	14.67	19.71
		1. Hib. batch	119 (16.8)	79.28	400 (56.7)	186 (26.3)	111.07	26.62	335.55	18.85	35.25
		2. Hib. subselect	48 (6.8)	83.25	529 (75.0)	128 (18.1)	16511.41	49.68	427.06	24.34	106.17
		3. Hib. batch/subsel.	55 (7.8)	78.65	409 (58.0)	241 (34.1)	6852.19	25.63	509.81	28.10	116.00
		4. Joins at arguments	22 (3.1)	84.27	172 (24.3)	511 (72.4)	1255.24	51.83	902.27	1061.08	128.14
		5. Guided batch (M)	189 (26.8)	84.35	498 (70.6)	18 (2.5)	114.00	24.16	247.00	18.47	25.13
		6. Guided batch (S)	145 (20.5)	79.70	536 (76.0)	24 (3.4)	112.04	20.54	246.49	18.47	25.13
		7. Guided batch (J)	153 (21.7)	81.46	520 (73.7)	32 (4.5)	111.59	21.08	252.12	197.63	25.10
Yellowgrass	179	0. Lazy						83.37	160.51	136.56	72.93
		1. Hib. batch	84 (46.9)	89.89	86 (48.0)	9 (5.0)	112.33	28.00	203.73	296.96	134.54
		2. Hib. subselect	4 (2.2)	83.75	2 (1.1)	173 (96.6)	441.08	33.45	201.35	307.48	130.34
		3. Hib. batch/subsel.	5 (2.7)	87.00	2 (1.1)	172 (96.0)	452.60	18.16	205.21	324.31	139.79
		4. Joins at arguments	1 (0.5)	97.00	1 (0.5)	177 (98.8)	143.70	135.40	200.93	276.15	125.68
		5. Guided batch (M)	89 (49.7)	90.36	81 (45.2)	9 (5.0)	109.33	31.89	200.93	276.63	124.98
		6. Guided batch (S)	163 (91.0)	87.20	12 (6.7)	4 (2.2)	116.75	21.28	200.93	276.63	124.98
		7. Guided batch (J)	162 (90.5)	86.38	12 (6.7)	5 (2.7)	113.00	21.28	200.93	276.63	124.98
WebDSL.org	94	0. Lazy						76.97	198.19	12.04	36.48
		1. Hib. batch	43 (45.7)	83.37	49 (52.1)	2 (2.1)	106.50	27.90	216.02	14.68	44.35
		2. Hib. subselect	2 (2.1)	95.00	37 (39.3)	55 (58.5)	297.93	76.97	198.19	12.04	36.48
		3. Hib. batch/subsel.	4 (4.2)	88.50	40 (42.5)	50 (53.1)	136.32	28.00	216.62	15.63	44.65
		4. Joins at arguments	9 (9.5)	92.67	42 (44.6)	43 (45.7)	116.58	69.05	185.33	58.04	40.32
		5. Guided batch (M)	49 (52.1)	85.02	42 (44.6)	3 (3.1)	111.00	23.10	184.29	0.54	40.19
		6. Guided batch (S)	49 (52.1)	83.94	42 (44.6)	3 (3.1)	112.33	19.17	184.29	0.54	40.19
		7. Guided batch (J)	24 (25.5)	88.38	63 (67.0)	7 (7.4)	102.43	39.76	181.07	62.15	36.48
Weblab	428	0. Lazy						55.94	61.27	17.82	14.09
		1. Hib. batch	49 (11.4)	81.22	302 (70.5)	77 (17.9)	111.73	32.49	79.24	44.77	33.98
		2. Hib. subselect	4 (0.9)	91.25	221 (51.6)	203 (47.4)	121.51	54.11	65.81	46.72	28.25
		3. Hib. batch/subsel.	45 (10.5)	80.96	237 (55.3)	146 (34.1)	128.68	32.40	78.30	56.32	41.78
		4. Joins at arguments	0 (0.0)		71 (16.5)	357 (83.4)	122.75	62.83	65.02	27.36	21.81
		5. Guided batch (M)	22 (5.1)	83.55	378 (88.3)	28 (6.5)	118.61	45.17	61.70	19.23	16.85
		6. Guided batch (S)	22 (5.1)	81.27	368 (85.9)	38 (8.8)	107.68	42.48	61.73	20.25	16.85
		7. Guided batch (J)	20 (4.6)	82.10	315 (73.5)	93 (21.7)	109.67	42.43	61.73	27.17	16.85

Table 8.1: Test results with normalized response times, where above 100% is slower and below is faster. The URLs are divided into the faster, unchanged and slower columns. The faster and slower columns also have a column showing the average normalized time. The final columns show the average number of executed queries, and the average number of fetched entities, duplicate entities and collections, for all URLs.

batch-fetching. Since the generated batches are small and do not have to be split up, Guided batching (max) can directly fit them into its prepared queries. Having to generate new queries for these small batches is too expensive. Therefore, technique 6 may be improved by only generating new queries for batches larger than 10 and using prepared queries for the smaller batches. Comparing the two techniques on Yellowgrass clearly shows that technique 5 has fewer faster URLs than technique 6. This is caused by the low maximum batch size, which significantly increases the average number of queries. Increasing the maximum batch size can make this technique faster, however, it is unlikely to improve the performance beyond that of Guided batching (single). Batch queries are normally not generated for every batch size, because of the memory requirement. Therefore, batches will still have to be split up, which still executes more queries than Guided batching (single), reducing the benefit of not having to generate a new query.

The performance of Guided batching (joins) is similar to Guided batching (single). However, the joins do fetch more duplicate entities, which is bad for performance. In Researchr there are a few cases where the joins perform better. These cases contain a recursion,

which is prefetched using joins. The joins trick guided batch-fetching into thinking that all properties have been prefetched by itself in a previous call, because it recognized that the join-fetched property has been prefetched already. This means that guided batch-fetching with joins stops earlier, which also results in more queries than guided batch-fetching without joins. This indicates that prefetching for up to four recursive calls is too aggressive for these cases.

Join-fetching at arguments is the final technique that uses the static analysis. This technique does not perform well. It is unable to improve the performance for any URL of Weblab and only one URL on Yellowgrass. An important reason is that this technique is unable to handle custom queries. When prefetching at the beginning of a function or template, then it is not possible to prefetch for custom queries inside of them, because those queries have not been executed yet. The analysis also does not provide information for it, because the custom query is not a traversal on one of the function or template arguments. As a result the called functions or templates within a for-loop will perform prefetching for their arguments. So the $1+n$ query problem is not always solved for custom queries. Join-fetching at arguments also fetches many duplicate entities, which is clearly shown in the results for Researchr.

Hibernate batching shows a performance improvement similar to guided batch-fetching, yet Hibernate batching has many more slower URLs for Researchr and Weblab. The URLs are slower, because more unused entities are fetched. These extra entities are fetched, because Hibernate can prefetch the entities for properties that are never accessed, as long as they have the same type as a property that is accessed. This is also the reason for the average number of fetched entities being higher, when compared to guided batch-fetching for all applications.

Hibernate subselect rarely improves performance, however, performance is decreased for a large number of URLs. Besides fetching more entities and collections, the decrease in performance is also caused by custom queries. These queries can be reused as a subquery, which reduces performance for expensive queries. Hibernate subselect can also use its own queries as new subqueries, causing the nesting of subqueries, making those queries even more expensive.

Combining Hibernate subselect with Hibernate batching does overcome the limitation that subselect-fetching can only prefetch collection properties. However, the combination of both Hibernate techniques still suffers from expensive subqueries. From the results it is clear that Hibernate batching alone performs better than the combination of both techniques.

Guided batching (max) and Guided batching (single) both show the best performance. Both techniques improve the performance on average, however, there are pages that become slightly slower. These pages would be better off with the lazy technique. A combination of both techniques, where some batch queries are generated at launch, will be even better, because generating new queries for small batches is too expensive. A more fine-grained method of choosing an optimization technique is required to improve the results further. Possible solutions are a more detailed static analysis to find patterns in templates where certain optimizations work best, or a more dynamic approach where the optimization technique can be swapped at run-time based on repeated benchmarks in the background.

Chapter 9

Related Work

Existing prefetching techniques are discussed in this chapter, which are divided into dynamic profiling and static approaches.

9.1 Dynamic Profiling Approaches

Dynamic profiling approaches gather and maintain extra information during runtime that is useful for prefetching. For example the AutoFetch, Fido and data compression approaches discussed below construct a probabilistic model that is used to predict future data accesses that should be prefetched.

9.1.1 AutoFetch

Ibrahim and Cook presented AutoFetch [11] that builds a probabilistic traversal profile for similar queries, by profiling the properties accessed on their results during runtime. The first time that a query is executed no prefetching is performed and a traversal profile will be constructed. When a query has a traversal profile and is about to be executed, then the traversal profile will be used to add joins to the query, to prefetch properties with a high probability of being accessed. Using joins instead of separate queries uses fewer queries and also increases the risk of fetching duplicates, as is discussed in Chapter 3. AutoFetch also prefetches a property for all entities returned by a query, where guided batch-fetching could use a detected condition to prefetch a property for only some elements. However, using probability to determine if a property should be prefetched allows AutoFetch to reason about complex conditions that are hard to translate into a query.

9.1.2 PrefetchGuide and Type-level Access Pattern

Han et al. [9, 10, 8] do not change queries and instead detect iterative and recursive access patterns on their results. The detected patterns can then be used to perform prefetching. They build a type-level path graph from a root node that represents the results of a query. In this graph the nodes are the types of the entities and the edges are property accesses. If a property access occurs that is already in the graph, then an iterative pattern has been detected

and the graph can be used to prefetch entities for the remaining iterations. The same is done for recursive patterns, which are detected when a cycle is introduced inside the type-level path graph. When a property is accessed that is not in the cache, then it requires fetching and prefetching will be performed for the captured access patterns on that property. This method of prefetching has also been refined to create views inside the database, minimizing disk I/O during prefetching [8].

The downside of this method is that once a property is recorded in an access pattern, then it will get prefetched, even when that property may only be required in very few cases. A probabilistic model, like the one that is used by AutoFetch, avoids this by assigning a low probability to such a property. Guided batch-fetching tries to prevent this by evaluating conditions during batch generation or by placing conditions inside queries.

9.1.3 Context-based Prefetch

Bernstein et al. [1] proposed prefetching properties for all entities in a context, whenever a property is accessed on one of them. Entitiess within the same context are similar, because they are all fetched by the same query or belong to the same collection. The set of identifiers, for entities within a context, are not placed inside queries, instead they are stored inside a temporary table, giving every context an identifier within the database. Using the context identifier and the temporary table in queries makes queries smaller for larger contexts, which may be faster and avoids a possible limit on query size, at the cost of inserting a context into the temporary table.

Context-based prefetch does not perform well when properties are only accessed for some entities within a context, just like PrefetchGuide. Guided batch-fetching can do better, by detecting the conditions under which properties are accessed. Guided batch-fetching also uses the actual collections that an applications loops over, when prefetching properties, instead of remembering the context in which an entity was fetched. Using the actual collection for prefetching is important when the context of an entity is changed, for example, when entities are removed from or added to query results.

9.1.4 Fido

Palmer and Zdonik [14] used a learning algorithm to predict future entity accesses. Given the last few entities accessed, the algorithm may recognize a previous pattern. The pattern may contain entities that are not inside the cache and these entities can be prefetched. They also used their own eviction policy, which tries to predict which cached entities are likely to be used last, so that those entities can be removed from the cache, making space for new entities. Since patterns are detected on entity instances, this approach only works well if the same entities are accessed repeatedly, because a pattern detected on an entity instance does not apply to other entities of the same type.

9.1.5 Data compression

Curewitz et al. [6] proposed using compression algorithms to detect patterns in database page accesses. Compression algorithms try to determine the probability that a character se-

quence will occur and encode sequences with a high probability using fewer bits. Curewitz et al. assign a character to each page, so that a page access sequence can be encoded as a character sequence. This allows a compression algorithm to be executed on a sequence of previous page accesses, and the resulting probabilities can be used to predict future page accesses. This technique only works well if the same pages are accessed repeatedly.

9.1.6 Sprint

Raman et al. presented Sprint [15], which creates and optimist and a pessimist version of an application. The pessimist is similar to the original program, except that it has points where it spawns the optimist. The optimist serves as a prefetcher for the pessimist and is executed on another thread. The optimist excludes code that reads input or has external side effects, like writing to a file. The optimist stays ahead of the pessimist by parallelizing one or more loops or recursive calls, making the optimist multi-threaded. The loops and calls to parallelize are annotated manually. A method is described for automatic annotation placement. The optimist adds entities to a batch instead of fetching them directly. The batch is fetched when the batch is full or when the pessimist forces it. As a result the pessimist is likely to read values from the cache, instead of remote servers. The optimist can speculate on input values and return values of recursive calls, however, this speculation can cause the optimist to prefetch the wrong data. This is detected by monitoring cache misses and restarting the optimist using the current data from the pessimist.

Sprint can be seen as another approach to batch generation and in that way it is similar to guided batch-fetching. Since Sprint uses an altered version of the original program to generate batches, it can use more complex conditions during batch generation. However, the evaluation of those conditions is not always accurate, because of the speculation about the values.

9.2 Static Approaches

Static approaches perform optimizations during compilation. This can be the reordering of operations, in order to bundle remote operations as with remote batch invocation, or a source code analysis from which prefetch queries are generated as with Query Extraction.

9.2.1 Remote Batch Invocation

For remote batch invocation [12, 5] a `batch` statement is added to Java, which specifies a remote root and a block of statements. The block of statements can contain a mix of local and remote operations. The operations are reordered, possibly duplicating loops and conditions, so that remote operations can be performed in one round trip to the server, which is only possible if there are no back and forth interdependencies between local and remote operations. The remote operations are then translated into a batch script that can be send to the server. Remote batch invocation can be used to optimize queries by translating these batch scripts to SQL queries. The returned values can then be used to execute the local operations. The advantage of remote batch invocation is that it can batch both read

and write operations. Aggregation operations are also supported, when they are performed using the provided functions, so that aggregation can be easily detected. The downside is that batches should be specified manually.

9.2.2 Query Extraction

Wiedermann et al. [19, 20] proposed a source-to-source transformation that transforms an object-oriented program that accesses persistent data transparently, into an equivalent program that contains explicit queries. Query extraction has two stages. First a path based analysis makes a description of the persistent data each method requires. Then the program is transformed so that each method can execute an explicit query that prefetches the persistent data specified by the analysis. The static analysis described in Chapter 6 is based on this technique, which is why these papers are discussed in more detail.

Traversal Summary

The analysis phase uses paths to describe persistent values that are used by a method. A path consists of four components:

1. A persistent root, from which a method can reach other persistent values.
2. A sequence of field names that the method traverses on the persistent root to reach a persistent value.
3. The condition(s) under which the method performs the traversals.
4. A data dependence flag, that indicates if the source statement or expression of the path can affect non-local state.

A set of paths is called a traversal summary.

Persistent root

Query extraction models the persistent data store as a rooted directed graph of persistent records, which represent persistent entities. The root of this graph is represented by a special root variable, which is a persistent root. Persistent method arguments and persistent values returned by method calls are also considered persistent roots. The analysis phase describes the persistent values a method uses, relative to these persistent roots.

Query Condition

Some persistent values are only required under certain conditions. For example, the condition of an if-statement determines if the persistent values that are used within the if-block are required. These conditions may be attached to paths if they can be shipped to the database as conditions inside a query, to filter the persistent data that is fetched. A condition can be placed inside a query if it meets the following requirements:

```
1 for(Department d : root.getDepartments()) {
2   for(Employee e : d.getEmployees()) {
3     if(d.size > 100 || e.salary > 35000)
4       System.out.println(d.name + " - " + e.name);
5   }
6 }
```

Figure 9.1: An example of two nested loops with a master-detail relationship.

1. The condition may contain only portable operators. Operators are portable if they can be executed and have the same semantics in both the database and the program. This may need some translation. For example, to make the handling of null values consistent.

2. The condition can be evaluated on every element of a collection, independently of all other elements in the collection.

3. Conditions in nested loops must participate in a master-detail relationship. The inner-loop is a detail of the outer-loop, if the inner-collection is a traversal on the iterator variable of the outer-loop. The condition on line 3, in figure 9.1, is a query condition, because the nested loops have a master-detail relationship.

4. Local variables used by the condition must be bindable. A variable is bindable if its last assignment occurs before the execution of the query that uses it. The Java prototype only executes queries at the beginning of methods, so in that case final arguments are always bindable.

Data Dependence Flag and Iterators

The data dependence flag is true for any statement or expression that can affect non-local state. For example, the statement on line 5 of Figure 9.2 is data dependent, because it increments a variable outside of its local scope. This statement does not use persistent data directly, however, it does depend on the existence of employees that have at least a certain salary. As such, it has an effect on the employees that need to be fetched. The analysis records data dependencies like this, by generating an extra path for every data dependent expression. This extra path records the data dependence on the last iterator in the iterator context. The iterator context maintains a list of inner-iteration variable paths that extend outer-iteration paths. The iteration context is also used to check the master-detail restriction for query conditions. The paths that the analysis would generate for Figure 9.2 are in Table 9.1. The last path in the table is generated by the statement at line 5. The path has the root and traversal of the iterator in the iterator context and the condition and dependence flag of the statement. The iterator is included in the paths to symbolize a traversal on the elements of a collection.

```
1 int countBySalary(final int salary) {
2   int count = 0;
3   for(Employee e : root.getEmployees()) {
4     if(e.salary >= salary) {
5       count++;
6     }
7   }
8   return count;
9 }
```

Figure 9.2: A Java method that counts the number of employees that make a certain salary.

Root	Traversal	Condition	Dependent
root	employees	true	false
root	employees.e	true	false
root	employees.e.salary	true	false
root	employees.e	root.employees.e.salary >= salary	true

Table 9.1: Traversal summary generated from Figure 9.2.

Method Calls

How is dealt with method calls depends on whether the called implementation is known statically. This is always the case for static or final methods, yet not for virtual methods. Virtual methods may be overridden by subclasses and the called implementation is determined at runtime, based on the actual type of the entity on which the method is called. If none of the subclasses override a virtual method, then the called implementation is also known statically, under the assumption that no additional classes are loaded and classes remain unchanged. This is called devirtualization.

If the called implementation is known statically, then the caller can prefetch the traversals made on the persistent method arguments of the callee. This is done by combining all the paths that represent an argument in the caller, with every path that is rooted at the argument in the traversal summary of the callee. The resulting paths should also be prefetched by the caller. If a path is defined as a $(r, \overrightarrow{f}, c, d)$ tuple, where the variables represent the root, fields, condition and data dependence flag, respectively, then two paths can be combined as follows:

$$(r, \overrightarrow{f}, c, d) \circ (r', \overrightarrow{f'}, c', d') = (r, \overrightarrow{f}.\overrightarrow{f'}, c \wedge c', d \vee d')$$

The left path represents a value that the caller passes on to the callee, and the right path is a path in the callee that is rooted at the corresponding argument. After combining conditions, they may contain arguments and local variables of the callee. Arguments should be replaced by their value in the caller and conditions containing local variables should be changed to true.

For method calls that cannot be devirtualized, only the traversals made to compute the arguments are prefetched by the caller. These traversals are also marked as data dependent.

The callee should perform its own queries to prefetch data to support traversals made from method arguments.

Return Values

Every method that returns a persistent value is changed to accept extra arguments. This allows a caller to pass on its traversals on the return value at runtime, which are detected by treating method calls as persistent roots. The callee combines these traversals with its own traversal summary and generates new prefetch queries that support these traversals. After combining the traversals, the conditions may reference local variables of the caller, which the caller also passes on to the callee. A caller can also prefetch its own traversals on a return value, when the return value of a callee is a traversal from one of its method arguments, also called a pass-through traversal. Another requirement is that the callee is not a virtual method or that it can be devirtualized. When a caller has performed prefetching for pass-through traversals, then the callee only has to perform addition prefetching for return values that are not pass-through traversals.

Recursion

For recursive method calls the traversal summary is not combined with itself recursively, because that will never terminate. Instead the traversals that are made from its arguments, to compute the arguments of the recursive call, are marked as recursive traversals in the traversal summary. When generating a query these recursive paths are unfolded a finite number of times.

Shortcomings

Query extraction may not prefetch all persistent data that is used by a method, because persistent values can escape through object properties. This does not break the program, because these values are fetched automatically by the persistence architecture through lazy-fetching.

The persistence framework does not know about filtered collections, so if a persistent collection escapes, then the persistence framework will not automatically reload the collection as an unfiltered collection. This changes the program when a filtered collection is used by a part of the program that expects a different filtering. One solution is the remove conditions from paths that represent collection values. Alternatively, it can be detected when a collection escapes and a reload can be forced when that happens, which is what guided batch-fetching does.

Difference with Guided batch-fetching

Query extraction executes queries at the beginning of methods, where guided batch-fetching executes queries just before for-loops. Query extraction uses join-fetching to prefetch properties, while guided batch-fetching executes another query instead of placing a join, to prevent duplicate entity fetches. Despite these differences both approaches use a similar static

analysis. However, the static analysis from Chapter 6 also considers a persistent for-loop iterator to be a persistent root, in order to collect information about individual for-loops. The static analysis for guided batch-fetching also makes some simplifications. For example, local variables are not allowed to occur in conditions and property accesses on values returned from method calls are only recorded for pass-through traversals. These features can be added, however, they are less important for WebDSL, because templates only allow variable assignment at the beginning of a template and templates do not return values.

Chapter 10

Conclusions and Future Work

This chapter will use the results of the evaluation to answer the research questions. Finally, some ideas for future work will be discussed.

10.1 Conclusions

In this thesis project, multiple prefetching techniques are implemented in the WebDSL compiler. Some simply enable the techniques that are included in Hibernate, while the others generate extra prefetching code, based on a static code analysis. All these techniques have in common that they try to reduce the response time by executing fewer queries. Some optimizations that are unrelated to prefetching are also implemented, as discussed in Appendix A, because without them negative effects of prefetching techniques could be exaggerated. The prefetching techniques are evaluated using the OO7 benchmark in Chapter 7 and using complete WebDSL applications in Chapter 8, which we will use to answer the following research questions.

How to best prefetch data under various circumstances?
As discussed in Chapter 3 there are two types of queries that can fetch more than one entity or collection at a time, join-fetch queries and queries using a less restrictive condition, like batch-fetch queries. Join-fetch queries contain joins to prefetch properties of the entity being fetched and batch-fetch queries use a list of identifiers to fetch multiple entities or collections using a single query. Hibernate subselect-fetching uses sub-queries inside conditions, which also results in a less restrictive condition.

Using multiple collection joins in a query has the risk of causing Cartesian products, which results in much data duplication in the recordset and can easily lead to an out-of-memory exception. To avoid these, collection properties are never join-fetched by the prefetching techniques implemented in WebDSL. The evaluation shows that the techniques that do use join-fetching for single-value properties are generally not the fastest, because they fetch more duplicate entities. Entities are fetched again, because a join always fetches the related entity, even when that entity has already been fetched.

Join-fetching is still useful when batch-fetching cannot be applied. For example, when

finding the root of a tree, by accessing the single-value property `parent` recursively. Using joins for recursive single-value properties can reduce the number of queries executed. However, the reduction in number queries should be significant in order to see any benefit, which would require a deep recursive data structure that is not often found in web applications.

When do the prefetching techniques included in Hibernate improve performance?
The prefetching techniques included in Hibernate improve performance when the same properties are accessed for all elements of a collection. Accessing a property on only some elements is likely to prefetch that property for other elements as well. As shown in subsections 7.2.6 and 7.2.7, this can result in increased response times for Hibernate batch-fetching, caused by fetching too many entities. Entities are prefetched based on their type, allowing any uninitialized proxy of the same type to be prefetched, even if it is unrelated to the property being accessed. Therefore, Hibernate batch-fetching is also slower when an entity has multiple properties of the same type and only one of them is used.

Hibernate subselect-fetching also prefetches a property for all elements in a collection. This technique is often similar to lazy-fetching, because it only applies to collection properties. On the OO7 benchmark, subselect-fetching performed well, because a full traversal of the assembly hierarchy is an ideal situation for subselect-fetching. However, when benchmarking the performance on complete applications, then the subqueries were often too expensive or prefetched too many entities.

Combining both Hibernate techniques does overcome the inability of subselect-fetching to prefetch single-value properties, yet it also inherits the flaws of both techniques. Meaning that subqueries can be quite expensive and the combination of techniques is likely to fetch even more entities than the individual techniques.

How do the prefetching techniques implemented in WebDSL compare to the techniques provided by Hibernate?
The prefetching code generated by WebDSL relies on a static code analysis from Chapter 6, to determine the properties that are accessed on a variable. The entities and collections fetched by guided batch-fetching reflect the actually used ones better than the entities and collection fetched by the Hibernate techniques. The match between fetched and used entities is better, because batch generation actually accesses the properties for which a batch is being generated, unlike Hibernate batch-fetching. The static analysis also detects conditions under which entities are used. These conditions are sometimes evaluated during batch generation or they are placed inside queries. Placing conditions in queries can even reduce the number of fetched entities below that of lazy-fetching.

From the four techniques that are implemented in WebDSL, two use join-fetching, which is bad for performance and memory consumption, because they fetch more duplicates. The other two are variations of guided batch-fetching and differ mostly in the number of queries used. Using a single query per generated batch, or by first splitting batches into smaller batches with a maximum size of 10. Generating a new query per batch is slightly more expensive, yet it guarantees only one query per batch. Generating new queries for small batches is too expensive, however, this can be prevented by combining both variations, using the static batch queries only for those smaller batches. Both techniques improve the

performance on the evaluated web applications on average, yet the results may be improved further with a more fine-grained method of choosing a prefetching technique.

Besides showing better performance than the Hibernate techniques, the guided batch-fetching techniques also provide more opportunities for a developer to influence the optimizations. The Hibernate techniques can be enabled for individual entities or properties, and that setting will affect the entire application. The WebDSL techniques allow a developer to place a manual prefetch specification, to add properties and conditions that the static code analysis missed. Adding a prefetch specification does not affect the entire application, because it is defined for a specific variable. The prefetch specification syntax can also be extended in the future to provide even more control. For example, by allowing a developer to force a sub-property to be join-fetched, or to exclude a property from prefetching.

10.2 Future Work

In this thesis project we have seen multiple techniques that apply prefetching, with as goal to improve performance, by reducing the number of queries. Using the second-level cache can also improve performance and reduce the number of queries, however, has not been thoroughly evaluated. Appendix A shows how the second-level cache can be used, yet leaves caching as an option to be enabled manually. Caching can be used in combination with prefetching. One way is to enable caching for all entities and collections and by trying to load a batch from the cache, before fetching any remaining elements using a query. This may still require a similar amount of queries, because entities and collections are evicted from the cache before they are required again. This may be avoided by caching only those entities and collections that are likely to be used soon or frequently. One method is to identify frequently called templates and to generate code that caches any entities or collections they use. Another way is to look at the links on a page to determine the next possible requests and to cache entities and collections they have in common.

Currently a WebDSL application is compiled using only one of the implemented prefetching techniques. The evaluation shows that no technique outperforms the others in every situation. Therefore it may be better to use different techniques in different situations, which does not need to be limited to techniques that are currently implemented and can, for example, also include techniques from related work (Section 9). The evaluation also shows that a combination of techniques can be better than its parts, which is the case for Hibernate batch-fetching and Hibernate subselect-fetching on the OO7 benchmark. An example of another possible combination would be guided batch-fetching with a profiling technique like AutoFetch [11], where the probabilities from the profile can be used by guided batch-fetching. This combination can avoid the prefetching of properties that are rarely used. The selection of a prefetching technique to be used does not have to be static and could be made dynamic, for example, by running a benchmark in be background. Dynamically selecting a technique may be useful for recursive structures, where it can be unclear how well a technique will perform. For example, when cycles are not present or very rare, then join-fetching may be the most effective technique. With many cycles join-fetching is likely to fetch many duplicates and another technique will be better suited.

Bibliography

[1] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch – an optimization for implementing objects on relations. *The VLDB Journal*, 9(3):177–189, December 2000.

[2] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 414–426, New York, NY, USA, 1994. ACM.

[3] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 12–21, New York, NY, USA, 1993. ACM.

[4] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. Technical report, University of Wisconsin-Madison, January 1994.

[5] William R. Cook and Ben Wiedermann. Remote batch invocation for sql databases. In *DBPL 2011*, DBPL '11, 2011.

[6] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 257–266, New York, NY, USA, 1993. ACM.

[7] The Apache Software Foundation. ab – Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html` , 2012.

[8] Wook-Shin Han, Woong-Kee Loh, and Kyu-Young Whang. Type-level access pattern view: Enhancing prefetching performance using the iterative and recursive patterns. *Information Sciences*, 180(21):4118–4135, November 2010.

[9] Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. PrefetchGuide: Capturing Navigational Access Patterns for Prefetching in Client/Server Object-Oriented/Object-Relational DBMSs. *Information Sciences*, 152(1):47–61, June 2003.

[10] Wook-Shin Han, Kyu-Young Whang, and Yang-Sae Moon. A formal framework for prefetching based on the type-level access pattern in object-relational dbmss. *IEEE Trans. on Knowl. and Data Eng.*, 17(10):1436–1448, October 2005.

[11] Ali Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *ECOOP 2006 Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 50–73. Springer Berlin / Heidelberg, 2006.

[12] Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 595–617, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole, and Hardy Ferentschik. *Hibernate Reference Documentation*, 2011.

[14] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *17th International Conference on Very Large Data Bases*, pages 255–264. Morgan Kaufmann, September 1991.

[15] Arun Raman, Greta Yorsh, Martin Vechev, and Eran Yahav. Sprint: speculative prefetching of remote data. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 259–274, New York, NY, USA, 2011. ACM.

[16] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[17] Eelco Visser. Program transformation with Stratego/XT. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer Berlin Heidelberg, 2004.

[18] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer Berlin Heidelberg, 2008.

[19] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 199–210, 2007.

[20] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 19–36, 2008.

Appendix A

Other Hibernate Optimizations

This chapter presents other optimizations that have been implemented in WebDSL. While these optimizations are unrelated to prefetching, they do influence test results. Section A.1 explains how the number of automatic flushes can be reduced. Section A.2 is about optimizations that prevent the fetching of unused entities. Without these optimizations the negative effect of prefetching too many entities can be exaggerated, because flushing will take longer and more uninitialized proxies may be fetched unintentionally. Section A.3 describes how Hibernate caching is used. Caching was used incorrectly, causing cache misses, resulting in more lazy-fetch queries.

A.1 Automatic Flushing

Flushing is the operation that writes new and changed entities back to the database. New entities could have been saved explicitly or they could have been assigned to a property of an existing entity. Entities have changed, when a value of one of its properties has changed. Therefore, Hibernate has to look at the properties of all loaded entities to find all the new and changed entities.

Flushing is not only performed automatically when committing a transaction, because flushing is also required before sending queries to the database, like those generated by the HQL or Criteria APIs. Without flushing the queries send to the database will run on old data and will not return the expected entities. For example, when saving a new entity, then that new entity cannot be found by queries until a flush has occurred. However, if nothing needs to be flushed to the database, then flushing is only overhead, because checking for new or changed entities is done needlessly.

Flushing before queries can be avoided, when it is known that there are no new or changed entities. Automatic flushing can be disabled by default, because initially there are no new or changed entities. Automatic flushing can be enabled again inside the setters and save-functions of entities, because those are the operations that will change entities or add new entities, respectively. After a flush has occurred, automatic flushing is disabled again, until another setter or save-function is called.

A.2 Unintentional Entity Fetching

When working with an ORM framework like Hibernate it is possible to unintentionally fetch entities. This is not desirable, because entities that are fetched unintentionally are not used at all and cause unnecessary overhead. The following subsections all explain optimizations that reduce the fetching of unused entities.

A.2.1 Load versus Get

One way to fetch an entity unintentionally is by calling the `get` method on the Hibernate session instead of using the `load` method. It is easy to mix up the two, because the difference is very subtle. The `load` method creates a proxy, which will not fetch the entity until it is used, while the `get` method will fetch the entity immediately. Therefore the `load` method should be used whenever possible, because this will not fetch entities that are not used.

However, the `load` method is not always the right choice, because in some situations where it is required that an entity is fetched immediately. For example, when an entity might be used after the Hibernate session has been closed. If a proxy is first used after its session has been closed, then it will raise a `LazyInitializationException` , because the entity cannot be fetched from a closed session. The `get` method will return the actual entity, and not a proxy, which can be used after the session has been closed. However, lazy properties may still contain proxies, so using entities without an open session still requires caution.

Another problem with the `load` method is that the existence of the loaded entity is not checked until it is used. The `load` method never returns a `null` value and always returns a proxy. The proxy will try to fetch the actual entity when it is used, which will raise an `ObjectNotFoundException` when that entity does not exist. So when it is uncertain if an entity exists, then it is better to fetch that entity using the `get` method. The `get` method will return a `null` value when an entity does not exist, allowing for existence checks.

A.2.2 Event Listeners and Interceptors

Another way to unintentionally fetch an entity is when custom event listeners or interceptors use entities that are passed on to them. Event listeners and interceptors can both be used to alter or extend Hibernate functionality. Most events have a default implementation, one example is the `DefaultFlushEntityEventListener` , which performs dirty checking and updates during flushes. One way to affect this implementation is by creating an interceptor that implements the `findDirty` method, to perform custom dirty checking. An alternative would be to replace the default event listener with a custom implementation.

Some event listeners or interceptor methods take an entity as argument, however, such an entity may also be an uninitialized proxy. Custom event listeners or interceptors should be careful not to initialize such proxies needlessly. Take for example, a custom subclass of the `DefaultSaveOrUpdateEventListener` that ensures that all entities have a version number of at least one before they are saved. That way a version number of one indicates that the entity has been saved. This custom event listener could fetch unused entities, because the `save-update` event is also raised for uninitialized entities and checking the

version number of those entities would initialize them. However, uninitialized entities have already been saved and therefore already have a version number of at least one. So the correct approach here is to skip the version number check for uninitialized entities.

A.2.3 Bidirectional One-to-One Associations

Take, for example, an entity with both a `previous` and `next` property, both referencing another entity of the same type. If a`.previous.next` and a`.next.previous` both always return entity a, given that there are no `null` values, then both properties are the inverse of each other. This type of relation is called a bidirectional one-to-one association and only requires one side of the association to be stored inside the database. Let's say that only the `previous` property is store inside the database, then that property is called the owning side. The `next` property is then the inverse side. In order to fetch the `next` property of entity a, a query needs to find an entity b, where the `previous` property points back to entity a. Only storing one side of the association enforces the inverse relation between the two properties.

The downside of storing only the owning side is that it becomes harder to configure lazy-fetching for the inverse side. For the owning side the property is stored inside the database row of the entity. If the column of the property is not `NULL` , then it is the identifier of the associated entity and that identifier is used to create a proxy for the property. For the inverse side this common scenario does not apply, because the association is not stored inside the row of the entity. Creating a proxy for the inverse side is now impossible, because the identifier of the associated entity is unknown, and more importantly, because it is unknown if there even is an associated entity. If there is no associated entity, then the inverse side property should be `null` , which it never is when a proxy has been assigned. Instead of checking the database for the existence of an associated entity, the entity is fetched immediately, because the fetch query is about as expensive and will prevent an extra round trip to the database, when the entity is lazy-fetched later on. This is why Hibernate eagerly fetches the inverse side of bidirectional one-to-one associations by default. When a single query fetches multiple entities with these inverse side properties, then the eager fetching will still use separate queries for each inverse side property of every entity, causing the $1+n$ query problem (Chapter 1). The $1+n$ query problem is even present when none of the fetched properties are used. This can be enough reason to store both properties inside the database, using two unidirectional one-to-one associations that do not enforce the inverse relation inside the database. Using two unidirectional associations is a performance trade-off, because unused entity fetches are reduced, at the cost of storing slightly more data and risking database inconsistencies.

Another solution is to not use proxies and instead change the getters and setters of inverse side properties, to fetch the associated entity the first time they are called. By adding the `@LazyToOne(LazyToOneOption.NO_PROXY)` annotation to a property, Hibernate knows that the getter and setter perform lazy-fetching. Hibernate expects that a bytecode instrumentation task is executed after compilation that changes the getters and setters, however, the instrumented code will fetch all `NO_PROXY` properties on the first `NO_PROXY` getter or setter that is called. Instead it is possible to implement the `FieldHandled` inter-

face on entity classes, which Hibernate takes as a sign of byte-code instrumentation. After that `NO_PROXY` properties are not fetched by Hibernate automatically and getters can perform their own lazy-fetching, without fetching other `NO_PROXY` properties. This does not work for the setters, because then changes would not get flushed back to the database. So setters still do what byte-code instrumentation did and fetch all `NO_PROXY` properties. After these changes the inverse side properties of bidirectional one-to-one associations are almost lazy. It is not completely lazy, because the associated entity is fetched by the getter, instead of the getter returning an uninitialized proxy. Setters are also not completely lazy, because they may fetch other `NO_PROXY` properties that remain unused.

A.3 Caching

Hibernate has three different caches. The session, second-level and query cache. The session cache keeps track of all entities that are managed by that session and is always enabled. The second-level cache can be used to cache entities across sessions. The query cache is similar to the second-level cache, yet caches identifiers of query results instead of entities. In this section we explain how the use of the second-level and query caches are improved.

A.3.1 Second-level Cache

The second-level cache can be used to cache entities across sessions. Caching for a Java entity class can be enabled by adding a `@Cache(...)` annotation, that should at least specify a caching strategy. The caching strategies vary in performance and the way they handle concurrent updates. The `read-only` strategy, for example, is very fast, yet does not handle updates, making it perfect for entities that do not change. The `read-write` strategy does update the cache to reflect changes made to entities. Write locks are used to prevent concurrent updates. As a result, the `read-write` strategy is slightly slower than `read-only`. In WebDSL the `read-write` strategy is always used, because it is a safe default. Caching can be enabled on a WebDSL entity by placing the `cache` keyword somewhere within the entity declaration.

Sub-entities always inherit cache settings from their super entity, so `@Cache(...)` annotations on sub-entities are always ignored. Cached entities are automatically put into the second-level cache by Hibernate as soon as they are fetched. The actual caching is handled by a cache provider, which may have more configuration options. Common examples are the cache size and eviction policy options. The cache size could, for example, specify the number of entities that may be cached at one time. And an example of an eviction policy is least-recently-used, which allows the least recently used entity to be evicted from to cache to make room for a new one. In most cases it should be left to the developer to specify which entities should be cached, because the trade-off between performance and memory usage is hard to make automatically. However, when generating code there may be easy ways to identify entities that benefit from caching. In WebDSL, for example, entities are generated for enumeration types. Enumeration types typically have few instances, which are also small and that is why they probably benefit from caching. Therefore, enumeration types are cached by default.

Caching entities alone is not always enough, because collection properties with entity elements are not cached by default. This can be enabled by adding a `@Cache(...)` annotation to the property in Java, just like with entities. It is important to note that this caches just the identifiers of the elements and not the actual entities. The cached identifiers are used to create a proxy for every element of the collection. The entities of these proxies are not fetched until they are used. A program will use them one by one, causing a lazy-fetch query for every proxy, resulting in the $1 + n$ query problem, unless all entities are found inside the session or second-level cache. Without caching the collection, all of its elements will be fetched using only one query, which is more efficient. That is why collections with entity elements should only be cached when all of its elements are also cached inside the second-level cache.

Simply enabling caching for the element-type of a collection, is not enough to ensure that all of its elements are cached inside the second-level cache. Since elements of a collection may be evicted from the cache, in order to make room for new entities of that type, without evicting the cached collection. As a result, lazy-fetch queries will still be executed for evicted elements. In order to avoid this problem, caching should only be enabled on entities when the cache is large enough to hold all of its instances. Under the assumption that a developer follows this guideline, collection properties are automatically cached when their element-type is cached.

A.3.2 Query Cache

The query cache only stores the identifiers of resulting entities, which makes caching queries similar to caching collections. It can also be automatically enabled for queries that return cached entities, under the assumption that there is enough room inside the cache to store all instances of the entity-type. There is, however, an extra complication for queries, because queries may eagerly fetch properties of resulting entities. These eagerly fetched entities may not be cached and the optimization that was intended by the query is lost when the results are loaded from the cache. Therefore, queries are not cached when they eagerly fetch properties and it is assumed that the optimization performed by the query is more important than caching the results.