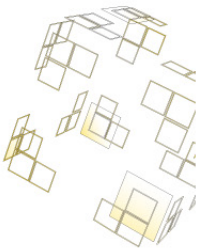
MASTER THESIS

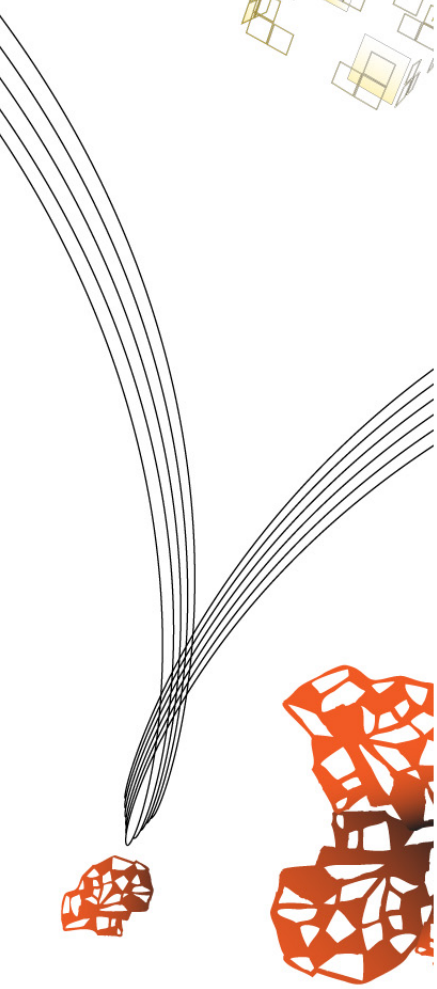
EVALUATING DATA STRUCTURES FOR RUNTIME STORAGE OF ASPECT INSTANCES



Andre Loker

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE (EEMCS)
CHAIR OF SOFTWARE ENGINEERING

EXAMINATION COMITTEE
Dr.-Ing. Christoph Bockisch
Dr. Maya Daneva



UNIVERSITEIT TWENTE.

OCT 2014

Abstract

In aspect-oriented execution-environments, aspect instances represent the state that is shared between multiple invocations of advice. Instantiation policies are responsible for retrieving the correct aspect instance for a specific execution of advice. Because this retrieval potentially happens many times during the execution of a program, it should be fast. In previous work, we have developed a unified model of aspect-instantiation policies that describes the semantics of instantiation policies independent of implementation details such as the underlying data structures. Strategies to optimise the execution speed of aspect-instance retrieval using JIT-compilation have been presented in previous work by Martin Zandberg. For specific instantiation-policy semantics, these strategies generate optimised machine code for the look-up procedure. The choice of data structures used to store the aspect instances is mostly left as an implementation detail. This choice, however, affects the execution speed of aspect-instance look-up and the memory footprint of the application. In this thesis, we evaluate different data structures for use as storage for aspect instances with respect to look-up speed and memory usage. Based on a benchmark, we suggest a two-level approach to implement aspect-instance storage: on a baseline level, data structures such as arrays, hash tables and prefix trees provide a widely applicable, but still fast solution, while on a second level, highly specialised data structures and access algorithms allow for even faster retrieval in certain special cases.

Contents

1 Introduction **1**
- 1.1 Problem statement and goals . . . 2
- 1.2 Approach . . . 3

2 Background and Related Work **5**
- 2.1 Instantiation policies . . . 5
  - 2.1.1 First responsibility: Aspect-instance retrieval . . . 5
  - 2.1.2 Second responsibility: Restriction . . . 8
- 2.2 Existing instantiation policies . . . 8
  - 2.2.1 Association aspects . . . 8
  - 2.2.2 Per-object instantiation . . . 13
  - 2.2.3 Singleton aspects . . . 16
  - 2.2.4 Other instantiation policies . . . 17
- 2.3 More related work . . . 19

3 Research objectives **20**
- 3.1 Research questions . . . 22
- 3.2 Approach . . . 25

4 Unified model of instantiation policies **28**
- 4.1 Definition of key tuples and query-key tuples . . . 30
- 4.2 The bind function . . . 31
- 4.3 Queries and the *find* function . . . 31
- 4.4 The *store* function . . . 34
- 4.5 isExact . . . 35
- 4.6 Implicit instantiation . . . 35
- 4.7 Generic Storage Function . . . 35
- 4.8 Summary . . . 37

5 Criteria **39**
- 5.1 Asymptotic time complexity . . . 39
- 5.2 Actually required time . . . 41
- 5.3 Memory usage . . . 45

6 Scenarios **46**
- 6.1 Aspect-instance look-up . . . 46

7 Data structures **49**
- 7.1 Arrays . . . 50
- 7.2 Linked Lists . . . 54

7.3 Tree-based data structures . . . 57
7.4 Tries . . . 60
7.5 Hash-based data structures . . . 63
7.6 Summary . . . 67

8 Benchmark **71**
8.1 Statistically rigorous performance evaluation . . . 71
8.2 Implementation of the statistically rigorous methodology . . . 72
8.3 More implementation details . . . 73
8.4 Selected data structures . . . 74
8.4.1 ArrayStorage . . . 74
8.4.2 HashMapStorage . . . 76
8.4.3 CustomHashMapStorage . . . 77
8.4.4 TrieStorage . . . 79
8.4.5 SingletonStorage . . . 81
8.4.6 PerObjectStorage . . . 81
8.4.7 AssociationAspectStorage . . . 83
8.5 System specifications . . . 85
8.6 Results . . . 87
8.6.1 Exact queries . . . 89
8.6.2 Partial-Range Queries . . . 96
8.6.3 Full-Range Queries . . . 104
8.6.4 Small datasets versus large datasets . . . 109
8.6.5 Baseline implementation versus specialisation . . . 109
8.6.6 Recommendations for single-query scenarios . . . 110
8.6.7 Complex scenarios . . . 112
8.7 Threats to validity . . . 114
8.7.1 Threats to internal validity . . . 114
8.7.2 Threats to external validity . . . 115

9 Summary and Future Work **117**

References **120**

1 Introduction

The primary goal of aspect-oriented software development (AOSD) [1] is the encapsulation of **crosscutting concerns**, that is, functionality which is required at potentially many different places in the code, into separated modules called **aspects**. An aspect describes the behavioural effect[1] of the crosscutting concern on the base program. In many cases the code that implements the functionality of the crosscutting concern needs to access state which is shared between multiple executions of that code. This shared state is encapsulated in **aspect instances**. Whenever the implementation of a crosscutting concern – the **advice** – is executed, the shared state that belongs to that occurrence has to be retrieved by the execution environment. Retrieving the correct aspect instance for a specific advice invocation is the primary task of **instantiation policies**. As the retrieval of the aspect instances can occur many times during the execution of the program, we want it to be fast. Although the aspect-instance retrieval has been optimised for certain instantiation policies, we need to find a way to make it efficient for as many instantiation policies as possible.

In this thesis we focus on aspect-oriented execution-environments that have an aspect-definition language following the pointcut-advice approach [2, 3], such as AspectJ [4], JAsCo [5], CaesarJ [6] or Compose* [7].

An aspect needs to define which places of the program it affects. This definition is given in form of a **pointcut** [1]. A pointcut is an expression that describes a set of **join points**, which refer to points in time during the execution of a program. Join points are those points in time at which advice has to be executed. The pointcut typically refers to specific places in the base-code, for example, all places where a certain method is called or all places where a certain field is written to. Those places in the base-code referred to by a pointcut are called **join-point shadows**. These join-point shadows are *static* concepts which can be determined during the **weaving** process. In contrast, join points are *dynamic* (runtime) concepts which can potentially occur at join-point shadows.

Aspects are comparable to classes in object-oriented languages in that they encapsulate the specification of state and behaviour. Like classes, aspects can be instantiated at runtime, with each aspect instance holding its own copy of state variables (*fields* or *instance variables* in object-oriented languages). The definition of the actual behaviour of the crosscutting concern is given by the **advices** of an aspect [2]. Advices are comparable to function bodies (or *method bodies* in object-oriented environments) in that they consist of executable code. Similar to methods, which are executed in

[1] Some aspect-oriented languages also support structural changes by means of inter-type declarations, but we will not further consider them here.

the context of an object, advice is executed in the context of an aspect instance. The state of the aspect instance can be used and modified from within the advice.

However, unlike functions or methods, advice is never invoked explicitly. Instead, the advice code is called implicitly whenever, during execution, a join point is reached that is matched by the pointcut associated with the advice. To execute the advice, the execution environment requires an aspect instance. Retrieving the aspect instance for a specific advice execution is the main responsibility of an **instantiation policy**. Multiple instances of an aspect can exist at runtime. An instantiation policy defines the rules that determine which instance to use for a specific join point. In addition, an instantiation policy may instantiate aspects **implicitly** if no existing aspect instance can be reused for the given join point. If such implicit instantiation is not supported or not possible[2], aspect instances need to be instantiated **explicitly**. As a secondary responsibility, instantiation policies can restrict the execution of advice: if the instantiation policy does not retrieve an aspect instance, the advice cannot be executed.

1.1 Problem statement and goals

Instantiation policies are themselves a cross-cutting concern in multiple respects. Firstly, they affect the application at runtime each time advice is executed. To preserve state between invocations, advice of an aspect is executed in the context of an instance of the respective aspect. It is the responsibility of the instantiation policy to determine the aspect instance each time the advice needs to be executed. That instance can be an existing instance or – if implicit instantiation is supported – a new instance. Because the instance look-up can happen potentially many times, it should be as fast as possible to reduce the overhead at runtime. The exact impact of aspect instance look-up is difficult to predict for the general case. However, Dufour et al. [8] have shown cases where setting up the arguments for advice execution – which the aspect-instance look-up is a part of – takes more than 26% of the total execution time. Optimizing the execution speed of aspect-instance look-up is therefore a desirable goal.

Secondly, instantiation policies are a cross-cutting concern in that they are part of the design-space of each aspect-oriented language which follows the pointcut-advice approach. The choice of available instantiation policies determines if and how easily certain usage scenarios are natively supported by an aspect-oriented runtime-environment. If an aspect-oriented runtime-environment does not support a

[2] Implicit instantiation may be impossible if creating an instance of the aspect requires information that is not available when the instance is required, such as specific constructor arguments.

specific instantiation policy, users of this runtime-environment need to implement equivalent logic on their own to achieve the same behaviour. For example, if an aspect-oriented runtime-environment only supports the `singleton` instantiation policy, each advice invocation takes place in the context of the same aspect instance. To emulate for example the `pertarget` behaviour (which uses a different aspect instance for each distinct target object of a method call), users of this environment need to come up with their own means to share state between advice invocations for the same method-call target. This can potentially lead to duplicate implementations of the same concept, which is exactly what aspect-orientation tries to avoid in the first place. Because instantiation policies are cross-cutting concerns, they should be implemented as isolated, reusable modules instead.

Instantiation policies should also ideally be implemented within the aspect-oriented runtime-environment itself. Depending on the desired behaviour and the runtime-environment, some instantiation policies can be difficult to implement manually. For example, instantiation policies like `per-control-flow` [1] or `per-data-flow` [9] are – if at all – rather difficult to implement efficiently on the base-code level. The aspect-oriented runtime-environment on the other hand typically has access to additional information that is difficult to acquire from within the base code. The aspect weaver can analyse the code, potentially allowing for macro-level and micro-level optimisations because of certain usage patterns and other criteria.

The existing aspect-oriented environments differ in the available instantiation policies. We also expect new aspect-oriented environments to be be developed in the future. Being able to implement new instantiation policies quickly or migrating existing instantiation policies from one aspect-oriented execution-environment to another is therefore a desirable goal. To simplify the implementation of instantiation policies, having a generally applicable, but still efficient way to store and retrieve aspect-instances would be beneficial. For special cases, aspect-instance look-up can then still be further optimised where possible.

Thus, we have two main goals for this research. Firstly, aspect-instance look-up should be *fast*, because it is expected to have considerable impact on the execution time. Secondly, implementing new instantiation policies for a given aspect-oriented execution-environment should be *easy*, because instantiation policies are cross-cutting concerns and are best implemented within the execution-environment.

1.2 Approach

The ultimate goal of this research is to enable a simple yet efficient implementation of instantiation policies within aspect-oriented execution-environments. Although

instantiation-policies can vary considerably in their semantics, they all have to implement aspect-instance look-up in one way or another. To make the implementation of new instantiation policies easier, we will suggest a baseline approach to implement aspect-instance storage and look-up. This baseline approach has to be generic enough to support as many instantiation policies as possible. At the same time, we want this approach to be efficient enough to be applicable in practice. Additionally, we will identify special cases in which aspect-instance look-up can be further optimised if more specialised implementations are used, and we will describe those specialised implementations.

As an outcome of this thesis, we will provide an evaluation of representative data structures and algorithms with respect to their suitability for aspect-instance storage and look-up. We expect some data structures to be more efficient in terms of execution speed and memory usage than others in the context of aspect-instance storage, depending on the usage scenario. Therefore, we will provide a representative overview of expected usage scenarios and recommend specific data structures and implementation details for each scenario.

We expect the results to be independent of any specific aspect-oriented execution-environment. Therefore, we will not provide an implementation of the baseline approach for a specific AOP framework or at the Java Virtual Machine (JVM) level. Instead, we will back up our findings with a representative benchmark running on the base code level. Although the results may differ quantitatively when applied on the JVM level or in an entirely different environment, we expect them to be transferable qualitatively. That is, data structures that perform well in theory and in our benchmark are supposed to still be a good choice when used in a low-level implementation inside the JVM or a different execution environment.

2 Background and Related Work

2.1 Instantiation policies

In aspect-oriented environments following the pointcut-advice approach, advice is typically executed in the context of an instance of the declaring aspect. This allows state to be shared between multiple invocations of the same piece of advice. An instantiation policy defines a set of rules that determine the aspect instance to be used for the execution of advice. An instantiation policy has two main responsibilities:

Aspect-instance retrieval When advice is applicable at a join point, the instantiation policy needs to determine the aspect instance used as the context in which the advice is executed. During this process the instantiation policy needs to determine if an existing aspect instance is reused (and if so, which instance) or if a new aspect instance needs to be created. If aspect instances can be reused, the instantiation policy needs to keep track of those instances it has created for subsequent look-ups.

Restriction An instantiation policy can define rules that restrict or enable the applicability of the advice when a join point covered by the pointcut of that advice has been reached. This responsibility allows an instantiation policy to affect the evaluation of a pointcut expression.

The semantics of an instantiation policy are determined by how the policy fills in these two responsibilities. We describe those responsibilities in more detail in the next subsections.

2.1.1 First responsibility: Aspect-instance retrieval

Because aspects often need to store state that is shared between individual invocations of their advice, they must be instantiated at runtime [4]. An advice is invoked if, during the execution of the program, a join point is reached that is matched by the pointcut expression belonging to the pointcut-advice. The instantiation policy used by the aspect needs to retrieve the aspect instance (or in some cases multiple instances) that is used for a particular invocation of the advice.

Instantiation policies typically determine which aspect instance to use by investigating parts of the context values available at a join point. Context values include the object in whose context the current method is executing, the target of the current method call, arguments passed to the current method, and so on. A specific

combination of context values determines the aspect instance to use. For example, the `pertarget` instantiation policy in AspectJ [4] investigates the target object of a method call. For each distinct target object, a different aspect instance is used. For each subsequent advice invocation at join points with the same target object, the same aspect instance is reused, allowing for state to be shared between invocations. How the context values are used to determine (shared) aspect instances therefore largely determines the semantics of an instantiation policy.

Some instantiation policies allow multiple aspect instances to be retrieved for a single join point. This can occur if the pointcut expression and/or context values do not unambiguously dictate one specific instance to be used, but rather a range of instances. In that case the advice is executed once for each aspect instance retrieved by the instantiation policy.

At some point before the advice invocation, the actual instantiation of the aspect needs to take place. An advice is typically given as a piece of code written in the same programming language as the base code. As such, it follows the same paradigms as the base code. For example, in AspectJ the advice code is object-oriented like the base code. In fact, the AspectJ aspect compiler translates aspects to conventional Java classes [4]. The advice becomes a method of this class. Variables that store the state of the aspect between advice invocations become fields in the aspect class. To instantiate the aspect, an instance of that class is created.

We differentiate between **implicit instantiation** and **explicit instantiation** [10]. In the case of implicit instantiation, aspect instances are created by the instantiation policy as needed. If, during aspect-instance retrieval, the instantiation policy evaluates the context values and finds that there is no aspect instance for the current execution context yet, it creates a new instance of the aspect – typically by instantiating the respective class – and uses that instance as the context object for advice execution. The policy also needs to remember the instances it has created so that those instances can be reused for subsequent advice invocations.

Example 1. The `pertarget` instantiation policy in AspectJ uses implicit instantiation. When a join point is reached that matches the pointcut, the instantiation policy tries to find an aspect instance associated with the target of the current method call. When no such instance exists, it implicitly creates a new instance and remembers it for future reuse. The next time a join point with the same target object of a method call is reached, the instantiation policy reuses the aspect instance it has created before. ▪

If an instantiation policy uses explicit instantiation, it will never create instances of the aspect on its own. Instead, it requires aspect instances to be created explicitly

from within the base code. The instances are then registered at the aspect-oriented execution-environment. When registering an aspect instance, the base code provides information about the cases in which to use the instance.

Example 2. Association aspects (Sakurai et al [10]), which use a different aspect instance per pair of objects, use explicit instantiation. From the base code, a new instance of the aspect is created and then associated with a specific pair of objects. The special `associated(a,b)` pointcut expression is used to declare that a join point is only matched if an aspect instance was registered for the pair of objects `a` and `b`, where `a` and `b` are variables that are bound to actual context values by other pointcut expression such as `target(a)`. If an aspect instance was registered for that pair of objects before, that instance is used for the execution of the advice. ▪

Implicit instantiation can only be performed if all information required for instantiating the aspect can be deduced at the moment of instantiation. For example, if the aspect depends on certain initialization arguments (like constructor arguments in Java), an instance can only be created if those dependencies can be resolved implicitly. Likewise, if the instantiation policy can potentially retrieve multiple aspect instances for a specific pointcut, it is typically impractical or even impossible to implicitly create the aspect instances because the set of required aspect instances cannot be determined unambiguously. Instantiation policies that support those cases need to rely on explicit instantiation.

Example 3. Association aspects support the use of **wildcards** in the `associated` expression. For example, the expression `associated(a,*)` would yield the aspect instances that have been registered to all pairs where the first element is `a`. In this situation, implicit instantiation would mean to implicitly create an aspect instance for each existing object that can appear as the second element of a pair of objects. As this is typically not a sensible approach, association aspects only support explicit instantiation. ▪

Different instantiation policies use different approaches to store and retrieve aspect instances. Often, those approaches are optimised for the specific policy.

Example 4. The `pertarget` instantiation policy implements aspect-instance storage by adding a special field to each class that is a potential call target according to the advice's pointcut[3]. When a new instance of the aspect is created, a reference to it is stored in the added field of the current call target. If some time later the object is again the target of a method call covered by the pointcut, the aspect-oriented execution-environment only needs to look at the added field of the current call target to find the aspect instance. ▪

[3] Because join point shadows are statically determinable in the source code, all classes of which methods are called can be determined statically. Thus, the modification of the classes can be performed at load time.

2.1.2 Second responsibility: Restriction

For advice to be applicable at a specific join point, all conditions in the pointcut expression of the pointcut-advice need to be satisfied. Explicit instantiation adds another condition, namely that an instance of the aspect must have been explicitly created for the current join point. If no suitable advice instance is found, the advice is not invoked. Therefore, an instantiation policy can provide additional restrictions to an aspect with respect to the applicability of advice. Unlike implicit instantiation, which does not restrict the applicability of an aspect, explicit instantiation can be used in cases where the aspect should only be applicable in selected circumstances.

Example 5. Association aspects can restrict advice execution. For example, a pointcut expression such as `associated(a,b)` requires that an aspect instance for the pair of objects `(a,b)` has been registered explicitly. If no such pair has been registered, no aspect instance can be retrieved and therefore the advice is not executed. .

2.2 Existing instantiation policies

Available instantiation policies vary between aspect-oriented environments. The rest of this section describes several of those policies. We provide a description of the policies' semantics and how the policies fulfil the two responsibilities introduced in Section 2.1.1, aspect-instance retrieval and restriction. In addition, we describe the instantiation policies from an abstract, conceptual point of view which allows us to generalise the concept of instantiation policies. This is necessary to recommend a generalised strategy for implementing new instantiation policies.

2.2.1 Association aspects

Overview

Sakurai et al. [10] present the concept of **association aspects** which associate aspect instances to combinations of n objects.

Association aspects extend the AspectJ language at several places. Firstly, the `perobjects` *per*-clause[4] declares an aspect to be an association aspect. The

[4] In AspectJ, the *per-clause* is an optional modifier of the aspect which determines the instantiation policy. It is placed after the name of the aspect. [11]

Listing 1: A simple association aspect [10]

```
1  aspect Equality perobjects(Bit, Bit) {
2    Bit left;
3    Bit right;
4
5    Equality(Bit l, Bit r) {
6      left = l;
7      right = r;
8      associate(l, r); // establish association
9    }
10
11   before(Bit l, Bit r) : call(* Bit.*(*)) &&
12                          this(l) &&
13                          target(r) &&
14                          associated(l,r) {
15     System.out.println(String.format("Bit %s called method of Bit %s", left, right));
16   }
17
18   after(Bit l) : call(void Bit.set()) &&
19                  target(l) &&
20                  associated(l,*) {
21     right.set(); // propagate value change to "partner"
22   }
23 ...
24 }
```

perobjects clause defines the number and the types of objects that take part in each association. Secondly, association aspects add a new pointcut primitive called associated. This primitive determines the (combination of) objects for which a specific aspect instance needs to be retrieved. Finally, association aspects introduce a method called associate which establishes the association between an aspect instance and the set of objects for which that specific aspect instance has to be used.

We explain the extensions to AspectJ made by association aspects and their semantics in more detail by means of two examples.

Example 6. Establishing an association between multiple objects can be useful to keep the state of those objects synchronised. For example, assume a class Bit [10] that represents a single bit. One might want to relate two bits in such a way that setting the value of one bit causes the value of the second bit to change accordingly.

Listing 1 shows parts of an association aspect named Equality that establishes this kind of relation. The *per*-clause perobjects(Bit,Bit) (line 1) is used to declare that this aspect is an association aspect which associates two objects of type Bit. The constructor of the aspect (lines 5–9) accepts the two Bit instances that take part in the association and stores the references for later use (lines 6–7). The call

Listing 2: The association aspect `NotificationLog`

```
1 aspect NotificationLog perobjects(Observable, Observer) {
2
3   int counter;
4
5   NotificationLog(Observable observable, Observer observer){
6     associate(observable, observer);
7   }
8
9   before(Observable observable, Observer observer) :
10            call(void Observer.notify(*)) &&
11            this(observable) &&
12            target(observer) &&
13            associated(observable, observer) {
14    counter++;
15  }
16 }
```

`associate(l,r)` (line 8) establishes the association between the two `Bit` instances and assigns the current aspect instance to this association.

The `Equality` aspect defines two pointcut-advice bindings. The first (lines 11–16) is supposed to log any calls of the left `Bit` to an associated right `Bit`. A `this` primitive is used to bind the variable l to the currently executing object (line 12). Likewise, the `target` expression binds the current call target to the variable r (line 13). The `associated` primitive in line 14 requires that an association has been established between l and r. If this is the case, the advice will be executed in the context of the aspect instance assigned to this association.

The second pointcut-advice pair (lines 18–22) establishes the synchronisation between two bits: whenever the `set` method is called on the left `Bit` of a bit pair, the right bit should be notified about the changes. The pointcut binds the current call target (which must be a `Bit`) to variable l (line 19). Again, the `associated` primitive is used to check for an association of bits (line 20). However, in this case, a wildcard (an asterisk, $*$) is used as the second argument to the `associated` primitive, causing the pointcut to be applicable for all associations in which the left `Bit` is l. For each such association the advice is executed once in the context of the respective associated aspect instance. ▪

Example 7. In the observer pattern [12] **observers** (types that implement `Observer`) can register with **observables** (types that register `Observable`) to be notified of certain events. An `Observable` notifies its `Observers` by calling their `notify` method. Assume that we want to count per individual pair of `Observer` and `Observable` how

Listing 3: Modified object layout for association aspects

```
1 class Bit {
2   // fields added during weaving
3   Map<Bit,Equality> rightPartnersToAspectInstances;
4   Map<Bit,Equality> leftPartnersToAspectInstances;
5   ...
6 }
```

often the Observable notifies the Observer. Listing 2 shows an exemplary implementation of `NotificationLog` using association aspects. The `perobjects` keyword (line 1) declares that this aspect associates a pair of `Observable` and `Observer`. Calling the intrinsic method `associate` (line 8) creates an association between the `Observable` and the `Observer` and assigns the current aspect instance to this association. The pointcut expression `call(void Observer.notify(*))` (line 10) defines that the advice will be executed when an `Observer` receives a call to the `notify` method. The object in whose context the call occurs is bound to `observable` (using `this(observable)`, line 11). The object receiving the method call is bound to `observer` (using `target(observer)`, line 12). And finally, the advice can only be executed if an aspect instance has been registered for this pair of `observable` and `observer` (`associated(observable, observer)`, line 13). ▪

Aspect-instance retrieval

When a pointcut is evaluated for an association aspect, the runtime tries to find an aspect instance associated with the pair of objects referred to by the `associated(. . . )` expression. Those objects are typically context values such as the call target (`target(. . . )`) or the object in whose context the current method is executing (`this(. . . )`). To keep track of aspect instances, the AspectJ runtime has been augmented to modify classes that can take part in an association (that is, those types used in the `perobjects(. . . )` declaration of the aspect) during the weaving process. For a given association aspect each type that can take part in the association gets additional fields with information about existing associations.

Example 8. For the `Equality` example, the AspectJ runtime adds two fields to the `Bit` class in a way similar to what is shown in Listing 3. The first field, `rightPartnersToAspectInstances`, represents associations where the current object plays the role of the “left” member. For each association with a second (“right”) bit the map contains an entry with that second bit as the key and the related aspect instance as the value. Likewise, the second field (`leftPartnersToAspectInstances`) stores aspect instance for associations in which the current object plays the “right” role. That is, for an aspect instance `aspect` associated with `associate(left,right)` the

Listing 4: Instantiation of `Equality` [10]

```
1 Bit left = ...
2 Bit right = ...
3
4 // Instantiate Equality aspect and
5 // establish association between left and right bit.
6 new Equality(left, right);
```

runtime effectively calls `left.rightPartnersToAspectInstances.put(right, aspect)` and `right.leftPartnersToAspectInstances.put(left, aspect)`.

Storing the associations twice is necessary to support wildcards. The first field can be used to find associated aspect instances for pointcuts that use an expression such as `associated(left, *)`, the second field is used for pointcuts with an expression such as `associated(*, right)`: the sought aspect instances are simply all values in the respective maps (`rightPartnersToAspectInstances` in the former case, `leftPartnersToAspectInstances` in the latter) and can be retrieved with a call to the `values()` method of the `Map` interface. If no wildcard is used, that is, both parts of the association are bound by an expression such as `associated(left, right)`, either the `rightPartnersToAspectInstances` of `left` or `theleftPartnersToAspectInstances` of `right` can be used: both `right.leftPartnersToAspectInstances.get(left)` and `left.rightPartnersToAspectInstances.get(right)` return the same aspect instance..

Association aspects always use explicit instantiation. An association needs to be established with a call to the method `associate`. The `associate` method is automatically added to the aspect class. The instance of the aspect on which `associate` is called is assigned to the established association. In the example given in Listing 1 the call is wrapped within the aspect constructor for convenience, thus the `Equality` aspect can be instantiated as shown in Listing 4.

Restriction

Because association aspects are instantiated explicitly, the applicability of advice can be restricted. An `associated` expression without wildcards will only match a join point if there is an association between the objects that are bound to that expression. There is either exactly one such association or none at all[5]. In the latter case no aspect instance can be determined. The advice will therefore not be executed. If wildcards are used, an `associated` expression can refer to zero or more associations. Again, if no association is matched by the expression, no advice is executed.

[5] It is not possible to associate multiple aspect instances to the same combination of objects.

Conceptual view

Association aspects have the most general instantiation policy of all policies presented in this thesis. Therefore, we base our unified model (see Section 4) mainly on this policy. Specifically, two properties of association aspects are of importance to our unified model:

1. Aspect instances are associated to a tuple of objects. That is, for a specific combination of context values, a separate aspect instance is used.

2. When retrieving aspect instances for a given pointcut, wildcards are support. That is, at a given join point, multiple aspect instances can be applicable.

Association aspects are always explicitly instantiated. While this provides the restriction function mentioned in Section 2.1.2, we also want to include implicit instantiation in our model.

Although association aspects do not support implicit instantiation, this could be added as an extension of the original concept for cases where no wildcards are used. When no wildcards are used inside the `associated` expression, at most one aspect instance can be retrieved[6]. If the association referred to by an `associated` expression has not been established before, the execution environment creates a new instance of the expected aspect, calls `associate` on it with the argument provided to the `associated` expression and uses the new aspect instance as the instance for the advice invocation.

2.2.2 Per-object instantiation

Overview

Per-object instantiation is supported by most aspect-oriented frameworks such as AspectJ [4], AspectWerkz [13, 14] or JAsCo [5]. Different flavours of per-object instantiation policies exist, but they have in common that they refer to a specific value in the context of the join point (for example, the object in whose context the current method is executing) and that they create a different aspect instance for each distinct instance of that context value. In this section, we concentrate on the flavours provided by AspectJ.

[6] We do not support cases where multiple aspect instances can be associated with the same tuple of context values and leave this for future work.

Listing 5: Examples for the **perthis** and **pertarget** instantiation policies.

```
1 aspect PerThisExample perthis(bitSetter()){
2   pointcut bitSetter(): call(void Bit.set());
3   before(): bitSetter() {
4     // advice body...
5   }
6 }
7
8 aspect PerTargetExample pertarget(bitSetter()){
9   pointcut bitSetter(): call(void Bit.set());
10  before(): bitSetter() {
11    // advice body...
12  }
13 }
```

AspectJ supports two different per-object policies, **perthis** and **pertarget** . The **perthis** policy, indicated by a `perthis` *per*-clause, creates a new instance of the aspect for each distinct object in whose context the current method is executing. The **pertarget**, similarly denoted by a `pertarget` *per*-clause, is used with pointcuts that refer to method calls. It creates a new aspect instance for each distinct target object of the method call, that is, the object that receives the method call.

Example 9. The first aspect in Listing 5 (lines 1–6) shows an exemplary use of the **perthis** policy. The aspect is declared to use the `perthis` policy (line 1). The `perthis` *per*-clause expects a pointcut expression as an argument. In this case, for each distinct object calling `Bit.set()` (line 2), a new aspect instance will be created. The second aspect in Listing 5 (lines 8–13) shows the use of the **pertarget** policy. In this example, a new instance of the `PerTargetExample` aspect is created for each distinct target of the call to `Bit.set()`. In other words, the aspect is instantiated once for each distinct `Bit` that has its `set()` method called. ▪

Aspect-instance retrieval

AspectJ adds support for per-object instantiation at the application-code level by modifying the object layout of classes, adding a field that can hold a reference to an aspect instance [15]. The class to which the field is added is determined by the exact type of policy and the pointcut expression passed as an argument to the *per*-clause. In the case of **perthis**, the field is added to each class that calls the method referred to by the pointcut. For example, if the pointcut is defined as `call(void Bit.set())`, each class calling `Bit.set()` in one of its methods receives the additional field. Similarly, in the case of **pertarget**, the additional field is added to the classes that may be the call-target of the referenced join point. For example, the pointcut

expression `call(void Bit.set())` will cause the `Bit` class to receive the field. When the aspect instance needs to be accessed, it is searched for in the added field. If the field has a value of `null`, no instance of the aspect has been created for the respective object yet.

Listing 6: Implementation of **pertarget**: an additional field is added to target classes.

```
1 class Bit {
2 ...
3   private PerTargetExample perTargetExampleInstance;
4 ...
5 }
```

Example 10. A simplified example for the implementation of `PerTargetExample` is shown in Listing 6. When weaving the aspect into the base code, the AspectJ weaver adds a field to the target class `Bit`. To retrieve the aspect instance for advice execution, the AspectJ runtime looks it up in the `perTargetExampleInstance` of the current call target (which must be of type `Bit`). ▪

Like all instantiation policies that AspectJ provides, **perthis** and **pertarget** only support implicit instantiation. At join points that are covered by the pointcut passed to the `perthis` or `pertarget` per-clause (e.g., in the case of `PerTargetExample`, calls to `Bit.set()`), the execution environment checks whether the generated field has an aspect instance assigned to it. If it does not, that is, its value is `null`, a new instance is created implicitly by the execution environment; otherwise the existing instance is used.

Although AspectJ does not support explicit instantiation, this restriction is not inherent for per-object instantiation. Other aspect-orientation execution environments support explicit per-object instantiation policies that allow aspects to be (deployed to and) instantiated for selected objects only [16].

Restriction

Per-object instantiation in AspectJ does not have a restriction function, because aspect instances are always created implicitly on demand. For implementations that do support explicit instantiation, the restriction function applies as described in Section 2.1.2.

Conceptual view

Per-object instantiation can be interpreted as a special case of association aspects in which a unary tuple of objects (that is, a single object) is associated to an aspect

instance. Aside from the implicit instantiation, the per-object policies of AspectJ (**perthis** and **pertarget**) can be completely expressed by association aspects as well. In both cases, the tuple used in the association is unary, that is, the tuple consists of only one element. The only difference is the context value used for this element. An example is shown in Listing 7. In the pointcuts, the policy is enforced by passing the current `this` (for `perthis`, line 2) or the current call target (for `pertarget`, line 6) to the `associated` expression. Both `perthis` and `pertarget`, however, support implicit instantiation, which is by default not supported by association aspects.

Listing 7: **perthis** and **pertarget** expressed as association aspects

```
1 aspect SomeAspect perobjects(Bit) {
2   before(Bit b) : this(b) && associated(b) {
3     // ...
4   }
5
6   before(Bit b) : target(b) && associated(b) {
7     // ...
8   }
9 }
```

2.2.3 Singleton aspects

Overview

Some aspects share their state between all advice invocations. These aspects are often called singleton aspects, because at most a single instance of the aspect is created in whose context advice is executed.

The singleton instantiation policy is called **issingleton** in AspectJ [4] and **perJVM** in AspectWerkz [13, 14], the latter stating more clearly that the aspect is instantiated only once per virtual machine instance. By default, when no other instantiation policy is used, aspects are singleton aspects in both AspectJ and AspectWerkz.

Aspect-instance retrieval

Because at most one instance of a singleton aspect can exist at any time, AspectJ stores the singleton aspects in an implicitly created field of the aspect class. Storing and retrieving aspect instances is therefore a matter of accessing that static field of the aspect. As with the backing field used in per-object instantiation strategies, a value of `null` in the static field indicates that no instance of the respective aspect has been created yet.

Instances of singleton aspects are implicitly created by the AspectJ execution environment the first time the advice needs to be invoked. To determine whether an instance exists, the execution environment only needs to check whether the created static field in the aspect class has an instance assigned to it. If not, a new instance is created and its reference is stored in that field for future reference. Otherwise, the object referenced by the field is reused.

Restriction

Like per-object instantiation (Section 2.2.2), singleton instantiation in AspectJ does not impose a restrictive function.

Conceptual view

Singleton aspects can be considered a trivial case of the instantiation policy used with association aspects. The arity of the associated tuples is zero in this case. Therefore, at most one association exists in this case: the association between the empty tuple and the singleton instance of the aspect. Listing 8 shows an example of how this can be represented conceptually. Again, the `perobjects` *per*-clause is used to make the aspect use the association-aspect instantiation-policy (line 1). However, the provided type tuple is empty (a 0-tuple). The pointcut in lines 2–3 uses the `associated` expression to check for an existing association of the 0-tuple. Because association aspects only support explicit instantiation, the pointcut will not be matched if no such association has been established before (with a call to `associate`). However, singleton aspects use implicit instantiation. To completely represent singleton aspects by association aspects, we assume that the association is implicitly established.

Listing 8: Conceptual example of singleton aspects as association aspects

```
1 aspect SomeAspect perobjects() { // associations of 0-tuples
2   before() : associated()
3         && call(void Bit.set()) {
4     // ....
5   }
6 }
```

2.2.4 Other instantiation policies

Other instantiation policies exist that are supported by only a few aspect-oriented languages or as prototypes only.

AspectJ supports the **percflow** (and **percflowbelow**) instantiation policy which associates a new aspect instance with each occurrence of a specific control flow whose start is defined by a given pointcut. Unlike the objects that are used as the key in the association to aspect instances, control flows are not represented as explicit objects in the Java runtime, which means that AspectJ cannot store aspect instances with **percflow** instantiation by modifying the layout of certain objects. Instead AspectJ uses a thread-local stack that keeps track of control flows that have been entered and exited. This strategy can be expressed by an association-aspects policy that establishes associations of aspects and unary tuples consisting of a “cflow”. However, because a control flow is not a first class entity in Java, it is a more abstract concept than a conventional Java object. Even though **percflow** conceptually fits into our generalised model, we will therefore not further consider this policy in our research.

Masuhara and Kawauchi [9] have suggested data-flow pointcut-expressions. A data flow is defined by the usage of the same object at two different join points, one that matches the current join point and one that has matched a join point in the past. Ensuring that data is encrypted while it flows through the system is an example use-case. Although the authors do not introduce a distinct instantiation policy for per-data-flow aspects, one can imagine aspects being instantiated once for each data flow identified in a data flow pointcut. A **per-dataflow** instantiation policy can be considered an extension of the **percflow** policy. Again, we will not include it in our research, even though it conceptually fits into our model.

Older versions of AspectWerkz [13, 14] provided support for a **per-thread** instantiation policy. This policy can be considered a special case of per control-flow instantiation with the control flow starting at the first stack frame of a specific thread. The support for per-thread instantiation was dropped in version 2 of AspectWerkz. JAsCo [5] also supports per-thread instantiation.

Other instantiation policies supported by JAsCo include **per-class** (one instance for each distinct type of the target object of a call) and **per-all** (one instance for each distinct join point shadow). **Per-class** instantiation can be simulated by association aspects as it is similar to the **pertarget** policy, if instead of the call target itself its type is used to determine whether a new or existing aspect instance is used.

JAsCo also supports completely custom instantiation policies by allowing the user to provide a customised aspect factory. At each matched join point the factory is asked for an aspect instance. Therefore, the factory can freely decide how to provide that instance.

2.3 More related work

In preparation to this thesis, we already established a first abstract model of instantiation policies [17]. Although the initial model already captures the essential ideas of our current unified model (see Section 4), it does not clearly distinguish between those parts of instantiation policies that define their semantics and those parts that are independent of any semantics and may therefore be more easily generalised. We also evaluated the asymptotic complexity of algorithms that can be used to represent aspect-instance storage.

In collaboration with Steven te Brinke and Christoph Bockisch, an article has been published for the 2012 FREECO Conference [18]. The unified model presented in that article defines instantiation policies in terms of an **implicit** flag and the functions **bind**, **find** and **store**. The unified model described in this thesis is based on and extends the model presented in that article.

Martin Zandberg evaluated optimisation of aspect-instantiation strategies using JIT compilation [19]. Based on the unified model, he identified properties of instantiation-policies:

- implicit versus explicit instantiation
- context-sensitive policies versus context-insensitive policies – that is: does the aspect-instance in the context of which advice is executed depend on context variables (such as the object that is calling the currently executed method)?
- exact queries versus range queries – that is: are there potentially multiple aspect instances applicable at the current join point?
- fixed associations versus non-fixed associations – that is: can the aspect instance for a specific join point be exchanged once it has been deployed?

For each combination of those properties[7], Martin Zandberg suggests an optimized implementation to access the underlying data storage for the aspect instances, when implemented in the ALIA4J framework using JIT compilation. Zandberg's thesis focuses on optimising the machine code generated for the look-up procedure, based on properties of the different instantiation policies. The underlying data structures are mostly left as an implementation detail. In contrast to that, this thesis focuses on the effect of the choice of data structures, independent of the compilation model or the aspect-oriented execution-environment. We expect the results of thesis to be combinable with the findings of Martin Zandberg.

[7] That is, for each combination that can occur in practice, as not all combinations are possible, such as implicit instantiation together with range queries.

3 Research objectives

Aspect orientation deals with crosscutting concerns in software systems [20]. The join-point shadows that result from the deployment of an aspect can potentially exist in parts of the program that are executed many times. Whenever the program execution reaches a join-point shadow, the aspect-oriented execution-environment needs to decide whether that shadow is an actual join point by evaluating the related pointcut of the aspect [4]. As described in Section 2.1, one part of this evaluation is the retrieval of the aspect instances for which the advice will be executed. This part of the pointcut-evaluation process, which is also depicted in Figure 1 on page 21, works as follows:

- First, the instantiation policy tries to find one or more aspect instances using a specific combination of values from the execution context (1). The combination of **context values** determines which aspect instances of all known instances to retrieve. Which context values are used depends on the policy's semantics (Section 2.1.1). For example, the **pertarget** instantiation policy defines that there is (at most) one distinct instance of the aspect for each different target object of the current method call. The instantiation policy will therefore retrieve the aspect-instance that is associated with that target object, if such an associated aspect instance exists. Specifically, the AspectJ implementation of the **pertarget** policy looks for an aspect instance in a field that has been added to the class of the target object during the weaving process.

- The initial look-up can be either successful or unsuccessful, that is, aspect instances matching the selection of context values may or may not have been found (2). If aspect instances have been found, the pointcut evaluation can continue (3). A pointcut can consist of several sub-expressions. A join-point shadow is only considered a join point (and therefore advice is executed) if all sub-expressions of the pointcut can be evaluated successfully and if aspect instances have been retrieved. Even if an aspect instance has been retrieved, other sub-expressions may fail to evaluate successfully. For example, the condition required by an **if** pointcut-primitive [11] may not be met at the moment of pointcut evaluation.

- If the look-up was not successful, the next step depends on the fact whether the instantiation policy supports implicit instantiation (4). If implicit instantiation is not supported (or not possible), no aspect instances can be retrieved for the current pointcut evaluation. Therefore, pointcut evaluation can stop immediately (7), as there is no way to execute the advice, even if all other sub-expressions of the pointcut are evaluated positively.

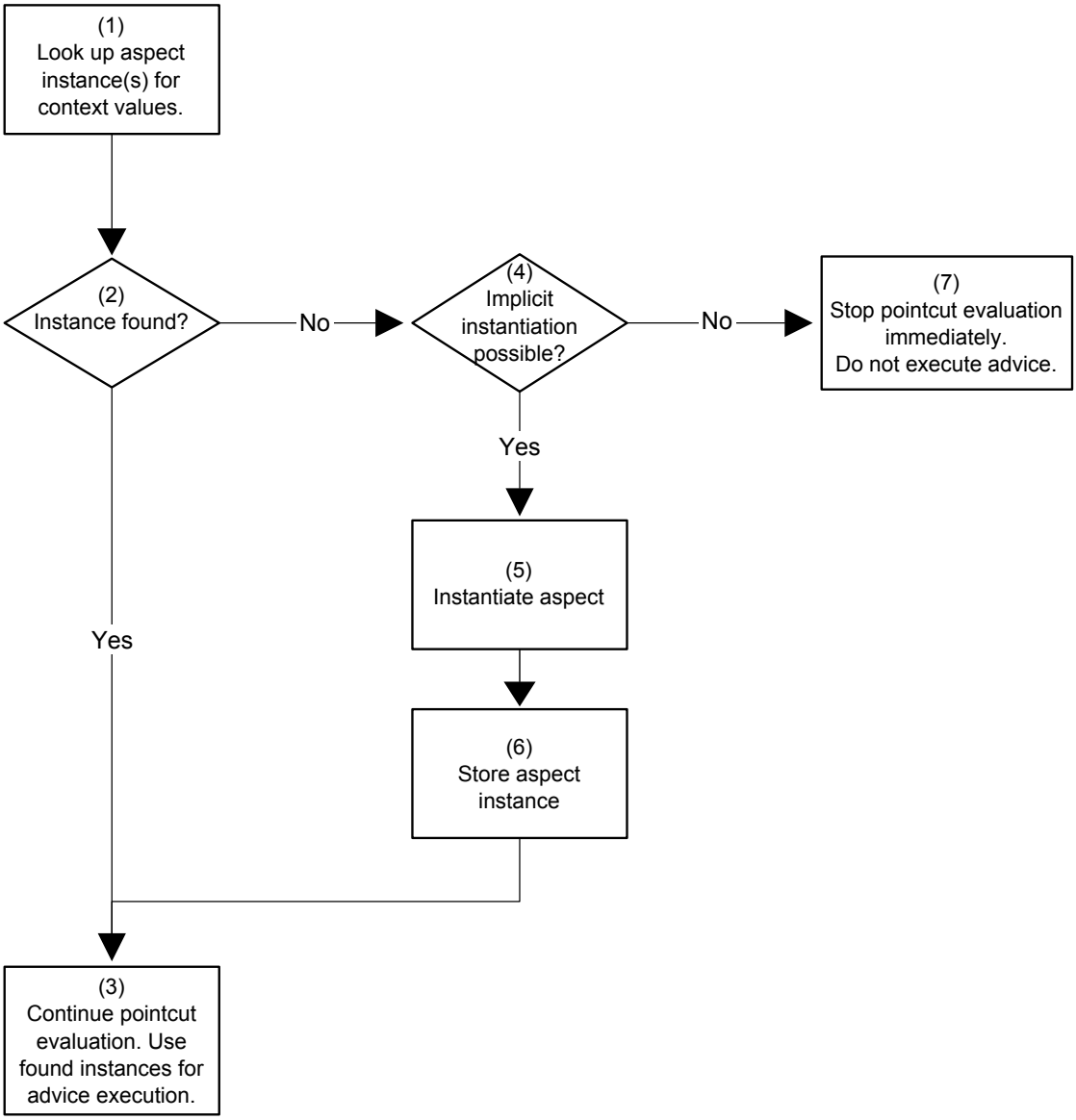


Figure 1: Aspect-instance retrieval as part of the pointcut evaluation.

- If implicit instantiation is possible, a new instance of the aspect is created (5) and stored for later retrieval (6). The pointcut evaluation may then continue (3). If all other sub-expressions of the pointcut are evaluated positively, the advice can be executed using the newly created aspect instance.

Aspect-instance retrieval needs to be performed on each evaluation of the join point shadow. Because this evaluation can occur many times during the execution of the program, we want it to be fast. If the evaluation of join points in general and the aspect-instance retrieval in particular takes too long compared to the actual advice execution, aspect-orientation may turn out to be inapplicable for performance critical applications. Therefore it is important to use data structures and algorithms that make aspect-instance retrieval fast.

3.1 Research questions

In this thesis we answer the following questions:

1. Which operations are required to implement instantiation policies in general?

We provide a unified model of instantiation policies that shows that an instantiation policy can be viewed as a function that maps tuples of context values to aspect instances. The unified model identifies the core properties and functions that define the semantics of an instantiation policy, namely **bind** and the **implicit** flag. In addition, an instantiation policy requires operations for aspect-instance **retrieval** and **storage**.

2. Which operations can be optimised without affecting the semantics of instantiation policies?

We can split the operations related to instantiation policies into two groups. First, those that implement the semantics of the instantiation policy and second, those that are common to instantiation policies in general. As we strive for a general approach to optimise aspect-instance retrieval, those operations that implement the semantics of an instantiation policy will typically require per-case optimisation and are more difficult to optimise in general. Instead, we will focus on those operations that are common to all instantiation policies, namely the operations related to aspect-instance retrieval and storage. Specifically, these operations are **exact queries** (look-up using a tuple of context values), full and partial **range queries** (look-up using wildcards,

that is, ignoring certain elements of the context-value tuple) as well as **insertion**. Of those, we are primarily interested in the query operations, as we expect querying to occur much more often than insertion and thus to have a greater impact on the program execution. We will therefore evaluate insertion as part of our theoretical analysis, but not as part of practical evaluation. There are additional operations that we intentionally left out, such as copying data from one data structure to another (basically a combination of a full-range query and insertion) and removal of elements. Even though some aspect-oriented execution-environments allow the removal of aspect instances or the full undeployment of aspects at runtime, we do not consider it to be a typical use case. We do not expect this omission to considerably affect the validity of our results (see Section 8.7).

3. What is the asymptotic computational complexity of the required operations when implemented for different data structures?

Operations can be classified in terms of their asymptotic computational complexity [21]. It can be used to quantify the time and space requirements of an algorithm when the size of the data that it works on approaches infinity. Operations with the same asymptotic complexity show a similar growth in time requirements with increasing input sizes. Specifically we are interested in having an upper limit of the growth of the time and space requirements of different algorithms. This upper limit is usually described using the Big-O notation [22]. Knowing their complexity classes allows us to classify different algorithms.

4. What is the data complexity of the respective data structures in those scenarios?

The execution speed is not the only criterion to evaluate the suitability of a certain data structure for storing aspect instances. Memory also needs to be considered. Depending on the requirements of the application and the environment it runs in (for example, mobile device), the amount of memory that can be used to store aspect instances is limited. Consequentially, the amount of memory used must also be considered.

5. How do query operations perform in scenarios where the number of aspect instances is comparatively small?

The asymptotic complexity of an operation only describes the limit of the behaviour, that is, the behaviour the operation shows when the size of the input dataset approaches infinity. However, in practice, the size of the input data for the operations

used by instantiation policies may be comparatively small. In the case of aspect-instance retrieval, the number of context variables (that is, the tuple size in our unified model) and the number of existing aspect instances determine the size of the input dataset for the operations needed by instantiation policies. Practical use cases are imaginable in which actually only a small number of aspect instances (for example, only a few dozen or hundred) exist simultaneously. Therefore, we expect the size of the input dataset to be far from "infinite"[8]. As a result, the asymptotic complexity may not give sufficient information about how the operations behave in practice. We expect different operations with the same asymptotic complexity to potentially perform notably different when the input size is comparatively small: constant overhead or other factors which are ignored by the Big-O notation can become important in those cases. Also, worst case scenarios may be more or less likely in practice, resulting in a different performance.

6. How efficient are operations on data structures that are generally applicable in all scenarios in comparison to operations on optimized data structures that can only be used in specific circumstances?

Most instantiation policies in current aspect-oriented execution environments are rather simple. For example, **singleton** aspects [4, 13, 14] can only be instantiated once. The **perthis** and **pertarget** instantiation policies in AspectJ [4] relate at most one object from the set of context variables to an aspect instance. Those simple cases provide opportunities for optimisation, for example by changing the object layout during the weaving process [15]. We want to find out how data structures that can be used for any instantiation policy perform in comparison to those optimized structures.

7. Which data structure is recommended in which practical scenario?

Our solution must be generally applicable. Although special cases can possibly be optimised, we want to provide a solution that is fast in the general case. "Fast" in this case is only relative. The time and space used by the operations and algorithms can be compared with each other. However, an absolute definition of "fast" is difficult, as it highly depends on the requirements of the respective use case. We can therefore only evaluate the efficiency of the operations and data structures in abstract, relative terms. Nevertheless we want to provide a recommendation that is applicable for current as well as future, more complex instantiation policies.

[8] Additionally, there are practical limits to the number of objects that can be instantiated at a time, such as the amount of system memory. However, we do not consider these limits to be of practical relevance for our research.

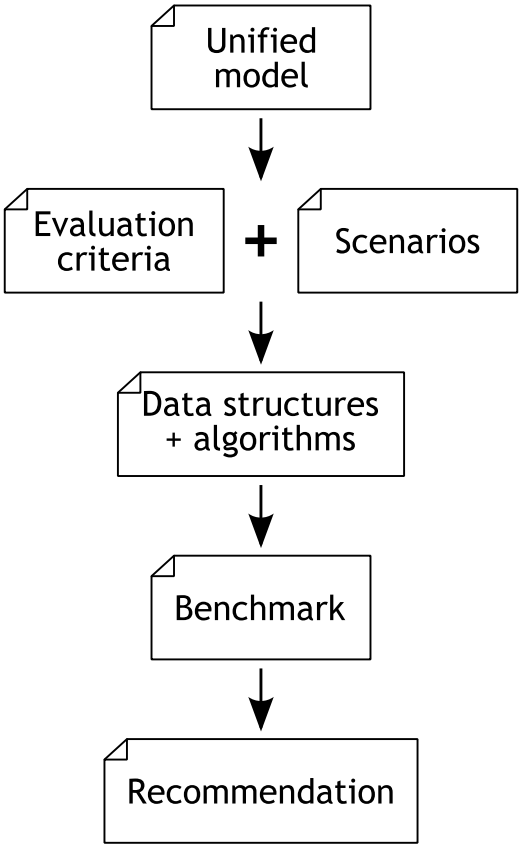


Figure 2: Products of the thesis. Arrows represent order of production.

Answering the research questions will help the developers of aspect-oriented execution-environments to improve the speed of their implementations. As a result the applicability of aspect-orientation in performance-critical parts of applications can be improved. Also, we expect the development of new, more complex instantiation policies to be simplified because our solution is generally applicable.

3.2 Approach

The goal of this thesis is the evaluation of data structures and operations for storing aspect instances at runtime to give a recommendation of the most suitable data structure for different scenarios. To give a sound recommendation, a number of intermediate products are required. The creation of those products determines the approach of the thesis. Those intermediate products and the order in which they were created are shown in Figure 2 on page 25.

As a first step, we establish a **unified model of instantiation policies** (see Section 4). We create the model as a refinement of our previous models ([17, 18]). It is based on the analysis of a selection of existing[9] instantiation policies. The model allows us to represent all instantiation policies we consider using one unified concept, which primarily is the mapping of an n-ary tuple to an aspect instance. The model influences the design of the benchmark and the pre-selection of data-structures to evaluate (see below). This step answers our first two research questions (answers are found in Section 4.8).

The operations defined in the unified model also help us establishing a set of **scenarios** that we use to evaluate the data structures (see Section 6). There is only limited information about the scenarios that occur in real-life applications. However, we chose a number of (what we think are) realistic scenarios that cover most of the possibilities of today's aspect-oriented execution-environments that use a pointcut-advice approach. Primarily we focus on aspect instance look-up using different key-tuple lengths and differently sizes of input data. The **evaluation criteria** that we use to evaluate the scenarios (asymptotic time complexity, actual execution time and memory usage) are already defined by the research questions 3 to 5. We give a more in-depth description of these criteria in Section 5.

With the scenarios available we assemble a selection of **data structures** to evaluate (see Section 7). Given the virtually infinite number of existing data structures and their derivatives, we are required to systematically select a short-list of data structures that we want to evaluate. The unified model suggests that instantiation policies are basically a function from tuples to aspect instances, or more generally from keys to values. This makes associative data structures *[23, ch. 4]* a good choice, that is, data structures that associate a *value* to a specific *key*. Typical examples for those data structures include hash maps and sorted trees ([24, 25]). Theoretically, any data structure can be used to store those associations, but some classes of data structures are better suited for this task because they are specifically designed for efficient key look-ups. In addition to associative containers we evalute simple data structures such as arrays and linked lists. The theoretical evaluation of the data structures answers question 3 and 4 (for the answers, see Section 7.6).

Next, we present the result of a **benchmark** that was used to execute the scenarios established earlier using the chosen data structures (see Section 8). The benchmark has been carefully developed to make the results reproducible and meaningful. However, the results are not meant to exactly quantify the execution speed of the operations or the amount of memory used, as those depend on many different factors. Instead, we were looking for qualitative statements about the relative behaviour of the different data structures as those are expected to be consistent in all real-life

[9] Either implemented in practice or described in literature.

Research question	Answered in
1. Which operations are required to implement instantiation policies in general?	Section 4.8
2. Which operations can be optimised without affecting the semantics of instantiation policies?	Section 4.8
3. What is the asymptotic computational complexity of the required operations when implemented for different data structures?	Section 7.6
4. What is the data complexity of the respective data structures in those scenarios?	Section 7.6
5. How do query operations perform in scenarios where the number of aspect instances is comparatively small?	Section 8.6.4
6. How efficient are operations on data structures that are generally applicable in all scenarios in comparison to operations on optimized data structures that can only be used in specific circumstances?	Section 8.6.5
7. Which data structure is recommended in which practical scenario?	Section 8.6.6

Table 1: Research questions and references to section in which they are answered.

cases. The benchmark together with the scenarios and evaluation criteria helps us to answer the research questions 5 (Section 8.6.4) and 6 (Section 8.6.5).

Finally, evaluate the results of the benchmark by comparing the influence of different variables (such as the number of aspect instances or the length of the query-key tuple) on the execution time of the queries performed by different algorithms. This allows us to qualitatively compare the performance of data structures and algorithms in different scenarios. The evaluation leads to a **recommendation** for the most suitable data structure to be used in specific scenarios according to our criteria (see Section 8.6.6). This final step answers the seventh, final research question.

As an overview, Table 1 on page 27 shows the research questions together with the section in which they are answered.

4 Unified model of instantiation policies

In Section 2.2 we presented a number of instantiation policies and gave a conceptual overview of how they work. Comparing existing instantiation policies, we found certain commonalities between them: instantiation policies have in common that they establish an *association* between a specific selection of context values and an aspect instance. That is, a specific selection of context values identifies a specific aspect instance. We call this selection of context values the **key-tuple** (an ordered sequence of n elements). The instantiation policy defines the rules how the key tuple is constructed from context values.

The semantics of an instantiation policy is defined by two properties. Firstly, whether the instantiation policy uses implicit or explicit instantiation. And secondly, how the key tuples that identify aspect instances are created during the pointcut evaluation. We call the property that defines whether an instantiation policy uses implicit instantiation or not, the **implicit flag** – a Boolean variable that has the value `true` for policies that use implicit instantiation and `false` for those that use explicit instantiation. We call the function that creates the key tuple at the moment of pointcut evaluation, the **bind function**. When the pointcut is evaluated, the bind function generates a key tuple consisting of values available from the execution context provided by the aspect-oriented execution-runtime. This key is used to search for candidate aspect-instances in whose context the advice is executed. The actual look-up operation is independent of a policy's semantics. Although the implementation of the look-up operation and the underlying data storage can often be (and in practice often is) optimised for specific instantiation policies, changing it does not alter the behaviour of the instantiation policy. This separation of semantically relevant and irrelevant parts allows us to modularise the implementation of instantiation policies.

Each of the two properties (**implicit** flag and **bind** function) affects both responsibilities of an instantiation policy (Figure 3 on page 29). Aspect-instance retrieval depends on the bind function, as this function creates the key tuples: only those aspect instances are retrieved that are identified by the generated key tuple. Also, the restriction responsibility is affected by the bind function, as restriction only takes place if the bind function generates a key tuple which does not identify any existing aspect instance. The implicit flag affects both the aspect-instance retrieval and the restriction responsibility, because it determines how to handle cases in which no existing aspect-instances are retrievable. If this flag is set to `true` a new instance will be created as part of the aspect-instance retrieval. If the flag is set to `false`, restriction comes into effect, causing the advice not to be executed at all.

A unified template for aspect-instance retrieval is shown in Algorithm 1. The first parameter of `RetrieveAspectInstances`, a `Context`, abstracts access to values in the

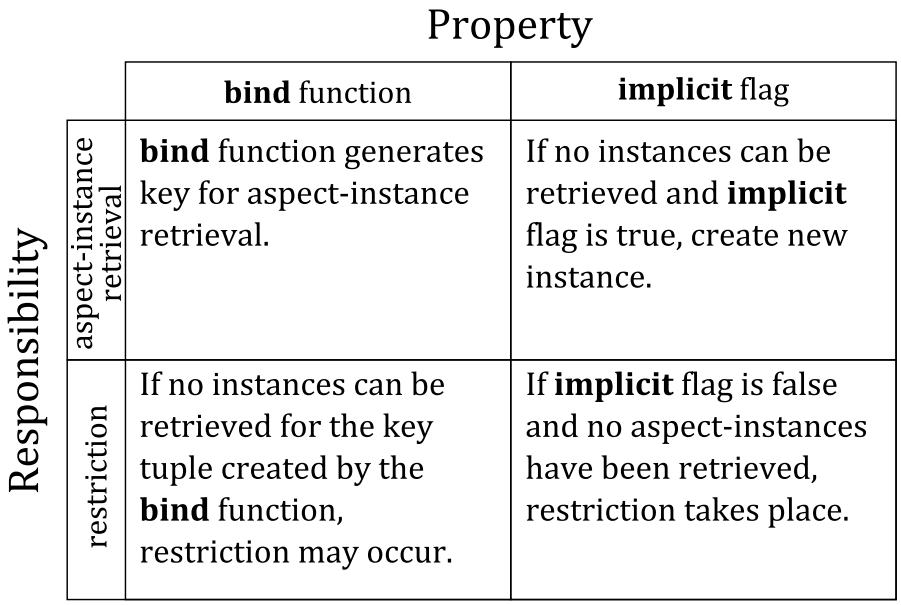


Figure 3: The relation between the semantic properties and responsibilities of instantiation policies.

context of the execution, such as local and global variables or the pointcut that is being evaluated. The implementation of `Context` is provided by the actual aspect-oriented execution-environment. The second parameter, an `InstantiationPolicy`, provides an abstraction of the four characteristics that vary between different instantiation policies: **bind**, **implicit**, **find** and **store**. **Bind** and **implicit** define the semantics of the policy. **Find** and **store** are respectively responsible for looking up and "remembering" associations between key tuples and aspect instances. We do not make assumptions about the implementation of **find** and **store** in this unified model. However, we provide a generic abstraction (Section 4.7) and suggest different ways to implement it (Section 8.4).

To retrieve aspect instances, `bind` creates – according to the semantics of the instantiation policy – a **query-key tuple** consisting of values from the execution context and possibly **wildcards**. `Find` then uses this query key to look up associated aspect instances. If no matching instances can be found and the policy supports implicit instantiation, a new aspect instance is created. In such a case, the runtime environment calls `store` to remember the association between the query key and the aspect instance for later look-ups.

The rest of this chapter describes the parts of the unified template in more detail.

Algorithm 1 Generic algorithm to retrieve all applicable instances of an aspect.

```
1: function RetrieveAspectInstances(c : Context, ip : InstantiationPolicy)
2:     keyTuple ← ip.bind(c)
3:     candidates ← ip.find(keyTuple)
4:     if candidates = ∅ and isExact(keyTuple) and ip.implicit then
5:         newInstance ← newA
6:         ip.store(keyTuple, newInstance)
7:         candidates ← {newInstance}
8:     end if
9:     return candidates
10: end function
```

4.1 Definition of key tuples and query-key tuples

Let each T_i for $i > 0$ represent an arbitrary type in the base code or the special type Any (which represents any type). Furthermore, let I_{T_i} denote the set of all instances of type T_i or any specialisation derived from it, that is, $I_{T_i} = \{o | o \, is \, a \, T_i\}$, where $x \, is \, a \, T$ if and only if x is of type T or a subtype (specialisation) of T. We define the (type of) **key tuples** K_A of one specific aspect A to be ordered sequences of n elements of the type: $K_A : I_{T_1} \times I_{T_2} \times ... \times I_{T_{n-1}} \times I_{T_n}$.

Example 11. In our `NotificationLog` example (Listing 2), each unique pair of an `Observable` and an `Observer` is associated with a distinct instance of the aspect. The set of key tuples for this aspect is defined as $K_{NotificationLog} : I_{Observable} \times I_{Observer}$. An exemplary key tuple k of this aspect is $k = (x, y)$, where $x \in I_{Observable}$ and $y \in I_{Observer}$, or put simply: a pair of an observable and an observer. ▪

Example 12. The `pertarget` instantiation policy creates an association between a single object (the target of a method call) and an aspect instance. As the instantiation policy does not constrain the actual type of the target object, the key tuple in this case can be defined as $K_A : I_{Any}$. ▪

In addition, let $*$ be a special **wildcard** value. We define **query-key tuples** K_A^* to be the set of ordered sequences of length n where each element is either an instance of type T_i as defined above or the wildcard value, that is $K_A^* : I_{T_1} \cup \{*\} \times I_{T_2} \cup \{*\} \times ... \times I_{T_{n-1}} \cup \{*\} \times I_{T_n} \cup \{*\}$. Note that $K_A \subseteq K_A^*$, so each key tuple is a valid query-key tuple. We define the boolean function $isExact(k : K_A^*)$ to return `true` if and only if $\forall p \in k (p \neq *)$, that is, if k does not contain a wildcard. If a query-key tuple is **exact**, it is also a valid key tuple. That is, $isExact(k : K_A^*) \Rightarrow k \in K_A$.

The size n of a (query-) key tuple depends on the instantiation policy. Some policies (such as **pertarget** or **perthis**) have a fixed key-tuple size. Other policies (such as association aspects) support key tuples of varying sizes, depending on the use case.

4.2 The bind function

During aspect-instance retrieval, the instantiation policy creates a query-key tuple to identify the aspect instances (if any) that are applicable at the current join point. The query-key tuple is constructed by the function $bind : Context \rightarrow K_A^*$. Given the current execution context, the function returns a query-key tuple, that is, a sequence of context values or wildcards. This key is used in a later step to retrieve matching aspect instances. The **bind** function is specific for an instantiation policy and possibly also for a specific pointcut.

Example 13. As shown in Algorithm 2, the **pertarget** policy defines `bind` to always return a query-key tuple containing the target of the current method invocation. This behaviour is equal for all usages of **pertarget**. It does not depend on the pointcut expression that is being evaluated for aspect-instance retrieval. ▪

Algorithm 2 The bind function of the *pertarget* policy.

```
1: function Bind(c : Context)
2:     target ← c.getCallTarget()
3:     tuple ← (target)
4:     return tuple
5: end function
```

Example: unlike the bind function of `pertarget`, the bind function used in the case of an association-aspect depends on the pointcut expression and the size of the key tuple. Association aspects use the explicit `associated(...)` expression to determine the content of the query-key tuple. The variables referred to by `associated(...)` need to have been bound to specific context values. In the case of `NotificationLog` (Listing 2), the two variables used in the `associated(...)` expression are bound to the current caller (`this`) and the target of the invocation of `notify` (the `target` or callee) respectively. The `bind` function for this aspect retrieves those objects from the context and creates a query key tuple from them. Algorithm 3 shows a possible implementation of `bind` for the `NotificationLog` example. The size of the generated tuple and the elements are specific for this example. If instead `associated(observable, *)` was used, the second element of the key tuple would be the wildcard value $*$. ▪

4.3 Queries and the *find* function

The function $find : (k : K_A^*) \rightarrow \mathcal{P}(I_A)$ takes a query-key tuple and returns a set of aspect instances. The set returned contains all existing aspect instances which are

Algorithm 3 The bind function of the `NotificationLog` aspect

```
1: function Bind(c : Context)
2:     this ← c.getCaller()
3:     target ← c.getCallTarget()
4:     tuple ← (this, target)
5:     return tuple
6: end function
```

associated with key tuples that are equal to k, taking into consideration the presence of wildcards. That is, the associated key tuples of the returned aspect instances are equal[10] to k at all places where the corresponding element of k is not the wildcard value $*$. We call the invocation of `find` with a certain query-key tuple a **query**. We distinguish different types of queries and use q to refer to the type of a query:

exact queries The key tuple does not contain wildcards. In this case, $q = E_m$, where m is the number of elements in the query-key tuple. For example, E_3 is an exact query with a query tuple of size 3. Exact queries can return either one aspect instance or none.

full-range queries The tuple contains only wildcards. In this case, $q = F_m$, where m is the number of elements in the query-key tuple. For example, F_2 is a full-range query for a query-key tuple of size 2. Full-range queries can return zero or more aspect instances.

partial-range queries The tuple contains both wildcards and non-wildcard values. We call the positions of the wildcards in a partial-range query the **layout** of the query-key tuple. Thus, $(x, y, z, *, *)$ has a different layout than $(x, *, y, z, *)$. When all wildcards appear after all non-wildcards (e.g. $(x, y, z, *, *)$), the query is a **prefix query**. We refer to prefix queries for key tuple sizes m containing w wildcards as $q = P_{m,w}$. For example, $P_{4,2}$ is a prefix query with a query-key tuple of length 4, having 2 wildcards at the end ($(x, y, *, *)$). When all wildcards appear before all non-wildcards, the query is a **suffix query**. We refer to suffix queries for key tuple sizes m containing w wildcards as $q = S_{m,w}$. For example, $S_{4,2}$ is a prefix query with a query-key tuple of length 4, having 2 wildcards at the beginning ($(*, *, x, y)$). Finally, if wildcards and non-wildcards appear in any order within the query, the query is a **mixed query**. We refer to mixed queries for key tuple sizes m containing w wildcards as $q = M_{m,w}$. For example,

[10] We use the term *equal* here without specifying how equality is determined. It can refer to identity (the two values compared are the same object) or equality of value (the two values represent the same value, but are not necessarily the same object). Which kind of equality is used is part of the semantics of an instantiation policy.

M is a prefix query with a query-key tuple of length 4, containing 2 wildcards (e.g. $(x, *, y, *)$ or $(x, *, *, w)$). Like full-range queries, partial-range queries can return zero or more aspect instances.

If a data structure benefits from a certain layout (for example, a prefix layout or a suffix layout), it can re-arrange the elements in the (query-)key tuple before it is used to store or look up aspect instances. This transformation is done by a **transform function** $T_A : K_A^* \to K_A^*$.

Example 14. Assume an association aspect that associates 3-tuples of object to an aspect instance. Assume further that the data structure used to store the existing aspect-instances can handle prefix queries especially fast. A pointcut expression of the aspect contains a mixed partial-range query of the form `associated(*,a,b)`. When the aspect is deployed, the aspect-oriented execution-environment may define a transform function that moves the wildcard to the end of the tuple, making it a prefix query. That is, the transform function T transforms an original query-key tuple such as $(x, *, z)$ into the query-key tuple $(x, z, *)$. ▪

There are a number of limitations with respect to the transform function:

- The same transform function must be applied when storing an aspect instance for a given key tuple and when retrieving aspect instances using a query-key tuple.

- Only one transform function can be used for a given aspect-instance storage. If multiple partial-range queries with different layouts need to be executed, all of them have to be transformed using the same transform function. For example, consider a case where the two queries `associated(*,a,b)` and `associated(x,*,y)` need to be executed on the same storage which prefers prefix queries. No single transform function can transform these queries both to prefix queries. For example, a transform function that transforms `associated(*,a,b)` into `associated(a,b,*)` (prefix layout) will transform `associated(x,*,y)` into `associated(*,y,x)` (suffix layout).

- The transform function cannot change as long as any aspect-instances are stored. If the transform function was replaced as long as aspect-instances are stored, subsequent retrieval attempts would create query-key tuples that may match no the aspect instances or the wrong aspect instances.

Depending on the semantics of the instantiation policy, `find` does not necessarily need to take the order of the query key tuple into consideration. It may instead choose to treat the key tuple as a set or a multi-set (a set which allows elements to

Listing 9: Setting up a *NotificationLog*

```
1 Observable observable = new Observable();
2 Observer observer = new Observer();
3 observable.addListener(observer);
4
5 NotificationLog log = new NotificationLog(observable, observer);
```

appear more than once) instead of a sequence. In this case, there is no practical difference between a prefix, suffix or mixed query in the first place, as order does not matter. Hence, no transform function is required.

Example: association aspects take the order of the key tuple sequence into consideration when finding associated aspect instances for a given query-key tuple. The expression `associated(x,y)` will only cause an aspect instance to be retrieved if an association has been established before, using `associate(x,y)`. An association established with `associate(y,x)` will be ignored in this particular case. For many scenarios this is a reasonable restriction. However, scenarios are imaginable where the order of elements is not important. Considering the `Bit` example (Listing 1), one may be interested in situations where one bit calls `set` on another bit, if and only if an association has been created involving those two bits, no matter which order. To achieve this using the original association aspects implementation, two symmetric `associated(x,y)` expressions are required (`this(l) && target(r) && (associated(l,r) || associated(r,l))`) . For scenarios involving larger key tuples, this can become increasingly complex. If association aspects had an option to ignore the order of the key-tuple elements, those cases would become simpler. ▪

4.4 The *store* function

As a counterpart to `find`, $store : (k : K_A, i : I_A)$ remembers a new association between a key tuple and an aspect instance. Note that unlike `find`, `store` expects a key tuple instead of a query-key tuple, that is, a tuple that does not contain wildcards. In our model, the relation between key tuples and aspect instances is one-on-one. Thus, if there already is an aspect instance associated with the provided key tuple, we assume that `store` replaces this association. The `store` function is only called by `RetrieveAspectIntances` if implicit instantiation is supported (see Section 4.6) and if the query-key tuple is exact (see Section 4.5). Only in the latter case can the query-key tuple be transformed into a key-tuple. If explicit instantiation is required, `store` has to be called explicitly to associate a key-tuple with an aspect instance.

Example 15. Association aspects require explicit instantiation of the aspects. An association is created by calling the special `associate` method. Listing 9 shows an

example how a `NotificationLog` would be set up in practice. Given an existing pair of an `Observable` and an `Observer`, a new instance of `NotificationLog` is created. The `Observable` and the `Observer` are passed as arguments to the constructor. The constructor calls `associate` as shown in Listing 2. The call to `associate` is equivalent to the call of *store(k,i)* with k being a tuple consisting of an `Observable` and an `Observer` and i being the instance of `NotificationLog` which receives the call to `associate`. .

4.5 isExact

As defined in Section 4.1, the `isExact` function determines if a query-key tuple contains any wildcards. This function can be implemented generically as shown in Algorithm 4.

Algorithm 4 Generic implementation of isExact

```
1: function isExact(k : KeyTuple)
2:     for e in k do
3:         if e is wildcard then
4:             return False
5:         end if
6:     end for
7:     return True
8: end function
```

For instantiation policies that never allow wildcards (such as **pertarget**), `isExact` may be replaced with a constant value, or the call may be skipped altogether. We consider this an implementation detail and optional optimisation.

4.6 Implicit instantiation

Implicit instantiation takes place when three conditions are met: first, `find` returns an empty set for the query-key tuple generated by `bind`. Second, the query-key tuple is *exact*, that is, it does not contain wildcards (if the query-key tuple is exact it can be represented as a key tuple, which is required by `store`). And finally, the instantiation policy supports implicit instantiation, that is, the **implicit** flag is `true`.

4.7 Generic Storage Function

Although we do not make assumptions about the actual implementation of `find` and `store`, we give a formal definition of a generic storage function in this section. Let I_A

be an instance of aspect A. At each point in time the set of all existing associations of key tuples and aspect instances[11] is defined by a relation of type $R : K_A \times I_A$ that contains associations between key tuples and aspect instances. Since we define that each key tuple is associated with at most one aspect instance, the relation is in fact a function. The domain of this function is only a subset of all possible key tuples. It consists of all key tuples which are associated with an aspect instance. Thus, the function is only partial. We call this partial function $S_A : K_A \nrightarrow I_A$ the **storage function of** A**.** We assume that no aspect instances are created that are not associated with a key tuple. Storing an association of a key tuple and an aspect-instance means defining the function for the key tuple. We can formalize the creation and removal of associations by defining several helper functions on S_A.

1. isDefined

$$isDefined : (s : S_A, k : K_A) \rightarrow Boolean \triangleq \begin{cases} true & \text{if } k \in \textbf{domain}\, s \\ false & \text{else} \end{cases}$$

The function $isDefined(s, k)$ takes a storage function and a key tuple and returns $true$ if $s(k)$ is defined (that is, k is in the domain of s) or $false$ otherwise.

2. define

$$define : (s : S_A, k : K_A, i : I_A) \rightarrow S_A$$

The function $define(f, k, a)$ "stores" a new association of a key tuple with an aspect instance. The function takes the storage function, a key tuple and an aspect instance and returns a new storage function s' which is equal to s, except that $s'(k) \triangleq a$. For each $t \in K_A$

$$t = k \Rightarrow isDefined(s', t) \wedge s'(t) = i \tag{1}$$
$$isDefined(s, t) \wedge t \neq k \Rightarrow isDefined(s', t) \wedge s'(t) = s(t) \tag{2}$$
$$\neg isDefined(s, t) \wedge t \neq k \Rightarrow \neg isDefined(s', t) \tag{3}$$

Equation 1 defines the new function for the key tuple k, causing it to return the aspect instance i for the key tuple k. Equation 2 causes the returned function to

[11] Instances of one specific aspect.

return the same value as s for all other elements in the domain of s. Line Equation 3 causes the returned function not be defined for any element that is neither in the domain of s nor the key tuple k.

3. undefine

Finally, we introduce a function to remove a definition of the function for a specific key tuple:

$$undefine : (s : S_A, k : K_A) \rightarrow S_A$$

$undefine(s, k)$ takes a storage function and a key tuple and returns a new storage function s' that is not defined for k and that return the same value as s for all other tuples. That is, for each $t \in K_A$

$$isDefined(s, t) \wedge t \neq k \Rightarrow isDefined(s', t) \wedge s'(t) = f(t) \tag{4}$$
$$\neg isDefined(s, t) \vee t = k \Rightarrow \neg isDefined(s', t) \tag{5}$$

Equation 4 ensures that the returned function s' is defined equally to s for all elements in the domain of s except for k. Equation 5 causes s' to be undefined for k or any key tuple for which s is not defined.

4.8 Summary

The unified model helps us to answer the first two of our research questions.

Question #1: Which operations are required to implement instantiation policies in general?

We identified four basic properties that are used to implement an instantiation property:

1. The **bind** function that defines how the query key-tuple is built from context values.

2. The **implicit** flag that determines if an instantiation policy supports implicit instantiation.

3. The **store** function that registers associations between key tuples and and aspect instances.

4. The **find** function that is used to find aspect instances for a given key-tuple.

We assume that aspects stay deployed once they have been deployed. Otherwise, a function to remove existing associations is required in addition.

Question #2: Which operations can be optimised without affecting the semantics of instantiation policies?

The **store** and **find** functions are independent of the semantics of an instantiation policy. Therefore, we are interested in optimising these two functions. The **bind** function, on the other hand, defines the semantics of an instantiation policy. We expect this function to vary for each instantiation policy.

5 Criteria

To compare data structures and the operations that are executed on them, criteria need to be defined. As it is our goal to make the look-up of aspect instances faster, our main criterion is the time that the different operations take to perform the operations defined in Section 4, specifically the **find** and **store** functions. We differentiate between the theoretic asymptotic time complexity complexity of those operations and the clock time used by actual implementations of the operations and data structures. However, the amount of memory that the data structures occupy in relation to the number of aspect instances is also of importance, as memory is as much a limited resource as time in certain usage scenarios (such as mobile devices).

5.1 Asymptotic time complexity

One way to describe and compare the theoretic amount of time different algorithms require for a specific task is the *(asymptotic) computational complexity* [21]. It is used to describe the time and space requirements of algorithms and computational problems with respect to the size of the input data. Because the actual computational complexity of an algorithm often depends on the input data, the actual time and space requirements depends on the individual case. As not all cases can be considered in practice, typically only the *best*, *worst* and *average* cases are investigated.

Example 16. Given an unsorted list of numbers of length N one wants to find out if a specific number is present in this list. Because the list is unsorted one needs to compare the elements in the list one by one until the desired element is found or the end of the list is reached. Let $L = [4, 2, 7, 5]$ be such a list where $N = 4$. Let us further assume that we always begin at the first element of the list when searching for a specific element. If we want to find out if 2 is an element of this list, we start by comparing the first element (4) to 2. They are not equal, so we proceed to the next element (2) and compare it to 2. The element is equal to the sought element. It took 2 comparisons to find the element. If we try to find out if 4 is an element of the list we need to compare the first element only, as this is already the sought element. We call this the *best case*, that is, the quickest way to complete the algorithm: at least one element in the list has to be compared[12]. On the other hand, if we try to find out if 8 is an element of the list we again need to compare the elements one by one to the sought value. In this case, however, four comparisons are required, one for each element in the list, because 8 is not an element in this list. This is *worst case:* it takes at most N comparisons to find an element in an unsorted list with N elements.

[12] If the list is not empty.

In addition to the best and worst case, the *average case* can also be considered. That is, on average, how many comparisons does it take to find an element in the list. Typically the average case is harder to determine because it is difficult to define the average input. It often depends on probabilistic properties. For example, if it is equally likely that the sought element is present at position 1, 2, 3, 4 or not part of the list, the average number of comparisons is $(1 + 2 + 3 + 4 + 4)/5 = 2.8$ [13]. ▪

The Big-O notation [22] is usually used to give an upper limit of the computational complexity and it is the measurement that is typically used when quantifying computational complexity. If the computational complexity of an algorithm is given by $f(n)$ (where n is a quantification of the data on which the algorithm works), then $f(n) \in O(g(n))$ means that the computational complexity of this algorithm does not grow faster than $M\, g(n)$ (where M is a constant). That is $|f(x)| \leq M\, |g(x)|$ for $x > x_0$: from a certain $x > x_0$, $f(x)$ is less than or equal to the value of $g(x)$ multiplied by a constant.

Example 17. If $f(N)$ gives the number of comparisons required to find a specific element in a list of length N, it will never be larger than N, no matter how long the list is – at most N comparisons are necessary. We can therefore say that $f(N) \in O(N)$. We can also define $f(N)$ to give the number of operations executed by a computer program that implements the algorithm to find the element. Such a program would basically perform a loop over all elements in the list to compare each element with the sought value until it is found or the whole list has been searched. The actual number of operations executed by such a program would be no more than $KN + C$, where N is the length of the list, K is the number of operations required to compare a single element and C is some constant overhead to set up loop variables etc. Still, $f(n) \in O(n)$, assuming $N > 0$, as this says that

$$\begin{aligned} & KN + C && \leq MN \\ \iff\ & K + {}^{C}/_{N} && \leq M \end{aligned}$$

For any combination of K and C, we can find an M such that $M \geq K + {}^{C}/_{N}$. For example, assume that $K = 8$ and $C = 4$. Then $M = 12$ would suffice this inequation for any $N > 0$. For $N = 1$, $12 \geq 8 + {}^{4}/_{1}$ and the fraction ${}^{4}/_{N}$ will become smaller the larger N gets. Therefore the inequation will still hold for $N > 1$.

We can therefore say that the time complexity of searching a list linearly has an upper limit of $O(N)$: if the list is twice as long, in theory searching in this list takes twice as long, too, in the worst case. ▪

A selection of typical *complexity classes* is shown in Table 2 on page 41. The complexity classes help us compare algorithms. A sorting algorithm that has a worst

[13] Two cases require 4 comparisons: 1) the element is at position 4 and 2) the element is not in the list

case complexity of $O(N \cdot log\, N)$ is expected to work faster for large input datasets than a sorting algorithm of class $O(N^2)$. This even holds for different implementations: for sufficiently large input data even a highly optimized $O(N^2)$ algorithm will take longer than a badly optimized $O(N \cdot log\, N)$ algorithm.

Table 2: Examples of complexity classes

Complexity class	Description
$O(1)$	The complexity does not depend on the size of the input dataset.
$O(N)$	The complexity grows linearly with the size of the input dataset.
$O(N^2)$	The growth of the complexity is quadratic to the size of the input dataset.
$O(log\, N)$	The growth of the complexity is logarithmically to the size of the input dataset.

The best, average and worse case can all be classified using the Big-O notation. That is, the Big-O notation does not automatically refer to the worst case complexity of an algorithm, although typically, the complexity of the worst case is used to classify algorithms. For example, the *Bubblesort* sort algorithm [25, p. 106] has a worst case complexity (if the list is reversely sorted) of $O(n^2)$, but a best case complexity (if the list is sorted) of $O(n)$. Still, Bubblesort is typically referred to as a $O(n^2)$ algorithm.

5.2 Actually required time

As described in Section 5.1 the asymptotic computational complexity describes the time and space requirements of an algorithm in the worst, best or average case. While the Big-O notation helps us compare algorithms, it does not suffice to predict the actual time an algorithm needs to perform its task in practice, that is, when implemented in software running actual scenarios. We can evaluate the execution time in practice either absolutely or relatively.

To evaluate the absolute execution time of a specific algorithm, we need to implement it as a software program that executes specific scenarios. Coming back to our searchable list from Section 5.1, we would have to implement such a search algorithm to measure its actual execution time. We could then run the algorithm on lists of different lengths and content and measure the required time. The time measured does not tell us much about the suitability of the algorithm for specific scenarios. The absolute time depends on many different variables, for example on the system that is used to execute the algorithm or how the algorithm is implemented exactly. A

faster system will execute the algorithm faster than a slower system. It is difficult to say if the algorithm is “good” if it is evaluated in isolation.

Therefore, we are more interested in the relative execution time of different algorithms. For example, consider two sort algorithms A and B. If A sorts any input set in less time than B (and if execution time is the only criterion we are interested in), then using A is certainly a better choice than B. We still cannot say if A is the “best” choice or “fast enough” for a specific use case, but we can safely say that A is superior to B. In practice, determining the better of two algorithms is more difficult, because one algorithm may perform better than another in certain scenarios but worse in other scenarios. Also, an algorithm may be faster but at the same time consume prohibitive amounts of memory, whereas a slower algorithm may consume considerable less memory. To evaluate algorithms and data structures, even relatively, we therefore need to take into account these factors.

As mentioned earlier, the Big-O notation is an instrument to compare the theoretic execution speed of different algorithms. In fact, it is a *simplification* in that it uses a simple function to describe the growth of space or time requirements of an algorithm depending on the size of the input data-set. The Big-O notation describes the asymptotic behaviour, that is, the growth of the space and time requirements as the size of the input approaches infinity.

Consider an algorithm that searches a list of elements. Let $f(n)$ return the maximum number of comparisons that an actual implementation of this algorithm executes, given an input data-set of n elements. For one specific implementation of the search algorithm, the number of comparisons may be $f(n) = 3n^2 + 18n + 25$. That is, given a list of n elements, the algorithm executes at most $3n^2 + 18n + 25$ comparisons. This algorithm falls into the class of $O(n^2)$. Intuitively, this tells us that starting from a certain (possibly large) value n, $f(n)$ can be approximated by the simpler function n^2, as both functions will show similar growth. For those values of n, the power n^2 has much more influence on the total value than the factor 3, the term $18n$ or the constant 25.

More formally, $f(n) \in O(n^2)$ as $n \to \infty$ says that $f(n) \leq Mn^2$ for some $n \geq n_0$ and for some constant M. This definition shows that the Big-O notation is not sufficient to compare two algorithms for all scenarios. Firstly, it is an inequation that only gives an upper limit for the algorithm and secondly, it includes constraints that have a significant influence in practice, namely the constant factor M and the choice of n_0.

As the Big-O notation only imposes an upper limit on the complexity of an algorithm, it cannot tell us about the actual complexity of the algorithm in specific cases. Depending on the scenario, a typical input data-set may be close to the best-case. The

actual complexity may therefore be much lower than what the upper limit describes. This is especially the case for algorithms for which the worst case, average case and best case belong to different complexity classes. For example, if a linear search algorithm operates on a list that happens to be mostly sorted and the elements that are typically sought appear at the beginning of the list, then the worst case (n comparisons) will happen less likely. To evaluate algorithms we therefore need to measure their actual execution time in realistic scenarios.

The Big-O notation only holds for values of n that are larger than a specific value n_0. However, this value n_0 may be larger than the actual size of the input in a specific use case, so the inequation may not even be applicable for practical use cases. Also, the factor M allows the upper limit of the complexity of two algorithms to potentially differ by a large factor.

When we expect n to be comparatively small (as it is the case with aspect-instance storage), we see that the simplification of the Big-O notation ignores factors and constants in the formula that may influence the number of operations for small values of n more than the limiting function.

Example 18. Assume two algorithms A and B that belong to $O(n^2)$ and let $f_A(n)$ and $f_B(n)$ be the maximum number of operations that A and B respectively execute for an input size n. Now assume for example:

$$\begin{aligned} f_A(n) =& 3n^2 + 18n + 25 \\ f_B(n) =& 5n^2 + 8n + 2 \end{aligned}$$

Figure 4 on page 44 depicts the number of operations needed by the two algorithms A and B for input sizes n, $1 \leq n \leq 10$. For $n < 7$ B is faster than A. However, for $n \geq 7$, A is faster. The Big-O notation thus does not sufficiently describe the two algorithms for small values of n. ▪

Example 19. As a second example consider the two algorithms C and D of the complexity class $O(n \log n)$ with the following respective complexity:

$$\begin{aligned} f_C(n) =& 2n \log n + 5 \\ f_D(n) =& n \log n + 20 \end{aligned}$$

Figure 5 on page 45 shows the actual number of operations required for small input sizes. For $n \leq 13$, algorithm C needs fewer operations than D, while algorithm D

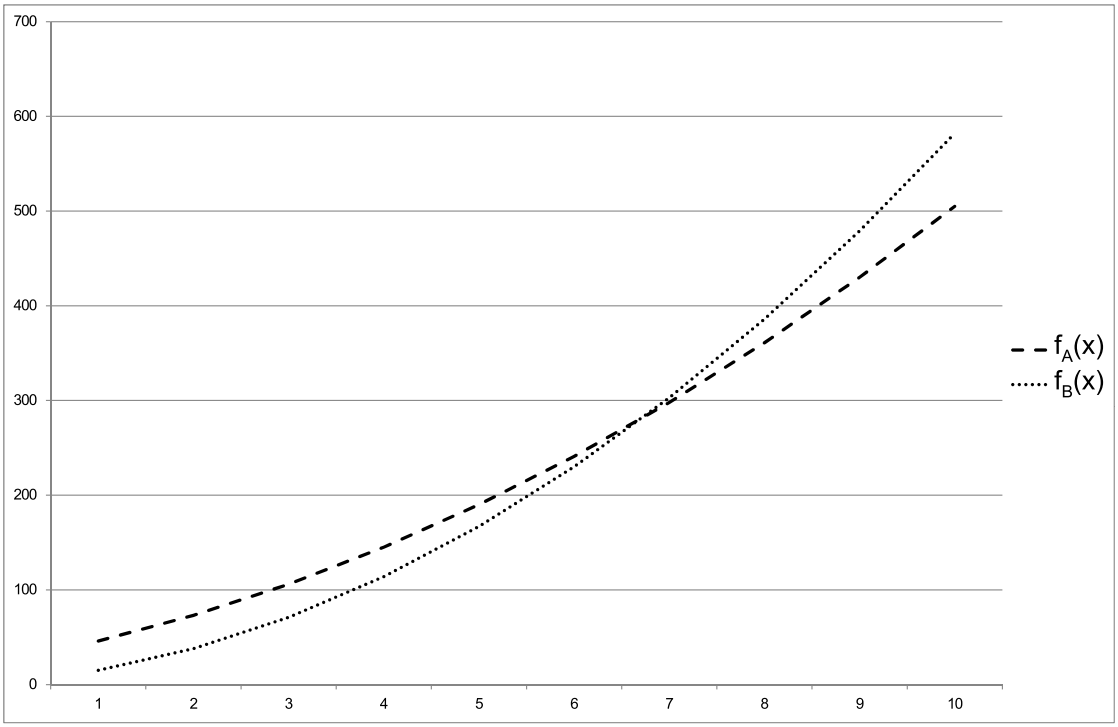


Figure 4: Number of operations (y-axis) required by A and B for a given input size n (x-axis)

uses fewer operations than C for $n \geq 14$. Although both algorithm are in the same complexity class, C is preferable for smaller input sizes and D is preferable for larger input sizes. ▪

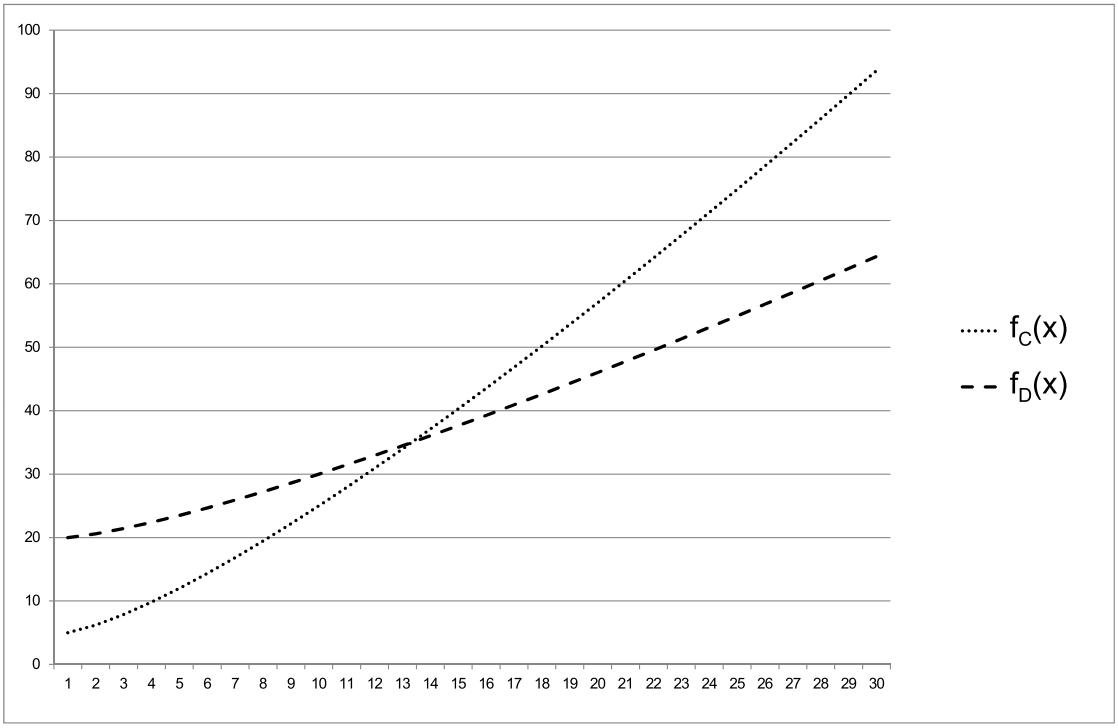


Figure 5: Number of operations (y-axis) required by C and D for a given input size n (x-axis)

5.3 Memory usage

Our major concern in this thesis is the execution speed of algorithms on different data structures. However, memory usage is a concern as well. If one algorithm achieves better execution speed only by using large amounts of memory the choice for this algorithm may not be as obvious. The actual memory requirements of different data structures and algorithm depends on a number of factors, such as the system architecture and implementation details. Instead of giving exact numbers (such as the number of bytes used with respect to the number of stored elements), we rather want to give a qualitative estimation of relative memory usage, that is, whether typical implementations of one data structure use notably more memory than other data structures. We consider Java as the execution platform and make a number of assumptions about the object layout.

6 Scenarios

The primary goal of this research is to recommend data structures that can be used in a reference implementation for aspect-instance storage and look-up. We evaluate possible data structures according to the criteria described in Section 5. We expect different data structures to perform significantly different depending on the scenarios in which they are used. That is, we do not expect a single data structure to outperform other data structures in all use cases. Therefore, we will evaluate the data structures in different scenarios and recommend specific data structures per scenario.

The choice of scenarios must reflect the fact that we want our evaluation to be qualitatively accurate. We do not assume a specific aspect-oriented runtime-environment or hardware platform. Instead, we want our results to be transferable to any environment. Many factors influence the actual execution speed of algorithms or memory consumption of data structures, such as the hardware architecture, the software platform and micro-optimisations. However, we expect the results to be *relatively* similar. An algorithm that is a good choice in a specific scenario in one environment should be a good choice when used in a different environment (but in the same usage scenario). We are aware that there are architectural differences that may influence the behaviour of algorithms, such as multi-threaded environments compared to single-threaded environments. An actual implementation of the reference implementation should therefore always we measured again to verify the suitability of the chose data structures and algorithms.

The scenarios we have chosen focus primarily on the implementation of the data storage for aspect instances, not on the environment in which they are used.

6.1 Aspect-instance look-up

The most important scenarios are those that represent aspect-instance look-up. One driving motivation of this research is the fact that aspect-instance retrieval needs to be fast, because it may happen many times during the execution of a program. As shown in Section 4, when a pointcut is evaluated, an instantiation policy creates a query-key tuple with context values (or wildcards). This query-key tuple is used to find all aspect-instances that are associated with matching key-tuples. We expect three factors to have the most significant influence on the execution speed of aspect-instance look-up: the number n of aspect instances stored in the data structures, the size m of the query-key tuple (that is, the number of elements in the key) and

the type q of the query (exact, full range or partial range and in the latter case the number and position of the wildcards).

Ideally, scenarios would be chosen in which those influencing factors represent real-life usage. This turns out to be difficult to achieve. We have no information about the "typical" number of aspect instances that exist during the execution of a program. Instead, we have chosen to use the simplest case ($n = 0$) and a number of powers of ten ($n \in \{1, 10, 100, 1000, 10000\}$) and evaluate how the execution speed scales. This allows us to predict results for different numbers of aspect-instances.

For the query-key tuple size, we first chose the sizes used by the instantiation policies described in Section 2.2. That is, $m = 0$ to represent singleton aspects, $m = 1$ to represent per-object instantiation policies and $m = 2$ to represent the simplest case of association aspects. To evaluate the scalability of the algorithms, we additionally evaluated key sizes of $m \geq 3 \wedge m \leq 5$. A case in which $m = 5$, that is, an aspect that needs to share state with respect to five different context values, is already a rather complex scenario. We do not expect m to be frequently much larger in practice.

With respect to query types, there are a couple of choices. A query can be either exact ($q = E_m$), full-range ($q = F_m$) or partial-range with the sub-types prefix, suffix and mixed query. We included both exact and full-range queries in our scenarios. As explained in Section 4.3, the layout of partial-range queries can be transformed to be beneficial for a specific data structure by using a transform function. We also showed that in cases where multiple queries with different layouts need to be supported, it can be impossible to transform all queries to the most beneficial form. To reflect that, we included two types of partial-range queries. One in which the query-key tuple has the most "beneficial" layout (best case), and one in which the layout of the query-key tuple is the least beneficial (worst case). The initial layout of the query is always the prefix layout. The transformation from the initial layout to the respective best and worst case is done by a transform function supplied by the data storage.

For aspect-instance look-up, we evaluate the following scenarios:

Singleton. In this scenario, the key length is fixed to 0 ($m = 0$), the query is always exact ($q = E_0$). We consider the case where an aspect instance was not stored yet ($n = 0$) and where an instance is already present ($n = 1$). This scenario represents singleton aspects and is the simplest scenario, in which state is shared between all invocations of the advice.

Per-object. In these scenarios, the key length is fixed to 1 ($m = 1$). The query can be either exact ($q = E_1$), returning a specific aspect instance, or full-range ($q = F_1$), returning all registered aspect-instances. This scenario is evaluated for $n \in \{0, 1, 10, 100, 1000, 10000\}$. This scenario represents

per-object instantiation (**pertarget**, **perthis**) in applications of varying complexity.

Pair-wise association. In these scenarios, the key length is fixed to 2 ($m = 2$). We test exact queries, ($q = E_2$, returning a specific aspect instance), full-range queries ($q = F_2$, returning all registered aspect-instances), and prefix queries with one wildcard ($q = P_{2,1}$) using a best-case and a worst-case transform function. This scenario is evaluated for $n \in \{0, 1, 10, 100, 1000, 10000\}$. This scenario represents the simplest case of association aspects (pairs of objects associated with an aspect instance) in applications of varying complexity.

Complex association. In these scenarios, key lengths of 3 to 5 are evaluated ($m \in \{3, 4, 5\}$). We test exact queries ($q = E_m$), full-range queries ($q = F_m$), and prefix queries with up to $m - 1$ wildcards ($q = P_{m,p}$ where $1 \leq p \leq m - 1$), using a best-case and worst-case transform function. This scenario is evaluated for $n \in \{0, 1, 10, 100, 1000, 10000\}$. This is the most complex and most general case, associating a number of objects to aspect instances in an application of varying complexity.

For all scenarios, we assume that the layout of all queries that need to be executed is known at deploy time. That is, no new queries are added at runtime while aspect instances are already deployed. In all cases where $q = E_m$ and $m \geq 1$, we also consider the case where the query-key tuple matches no existing aspect instances. This scenario is especially interesting for instantiation policies that support implicit instantiation. Finding out that no matching aspect instance exists should be as fast as possible.

7 Data structures

According to the unified model described in Section 4, instantiation policies create an association between a key tuple and an instance of an aspect. During the execution of the program those associations need to be stored and maintained by the instantiation policy. An association consists of a key tuple and an associated aspect instance. Because the key tuple uniquely identifies an association, finding an association within a data structure only involves comparison of the key tuple. The associated aspect instance is considered the payload (or *value*) associated with the key tuple (or *key* hereafter). We use "finding an association" synonymously to "finding an association with a specific key tuple".

A data structure used for storing the associations between key tuples and aspect instances needs to support at least the following operations:

1. *Exact queries* – retrieving an aspect instance given an exact query-key tuple, that is, a key tuple without wild cards. This operation returns the aspect instance that is associated with a key tuple equal to the provided query-key tuple.

2. *Partial-range queries* – retrieving all aspect instances given a query-key tuple with wildcards. The operation returns all aspect instance associated with key-tuples that matches the query-key tuple for all non-wildcard elements, that is, where each non-wildcard element in the query-key tuple is equal to the respective element in the associated key-tuple.

3. *Full-range queries* – retrieving all aspect-instances stored in the data structure.

4. *Insertion* – adding a new association to the data structure. This operation is performed whenever a new association between a key tuple and an aspect instance is established.

5. *Removal* – removing an association from the data structure given at least the key tuple of the association. This operation is needed when an association between a key tuple and an aspect instance is explicitly destroyed.

Data structures are conceptually independent of programming languages and runtime environments. The actual memory usage of data structures depends on the environment of the implementation, the implementation of the data structure and in some cases the stored data itself. To make the expected memory usage comparable, we estimate the average memory usage of a typical implementation in Java. We use the following methodology for our estimation:

- $sizeof(Object)$ refers to the minimum size of an object, that is, the memory used by an object without any explicitly defined fields. In Java, this is typically the size of the object header, consisting of two words, that is, 2x4 bytes on a 32 bit runtime and 2x8 bytes on a 64 bit runtime.

- $sizeof(Ref)$ refers to the size of a reference to an object. This may be the native pointer size for the respective platform (4 bytes on a 32 bit system, 8 bytes on a 64 bit system), although this is not necessarily guaranteed.

- $sizeof(Word)$ refers to the native word size, that is, typically 4 bytes on 32 bit systems or 8 byte on 64 bit systems.

- We assume gapless alignment of fields. That is, all fields of an object are stored in consecutive order without any free bytes for alignment between them.

- We assume that an array of length n uses (in addition to the object header) a word to store the length + n times the size of an element.

- We never consider the size of the aspect instance itself nor the size of objects representing context values, as those do not vary across different instantiation policies. We do, however, consider the size of any helper objects that are used to store references to context values or aspect instances, such as an array that represents the key-tuple.

To illustrate the differences between the data structures, we pick up the `Equality` example given by Sakurai et al. [10] as described in 2.2.1 on page 8.

In the remainder of the section we will compare different data structures with respect to the asymptotic time complexity of those operations and the memory complexity of the data structure. The data structures we describe in this section are typical choices for searchable data storages that are used in search problems.

7.1 Arrays

An array is a consecutive block of memory that can hold a number of homogeneous elements. Arrays allow direct access to each element by using the element's index. We distinguish unsorted arrays from sorted arrays. In unsorted arrays, the elements have no specific order. In sorted arrays, the elements are sorted according to some sort order.

Asymptotic time complexity of basic operations on arrays

Table 3 on the next page summarises the average complexity of the basic operations on arrays.

Exact query In an unsorted array, finding the element with a given key tuple has linear complexity. Potentially all elements need to be visited to find the element with a specific key tuple. If all key tuples are sought equally frequently, on average half of the elements need to be visited. As soon as the first element with a matching key tuple is found, the search procedure can terminate because no other element can have the same key tuple.
In a sorted array, finding an element given an exact key has logarithmic complexity, because a binary search algorithm can be used to find the element.

Partial-range query Partial-range queries in unsorted arrays are comparable to exact queries in that they have linear complexity. However, because multiple elements can exist that have a key tuple matching the sought key tuples, all elements have to be visited every time. That is, the average and worst case complexity are equal.
Partial-range queries in sorted arrays also have linear complexity in general. If the query-key tuple can be transformed into a prefix form and the elements are sorted by this prefix, all matching values occur consecutively in the array. Finding the first element can be done in $O(log\, n)$. The average complexity is $O(log\, n)$, if the average number of found elements is independent of n.

Full-range query Full range queries have linear complexity in both sorted and unsorted arrays because each value must be returned. However, no comparison of key elements is necessary because each value is returned indiscriminately.

Insertion In an unsorted array, new elements can be inserted at the index following the index of the last occupied location (assuming that we do not allow gaps in the array, that is, empty locations between occupied locations). If the array is not completely occupied, the time complexity for insertion is constant ($O(1)$). However, if all locations are occupied, the array needs needs to be resized. Because arrays can typically not be resized in-place, a new array with a larger size needs to be allocated and all elements have to be copied from the old array to the new array. This operation has linear time complexity with respect to the number of elements in the array. The amortised complexity of insertion is still considered to be constant.
If the array is sorted, insertion is done in two phases: first, the location is determined at which the element needs to be inserted. Using a binary search algorithm, this can be done with logarithmic complexity ($O(log\, n)$). After that,

Table 3: Average asymptotic time complexity of basic operations on an array.

Operation	Unsorted array	Sorted array
Exact query	$O(n)$	$O(log\, n)$
Partial-range query	$O(n)$	$O(n)$, $O(log\, n)$ (prefix search)
Full-range query	$O(N)$	$O(n)$
Insertion	$O(1)$ (amortised)	$O(n)$
Removal	$O(n)$	$O(n)$

if the insertion index refers to an already occupied location, the element at this and all following locations needs to be moved by one position in the direction of increasing indices, which again has linear complexity ($O(n)$). Also, the array might need to be resized as in the case of an unsorted array. The overall complexity is thus $O(n)$.

Removal Removing an element from an unsorted array has linear time complexity. First, the position of the element to remove needs to be found by iterating linearly through the array and comparing the key tuple of the element with the key tuple that defines the element to remove. If the position is found, all elements coming after the removed position need to be moved one position in the direction of decreasing indices to guarantee no gaps in the array.
If the array is sorted, the first step (finding the position to remove) can be done in logarithmic time by using a binary search algorithm that compares the key tuple of the elements with the provided key tuple that identifies the element to remove. However, closing the gap has linear complexity as for unsorted array, causing an overall linear complexity in the case of sorted arrays as well.

Space complexity

In Java, an array of objects does not store the objects themselves consecutively in memory. Instead, the array only contains references to the objects. Exceptions are arrays of primitive types (int, float etc.) which contain the primitive data directly. In our case we only consider the case of arrays storing object references.

A possible way to store an association in an array is to use instances of a class as shown in Listing 10 as elements. The key tuple is represented as an array of object references.

We assume that an array of object references with a length N consists of a word that holds the size of the array, a continuous memory block of N object references of size $sizeof(Ref)$, plus the normal object overhead itself. That is,

Listing 10: A generic Java class that represents an association between a key tuple and an aspect instance of the provided type $Aspect$.

```
1 class Association<Aspect> {
2   public Object[] keyTuple;
3   public Aspect aspectInstance;
4 }
```

$$sizeof(Ref[N]) = sizeof(Object) + sizeof(Word) + N * sizeof(Ref)$$

The size of an instance of `Association` for key tuples of length M is therefore:

$$sizeof(Association_M) = sizeof(Object) + sizeof(Ref[M]) + sizeof(Ref)$$

When a new association needs to be stored and the array is fully occupied, a new, larger array must be created that can hold all previous associations and at least the new association. Because arrays can typically not be resized in-placed and the required number of entries (that is, the number of aspect instances) might not be predictable, arrays are often over-allocated in practice to reduce the frequency of array allocations when the array becomes completely occupied. A typical choice of the size of the new array is to double the size of the current array. That is, if the current array of N elements is completely occupied, $2N$ is chosen as the size of the new array.

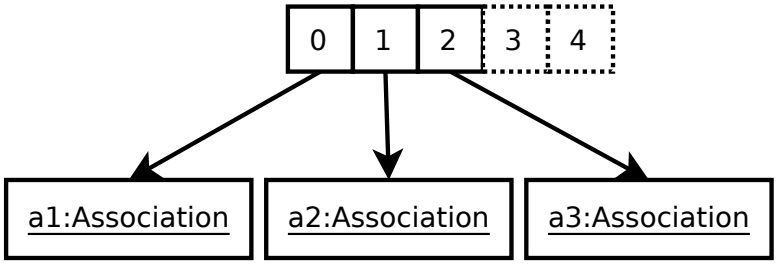
The total memory footprint for an array-based storage with a capacity of C elements, a key-tuple size of M and N stored associations is therefore:

$$\begin{aligned} &sizeof(ArrayStorage_{N,M,C}) = sizeof(Ref[C]) + N * sizeof(Association_M) \\ &= sizeof(Ref[C]) + N * sizeof(Object) + N * sizeof(Ref[M]) + N * sizeof(Ref) \end{aligned}$$

Ideally, we want N to be as close to C as possible, that is, we want the array to be as fully occupied as possible to reduce the amount of "wasted" memory.

Instead of using an array of `Association` instances, an array based storage can be implemented by using two arrays: one array of key-tuple arrays (`Object[][] keyTuples`) and another array holding references to the aspects (`Aspect[] aspectInstances`). The same index is then used for both `keyTuples` and `aspectInstances` to refer to

Figure 6: Array of length 5. Boxes with dotted outlines refer to unused array positions. Arrows indicate references to objects.



the same association. This removes the per-element overhead associated with the `Association` instance, resulting in a size as follows:

$$sizeof(ArrayStorage_{N,M,C}) \quad = \quad 2 \ * \ sizeof(Ref[C]) \ + \ N \ * \ sizeof(Ref[M])$$

As long as the number of associations stored in the arrays is high enough ($N \gtrsim C/3$), this leads to a smaller total memory footprint than the approach using `Association` instances.

Example

In the *Equality* example, the instantiation policy can store each established association between two bits and an instance of `Equality` in an array of `Association` instances. Figure 6 on page 54 shows a schematic view of the array and the containing references. The first three positions of the array contain references to three distinct instances of `Association`, the last two positions are unused ($null$ in Java). Note that while only three positions in the array are actually used, the array's capacity is 5.

7.2 Linked Lists

Unlike arrays, linked lists [26] do not store all elements as a consecutive block. Rather each element is stored independently together with a pointer (a link) to the next element of the list. By following one link after another, all elements of the linked list can be accessed. Different linking strategies exist. In a single-linked list, each

element has a single pointer to the next element. A double-linked list also maintains a pointer to the previous element. A more complex linked list is the so-called skip list, in which each element store links not only to the next element but also one or more links to elements farther away, which allows more efficient navigation in certain cases.

Asymptotic time complexity of basic operations on linked list

Exact query Searching for an element with a specific key requires navigating along the *next* pointers of the elements until the element with the requested key tuple has been found. This bears linear complexity.

Range query Like range queries on unsorted arrays, range queries on linked lists require all elements to be visited which has linear complexity.

Insertion Adding an element to a single-linked list can be done in constant time if a reference to the last element of the list (the tail) is available.

Removal Removing an element from a single-linked list requires two steps. First, the element to remove needs to be found using an exact query (with linear complexity). Second, the element needs to be removed. This can be done in constant time if the predecessor and the successor of the removed node are known: the successor node of the removed element is simple set as the successor node of the predecessor node of the removed node. The overall complexity is thus linear with respect to the number of elements in the list.

Space complexity

Listing 11: A generic single-linked list node to store associations of aspect-instances and key tuples

```
1 class ListNode<Aspect> {
2   public Object[] keyTuple;
3   public Aspect aspectInstance;
4   public ListNode<Aspect> next;
5 }
```

The per-element memory overhead of linked lists depends on the number of links that is maintained per element: single-linked lists store one additional pointer per element, double-linked lists store two additional pointers and skip lists might store even more pointers per element. Single-linked lists are typically implemented by

using node objects which hold the value and a link to the next element in the list, that is, the next node. An example of such a node usable for aspect-instance storage is shown in Listing 11. The node consists of the key tuple, a reference to the aspect instance and a reference to the next node. Thus, the size is of a node with key-tuple size M is:

$$sizeof(ListNode_M) = sizeof(Object) + 2 * sizeof(Ref) + sizeof(Ref[M])$$

When a single-linked list is used, over-allocation typically does not occur, that is, only the number of required nodes is created. As a minimum requirement, an aspect-instance storage using a linked-list needs to maintain a reference to the first node. Thus, the total size of an aspect-instance storage for key-tuples of size M, holding N aspect instances can be estimated with:

$$\begin{aligned} sizeof(LinkedListStorage_{N,M}) &= sizeof(Ref) + N * sizeof(ListNode_M) \\ &= (2N+1)sizeof(Ref) + N * sizeof(Object) + N * sizeof(Ref[M])) \end{aligned}$$

Example

An element of a single-linked list that can be used to store associations is shown in Listing 11. The field `next` holds a reference to the next element in the linked list.

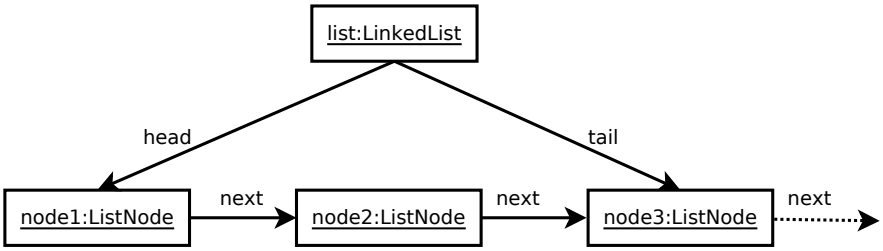
Listing 12 shows part of a single-linked list. The fields `head` and `tail` point to the first and last node in the list respectively, `count` refers to the current number of elements in the list. This list implementation adds slightly more overhead than the minimum described in the previous section. However, the overhead is constant.

Listing 12: A simple single-linked list.

```
1 class LinkedList<Aspect> {
2   private int count;
3   private ListNode<Aspect> head;
4   private ListNode<Aspect> tail;
5 }
```

Figure 7 on page 57 shows the objects created for a single-linked list with three elements. The list's $head$ references the first node in the list; the $tail$ references the last node. The nodes are connected with each other by the $next$ reference. Traversing through the list is achieved by starting at the node pointed at by $head$ and following

Figure 7: A linked list with three elements. Arrows depict references. Dashed arrows mean $null$



the $next$ references until the node equals $tail$ (or $next$ is $null$). To add an element, a new node is created and its reference is assigned to the $next$ reference of the current $tail$ and to the $tail$ variable itself.

7.3 Tree-based data structures

Trees are data structures which organise elements in a hierarchy. A tree consists of nodes that may have links to child nodes and to a single parent node. A node without children is called a *leaf*. The topmost node, which has no parent, is called the *root*. The children of a tree can be ordered or unordered. The number of links to follow from the root to a certain node determines the *depth* of that node. The depth of the whole tree is defined as the maximum depth of all leaves. Trees in which the depth of all leaves differs by at most one are called *balanced trees*.

Various different tree structures exist with different benefits in different scenarios. A common tree structure is the binary search tree (BST) [26]. Binary trees in general are trees with at most two child nodes (left and right). Binary search trees are binary trees in which all nodes of the left subtree of any node have a value less[14] than the value of the node and the nodes in the right subtree have a value greater than or equal to the value of the node. Red-Black trees and AVL-trees are variants of binary search trees that keep the tree balanced to reduce the complexity of searching values in the tree.

Asymptotic time complexity of basic operations on binary search trees

We only consider binary search trees (balanced and unbalanced) in this analysis. In general, the complexity of many operations on binary search trees directly depends

[14]The meaning of a value being “less than” or “greater than” another value depends on the use case.

on the depth of the tree. This explains why the operations on balanced trees typically have a lower worst case complexity than the same operations on unbalanced trees.

Exact query The number of comparisons required to find a node with a specific key has an upper bound determined by the depth of the tree. Because an unbalanced BST with *n* nodes may have a depth of *n* (in case all nodes have only one child), the worst case complexity of finding a node in such an unbalanced BST is linear with respect to the number of nodes. In fact, such a tree is basically a linked list.
Contrarily, in a balanced tree, the complexity is never worse than $O(log\, n)$: at each node, only either the left or the right link is followed during traversal which causes half of all remaining nodes to be removed from the set of possible candidate nodes at each level of the tree.

Range query Whether range queries can be executed efficiently in a BST depends on the kind of those queries. If the range query asks for nodes with a key less than a given value or greater than a given value, finding those nodes can be done efficiently, because this is exactly the criterion that defines whether nodes are inserted as the left or right child of their parent. For example, to find all nodes with a key less than *k*, one needs to first find the smallest node with a key greater than *k* and then traverse all child nodes to the left of that node. Finding such a node can be done with logarithmic time complexity.
Other range queries require the full traversal of the tree to find all nodes that match the query. This has linear complexity. As a consequence, finding a sort order of the node keys that supports the desired range query is necessary to allow efficient range queries.

Insertion Adding a node into an unbalanced BST has a worst case complexity of $O(n)$. For example, if nodes are inserted in increasing order, each node will only have one child and each insertion requires traversal over all previously inserted nodes.
Contrarily, inserting nodes into a balanced tree can be achieved with logarithmic ($O(log\, n)$) worst case complexity, even with the additional cost of keeping the tree in its balanced state.

Removal Removing a value from the tree first requires an exact query to be executed to find the respective node. Removing the actual node can take linear complexity for unbalanced BSTs. For balanced trees, this can be achieved with logarithmic worst-case complexity.

Listing 13: A binary search-tree node for a tree that stores associations between aspect instances and key tuples.

```
class BstNode<Aspect> {
  public Tuple keyTuple;
  public Aspect aspectInstance;
  public BstNode<Aspect> left;
  public BstNode<Aspect> right;
}
```

Space complexity

A binary search tree is comparable to a linked list, only that its nodes do not refer to a “next” element, but instead provide two links, one to the “left” subtree and one to the “right” subtree. The size of a node is therefore:

$$sizeof(BstNode_M) = sizeof(Object) + 3 * sizeof(Ref) + sizeof(Ref[M])$$

As with linked lists, no overallocation of nodes takes place when using a binary search tree. As a bare minimum, a data storage using a binary tree as a data structure must maintain a reference to the root node of the tree. The total size of the storage can therefore be estimated with:

$$\begin{aligned} sizeof(BstStorage_{N,M}) &= sizeof(Ref) + N * sizeof(BstNode_M) \\ &= (3N+1)sizeof(Ref) + N * sizeof(Object) + N * sizeof(Ref[M])) \end{aligned}$$

Note that the fact whether the tree is balanced or not does not affect the memory footprint, as the number of nodes stays the same.

Example

Listing 13 shows a possible node of a binary search tree that stores associations for the `Equality` aspect. The *key* of the node is determined by `keyTuple`.

To determine the relative order of two keys in the BST, `Tuples` need to be comparable. `Tuple` may implement the `Comparable` interface as shown in Listing 14. To make this implementation work, the `compare` method needs to be able to determine the relative

order between two arbitrary objects. This order can be artificial, for example by assigning a unique identifier to each object and comparing those identifier .

Listing 14: A sortable `Tuple` implementation for use in a BST

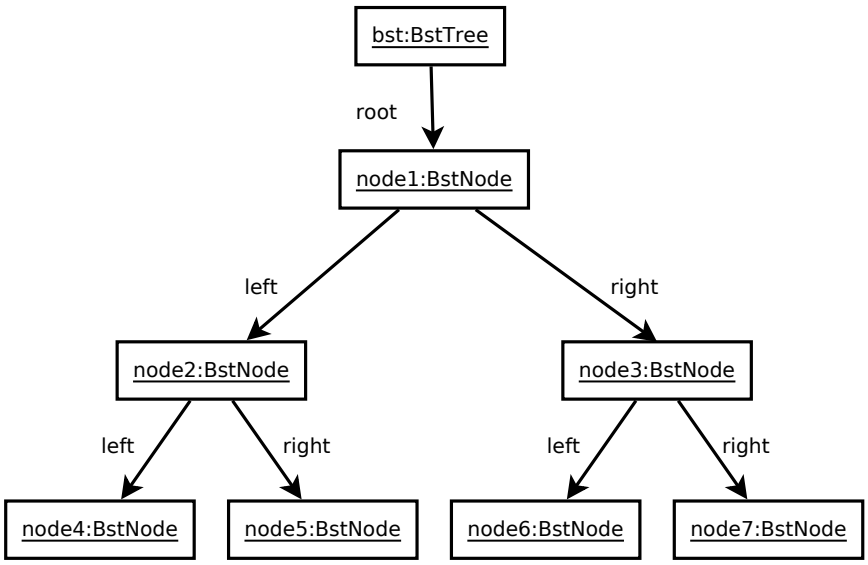
```
class Tuple implements Comparable<Tuple> {
  public Object[] elements;

  public int compareTo(Tuple other) {
    assert elements.length == other.elements.length;

    for(int i = 0; i < elements.length; ++i){
      // compare i-th element
      int comp = compare(this.elements[i], other.elements[i]);
      if(comp != 0){
        return comp;
      }
    }

    // both tuples are equal
    return 0;
  }

  // Returns
  // - a negative value if a < b
  // - a positive value if a > b
  // - 0 if a = b
  private static int compare(Object a, Object b){
    // determine relative order between elements a and b
  }
}
```

7.4 Tries

A special tree structure that uses string-like keys are so-called tries (prefix trees) [27, 26]. In a trie, the *key* of a node is interpreted as a string of m elements. The root node (first level) contains a child for each distinct value of the first element in the key string. Each node on the second level contains a child node for each value of the second key element. On the third level, each node contains a child for each distinct value of the third element in the key string, and so on. Generally, the nth level of the tree discriminates the keys based on the nth element in the key string. The depth of a leaf node in a trie therefore corresponds to the length of the key that leads to the leaf. Only the leaf nodes contain actual values associated with a key.

Each node must maintain references to child nodes, at most one for each distinct value of the element in the key at the position corresponding to the level of the node.

Figure 8: Objects of a BST. Arrows depict references.



Listing 15: A node used in a trie that stores strings of Booleans.

```
class BinaryTrieNode<Aspect>{
  private BinaryTrieNode trueChild;
  private BinaryTrieNode falseChild;
  private Aspect data;
}
```

If the number of those distinct values is small, each node may contain one explicit reference for each possible element value. For example, in a trie that stores strings of Boolean values, each node can have at most two children. One for the child that represents strings where the next element is true and one for the child for strings where the next element is false. In this case, both references can be provided as distinct fields (Listing 15).

However, if the number of children is unknown beforehand or if the number of possible distinct values for the next level is considerably larger than the actual number of children, a trie node can use a secondary storage to store the references to those children, for example using a search tree or a hash map (see Listing 16).

Asymptotic time complexity of basic operations on tries

Generally, the exact complexity of the operations on tries depends on the secondary data structure used in each node to store the children.

Listing 16: A node for a trie that uses key-tuples with elements of type *Object.*

```
class TrieNode<Aspect>{
  private Map<Object, TrieNode<Aspect>> children;
  private Aspect data;
}
```

Exact query Finding the node with a key of length k requires k searches in the secondary data structures, one for each element in the key string. For example, if a hash table is used as the secondary data structure, the complexity for an exact query is constant (see Section 7.5).

Range query Tries are also called prefix trees because finding all nodes with a key that starts with a specific sequence is efficient. For a prefix of length p, p exact queries in the secondary data structures need to be executed to find the node that represents the key with the requested prefix. All leaf nodes that can be reached from this node are then part of the requested range. Again, if hash tables are used as secondary storage, this can be done in constant time (with respect to the number of elements in the trie).

Insertion Finding the node at which a new value needs to be added is similar to an exact query. For a node with a key of length k, k exact queries in the secondary storage are required.

Removal Similarly, removing a node requires k searches in the secondary data structures to find the node to remove and an additional removal in the secondary data of the last node to remove the node.

Space complexity

The space complexity of tries largely depends on the choice of the secondary storage that is used at each intermediate node on the paths from the root to the leafs. In the worst case, no keys of the inserted values share a common prefix. In this case the number of nodes grows linearly with the number of inserted values times the length of the key string. The size of each secondary storage naturally depends on the choice of the data structure for the secondary storage and is, as such, an implementation detail.

The size of a single node that contains P children can be estimated with:

$$sizeof(TrieNode) = sizeof(object) + sizeof(Ref) + sizeof(SecondaryStorage_P)$$

Because we assume that the length of a key tuple is fixed, we could use two different kind of nodes: one intermediate node that does not store a reference to any data, and one leaf node that does contain that reference but lacks a secondary storage, because it cannot have children. For simplicity, we use the same node everywhere.

The number of nodes required for storing a trie depends on how much the key tuples of the stored associations share prefixes. Because this is not easy to predict, we consider the worst and best case.

In the worst case, no key tuple shares a prefix. In this case, the number of nodes is the number of associations times the length of the key tuples, that is $M * N$. In the best case, the key tuples of all stored associations are equal except for the last element in the key tuple, that is, the length of the shared prefix is $M - 1$. In this case, the number of nodes is $N + (M - 1)$. We can therefore estimate an upper and lower boundary for the size of a trie:

$$(N+M-1)*sizeof(TrieNode) \leq sizeof(TrieStorage_{M,N}) \leq MN*sizeof(TrieNode)$$

Example

Listing 17 shows a node of a trie. The secondary storage can for example be a hash table. To find an aspect instance given a tuple, the protected `find` method receives the current level in the trie (this removes the need to store it in the node) and the whole tuple. If the current level equals the length of the tuple, the data of the current node is the sought aspect instance. Otherwise the method continues the search with the child node that is associated to the element of the tuple corresponding to the current level.

Figure 9 on page 64 shows the nodes of a trie that associates text strings with objects of type `Data`. In the figure, the trie contains the keys *bar*, *baz* and *bob*. The paths in the trie leading to the leaf nodes for *bar* and *baz* share the common prefix *ba* (represented by *node2*). All leaf nodes have a common root node *b* (*node1*).

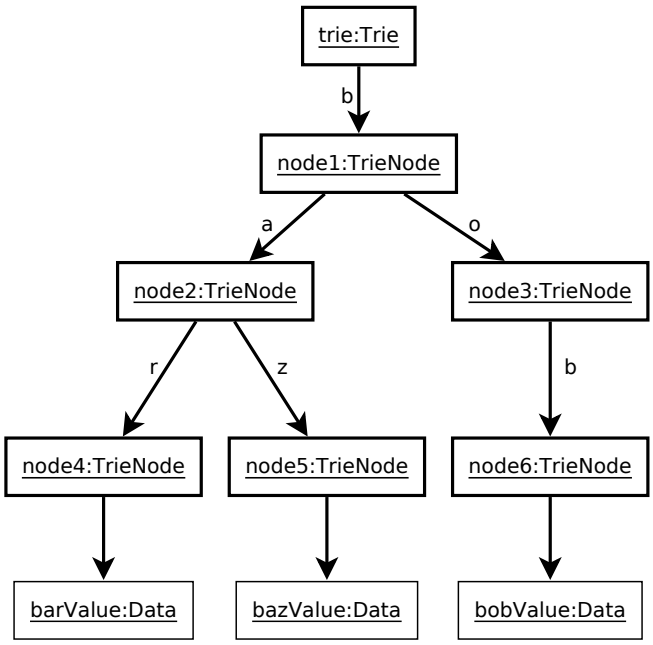
7.5 Hash-based data structures

Hash-based data structures (hash tables [26], [25]) store their entries in a number of so-called buckets. For each entry, the index of the bucket in which it needs to be

Listing 17: A node for a trie that associates key tuples with aspects of type *Aspect* .

```
class AspectTrieNode<Aspect> {
  private Map<Object, AspectTrieNode<Aspect>> children;
  private Aspect data;

  public Aspect find(Tuple tuple){
    return find(0, tuple);
  }

  protected Aspect find(int level, Tuple tuple){

    // for the given tuple, this is a leaf node:
    if(tuple.elements.length == level){
      return data;
    }

    // move to the next element in the tuple and find the proper child node
    AspectTrieNode<Aspect> nextLevel = children.get(tuple.elements[level]);

    // continue search in child node
    return nextLevel.find(level + 1, tuple);
  }
}
```

Figure 9: Nodes forming a trie. Arrow labels show the element in the string that is used to discriminate children. Objects with thin borders depict data referenced by leaf nodes.



stored can be calculated from the content of the entry (or the key of the entry) using a hash function $h(k)$ that returns the bucket index for a key k.

If h hashes multiple keys to the same index, a collision occurs. Different strategies exist to handle these collisions. One common strategy (*Separate Chaining*) is to use linked lists as buckets. All elements hashed to the same bucket index are added to this linked list. This adds memory overhead and can potentially increase the time complexity when adding, removing or finding an element. However, a good hash function reduces the chance of collisions, rendering this overhead negligible. A second collision resolution strategy, called *Open Addressing,* is to find another, empty bucket for an element if the bucket to which the key hashes is occupied. This can be accomplished for example by searching for empty buckets starting from the original bucket in a specific way (such as linearly or with step sizes increasing quadratically). Alternatively, additional, different hash functions can be used to find alternative locations.

A notable algorithm to handle collisions, called Bidirectional Linear Probing, has been introduced by Amble and Knuth [28]. The basic idea of this approach is to keep all keys hashing to the same bucket in sorted order. When a new item is inserted which hashes to a location that is already occupied, the direction at which a new location is sought for depends on the value of the key to insert and that of the key at the suggested bucket. If the bucket's key is higher than the key to insert, the algorithms performs a downward search for a free bucket, if it is larger than the key to insert an upward search is performed. This approach can of course require parts of the table to be moved to insert a key between two other keys. Effectively this approach clusters keys around the suggested bucket index. This approach can considerably reduce the number probings required to find a key. First, because ideally the key at the suggested bucket index partitions all colliding keys in two equal halves in case of which at most half the number of colliding keys has to be probed. Second, due to the keys being stored in sorted order probing can stop once a key too high (when searching upwards) or too low (when searching downwards) has been found.

The chance of collision can be reduced by using a hash function that distributes keys evenly over the available bucket indices. To further reduce the chance of collision, hash tables often define a specific fill factor, that is, the maximum percentage of elements in the hash table with respect to the number of buckets. Once the fill factor is exceeded, the bucket array size needs to be increased and all elements need to be hashed to the new bucket array, which is a comparatively costly operation.

Asymptotic time complexity of basic operations on hash tables

Exact query Because the position of an entry in the hash table is calculated from the content of the key, finding a specific entry does not depend on the number of elements in the hash table, assuming that collisions are comparatively seldom. At worst, the complexity is linear with respect to the maximum number of steps necessary to resolve collisions. For example, when separate chaining is used, the complexity is linear with respect to the longest linked list in the hash table. In the case of open addressing, the complexity is linear with respect to the maximum number of times a new address needs to be probed until the key is found (or no key is found at all).
Because the hash function is expected to distribute hashes evenly over the buckets, collision resolution is neglected and hash tables are considered to have constant time complexity for exact queries.

Range query Hash tables do not provide special support for range queries. Typically, hash functions do not return similar indices for similar keys to provide even distribution of bucket indices. As a consequence, range queries always require the examination of all entries in the hash table, which has linear complexity.

Insertion Inserting an element into a hash table can be achieved with amortised constant complexity, that is, the complexity does not depend on the number of elements in the hash table. This, again, only holds under the assumption that entry keys do not cause too many collisions.

Removal Like insertion, removal can be achieved with constant complexity.

Space complexity

The memory required by hash based data structures strongly depends on the internal implementation, especially on the collision resolution strategy. We consider a simple implementation here (see Listing 18) using an array to store `Association` references and using linear probing for collision resolution. In this implementation, the internal array size is increased whenever the number of associations reaches a specific *fill factor*. This maximum fill factor is typically around 0.75, that is, 75%. If the hashtable is filled up to the maximum fill factor, we estimate the size with:

$$sizeof(Hashtable_{M,N}) = sizeof(Ref[N/FillFactor]) + N * sizeof(Association_M)$$

That is, the internal array has a capacity of $N/FillFactor$ (where $FillFactor < 1$) and holds N instances of `Association`. The maximum fill factor typically used by hash based container causes some memory overhead because parts of the allocated buckets will never be used. Furthermore, using *Separate Chaining* as a collision resolution strategy makes use of secondary data structures such as linked lists which have additional memory overhead.

Example

Listing 18 shows a partial implementation of a hash table that associates tuples with aspect instances. The `hash` method calculates an integer from the value of the key (the `BitTuple`). The `indexOf` method ensures that the integer returned from `hash` stays within the range of valid integer indices of the `buckets` array. To insert an entry, `insert` maps the content of the entry's key to an index in the bucket array. If the bucket array is already occupied with a different entry, a collision has occurred that needs to be resolved. In this case, the implementation uses open addressing to find another free slot by searching linearly for the next free bucket. Finding works similarly. First, the expected index is calculated for the tuple. If the bucket at that index is empty, the entry was not found. If it is not empty and has a matching key tuple, the entry was found and is returned. If the key tuple of the entry at the expected index does not match the given tuple, this has been caused by a collision in the past. As in `insert`, the buckets array is searched linearly starting from the expected bucket for the entry with the expected key. If an empty bucket is found or all buckets have been examined, the entry has not been found.

7.6 Summary

In this section we provide an overview of typical data structures that are used for problems related to searching. Specifically, we can now answer research questions #3 and #4.

Question #3: What is the asymptotic computational complexity of the required operations when implemented for different data structures?

In this section, we have described several common data structures and evaluated the asymptotic computational complexity of operations such as querying, insertion and removeal. An overview of the asymptotic computational complexity is given in Table 4 on page 69.

Listing 18: Parts of a hash table to store *Association*s

```
class AspectHashtable<Aspect> {
  private Association<Aspect>[] buckets;

  public void insert(Association<Aspect> entry){
    int index = indexOf(entry.keyTuple);
    if(buckets[index] != null &&
          !buckets[index].keyTuple.equals(entry.keyTuple)){
      // collision, use open addressing with
      // linear probing to find free bucket
      do {
        index = (index + 1) % buckets.length;
        if(buckets[index] == null){
          break;
        }
      while(true);
    }
    buckets[index] = entry;
  }

  public Association<Aspect> find(Tuple tuple){
    int index = indexOf(tuple);
    if(bucket[index] == null){
      // not present
      return null;
    }

    if(bucket[index].keyTuple.equals(tuple)){
      return bucket[index];
    }

    // key of stored entry does not match
    // expected key, due to a collision
    // Use same open addressing method
    // as in insert() to find the bucket with the
    // expected key
    int originalIndex = index;
    do {
      index = (index+1) % buckets.length;

      if(buckets[index] == null){
        return null;
      }

      if(buckets[index].keyTuple.equals(tuple)){
        return buckets[index];
      }
    } while(index!=originalIndex);
  }

  private int hash(Tuple tuple) {
    // assuming Tuple implements hashCode correctly
    return tuple.hashCode();
  }

  private int indexOf(BitTuple tuple) {
    return hash(tuple) % buckets.length;
  }
}
```

	Exact	Partial-range	Insertion	Removal
Array	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Sorted Array	$O(log\, n)$	$O(n)$ $O(log\, n)^{\dagger 2}$	$O(n)$	O(n)
Linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$
BST	$O(log\, n)$ (avg) $O(n)$ (worst)	$O(log\, n)$ (avg) $O(n)$ (worst)	$O(log\, n)$ (avg) $O(n)$ (worst)	$O(log\, n)$ (avg) $O(n)$ (worst)
Trie$^{\dagger 1}$	$O(k)$ (avg) $O(n)$ (worst)	$O(k)$ (avg) $O(n)$ (worst)	$O(k)$ (avg) $O(n)$ (worst)	$O(k)$ (avg) $O(n)$ (worst)
Hashtable	$O(1)$ (avg) $O(n)$ (worst)	$O(1)$ (avg) $O(n)$ (worst)	$O(1)$ (avg) $O(n)$ (worst)	$O(1)$ (avg) $O(n)$ (worst)

Table 4: Asymptotic computational complexity of different operations on data structures with n elements and key-tuples of length k. Full range queries are omitted, because those queries simply mean "full traversal" and is $O(n)$ in all cases.
$^{\dagger 1}$ using hash tables as secondary storage.
$^{\dagger 2}$ if prefix query

Question #4: What is the data complexity of the respective data structures in those scenarios?

The actual memory consumption of the different data structures in a running system is difficult to estimate beforehand, as it depends on the implementation of those data structures as well as other factors such as the processor architecture (32 bit versus 64 bit) and in same cases the values of the key tuples at runtime. We can therefore only estimate the relative memory consumption of different data structures. We can qualitatively assess the data complexity as follows: **arrays** have a minimum memory overhead, especially when they are completely occupied. When two arrays are used for key tuples and aspect instances, the memory overhead per association is as low as two words plus the key tuple array (plus over allocation). **Linked lists** have a similar memory footprint as arrays using the `Association` instance approach. The overhead per association is about four words plus the key-tuple array. On the other hand, linked lists do not suffer from over-allocation. **Binary search trees** add another word per association, but also do not suffer from over-allocation. **Hashtables** can be implemented as arrays using the same techniques as pure array storages. That way, the overhead per association can be rather small. However, the fill factor determines how many array slots are unused. Typically at least 25% of the array slots need to be kept open to reduce the chance of collision. **Tries** are more complex and the memory footprint largely depends on the secondary storage used inside the node. In the worst case, $N * M$ secondary storages need to be created for N associations

with key tuples of length M. We therefore expect the total memory consumption of tries to be several times that of an array based approach in practice.

8 Benchmark

We implemented a benchmark application in Java to measure the execution speed of search algorithms for different data structures. The results of the benchmark are used to justify our recommendation for choosing certain data structures in specific scenarios.

8.1 Statistically rigorous performance evaluation

Our benchmark is an artificial measurement of the execution speed of algorithms that are used on different data structures. It is artificial in that it does not measure a real-world application. Instead, we create the desired scenarios on purpose. Also, a real implementation of aspect-instance look-up would probably be part of the aspect-oriented execution-environment. In the case of environments that run inside a virtual machine (such as Java), the aspect-oriented execution environment may even be part of the virtual machine (Bockisch et al. [29], Haupt [30]). Therefore, our benchmark does not necessarily represent what would be executed in a real-world application (see Section 8.7 for a discussion on this and other threats to validity).

However, we want the results of the benchmark to be transferable to real implementations. While the actual times may be different in practice, the results should qualitatively justify the decision to use or not to use a specific data structure for a specific scenario. To give dependable recommendations we used a benchmarking methodology that is based on the *statistically rigorous* approach suggested by Georges et al [31]. In their paper, Georges et al. introduce a methodology that reduces the influence of non-deterministic factors during the execution of benchmarks, such as JIT-compilation, thread scheduling or garbage collection.

We are interested in the execution time of algorithms in the application's *steady state*, that is, after the application has completed the start-up phase. In the steady state, most methods have been optimised and compiled by the JIT compiler. Measuring the steady-state performance is most useful for applications that are expected to run for a considerably longer time than the start-up phase. There is less variability during the steady state of an application, which makes benchmarking easier and more reliable [31].

For steady-state performance measurements Georges et al. suggest a four step methodology that uses multiple Virtual Machine (VM) invocations with multiple benchmark iterations in each VM invocation. Inside a single VM instance, the benchmark is repeated until the steady-state has been reached. Georges et al. suggest

that steady-state can be considered reached if the last k iterations of the benchmark have a low *coefficient of variation*, that is, when the standard deviation σ of the last k samples is relatively low compared to the mean $\bar{x}$ (that is, for example, $\sigma/\bar{x} < 0.02$). After p independent invocations of the VM – each returning the mean of k samples during steady-state – the total mean of the p results and the confidence intervals of the desired levels can be calculated. This makes the results more dependable. Whereas the measurements within a single VM invocation are not independent, the measurements across multiple VM invocations are independent.

8.2 Implementation of the statistically rigorous methodology

The benchmark follows the methodology for measuring steady-state performance suggested by Georges et al. A **driver** program generates **cases**. A case is the combination of a specific **scenario** (for example: execution of an exact query) with specific **parameters** (for example, n existing aspect instances and key tuples of length m) and a specific **storage** implementation (for example, an array based storage or a hash-table based storage). Each case is executed 30 times (parameter p in the four-step methodology suggested by Georges et al.). For each of the p executions per case, the driver starts a **runner** application in a new Java VM. This way, each case runs in its own, new Java VM. Case executions are therefore isolated from each other, avoiding any possible mutual influence.

When a case is executed in its own virtual Java VM, the runner application creates and executes **test batches**, each of which consists of many actual **invocations** of the code to measure. A single invocation (for example, a look-up operation) can – depending on the test machine, the scenario and the storage – take as little as only a few nano-seconds. However, accurate measurement of times in the order of nanoseconds is typically not possible on most systems. For example, on the primary test machine, the resolution of `System.nanoTime()` (the function used to measure time) is about 300 nanoseconds. Instead, a test batch performs the invocations multiple times and measures the whole timespan for all invocations. From the accumulated duration of all invocations and the number of invocations per test batch we can calculate an average duration per invocation. Each single look-up is done with a different query key. This better reflects the usage in practice and reduces the influence of CPU caching on the outcome.

The number of invocations per batch is variable. The runner starts with batches of 100 invocations. It runs the batch a number of times (warm up) and then measures the average batch time across a number of batches. If the average time is lower than a certain threshold (1 millisecond, or 1,000,000 nanoseconds), the batch size is

increased (multiplied by 10, up to 10,000,000 invocations per batch) and measurement is restarted. If a batch runs for longer than the threshold, the error caused by the limited resolution of the timing method is considered low enough to be ignored. For example, with a resolution of 300 nanoseconds the error will not be higher than $300ns/1,000,000ns = 0.03\%$.

Once a sufficiently large batch size is determined, the actual measurement is performed. Batches are executed and measured until a steady-state has been reached. Georges et al. suggest using he coefficient of variation (CoV, the standard deviation divided by the mean) to determine when steady state has been reached. However, during our tests, the suggested CoV of 1% or 2% were often never achieved. Therefore, we changed the criteria slightly. For the calculation we take the last 300 measurements and discard the highest and lowest 5% of those measurement to ignore outliers. We consider a steady state to be reached if the CoV falls below 7% and the 95% confidence interval for the execution time of the remaining 270 samples is smaller than $\pm 1\%$ of the mean execution-time.

The result of a single case is the calculated mean execution-time of a single invocation. Each case is executed 30 times by the driver (each time using a separate Java VM), leading to 30 different, independent measurements of mean execution-times per case. From this data set we calculate the overall mean and the 95% confidence intervals per case.

8.3 More implementation details

Our benchmark compares the execution speed of different algorithms in different scenarios. To make results for a single scenario comparable, the only code that changes is the algorithm under test. The rest of the code that is being executed stays the same. To achieve this, we separated the code that defines a specific scenario from the code that implements a specific data structure and its related algorithms. We call the abstraction of a data structure and its algorithm a **storage**. To define the data structure for a specific scenario, the object implementing that scenario receives an instance of the storage under test. The scenario accesses the storage through an interface without knowing about any implementation details of the storage itself. This also reflects our unified model, in which the data storage is an implementation detail that can and should be separated from the semantics of the instantiation policy.

When a batch is created, the scenario notifies the storage about the queries that it can expect. This allows the storage to initialize and prepare itself for a specific type of query. In practice, the aspect-oriented execution-environment also knows which queries to expect by parsing the pointcut expressions of an aspect.

Figure 10 on page 75 shows a part of the benchmark's design. Scenarios are abstracted by the interface `Scenario`. Classes that implement `Scenario` act as factories for actual instances of the scenario (`ScenarioInstance`). A storage implementation is accessed by specific `ScenarioInstances` through its abstraction `Storage`.

Key tuples and query key tuples are represented by the `KeyTuple` class. A `KeyTuple` contains an array of length m, holding the elements of the key tuple. In our benchmark application, the key elements are always of type `KeyItem`. We use this type to simulate some of the optimisation techniques found in existing instantiation policies (see Section 8.4.6 and Section 8.4.7). `KeyTuple` overrides `equals` and `hashCode`. We consider two `KeyTuples` to be equal if their key items are identical (that is, the references point to the same objects). The `KeyTuples` themselves do not have to be the same. The hash code of a `KeyTuple` is a hash of the hash codes of the key items[15]. The hash code of a `KeyItem` is the default hash code as assigned by the Java runtime.

8.4 Selected data structures

We implemented a number of different storages based on data structures discussed in Section 7. These data structures are considered general purpose storages that are applicable to a wider range of scenarios. We specifically do not rely on advanced techniques such as changing the layout of classes. In addition, we implemented storage variations that imitate some of the optimisation techniques used by existing instantiation policies. This way we can compare the performance of those specialisations to the performance of general purpose storages.

8.4.1 ArrayStorage

The `ArrayStorage` implements storage using a simple array. Each element in the array is an `Association`, which is a simple class holding a key tuple and the associated aspect instance (see Listing 19). The size of the array is fixed in our benchmark and is equal to the expected number of aspect instances n for a specific scenario. In a real-world application, the required size of the array is not known beforehand. In practice a "resizeable"[16] array (such as the `ArrayList`) should therefore be used. When an element is stored in the `ArrayStorage`, it is placed at the first free position in the array. Because `ArrayStorage` (like all storage implementations) does not allow

[15] We use a general purpose hash function to combine the hash codes of all key items to a single hash code. See `http://www.partow.net/programming/hashfunctions/`

[16] Arrays in Java cannot be resized. Instead, a new array with the desired new size needs to be created and existing elements need to be copied over from the old to the new array.

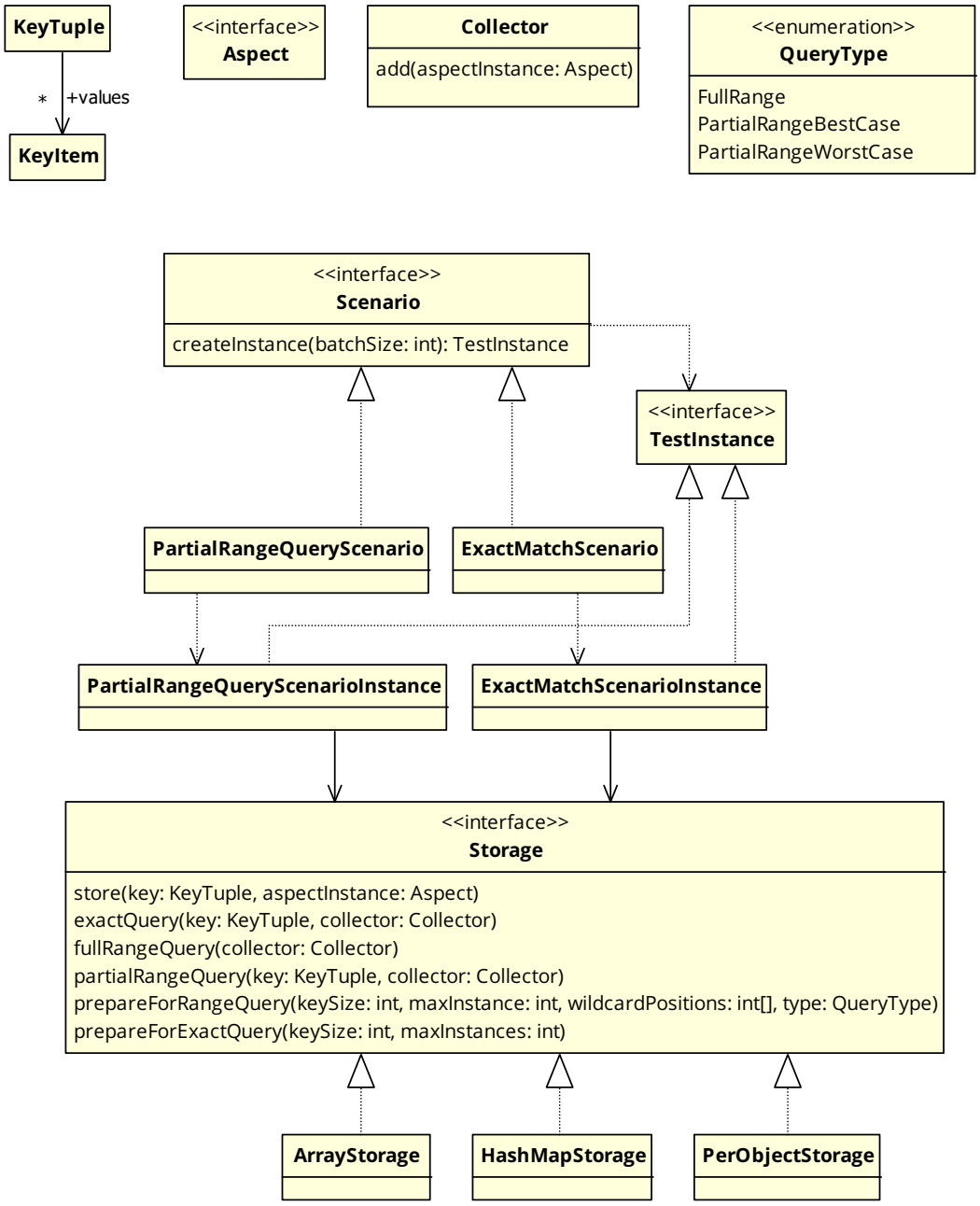


Figure 10: UML diagram showing a part of the design of the benchmark. Scenarios and storages can be combined freely, allowing new scenarios or storages to be added easily.

Listing 19: Association class that relates a key tuple to an aspect instance.

```
1 public class Association {
2     public final KeyTuple key;
3     public final Aspect aspect;
4 }
```

the removal of elements, the index of the next free position equals the number of elements currently stored, which makes bookkeeping easier.

To execute an exact query, the storage iterates over all elements in the array and compares the `key` field with the query-key tuple. If the `key` and and the query-key tuple are equal, the aspect instance has been found. In this case, the content of the `aspect` field of the respective array element is returned and the search stops immediately. If all elements in the array (up to the number of stored elements) have been examined, but no element with a matching `key` field has been found, no aspect instance matches the query.

To execute a full-range query, the storage iterates over all elements in the array and adds the value of the `aspect` field to the result list. No keys need to be compared in this case. To execute a partial range query, the storage iterates over all elements in the array and compares the `key` field to the query-key tuple. If all non-wildcard elements of the query-key tuple match the respective elements in the `key` field, the value of the respective `aspect` field is added to the result list.

8.4.2 HashMapStorage

The `HashMapStorage` stores associations of key tuples and aspect instances in a `java.util.HashMap`. When a new association is stored, the storage adds an entry to the internal hash map, using the key tuple as the key and the aspect instance as the value.

To execute an exact query, the storage calls `get` on the internal hash map using the provided query-key tuple as the key. The `get` method either returns the sought element or `null`, which indicates that no element is stored for the query-key tuple.

To execute a full-range query, all elements in the value collection returned by the internal hash map's `values()` method are added to the collector. To execute a partial-range query, the storage iterates over all elements in the hash map using the `entrySet()` method. The key element of each element is compared to the query-key tuple. If all non-wildcard elements of the query-key tuple match the respective

elements in the key, the respective value (the aspect instance) is added to the collector.

Iterating over the values instantiates a new instance of an `Iterator<Aspect>`. That is, a full-range query required memory allocation. Although memory allocation is typically comparatively fast in Java, the instantiated objects need to be garbage collector at some point which may adversely affect the execution speed of the benchmark.

8.4.3 CustomHashMapStorage

Java runtimes such as the Oracle implementation typically offer different garbage collection algorithms and settings to tune an application for specific use cases when necessary. Because of common usage patterns in applications, memory allocation and garbage collection is typically fast for short-lived objects (especially when a generational garbage collector is used), that is, objects that are only used for a short duration after which they can be garbage collected [32]. Even though we expect the cost of allocating and collecting short-lived objects to be small, we added a custom implementation of a hash map that does not require any allocation to iterate over its elements. This allows us to quantify the possible impact of the `Iterator` allocation.

Our custom implementation of the hash map is for the most part identical to the stock `HashMap` as provided by the Java Runtime Library[17]. That is, it uses the same internal hashing function, the same maximum fill factor of 0.75 and separate chaining for hash-collision resolution [25, p. 542]. The main difference lies in the fact that our implementation exposes the internal array of buckets which allows users of the class to manually iterate over all stored entries without the need to allocate an `Iterator`.

Exact queries work the same way as in `HashMapStorage`. Full range queries use the `getAll` method of the custom hash-map implementation (`SimpleHashMap`), as shown in Listing 20.

Partial range queries work similarly, using a similar collection function (see Listing 21). The partial-range collection function differs only in the fact that aspect instances are only collected if the associated key matches the query-key tuple. This check is implemented by `TupleEqualityComparer.partialMatch()`.

[17] We use the official Java runtime provided by the Oracle Corporation. Available at `http://java.oracle.com`

Listing 20: Collecting all elements in the custom `SimpleHashMap` without allocating an `Iterator`

```
public void getAll(Collection<TValue> collector) {
  for (int i = 0; i < buckets.length; ++i) {
    Bucket<TKey, TValue> bucket = buckets[i];
    while (bucket != null) {
      collector.add(bucket.value);
      bucket = bucket.next;
    }
  }
}
```

Listing 21: Executing a partial-range query on a `SimpleHashMap`.

```
public void collectPartialMatches(KeyTuple queryKeyTuple, Collector collector, int[]
     rangeQueryMatch) {
  SimpleHashMap.Bucket<KeyTuple, Aspect>[] buckets = buckets();
  for(int i= 0; i < buckets.length;++i){
    SimpleHashMap.Bucket<KeyTuple, Aspect> bucket = buckets[i];
    while(bucket!=null){
      KeyTuple key = bucket.key;
      if (TupleEqualityComparer.partialMatch(rangeQueryMatch, key, queryKeyTuple)) {
        collector.add(bucket.value);
      }

      bucket = bucket.next;
    }

  }
}
```

Listing 22: `findNode` finds the node for a specific key prefix.

```
1  private Node findNode(KeyItem[] key, int levels, boolean createMissingNodes) {
2    assert levels <= depth - 1;
3    Node currentNode = root;
4    for (int level = 0; level < levels; ++level) {
5      KeyItem levelKey = key[level];
6      Node child = (Node) currentNode.get(key[level]);
7      if (child == null) {
8        if (createMissingNodes) {
9          child = new Node();
10         currentNode.put(levelKey, child);
11       } else {
12         return null;
13       }
14     }
15     currentNode = child;
16   }
17   return currentNode;
18 }
```

8.4.4 TrieStorage

The trie storage makes use of a custom implementation of a trie. The trie in our benchmark has a fixed depth, which equals the length m of the key-tuple. Our trie implementation cannot be used for singleton storage, because in those scenarios the key length is zero. At level k of the tree (where $1 \leq k \leq m - 1$), each node is a hash map that maps the kth key-tuple element to nodes of the next level $k + 1$. At the last level m, the nodes are hash maps that map from the last key-tuple element to aspect instances.

Listing 22 shows the core method of the fixed depth trie, `findNode`. This method returns the `Node` (essentially the same implementation of the hash map as described in Section 8.4.3) of the trie that represents the first `levels` elements of the provided `key`. If no such node exists and `createMissingNodes` is `false`, `null` is returned. Otherwise, the missing nodes are added to the trie as required. The method can be used for different purposes.

To insert a node, `findNode` is called with `levels` being one less than the key length and `createMissingNodes` set to `true`. The resulting node is the `Node` (hash map) that will store the aspect instance associated with the last key element. Listing 23 shows the implementation of this insertion method. The method also detects if an aspect has been stored for the same key tuple. In that case, the old aspect instance is returned.

Listing 23: Method to add an aspect to the trie.

```
public Aspect put(KeyTuple keyTuple, Aspect value) {
  assert keyTuple != null && keyTuple.size() == depth;
  KeyItem[] key = keyTuple.values;
  Node node = findNode(key, depth - 1, true);
  KeyItem element = key[lastElementIndex];
  Aspect result = (Aspect) node.put(element, new Association(keyTuple, value));
  if (result == null) {
    ++size;
  }
  return result;
}
```

Listing 24: Finding a value by a key in a trie.

```
public Aspect get(KeyTuple keyTuple) {
  assert keyTuple != null && keyTuple.size() == depth;
  KeyItem[] key = keyTuple.values;
  Node node = findNode(key, depth - 1, false);
  if (node == null) {
    return null;
  }
  KeyItem element = key[lastElementIndex];
  Association item = (Association) node.get(element);

  if (item != null) {
    return item.aspect;
  }
  return null;
}
```

Looking up an instance uses `findNode` to find the leaf node inside which the sought value must have been stored (see Listing 24). If such a node exists and if it contains an entry for the last key-tuple element, the search was successful.

A prefix query on a trie also uses the `findNode` method (seeListing 25). In this case, the method is used to find the node that represents the key-prefix. If such a node exists, all aspects stored in this node and any descending node are the nodes to collect. This collection process starting from a specific node is done by the method `recursiveCollect`.

A full-range query can be executed similarly by calling `recursiveCollect` on the root node. This process requires the traversal over all hash tables (that is, nodes) in the trie. Similarly, if a partial range query needs to be executed that is not a prefix query, all nodes have to be traversed. In this case, however, only aspect instances are collected for which all non-wildcard elements of the query-key tuple match the

Listing 25: Implementation of a prefix query in the fixed depth prefix tree implementation

```
1 public void collectPrefixMatches(KeyTuple key, int prefixLength, Collector collector) {
2   Node node = findNode(key.values, prefixLength, false);
3   if (node != null) {
4     recursiveCollect(collector, node, prefixLength);
5   }
6 }
```

Listing 26: Exact query on a `SingletonStorage`

```
1 public void exactQuery(KeyTuple queryKeyTuple, Collector collector) {
2   Aspect instance = GenericAspect.aspectOf();
3   if (instance != null) {
4     collector.add(instance);
5   }
6 }
```

respective key-tuple elements in the aspect instance's key.

8.4.5 SingletonStorage

The `SingletonStorage` represents the way AspectJ implements **singleton** aspects [15]. The aspect class (`GenericAspect` in our class) has a static field that can hold a reference to the singleton instance of the aspect. This instance is readable with the static `aspectOf()` method. To set the value of the field, `setAspectOf()` is used. The code for the exact query is shown in Listing 26. Full range queries are executed exactly the same way, as there is at most one instance. Partial range queries are not supported, because the size of the query-key tuple is always 0, that is, it can never contain wildcards.

8.4.6 PerObjectStorage

Similar to the `SingletonStorage`, the `PerObjectStorage` imitates the way AspectJ implements **perobject** instantiation policies (that is, **perthis** and **pertarget**). In our benchmark, elements in the key tuple are always instances of `KeyItem`. In a real-world application, the items in a key tuple can be any context values. When **perobject** instantiation policies are used, AspectJ changes the class layout of affected classes and adds a field that holds a reference to the associated aspect instance (see Section 2.2.2). We simulate this by having a field in the `KeyItem` class called `perObjectAspect`

Listing 27: Parts of `KeyItem` that are relevant for `PerObjectStorage`

```
public class KeyItem {

  private Aspect perObjectAspect;

  public Aspect getAspect() {
    return perObjectAspect;
  }

  public void setAspect(Aspect aspect) {
    this.perObjectAspect = aspect;
  }
}
```

Listing 28: Storing and retrieving aspect instances in `PerObjectStorage`

```
@Override
public void store(KeyTuple keyTuple, Aspect aspect) {
  assert keyTuple.values.length == 1;
  keyTuple.values[0].setAspect(aspect);
  ++size;
}

@Override
public void exactQuery(KeyTuple queryKeyTuple, Collector collector) {
  assert queryKeyTuple.values.length == 1;
  Aspect instance = queryKeyTuple.values[0].getAspect();
  if (instance != null) {
    collector.add(instance);
  }
}
```

that references an aspect instance. Listing 27 shows an excerpt of `KeyItem` with the parts that are relevant for `PerObjectStorage`: the field that holds the aspect instance and a getter/setter per to read and write the value of the field.

Listing 28 shows the parts of `PerObjectStorage` that make use of these elements in `KeyItem`. `PerObjectStorage` uses `setAspect` to store the aspect instance in the key item which it is associated with. That key item is always the first (and only) element in the key tuple. Similarly, to perform an exact query, `PerObjectStorage` reads the aspect instance stored in the first element of a query-key tuple.

`PerObjectStorage` does not support any range queries. It requires a specific context value from which the respective aspect instance can be read. In the case of range queries, no context values are provided[18].

[18] `PerObjectStorage` can only be used when the key length m is 1, so a query-key tuple in a partial-

8.4.7 AssociationAspectStorage

For association aspects that associate a pair of object to an aspect instance, Sakurai et al. suggest an optimisation that also involves the modification of the layout of affected classes [10]. The `perobjects(TypeA, TypeB)` *per*-clause associates pairs of objects consisting of an instance of `TypeA` and an instance of `TypeB` with aspect instances. The modified AspectJ compiler adds a field containing a hash table to `TypeA`. The hash table maps instances of `TypeB` to aspect instances (we call this table the "right partners" table). Similarly, a field holding a hash table is added to `TypeB`, mapping instances of `TypeA` to aspect instances (the "left partners" table). If `TypeA` and `TypeB` are the same type (like they are in our Equality example as shown in Listing 1), both fields are added to the class.

In our benchmark, we used instances of `KeyItem` as elements in the key tuple. Therefore, we added the hash tables to the `KeyItem` class, together with methods to set and retrieve the content of those hash tables (Listing 29). We use our simplified hash-table implementation to avoid the memory allocation when iterating over the content of the hash table.

To create an association between a pair of objects and an aspect instance, we use the method associate as show in Listing 30.

Listing 30: `AssociationAspectStorage`: storing aspect instances

```
1 public static void associate(KeyTuple keyTuple, Aspect aspect){
2   assert keyTuple.size() == 2;
3
4   KeyItem left = keyTuple.values[0];
5   KeyItem right = keyTuple.values[1];
6
7   left.associateRight(right, aspect);
8   right.associateLeft(left, aspect);
9 }
```

To execute an exact query, the storage needs to perform a single look up in the hash table stored with the first element on the query-key tuple. The key used for this look-up is the second element of the query-key tuple (Listing 31).

Listing 31: AssociationAspectStorage: exact query

```
1 @Override
2 public void exactQuery(KeyTuple queryKeyTuple, Collector collector) {
3   KeyItem left = queryKeyTuple.values[0];
```

range query would only contain a single wildcard, but no context values. Such a query would be indistinguishable from a full-range query. In fact, we only considered partial-range queries where the number of wildcards $w < m$.

Listing 29: Parts of `KeyItem` that are relevant to `AssociationAspectStorage`

```
public class KeyItem {

  private SimpleHashMap<KeyItem, Aspect> leftPartners;
  private SimpleHashMap<KeyItem, Aspect> rightPartners;

  public void associateRight(KeyItem partner, Aspect aspect) {
    if (rightPartners == null) {
      rightPartners = new SimpleHashMap<KeyItem, Aspect>();
    }
    rightPartners.put(partner, aspect);
  }

  public void associateLeft(KeyItem partner, Aspect aspect) {
    if (leftPartners == null) {
      leftPartners = new SimpleHashMap<KeyItem, Aspect>();
    }
    leftPartners.put(partner, aspect);
  }

  public Aspect getByRightPartner(KeyItem partner) {
    if (rightPartners == null) {
      return null;
    }
    return rightPartners.get(partner);
  }

  public Aspect getByLeftPartner(KeyItem partner) {
    if (leftPartners == null) {
      return null;
    }
    return leftPartners.get(partner);
  }

  public void collectAllRightPartners(Collector collector) {
    if (rightPartners != null) {
      rightPartners.getAll(collector);
    }
  }

  public void collectAllLeftPartners(Collector collector) {
    if (leftPartners != null) {
      leftPartners.getAll(collector);
    }
  }
}
```

```
  KeyItem right = queryKeyTuple.values[1];

  Aspect a = left.getByRightPartner(right);
  if (a != null) {
    collector.add(a);
  }
}
```

`AssociationAspectStorage` does not support full-range queries. Because aspect instances are stored inside the objects that take part in the association, at least one of these objects is required.

Partial-range queries with one wildcard are supported. To find all the aspect instances for this query, the storage simply needs to collect all entries in the hash table of the non-wildcard key-item. Listing 32 shows the code to perform a partial range query. If the query is a prefix query, that is, the second element of the query-key tuple is a wildcard, the storage uses the first element of the query-key tuple. If the query is a suffix query instead, that is, the first element in the query-key tuple is a wildcard, the storage uses the second element of the query-key tuple. Depending on the `KeyItem` that is chosen, the storage collects the values of the hash table for "right partners" or "left partners" respectively.

Listing 32: `AssociationAspectStorage`: performing partial-range queries

```
@Override
public void partialRangeQuery(KeyTuple queryKeyTuple, Collector collector) {
  if (isPrefixQuery) {
    queryKeyTuple.values[0].collectAllRightPartners(collector);
  } else {
    queryKeyTuple.values[1].collectAllLeftPartners(collector);
  }
}
```

8.5 System specifications

The benchmark was executed on a system with the following specifications:

Hardware

- Intel Core i7-2600K quad core CPU, running at 3.4GHz
- 16 GB RAM

- Crucial C4 128GB solid state drive (operating system)
- Seagate Barracuda 3TB hard drive (benchmark application)

Software

- Microsoft Windows 8 Pro, 64 bit
- Oracle Java SE Runtime Environment 1.7.0.21
  - Java HotSpot 64-Bit Server VM
  - maximum heap size: 2 GB

8.6 Results

In this section we present the results of our benchmarking application. In the first three subsections, we evaluate the results for exact queries, partial-range queries and full-range queries respectively. In each of these subsections, we first compare the results of the different algorithms with respect to different relevant variables (for example, with respect to changing numbers of elements N in the storage). In Section 8.6.4, we specifically discuss the results for small input sizes to answer research question #5. Likewise, Section 8.6.5 discusses differences between the baseline approach – generic data structures that can be used in all cases – versus specialised optimisations to answer research question #6. We then aggregate the single results into a decision making process that recommends specific data structures for different scenarios (Section 8.6.6). Because we do not know a priori which scenario is applicable in practice, we leave it to actual implementers to choose the data storage for the respective use case. In Section 8.6.7 we provide a higher level of evaluation that considers the need to have multiple query types.

We use the following abbreviations consistently throughout the evaluation:

- N is the number of elements currently stored in the storage.
- M is the length of the key-tuple, that is, the number of elements in the tuple.
- W is the number of wild cards in a query-key tuple.

If not mentioned otherwise, the 95% confidence interval of the provided statistics is below $\pm 5\%$ within the mean and are not shown in the charts.

Applicability of storages

Not all storage strategies are shown in each of the results. This is due to the inherent limitation of the applicability of some storages.

- `SingletonStorage` only supports keys with length $M = 0$
- `PerObjectStorage` only supports keys with length $M = 1$
- `AssociationAspectStorage` only supports keys with length $M = 2$
- `TrieStorage` requires the key length M to be at least 1.

Also, some storage implementations (specifically `PerObjectStorage` and `AssociationAspectStorage`) are only applicable if the equality of key-tuples only relies on the identity of key elements. For example, the `PerObjectStorage` strategy stores the aspect instance inside the related context value (such as the current call target or the object in whose context the current method is called). During aspect-instance retrieval, the same aspect-instance will only be returned if the (only) context value inside the query-key tuple is exactly the same object. Although this restriction is often unproblematic, there are scenarios where this is not the case, for example, if the object that has a reference to the aspect-instance is serialised and then deserialised. Also, scenarios are imaginable where we intentionally do not want to associate aspect instances to a specific *identity*, but to a specific *value*. In those cases, optimisations that modify the object layout are not applicable as easily[19].

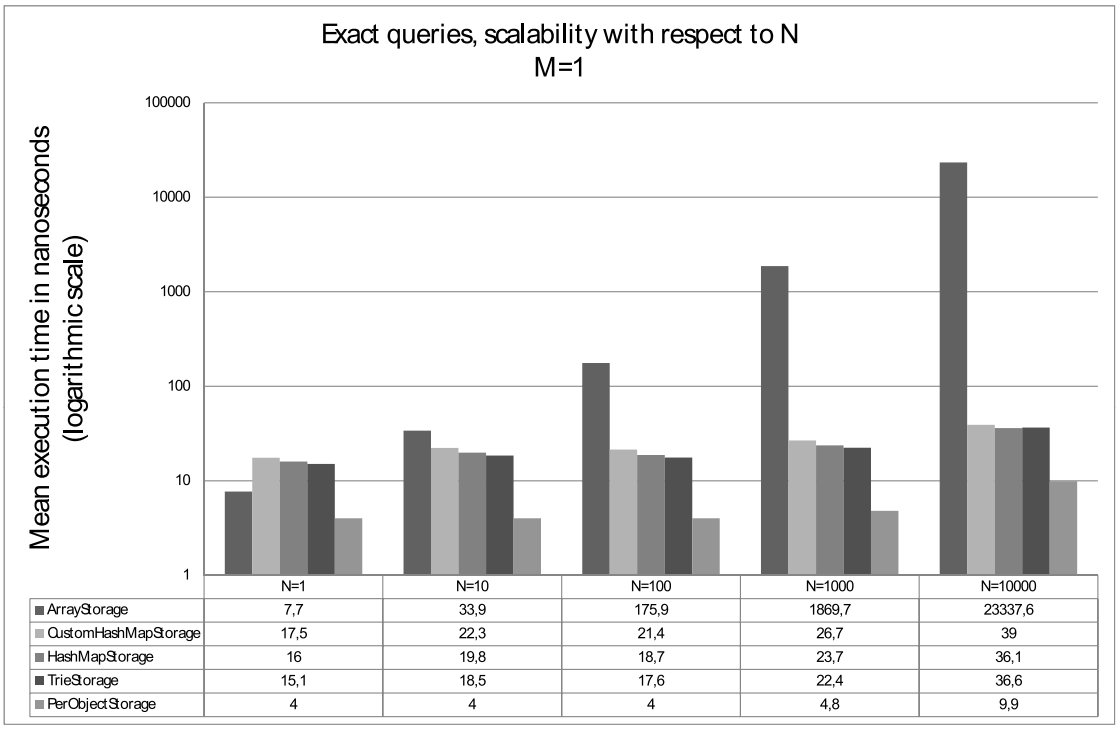
Example 20. Consider an application that connects to multiple databases. A single database-connection is represented by an object of the class `Connection`. Multiple `Connections` can connect to the same database. Two `Connections` are considered *equal* if they connect to the same database (identified for example by a connection string). We want to have a different aspect-instance for each distinct database (not for each distinct connection, though), for example, to aggregate usage statistics by logging all calls to instances of `Connection`. In this scenario, we need the query-key tuple to compare the *value* of the key elements, not the *identity* to retrieve the associated aspect instance. ▪

[19] It is, in fact, possible to circumvent this issue, but it requires another indirection and makes aspect-instance more complicated.

8.6.1 Exact queries

The first group of scenarios involves exact queries, that is, finding the single aspect-instance that is associate with an exact query-key tuple. The relevant variables for these queries are N (the number aspect-instances stored in the storage) and M (the length of the key tuple). For the initial evaluation, we keep one variable fixed and compare the influence of the respective other variable on the different storages.

Fixed key length M, varying number of elements N



	N=1	N=10	N=100	N=1000	N=10000
ArrayStorage	7,7	33,9	175,9	1869,7	23337,6
CustomHashMapStorage	17,5	22,3	21,4	26,7	39
HashMapStorage	16	19,8	18,7	23,7	36,1
TrieStorage	15,1	18,5	17,6	22,4	36,6
PerObjectStorage	4	4	4	4,8	9,9

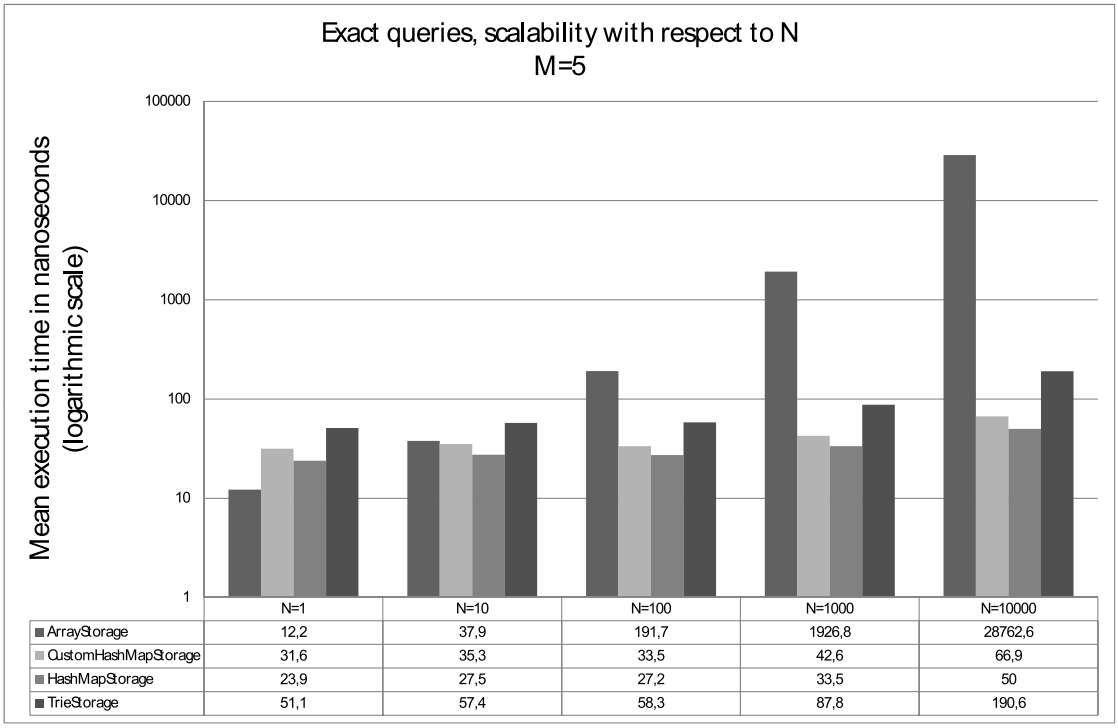
Figure 11: Mean execution time of exact queries for different N with fixed query-key length $M = 1$.

Figure 11 compares the mean execution time of exact queries for different values of N when the key-tuple size M is 1.

- As expected, the execution time for the array-based storage scales roughly linearly with the number of elements it stores. For $N < 10$ it performs in the same order as the other storages. For larger values it becomes considerably slower.

- The `CustomHashMapStorage`, `HashMapStorage` and `TrieStorage` are all based on hash tables. When $M = 1$, the trie used by `TrieStorage` has only one level. Therefore, all three storages perform a single hash-table look-up for the exact query. As expected, the results are similar.

- The `PerObjectStorage` always requires only a single field access to perform the exact query. This is always the fastest choice. However, when $N = 10000$, the absolute execution time more than doubles. A possible explanation for this is an increase of cache misses due due to reduced locality of data. With an increasing number of existing objects, it is more likely that an object that is being accessed is not in one of the caches that are closer to the CPU.



	N=1	N=10	N=100	N=1000	N=10000
ArrayStorage	12,2	37,9	191,7	1926,8	28762,6
CustomHashMapStorage	31,6	35,3	33,5	42,6	66,9
HashMapStorage	23,9	27,5	27,2	33,5	50
TrieStorage	51,1	57,4	58,3	87,8	190,6

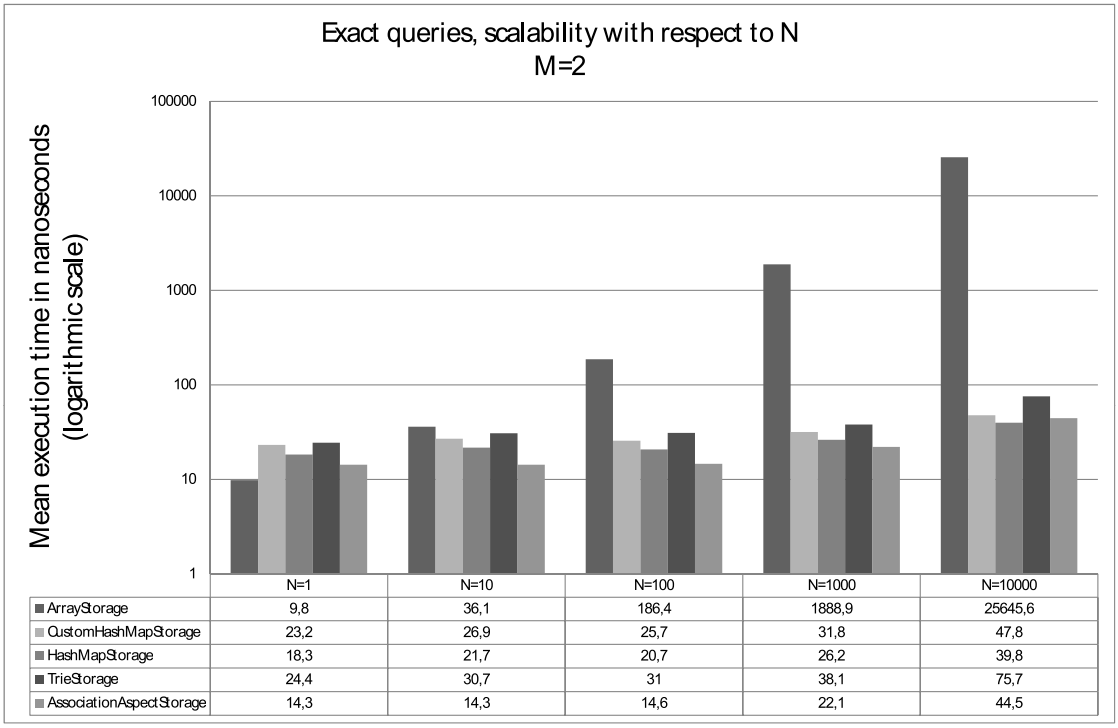
Figure 12: Mean execution time of exact queries for different N with fixed query-key length $M = 5$.

When the size of the query-key tuple is increased to 5, the execution times change as shown in Figure 12.

- For `ArrayStorage`, the situation does not change much. For $N < 10$ it performs similar or better than the alternative, for higher values of N it becomes considerably slower.

- The two hash table implementations `CustomHashMapStorage` and `HashMapStorage` have not become much slower: even though the key-tuple size increased from 1 to 5, the execution time roughly only doubled. Thus, these storages do not depend that much on the size of the query-key tuple.

- The `TrieStorage` on the other hand clearly depends on the key-tuple size. Each additional element in the query-key tuple adds another level to the trie and therefore another hash table look-up. The increase of the execution time clearly supports this, especially when $N = 10000$. In this case the execution time is roughly five times larger for $M = 5$ than for $M = 1$. For lower values of N, this factor is not as high, though, ranging from 3.4 to 3.9.

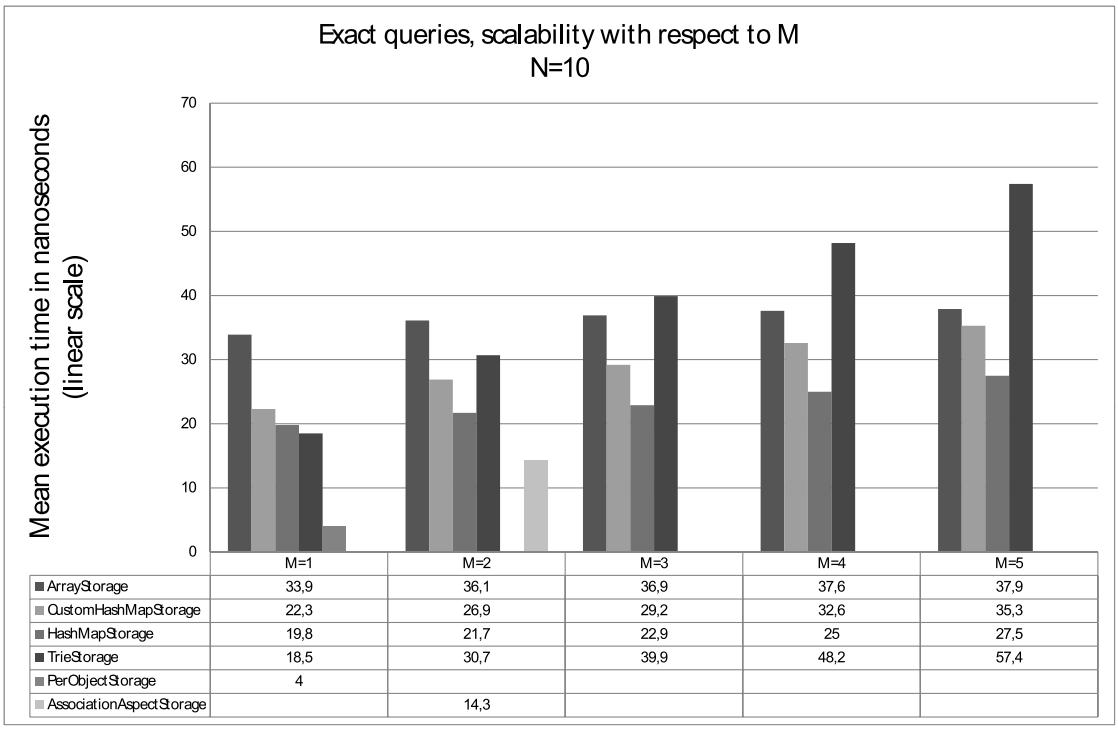


	N=1	N=10	N=100	N=1000	N=10000
ArrayStorage	9,8	36,1	186,4	1888,9	25645,6
CustomHashMapStorage	23,2	26,9	25,7	31,8	47,8
HashMapStorage	18,3	21,7	20,7	26,2	39,8
TrieStorage	24,4	30,7	31	38,1	75,7
AssociationAspectStorage	14,3	14,3	14,6	22,1	44,5

Figure 13: Mean execution time of exact queries for different N with fixed query-key length $M = 2$.

For $M = 2$ (Figure 13) we can make use of the `AssociationAspectStorage` which uses the optimisation described in Section 2.2.1 and Section 8.4.7. Because an exact query in this case is again a single look-up in a hash table, the results are similar to those for `CustomHashMapStorage` and `HashMapStorage`. Storing the aspect instances with the related objects apparently gives a minor performance benefit for exact queries when $N \leq 1000$. For $N = 10000$, the performance is similar to the other hash-map storages.

Fixed number of elements N, varying key-length M



	M=1	M=2	M=3	M=4	M=5
ArrayStorage	33,9	36,1	36,9	37,6	37,9
CustomHashMapStorage	22,3	26,9	29,2	32,6	35,3
HashMapStorage	19,8	21,7	22,9	25	27,5
TrieStorage	18,5	30,7	39,9	48,2	57,4
PerObjectStorage	4				
AssociationAspectStorage		14,3			

Figure 14: Mean execution time of exact queries for different M with fixed number of elements $N = 10$.

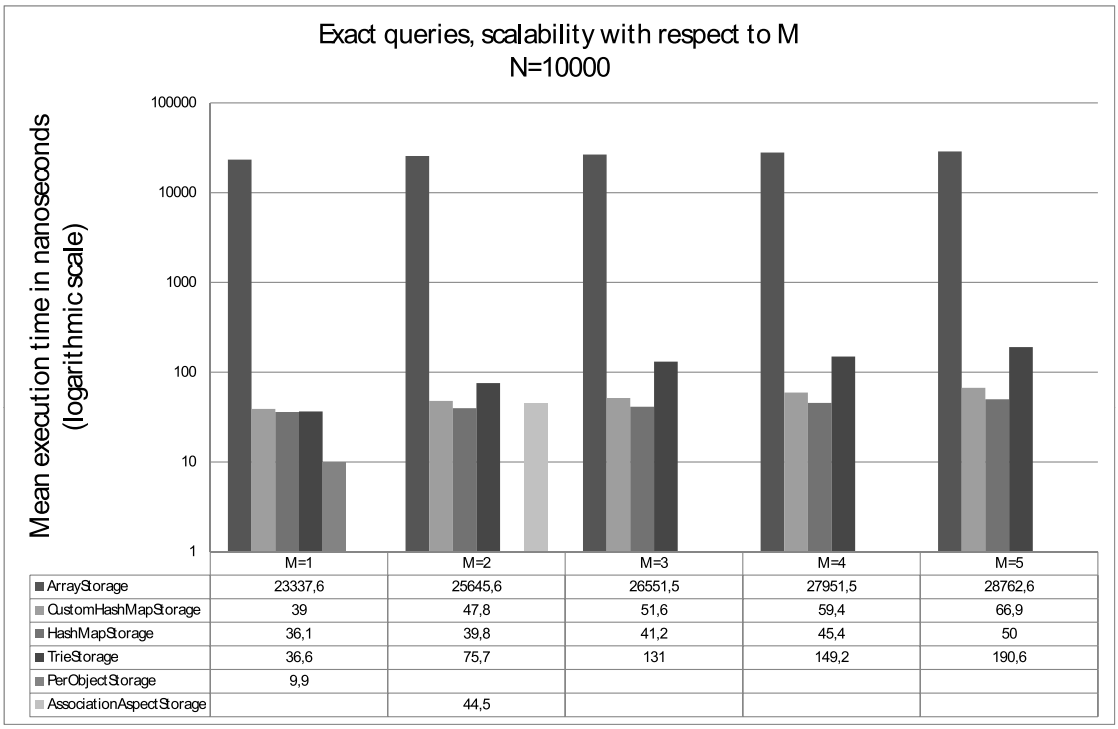
Figure 14 also shows the results for exact queries, but compares the influence of M for a constant $N = 10$.

- The two optimisations (`PerObjectStorage` and `AssociationAspectStorage`) are clearly the fastest, if the respective M is supported.

- `ArrayStorage` does not depend much on M. The comparison of the query-key tuple to the stored key tuple apparently does not affect the overall execution time for values of M between 1 and 5. In the scenarios we executed, no key tuple shared common elements. As a result, the key comparison will only need compare the first key element if the evaluated array element is not the sought one.

- `CustomHashMapStorage` and `HashMapStorage` moderately depend on M. This may be due to the fact that the calculation of the hash value of the query-key tuple takes longer for larger key tuples.

- `TrieStorage` depends the most on M, because the number of hash table look-ups directly corresponds to the number of elements in the key tuple



	M=1	M=2	M=3	M=4	M=5
ArrayStorage	23337,6	25645,6	26551,5	27951,5	28762,6
CustomHashMapStorage	39	47,8	51,6	59,4	66,9
HashMapStorage	36,1	39,8	41,2	45,4	50
TrieStorage	36,6	75,7	131	149,2	190,6
PerObjectStorage	9,9				
AssociationAspectStorage		44,5			

Figure 15: Mean execution time of exact queries for different M with fixed number of elements $N = 10000$.

Figure 15 shows the results when $N = 10000$.

- The overall effects of M are similar to those in Figure 14.
- `ArrayStorage` becomes slow for larger values of N, but stays independent of M.
- Hash based storages only become slightly slower. The impact of an increased key length is similarly moderate.
- `TrieStorage` isas highly dependent on M as for $N = 10$.

Singleton scenario

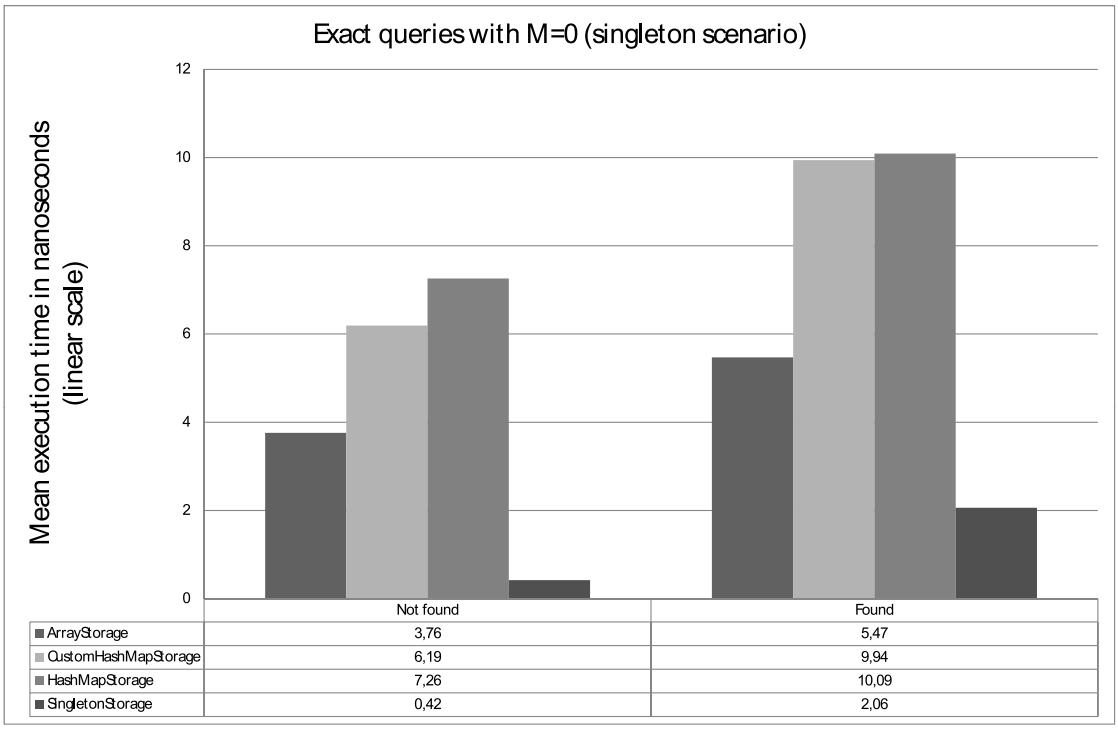


Figure 16: Singleton scenario. $M = 0$. Comparing mean look-up times a) when no instance has been stored yet (“Not found”) and b) when an instance has been stored (“Found”)

Singleton aspects are a special case in which all advice invocations use the same aspect instance. In this case, the query-key tuple has a length of zero, that is, the aspect-instance retrieval does not depend on any context value. Figure 16 compares the mean execution time for the general purpose storages (`ArrayStorage`, `CustomHashMapStorage` and `HashMapStorage`) as well as for the optimised storage `SingletonStorage`. The left set of results (labelled “Not found”) shows the case where no instance has been stored in the storage yet. The right set of results(labelled “Found”) represents the case where an instance which has earlier been stored is retrieved.

- As expected, the `SingletonStorage` is the fastest solution by a large margin. It simply looks up a value in a field.
- The additional 2–3 nanoseconds required by all storages in the “Found” case is caused by the fact that the storages add the aspect instance to a `Collector`

object, a collection holding the result of the query. Although this overhead is negligible for more complex queries, it is apparently relevant for simple cases such as the singleton scenario. In practice, the implementation of an exact query may instead choose to return the found aspect instance as a result directly, without the need to add the result to a collection.

Conclusions on exact queries

- `PerObjectStorage` and `AssociationAspectStorage` are good choices for all tested values of N. Naturally, they are not applicable in all cases (specific key-lengths M are required and key-tuple elements cannot be compared by value, but only by identity), but if they are applicable, they offer execution speed that is similar to or faster than the speed of other storages. There is, however, the memory overhead of adding a field to every object of the relevant classes. Also, the hash table used in AssociationAspectStorage has a bigger relative memory footprint than for example a simple array.

- When only a small number of aspect-instances needs to be handled ($N \leq 10$), array based storages are a reasonable choice. The look-up speed is comparable to most other storages, it is almost independent of the key-tuple length and the storage has the smallest memory foot-print of the implemented storages.

- As a general purpose implementation for exact queries, hash table based storages (`CustomHashMapStorage`, `HashMapStorage`) offer a good performance that is neither affected much by the number of aspect-instances N nor the length of the key-tuple M. The main drawback of these storages is the memory overhead caused by the hash table. Hash tables typically have a maximum fill factor of around 75% to avoid hash collisions, so at least 25% of the array slots are left empty. Also, when using separate chaining, each element stored in the hash table requires an additional bucket object to be instantiated.

- For small values of M (key-tuple length), `TrieStorage` performs comparably to `CustomHashMapStorage` and `HashMapStorage`. However, the trie never offers an actual advantage. The longer the key-tuple gets, the more hash table look-ups are required for exact queries.

8.6.2 Partial-Range Queries

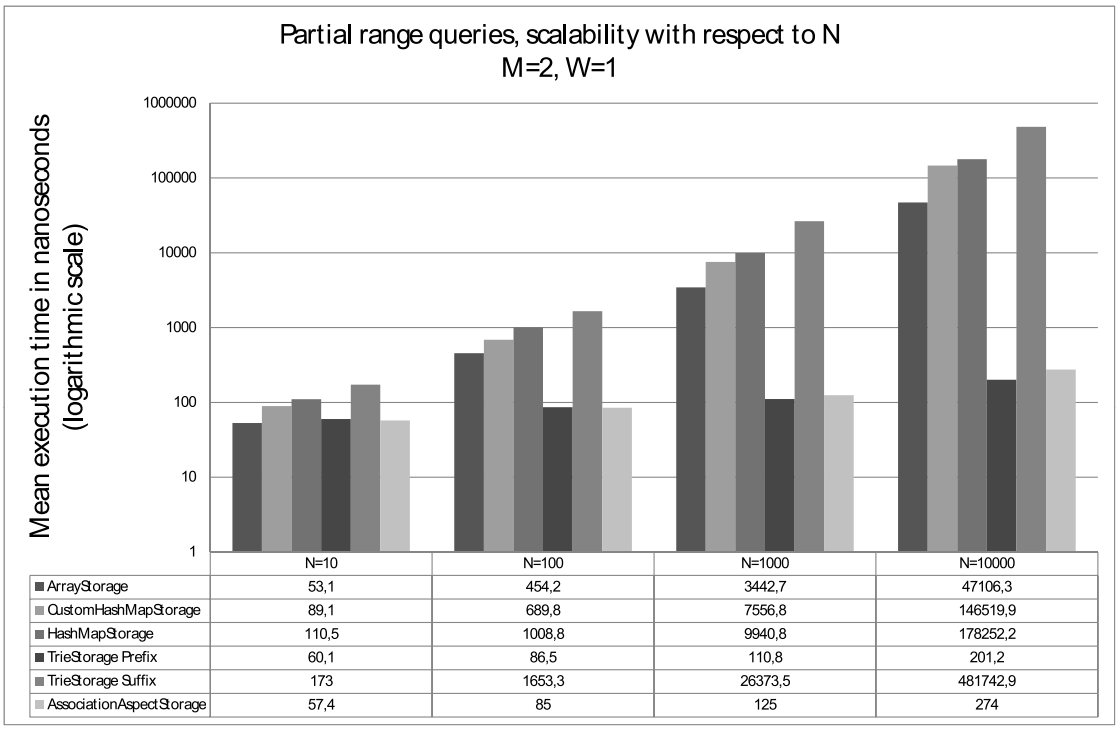
In this section we evaluate different scenarios involving partial-range queries, that is, queries with a query-key tuple containing one or more wildcards.

`TrieStorage` is the only storage that behaves drastically different depending on the layout of the query-key tuple. Therefore, we have included the results of the `TrieStorage` for prefix queries (`TrieStorage Prefix`) and suffix queries (`TrieStorage Suffix`) separately. The suffix results are only relevant if multiple different partial queries that can not both be transformed to prefix queries need to be executed on the same `TrieStorage`. If only a single query type needs to be executed, the key tuples can always be transformed into a prefix format using the transform function (see Section 4.3).

The other storages are not affected by a specific key-tuple layout, therefore we have only included the results for a default key-tuple layout.

Partial-range queries depend on a number of variables. The main variables are: the number of aspect-instances (N), the key-tuple length (M) and the number of wildcards in the query-key tuple (W). We started the evaluation of partial-range queries by keeping two of the variables fixed, while varying the respective third variable and compared the influence of this third variable on the performance of the different storages. Unless otherwise noted, each partial query that is executed in the benchmark returns exactly 10 matching aspect instances.

Fixed query-key length M and wildcard count W, varying number of elements N



	N=10	N=100	N=1000	N=10000
ArrayStorage	53,1	454,2	3442,7	47106,3
CustomHashMapStorage	89,1	689,8	7556,8	146519,9
HashMapStorage	110,5	1008,8	9940,8	178252,2
TrieStorage Prefix	60,1	86,5	110,8	201,2
TrieStorage Suffix	173	1653,3	26373,5	481742,9
AssociationAspectStorage	57,4	85	125	274

Figure 17: Mean execution time of partial-range queries for different N with fixed query-key length $M = 2$ and one wildcard.

Figure 17 shows the mean execution time of partial-range queries when $M = 2$ and $W = 1$ (one wildcard), for different values of N. This resembles the example partial-range query shown in Listing 1 on page 9 (second pointcut).

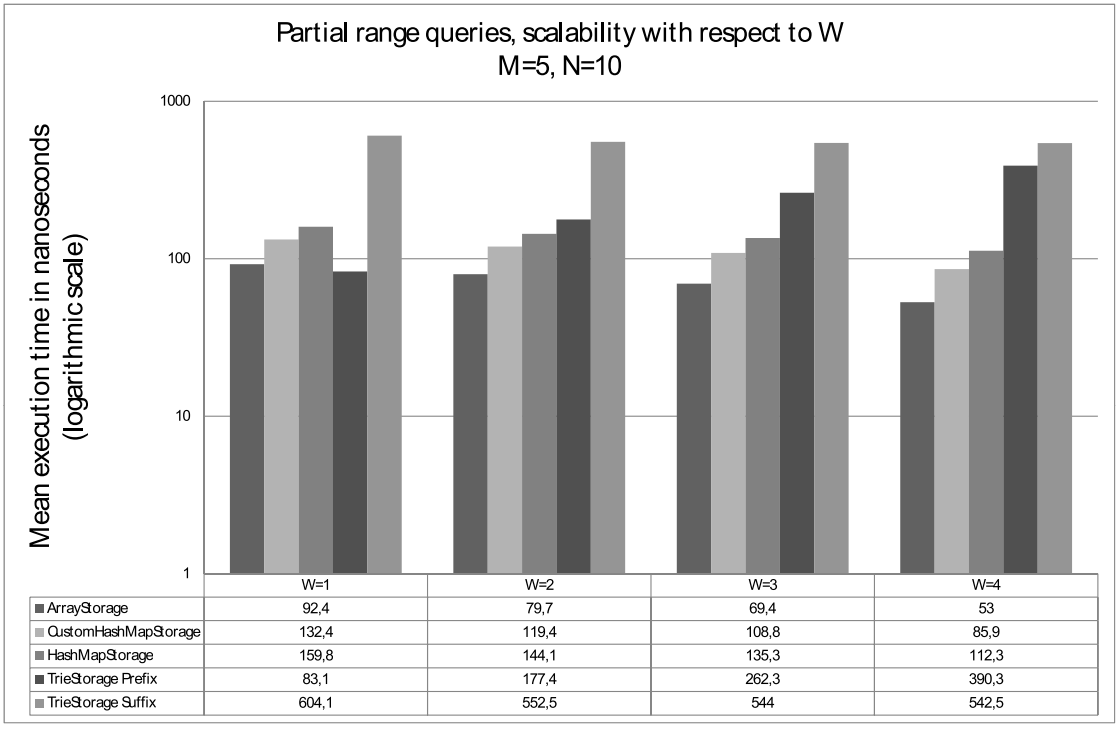
- The results `ArrayStorage` scale linearly with the number of elements in the storage. For low values of $N \leq 10$, it is among the fastest options.
- The two hash map implementations `CustomHashMapStorage` and `HashMapStorage` perform notably worse than `ArrayStorage` for larger values of N.
- All three storages need to iterate over all their elements to find the matching aspect instances. Iterating over an array is apparently faster than iterating over the buckets of a hash map. The `ArrayStorage` uses a *dense* array, that is, each "cell" of the array up to an index $N - 1$ contains a value. The array that holds the buckets of the hash maps on the other hand necessarily contains gaps. Also, each array element is a potential start of a linked list that holds all elements

that hash to the same array index. Therefore, not only is it necessary to skip over gaps in the array, it is also possibly necessary to iterate over multiple linked lists.

- When the query-key tuples have a prefix layout, the `TrieStorage` shows its benefits. It is optimised for this kind of query. When $M = 2$ and $W = 1$, a partial-range query consists of a single hash table look-up and an iteration over a hash table. A node in a trie represents all elements that have a key with the same prefix. Ideally, this number is small compared to the N. The higher this number is with respect to N, the more child nodes will have to be traversed.

- If the layout of the query-key tuple is not a prefix layout, the trie needs to be traversed completely, which is even more complex than iterating over a single hash table. As a result, for this scenario, the trie is actually the worst choice.

Fixed element count N and query-key tuple length M, varying wildcard count W



	W=1	W=2	W=3	W=4
ArrayStorage	92,4	79,7	69,4	53
CustomHashMapStorage	132,4	119,4	108,8	85,9
HashMapStorage	159,8	144,1	135,3	112,3
TrieStorage Prefix	83,1	177,4	262,3	390,3
TrieStorage Suffix	604,1	552,5	544	542,5

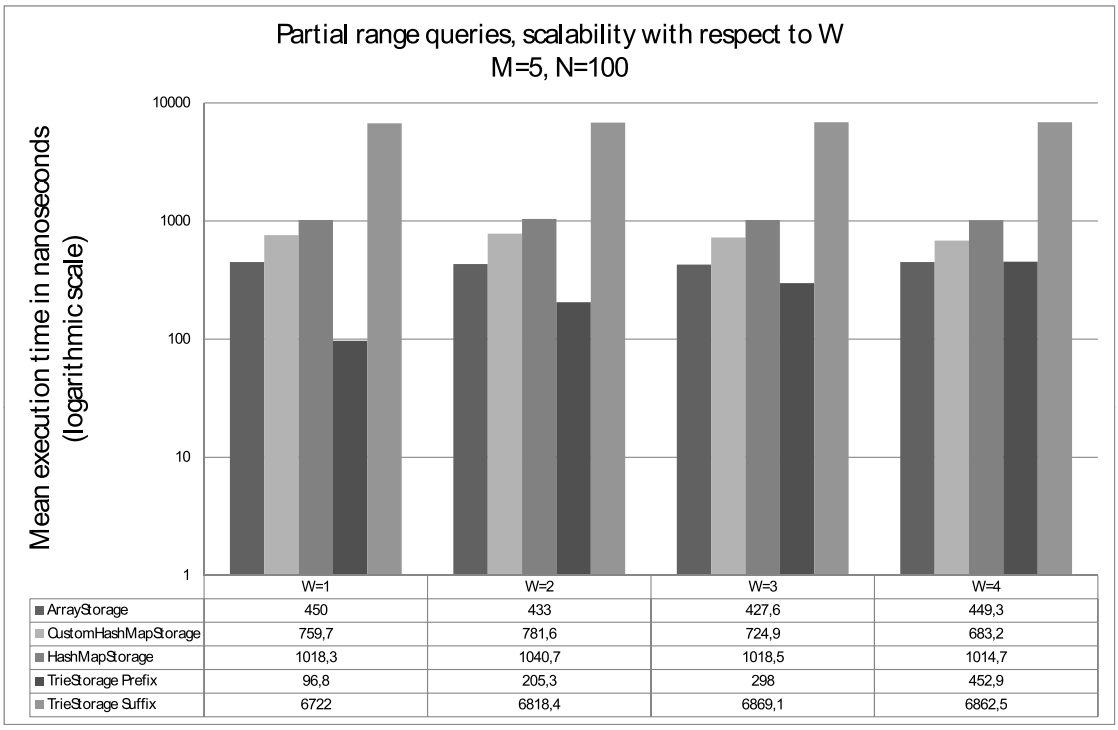
Figure 18: Mean execution time of partial-range queries for different number of wildcards W with fixed query-key length $M = 5$ and $N = 10$.

When we increase the size of the query-key tuple we can compare the influence of the number of wildcards on the execution time. Figure 18 on the preceding page shows this influence for $N = 10$.

- For all storages except `TrieStorage`, the execution time *decreases* when the number of wildcards *increases*. This is due to the fact that the comparison of a stored key-tuple with the query-key tuple becomes simpler if the number of wildcard increases: fewer elements need to be compared.

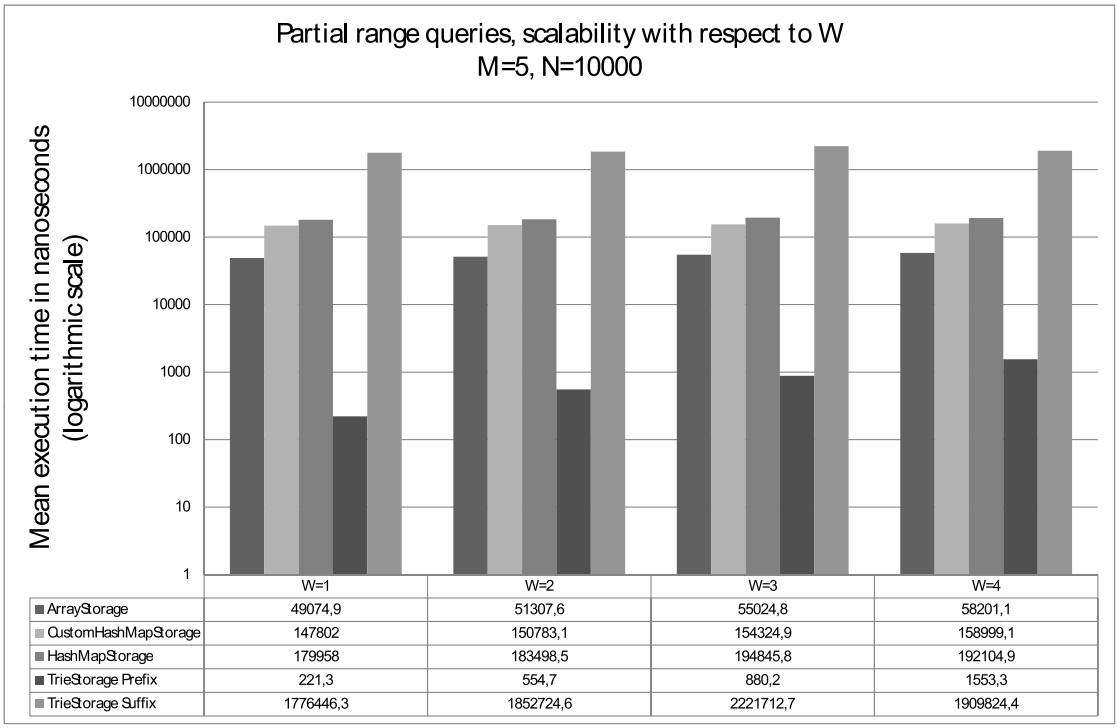
- In all cases, `ArrayStorage` is the fast, because iteration of an array is comparatively simple.

- The execution time for partial-range queries *increases* for `TrieStorage` (prefix) when the number of wildcards increases. A query-key tuple with more wildcards means shorter prefixes. Shorter prefixes result in larger sub-trees of the trie to traverse, that is, more hash tables need to be iterated. In this example – the tries in the benchmark are rather well balanced – the execution time for tries increased roughly linearly with the number of wildcards. For more than one wildcard, the `TrieStorage` is the slowest storage solution.



	W=1	W=2	W=3	W=4
ArrayStorage	450	433	427,6	449,3
CustomHashMapStorage	759,7	781,6	724,9	683,2
HashMapStorage	1018,3	1040,7	1018,5	1014,7
TrieStorage Prefix	96,8	205,3	298	452,9
TrieStorage Suffix	6722	6818,4	6869,1	6862,5

Figure 19: Mean execution time of partial-range queries for different number of wildcards W with fixed query-key length $M = 5$ and $N = 100$.



	W=1	W=2	W=3	W=4
ArrayStorage	49074,9	51307,6	55024,8	58201,1
CustomHashMapStorage	147802	150783,1	154324,9	158999,1
HashMapStorage	179958	183498,5	194845,8	192104,9
TrieStorage Prefix	221,3	554,7	880,2	1553,3
TrieStorage Suffix	1776446,3	1852724,6	2221712,7	1909824,4

Figure 20: Mean execution time of partial-range queries for different number of wildcards W with fixed query-key length $M = 5$ and $N = 100$.

When we increase the number of elements in the storage, tries start to become faster when compared to the alternatives as shown in Figure 19 on the previous page and Figure 20.

- When $N = 100$ (Figure 19), the `TrieStorage` is the fastest option for $W \leq 3$ and close second when $W = 4$. Again, the lower the number of wildcards, the better `TrieStorage` performs (up to 9 times faster than `HashMapStorage` for $W = 1$).
- The more elements we put into the storage, the larger the advantage of `TrieStorage` becomes. When $N = 10000$ and $W = 1$, the `TrieStorage` outfperforms the other storages by a factor of about 220 (with respect to `ArrayStorage`) to 760 (with respect to `HashMapStorage`).
- Even when $W = 4$, `TrieStorage` performs the partial-range query on average about 45 times (with respect to `ArrayStorage`) to 150 times (with respect to `HashMapStorage`) faster.

Varying size of result set

As mentioned earlier, we expect the execution time for partial range queries in prefix trees to be dependent on the number of matching elements, that is, the number of elements that share the same prefix. Figure 21 compares the influence of the number of elements that match the partial-range query in a use case where the key-tuple length is 5 and 10,000 elements are stored in the storage. We use the variable Q to refer to the number of elements that are returned from the executed partial-range query. That is, a value of $Q = 100$ means that the partial-range query returns 100 elements, or – specifically important for the `TrieStorage` – that 100 aspect instances are associated with keys that share the same prefix.

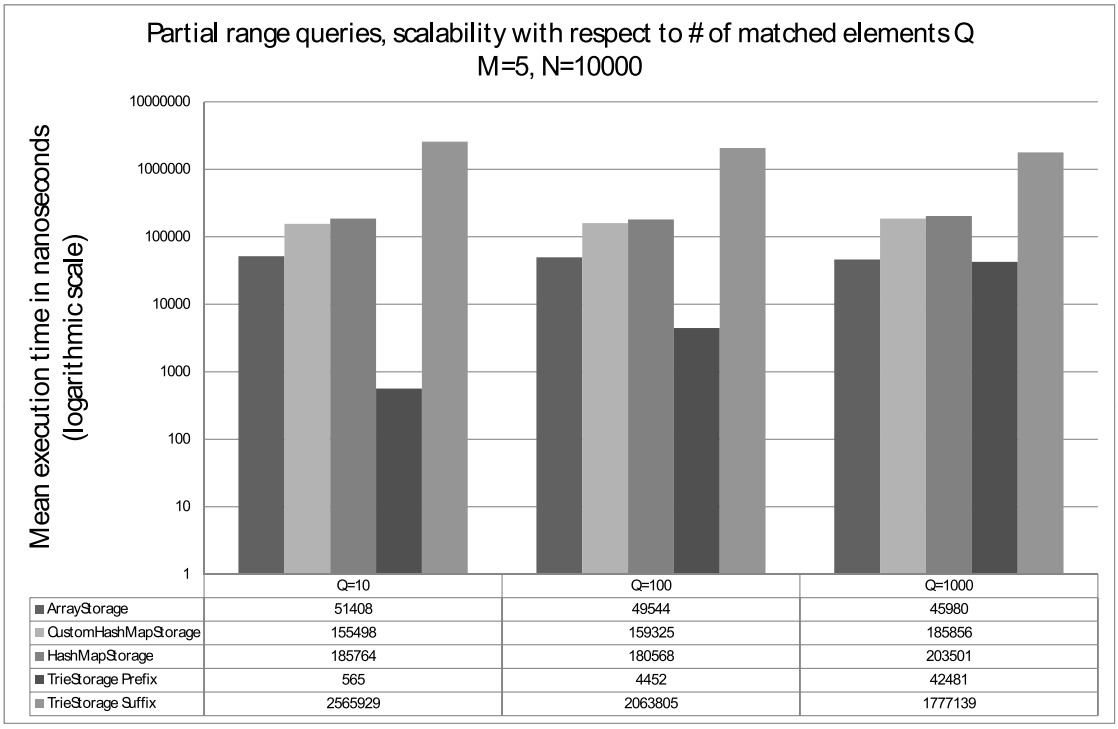


Figure 21: Mean execution time of partial-range queries for different number of matched elements Q with fixed query-key length $M = 5$ and $N = 10000$.

- `ArrayStorage`, `CustomHashMapStorage` and `HashMapStorage` are not much influenced by the number of results. This is not unexpected, as the layout of the data structure does not change if multiple elements sharing the same key-tuple prefix are stored. In each case, the data-structure must iterate over all its elements to find the matching entries.
- `TrieStorage Prefix` is clearly affected by the number of matched elements. The

execution time scales almost linearly with Q. This is, again, not unexpected. The part of the query that finds the node representing the prefix of the query-key tuple will take about the same in all cases. However, the collection of all entries below this node involves the iteration of multiple hash tables, and these hash tables may also be larger. Naturally, the collection of elements below a node takes the longer the more elements need to be collected.

- For the `TrieStorage Suffix` storage, the execution time of the query decreases with an increasing value of Q. This can be explained by the fact that the first two levels of the trie will have significantly fewer nodes, because more elements share the same prefix.

Conclusions on partial-range queries

- For partial-range queries with $M = 2$, we were not able to find an advantage of the association-aspect optimisation compared to the prefix tree implementation. This is not surprising, given that in both cases, the query consists of a single hash-table look-up followed by an iteration over another hash table. Prefix trees can therefore probably be used as a substitute for the association-aspects optimisation without any penalty.

- When the number of aspect instances is small ($N \leq 10$), `ArrayStorage` is a good choice and will perform at least as good as any other strategy.

- `ArrayStorage` is also a good choice when the number of aspect instances is not more than about 100, but the key-length is large ($M = 5$) and the query contains more than 2 wildcards.

- For all scenarios where $N \geq 100$, the `TrieStorage` offers the best performance and scaling behaviour as long as the query is a prefix query. Queries on a prefix tree are far less dependent on the number of elements. The fewer wildcards the query contains, the larger is the advantage. However, even if four of five elements in the query-key tuple are wildcards, our prefix tree implementation can be queried about 37 times faster than the closest competitor when $N = 10000$. This makes the `TrieStorage` the clear recommendation for all scenarios where the number aspect instances is expected to be larger than 100.

- `TrieStorage` can execute partial-range queries the faster the fewer elements in the trie share the same key-tuple prefix. In the worst case, where all elements share the same key-tuple prefix, the performance of partial-range

queries in `TrieStorage` is expected to be comparable to or even worse than `HashMapStorage`.

- If the query-key tuple cannot be transformed into a prefix layout, `TrieStorage` is by far the worst choice. It is much slower than alternative solutions (often by a whole magnitude) and has a considerable memory footprint due to the many hash tables it creates.

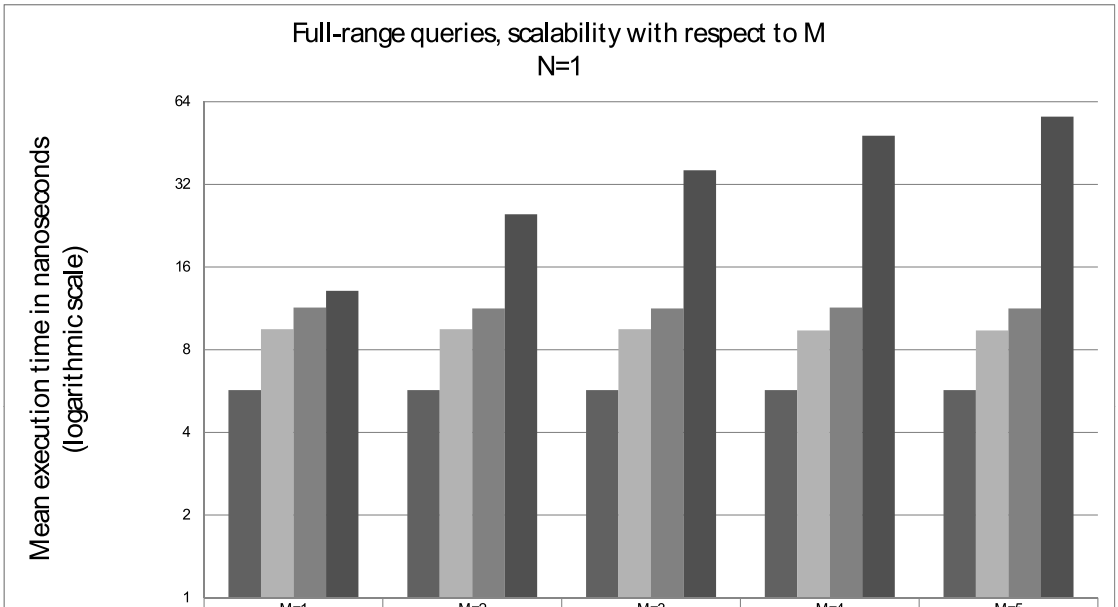
- We have found a slight benefit of our custom hash-map implementation (which does not require the allocation of an `Iterator` to traverse over all entries) over the default `HashMap` implementation . Partial-range queries using the custom hash map are up to 25% faster than those queries using stock `HashMaps`.

8.6.3 Full-Range Queries

In this section we evaluate the results for full-range queries. Those queries always return all instances of a specific aspect. Some storage implementations inherently do not allow these full-range queries. `PerObjectStorage` and `AssociationAspectStorage` save aspect-instances in the objects that are used as context values. They do not maintain a list of all instances. Therefore, a full-range query is not possible with these storages.

Again, the relevant variables for these queries are N (the number of aspect-instances stored in the storage) and M (the length of the key tuple). Like we do for exact queries, we keep one variable fixed and compare the influence of the respective other variable on the different storages.

Fixed number of elements N, varying key-tuple length M



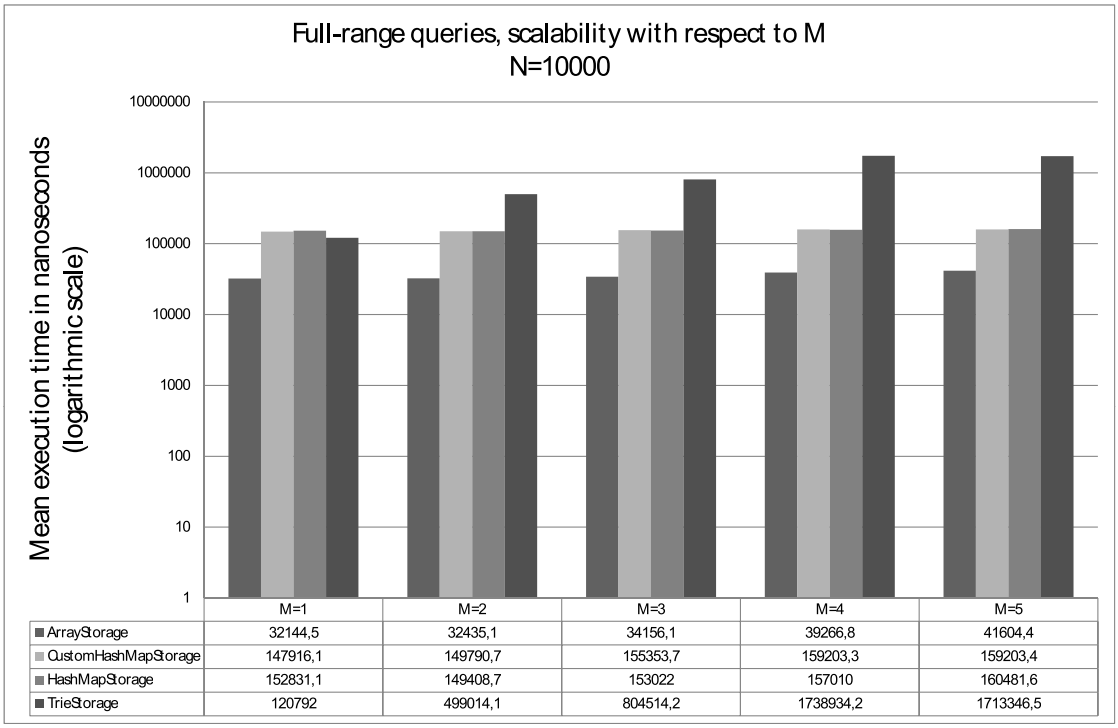
	M=1	M=2	M=3	M=4	M=5
ArrayStorage	5,7	5,7	5,7	5,7	5,7
CustomHashMapStorage	9,5	9,5	9,5	9,4	9,4
HashMapStorage	11,4	11,3	11,3	11,4	11,3
TrieStorage	13,1	24,9	36	48,1	56,5

Figure 22: Mean execution time of full-range queries for different key-tuple sizes and a fixed number of elements $N = 1$.

Figure 22 on the preceding page compares the mean execution time for full-range queries with different key-tuples lengths on storages that hold one element ($N = 1$).

- `ArrayStorage`, `CustomHashmapStorage` and `HashMapStorage` are not affected by the length of the key-tuples that have been used to store the elements. A full-range query only involves an iteration over all elements in the storage without the need to compare the key tuples in any way.
- `ArrayStorage` is the fastest in all cases, even though the absolute difference to `CustomHashmapStorage` and `HashMapStorage` for this small case where $N = 1$ is minuscule (4 to 6 nanoseconds).
- `TrieStorage` becomes notably slower for larger key-tuple sizes. This can be explained by the fact that each element in the key tuple adds a level to the prefix tree used internally. Each additional level results in exactly one additional hash table to iterate over when $N = 1$. The increase in execution time is almost linear to the key-tuple length.

This pattern appears for other values of N as well. For example, Figure 23 shows the results when $N = 10000$.



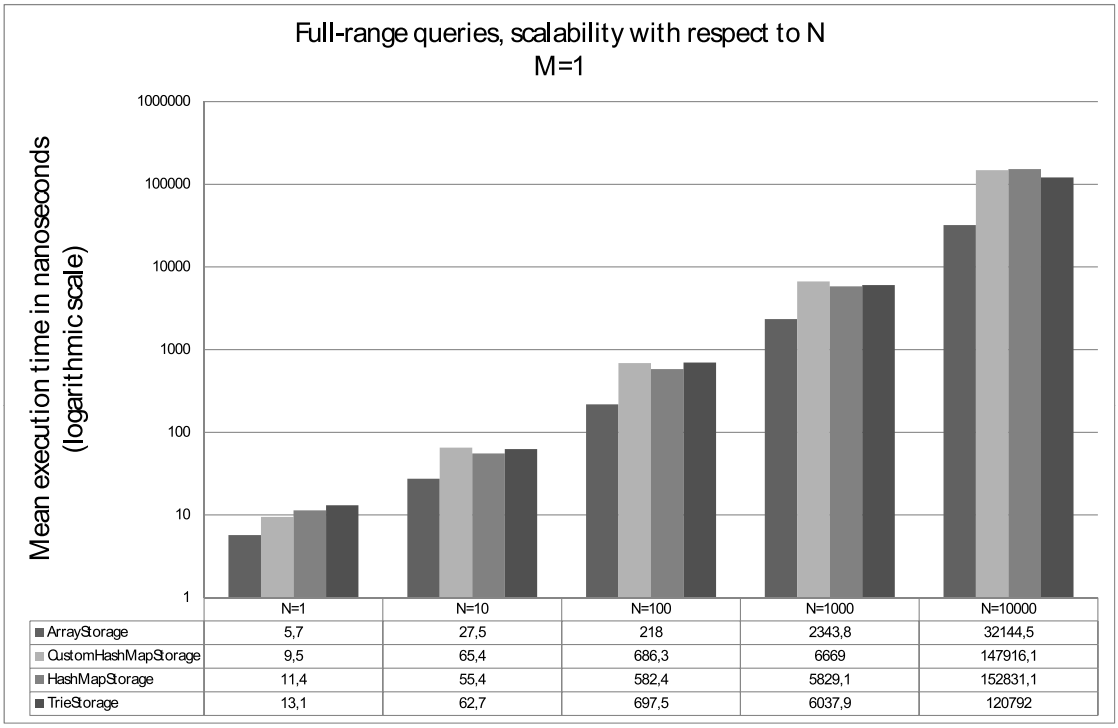
	M=1	M=2	M=3	M=4	M=5
ArrayStorage	32144,5	32435,1	34156,1	39266,8	41604,4
CustomHashMapStorage	147916,1	149790,7	155353,7	159203,3	159203,4
HashMapStorage	152831,1	149408,7	153022	157010	160481,6
TrieStorage	120792	499014,1	804514,2	1738934,2	1713346,5

Figure 23: Mean execution time of full-range queries for different key-tuple sizes and a fixed number of elements $N = 10000$.

- `ArrayStorage`, `CustomHashmapStorage` and `HashMapStorage` are still not affected by the key-tuple length.

- Full-range queries on `TrieStorage` again become slower for larger values of M. However, the execution time does not increase linearly with respect to the key-tuple length any more. The execution time does not appear to increase for$M > 4$.

Fixed key-tuple length M, varying number of elements N



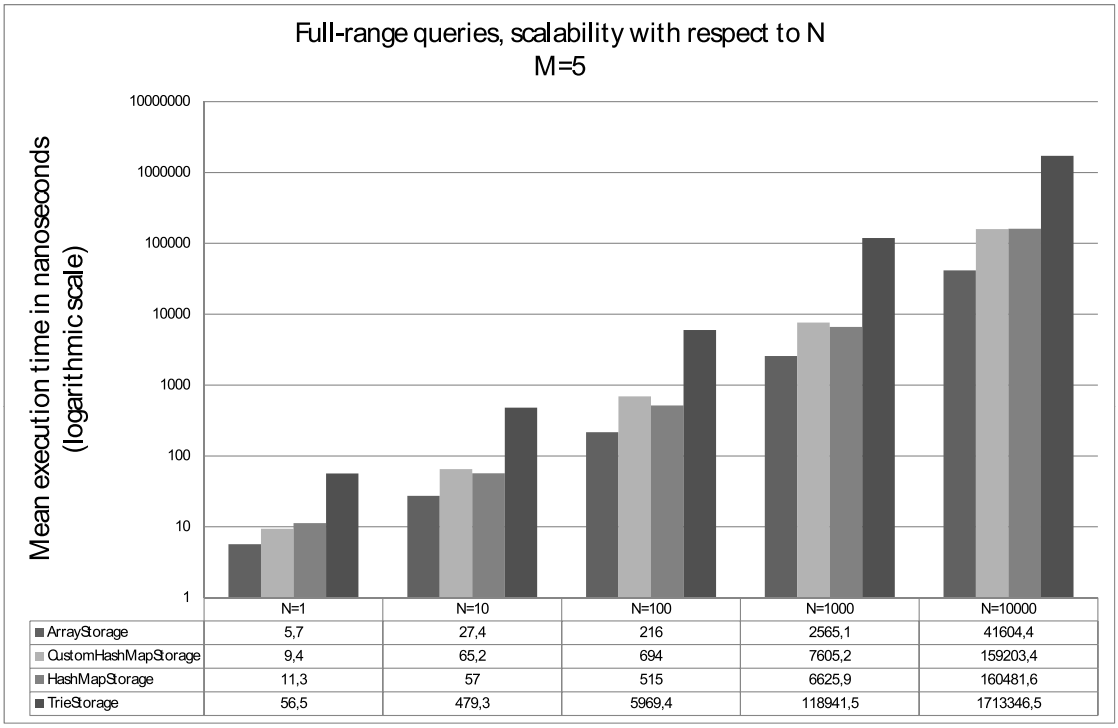
	N=1	N=10	N=100	N=1000	N=10000
ArrayStorage	5,7	27,5	218	2343,8	32144,5
CustomHashMapStorage	9,5	65,4	686,3	6669	147916,1
HashMapStorage	11,4	55,4	582,4	5829,1	152831,1
TrieStorage	13,1	62,7	697,5	6037,9	120792

Figure 24: Mean execution time of full-range queries for different numbers of elements and a fixed key-tuple length $M = 1$.

Figure 24 compares the storages for different numbers of elements, assuming a fixed key-tuple length $M = 1$.

- As in all previous results regarding full-range queries, `ArrayStorage` is the fastest to execute the query. For larger N, this advantage becomes significant. For example, for $N = 10000$, `ArrayStorage` is about four times faster than the fastest other storage (`TrieStorage`).

- The other three storages perform comparably.
- Between $N = 10$ and $N = 1000$, the execution times for all storages scale almost linearly to the number of elements. However, for $N = 10000$, the increase is larger, especially for `CustomHashMapStorage`, `HashMapStorage` and `TrieStorage`, which all take more than 20 times longer for $N = 10000$ than for $N = 10000$, even though the number of elements is only ten times larger. We suspect cache misses and other issue with the lack of data locality to be the cause for this.



	N=1	N=10	N=100	N=1000	N=10000
ArrayStorage	5,7	27,4	216	2565,1	41604,4
CustomHashMapStorage	9,4	65,2	694	7605,2	159203,4
HashMapStorage	11,3	57	515	6625,9	160481,6
TrieStorage	56,5	479,3	5969,4	118941,5	1713346,5

Figure 25: Mean execution time of full-range queries for different numbers of elements and a fixed key-tuple length $M = 5$.

When we increase the key-tuple size to $M = 5$, we get the results shown in Figure 25.

- The general behaviour is comparable to that for $M = 1$.
- As seen earlier, the execution time for `TrieStorage` notably increases compared to $M = 1$ due to the additional levels in the trie.
- The non-linear increase of the execution time for larger values of N is present for $M = 5$ as well. However, for `TrieStorage`, this effect already occurs for lower values of N: the execution time for $N = 1000$ is already about 20 times higher than for $N = 100$.

Conclusions on partial-range queries

- The `ArrayStorage` is the fastest solution for all cases of full-range queries we benchmarked.
- The more complex data structures, especially `TrieStorage`, provide no advantage over simple arrays for full-range queries. On the contrary, they execute full-range queries considerably slower and have a higher memory footprint.

8.6.4 Small datasets versus large datasets

In this section, we focus on the difference between data stores when the datasets are small in order to answer research question #5.

Question #5: how do query operations perform in scenarios where the number of aspect instances is comparatively small?

As mentioned in Section 5.1, the asymptotic computational complexity that can be used as a means to describe the time requirements of algorithms. However, these descriptions are theoretical statements for large dataset sizes. The results presented earlier in this chapter show that for small dataset sizes, the relative execution time of different algorithms can be close even though the asymptotic computational complexity is rather different. For example, even though an exact query for arrays belongs to $O(n)$ while for hash tables it belongs to $O(1)$, the latter is not always the better choice. Using an array for exact queries or partial range queries can lead to faster queries than using a hash-table, if the number of elements in the array is small (for example, less than 10) and the key-tuple size is not too large (for example less than 5). This confirms our assumption that for small dataset sizes, using a simple data structure can be both beneficial with respect to execution time *and* memory consumption.

8.6.5 Baseline implementation versus specialisation

In the previous sections, we presented data stores that are generally applicable (baseline approach) as well as implementations of specialisations that are applicable in specific scenarios only. In this section we discuss the differences between the two approaches in order to answer research question #6.

Question #6: How efficient are operations on data structures that are generally applicable in all scenarios in comparison to operations on optimized data structures that can only be used in specific circumstances?

Our results show that using a specialised implementation (that is, `PerObjectStorage`, `AssocationAspectStorage`, `SingletonStorage`) is practically always a good choice if applicable. Especially the singleton approach — which we expect to be a common case in practice — has proven to be significantly faster than any baseline implementation, running in only a fraction of the time other approaches take (up 17 times

faster). Likewise, `PerObjectStorage` performs significantly faster than any generic baseline implementation. The advantage of `AssociationAspectStorage` is slightly less obvious. For small numbers of associations ($N = 1$), it is even slightly slower than an array based approach, but for all other scenarios it is at least as fast as any other implementation.

As expected, specialised implementations are applicable in a narrow scope only. However, *if* they are applicable, they can provide a remarkable advantage over the baseline implementation.

8.6.6 Recommendations for single-query scenarios

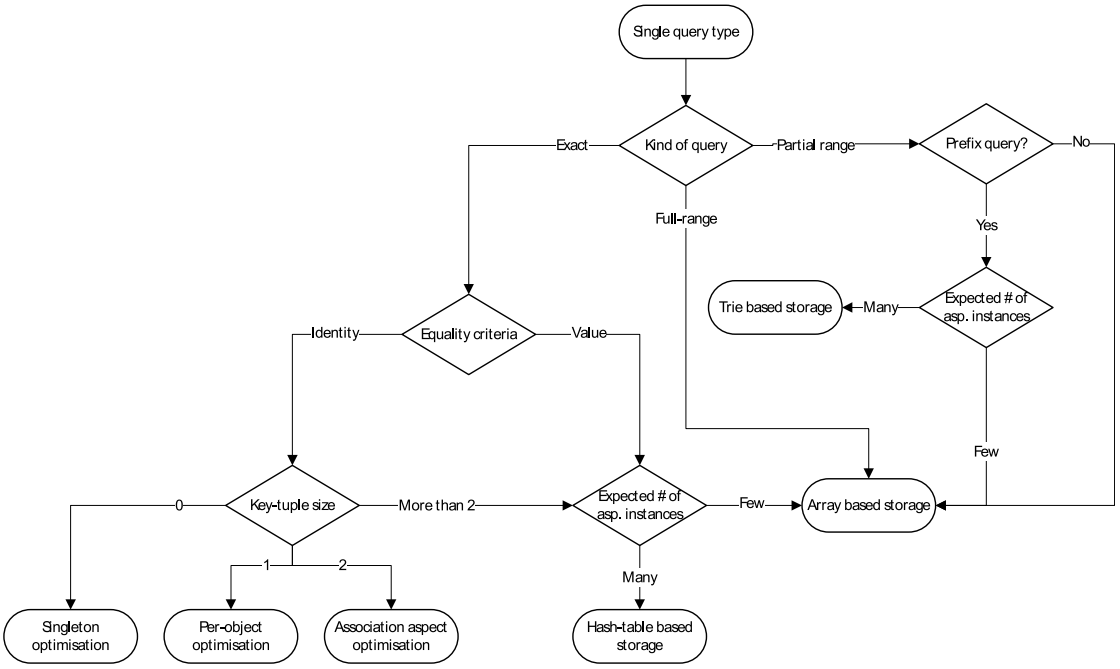


Figure 26: Choosing a data structure when only one type of query is required.

With the evaluation results in the previous subsections we can now establish a recommendation for a specific data structure depending on the use case, answering our final research question.

Question #7: Which data structure is recommended in which practical scenario?

No single data structure that we evaluated came off as the definite choice for all scenarios. Therefore, the choice for a certain data structure must be made depending on the requirements of the specific use case. We outline this decision-making process in Figure Figure 26 on page 110. Some of the decisions in the process are based on the number of expected aspect-instances. That is, there is a specific threshold above (or below) which a different storage is more beneficial than another. We classify the number of aspect-instances with abstract quantifications ("few" and "many") instead of specific numbers. On our test system, the threshold between "few" and "many" was often at around 10 aspect-instances. However, this number may be different on other systems.

- For exact queries, a hash table based storage is a good general-purpose choice for basically all scenarios. The execution speed scales well for all values of N and M. Also, this storage does not require the modification of object layouts. We highly recommend a hash-table based storage as a baseline approach.

- For exact queries in scenarios in which the equality of context values is based on the identity of the object rather than their "value", the optimisation techniques for $M = 0$ (`SingletonStorage`), $M = 1$ (`PerObjectStorage`) and $M = 2$ (`AssociationAspectStorage`) are good choices. Aside from a specific key-tuple length, `PerObjectStorage` and `AssociationAspectStorage` have additional requirements that involve the modification of the memory layout of class. If the aspect-oriented execution-environment does not provide these mechanisms out of the box, an implementation of these storages can potentially be a challenging task. Also, adding fields to a class increases the size of the objects in memory. If many of these objects are instantiated, but only few of them are actually used to store references to the aspect instances, there is a certain "waste" of memory caused by this design.

- If it is known that only a small number of aspect instances will exist at runtime, an array-based storage may be sufficient for any kind of query. The simple structure of arrays makes many operations on arrays similarly fast as on other data structures (or even faster) as long as the number of elements is small. Also, arrays have little memory-overhead compared to other data structures.

- If only full-range queries are required, arrays are the best choice. The elements can be iterated quickly and the memory overhead is small.

- For partial-range queries where the query-key tuple can be transformed into a prefix layout, tries (prefix trees) are highly recommended, as they are specifically made to handle prefix queries efficiently. There are factors that can cause

tries to become less advantageous (such as a high number of wildcards, or many aspect-instances being associated to keys with the same prefix). However, these factors never cause tries to be a significantly bad choice.

8.6.7 Complex scenarios

The recommendations in Section 8.6.6 assume that only one type of query has to be executed for a specific deployed aspect. Aspect-oriented languages such as AspectJ may also support multiple pointcut-advice pairs for a single aspect. Depending on the exact case, this scenario may require different queries to be executed for different pointcut evaluations. The `Equality` association aspect shown in Listing 1 is such a case. Two queries must be executed: an exact query for the first pointcut (lines 11–14) and a partial-range query for the second pointcut (lines 18–20).

As we have shown earlier, no single data structure is able to handle all scenarios equally well, so choosing data structures for scenarios in which multiple queries need to be executed is a trade-off. This trade-off is typically about execution speed versus memory footprint.

Redundant data structures

One choice is to use a distinct data structure for each query type. In the `Equality` example one could choose a data structure that supports fast exact queries (such as a hash-table based data structure) for the first pointcut and another data structure that supports efficient partial-range queries (such as trie-based storage) for the second pointcut.

Advantages:

- Each query can be executed against a data storage that is optimised for that kind of query. None of the queries suffers from a penalty due to an sub-optimal storage strategy.
- The relative frequency of occurrence of each query type is not important as each query type receives its own data storage. No heuristics are required to determine the “most important” query.

Disadvantages:

- Multiple data storages need to be maintained. When an aspect instance is created, it needs to be added to each of the data structures. Likewise, when an aspect instance is removed at runtime, it needs to be removed from multiple data structures.
- The memory footprint is typically higher compared to a solution that uses only one data structure. The more different queries need to be supported, the higher the memory footprint is expected to be.

Shared storage

Another choice is to choose one data structure as the only storage which is then used for all required queries.

Advantages:

- The memory footprint is expected to be smaller, as data is stored in one storage only.
- Maintenance is simpler, because an aspect instance only needs to be added to or removed from one data structure.

Disadvantages:

- Some queries are likely to be executed more slowly. For example, the `Equality` example requires partial-range queries and exact queries. If only a trie-based storage is used, partial-range queries can be executed quickly (see Figure 17 on page 97). However, for the exact queries, the trie is expected to be slower than for example a hash-based storage (see Figure 13 on page 91).
- Knowledge about the relative frequency of the queries is required a priori to make the best choice. In general, a data structure should be chosen in such a way that the overall penalty for using only one storage is as small as possible. For example, if a specific type of query is executed much more often than another query type, then it is probably advisable to choose a data structure that makes the first query type execute the most efficiently.

Adapting to runtime changes

A more complex strategy to select a storage is to use an adaptive approach. In this approach, the data structure that is used to store the aspect instances is exchanged

with a different data structure when the usage pattern of the application changes. For example, consider an application that has a clear distinction between a complex initialisation state and a steady state. The frequency of different queries may change when the application changes from one state to another. In this case, changing the data storage when this change happens may offer a considerable improvement of the performance. We expect that finding a heuristic which automatically determines when to change the storage strategy to be one of the biggest challenges in this adaptive approach.

8.7 Threats to validity

8.7.1 Threats to internal validity

As described in Section 8.2, our benchmark is specifically designed to minimize the systematic errors of its results. The methodology is mostly determined by the rigorous measurement approach suggested by Georges et al. [31]. Our means to avoid measurement bias include:

- We measure durations that are long enough (at least 1 millisecond) to minimise the effect of the limited timer resolution (300 nanoseconds on the test system).
- Execution times are measured until a steady state is reached.
- Outliers are removed before statistics are calculated.
- We recreate all relevant object for each test invocation to reduce the influence of memory caches.
- We randomise all hash values to avoid coincidental advantageous or disadvantageous scenarios.
- We perform multiple VM invocations to get independent results.

There are still a number of factors that may have influenced our results.

- Within a single VM invocation, we throw away the lowest and highest 5% of all measurements (10% in total) to remove outliers. This is a simple heuristic to handle outliers and may affect the results. We cannot tell exactly if we remove too few samples (that is, the sample set still contains outliers) or too many samples (that is, we throw away valid samples). Also, we expect more outliers with high values than with low values, for example, due to interruptions caused

by the operating system or by garbage collection. We chose our approach because it caused the detection of the steady state to terminate reliably. If we were too rigorous in this respect, the detection of the steady state may have taken too long for practical purposes, or it may frequently not terminate at all.

- Query invocations are executed in a partially unrolled loop that performs 100 invocations per loop. There is still a small overhead that is caused by the loop itself (such as a variable comparison and a jump instruction) which we do not exclude from the results. However, we expect this error to be comparatively small. Also, we perform all tests using the same loop. Only the number of repetitions varies for different batch size. We therefore expect the error caused by the loop overhead to be equal for all results.

8.7.2 Threats to external validity

Our primary goal is to provide a recommendation for certain data structures given specific scenarios. This requires our results – which were gathered on a single system – to be transferable to other environments. Certain factors may threaten the validity of this generalisation.

- The software, including the base code, the aspect-oriented execution environment and the implementation of the storage, will most likely run on systems that differ considerably from our testing system. Hardware and software configurations can vary substantially, including different processor architectures or operating systems. The complexity of these systems and the many variables that may influence the programs running on them certainly threaten the amount to which we can reliably generalise our results. Nevertheless, we consider most computer systems used today to be sufficiently similar in their general method of operation that our results should be qualitatively repeatable on a wide range of systems. For further confirmation we recommend more comparative measurements on different systems.

- We do not include the insertion or removal of elements in our results. However, we expect most practical applications to follow a “write few, read many” paradigm. That is, the number of aspect-instance retrievals is expected to be much higher than any changes to the data structures by insertion or removal.

- The decision process described in Section 8.6.6 assumes that certain variables, such as the length of the key-tuple or the number of aspect instances, are known in advance. For some variables, this may be a difficult requirement to meet. We especially expect the number of aspect instances to be difficult

to predict. Instead of using such a prediction, an aspect-oriented execution-environment may have to implement heuristics to adapt to the situation at runtime (as described in Section 8.6.7).

9 Summary and Future Work

Aspect-instance look-up (as part of an instantiation policy) can have a considerable effect on the total execution time of a program [8]. Therefore, we want this look-up to be as fast as possible. Existing instantiation policies such as **perobject**, **singleton** or **association aspects** try to minimise the time required by aspect-instance look-up by using optimisations that are tailored to the respective policy. Some of these optimisations make use of rather advanced techniques such as modifying the layout of classes at load-time.

These optimisations are typically only applicable in a specific set of scenarios. For example, the **perobject** optimisation used by AspectJ (that is, adding a field to the related classes which stores the aspect instance) is only applicable if the choice for a specific aspect instance depends on only one context object and only on the identity of that context object (and not the value). Also, this optimisation only allows exact queries. Range queries are inherently not supported.

Such limitations prohibit the use of those optimisation strategies in new instantiation policies. Also, not all optimisations may be transferable to all execution environments (for example, if the modification of class layouts is not supported). The goal of this thesis is therefore twofold: firstly, we want aspect-instance look-up to be fast and secondly, we want to have an aspect-instance storage-strategy that is applicable in as many scenarios as possible. We understand that these two goals at least partially exclude each other: a solution that is applicable in many scenarios is typically not as efficient as a solution specifically tailored to a certain scenario.

Our unified model shows that the storage strategy, which aspect-instance look-up is a part of, can be separated from the semantics of an instantiation policy. This allows us to consider storage strategies isolated from semantics.

Based on our benchmark, we suggest a solution for aspect-instance storage following a **two-level approach**. The first level is the **baseline** solution, a storage strategy applicable in a wide range of scenarios. The second level is an **optimised** solution which is limited in scope, but more efficient than the baseline approach.

For the first level, the **baseline** solution, we suggest the implementation of the aspect-instance storage using data structures such as arrays, hash tables and prefix trees (tries). The choice for a specific data structure mainly depends on the types of the queries that need to be executed on the storage. If only one type of query needs to be executed, we suggest the use of the following data structures:

- For exact queries: a hash-table based storage

- For partial-range queries (prefix queries)[20]: a trie-based storage
- For full-range queries: an array

If multiple queries of different types need to be executed, a trade-off needs to be made:

- If execution speed is the more important criterion: for each query type, maintain a distinct storage that is the most efficient for that query type (see above).
- If memory usage is the more important criterion: pick a storage type that provides the best overall execution speed for all required query types. For example, a trie can execute both a partial-range query and an exact query comparatively fast.

The choices made in on this first baseline level are not necessarily the fastest for all scenarios, but they provide a solution that is simple to implement, yet reasonably fast.

The second level, the **optimised** solution, requires more knowledge of the usage scenario, but provides more opportunities for faster execution speed. The optimisations found in AspectJ belong to this level: they are tailored to a specific set of scenarios, but may provide a considerable benefit in terms of execution speed, as our benchmark results suggest. According to the benchmark results, the optimisations were always about as fast or faster than any non-optimised (baseline) solution, so we suggest to use those optimisations if they are applicable and supported by the aspect-oriented execution-environment. The applicability of those optimisations only depends on the length of the key-tuple and the ability of the execution-environment to modify the class layout, so it should be comparatively easy to determine when to use them.

Other optimisations are not necessarily as easy to apply, because they require extra knowledge, not only about statically acquirable information (such as the size of the key-tuple or the types of the queries), but also knowledge of specific runtime statistics, such as the expected number of aspect-instances. It is mostly the boundary cases (such as having only a couple of aspect instances at runtime) in which the storages show performance behaviour that is not described by their asymptotic time complexity. For example, if the number of aspect instances is predicted to be small, a storage based on a simple array may provide a viable solution which has a small memory footprint and is fast for any kind of query. We leave it to future development to come up with mechanisms to make full use of these boundary cases.

[20] This implies that the size of the key-tuple is at least 2, otherwise no partial-range query would be possible.

The knowledge required to take advantage of boundary cases must be either predicted, which may or may not be possible, or may be taken from actual statistics gathered at runtime. Ideally, the execution-runtime would adapt to the actual situation during the execution of a program and pick the storage strategy that provides the fastest access, comparable to some Java execution environments which selectively optimise certain methods based on actual usage. Again, we leave it to future research to design and implement such an adaptive approach.

The results of this thesis provide guidance for implementers of instantiation policies. With the suggested two-level approach, aspect-instance look-up can be implemented quickly and efficiently on a baseline level with an optional optimisation stage as a second level, where applicable.

For future research, we suggest to execute the benchmark on multiple systems to further confirm the external validity of our results. Additionally, we suggest to implement the benchmarked storages inside actual virtual machines, such as the Jikes RVM [33] or Steamloom [29]. Martin Zandberg has already suggested approaches to optimise aspect-instantiation strategies in ALIA4J using JIT compilation [19]. As a future work, we suggest the incorporation of the choice of data structures for aspect-instance storage into his work.

References

[1] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[2] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," *Compiler Construction*, vol. 2622 of Springer Lecture Notes in Computer Science, pp. 40–60, 2003.

[3] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 5, pp. 890–910, 2004.

[4] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 26–35.

[5] D. Suvée, W. Vanderperren, and V. Jonckers, "Jasco: an aspect-oriented approach tailored for component based software development," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2003, pp. 21–29.

[6] I. Aracic, V. Gasiunas, M. Mezini, and K.Ostermann, "Overview of CaesarJ," *LNCS Transactions on Aspect-Oriented Software Development*, vol. 3880, pp. 135–173, 2006.

[7] A. J. de Roo, M. F. H. Hendriks, W. K. Havinga, P. E. A. Durr, and L. M. J. Bergmans, "Compose*: a language- and platform-independent aspect compiler for composition filters," in *First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, Paphos, Cyprus*. Cyprus: No publisher, July 2008.

[8] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge, "Measuring the dynamic behaviour of AspectJ programs," *SIGPLAN Not.*, vol. 39, no. 10, pp. 150–169, 2004.

[9] H. Masuhara and K. Kawauchi, "Dataflow pointcut in aspect-oriented programming," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2003.

[10] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya, "Association aspects," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 16–25.

[11] Aspectj 5 quick reference. [Online]. Available: http://www.eclipse.org/aspectj/doc/next/quick5.pdf

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] J. Bonér, "What are the key issues for commercial aop use: how does AspectWerkz address them?" in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 5–6.

[14] J. Bonér. (2004) Aspectwerkz - dynamic aop for java. [Online]. Available: http://web.archive.org/web/20050527035543/http://www.codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf

[15] R. M. Golbeck, S. Davis, I. Naseer, I. Ostrovsky, and G. Kiczales, "Lightweight virtual machine support for aspectj," in *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2008, pp. 180–190.

[16] M. Fleury and F. Reverbel, "The jboss extensible server," in *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 344–373.

[17] A. Loker, "A generalised model for instantiation policies," 2010.

[18] A. Loker, S. te Brinke, and C. Bockisch, "A unified model of aspect-instantiation policies," in *Proceedings of the 3rd international workshop on Free composition*, ser. FREECO '12. New York, NY, USA: ACM, 2012, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/2414716.2414718

[19] M. Zandberg, "Optimization of aspect-instantiation strategies in the jit-compiler," Master's thesis, University of Twente, January 2012.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Springer Berlin / Heidelberg, 1997, vol. 1241, pp. 220–242, 10.1007/BFb0053381. [Online]. Available: http://dx.doi.org/10.1007/BFb0053381

[21] J. Hartmanis and R. E. Stearns, "On the computational complexity of algorithms," *Transactions of the American Mathematical Society*, vol. 117, pp. pp. 285–306, 1965. [Online]. Available: http://www.jstor.org/stable/1994208

[22] N. G. de Bruijn, *Asymptotic Methods in Analysis*. Dover Pubn Inc, 1981.

[23] K. Mehlhorn and P. Sanders, *Algorithms and data structures: The basic toolbox*. Springer, 2008.

[24] D. E. Knuth, *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[25] ——, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[26] T. A. Standish, *Data Structures in Java*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[27] R. De La Briandais, "File searching using variable length keys," in *IRE-AIEE-ACM '59 (Western): Papers presented at the the March 3-5, 1959, western joint computer conference*. New York, NY, USA: ACM, 1959, pp. 295–298.

[28] O. Amble and D. E. Knuth, "Ordered hash tables," *Computer Journal*, vol. 17, no. 2, pp. 135–142, 1974.

[29] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann, "Virtual machine support for dynamic join points," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 83–92.

[30] M. Haupt, "Virtual machine support for aspect-oriented programming languages," Ph.D. dissertation, Technische Universität Darmstadt, 2005.

[31] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. New York, NY, USA: ACM, 2007, pp. 57–76.

[32] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: the performance impact of garbage collection," in *Proceedings of the joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM, 2004, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1005686.1005693

[33] (2007) Jikes research virtual machine. The Jikes RVM Project. [Online]. Available: http://jikesrvm.org/

Online resources have been checked for availability on 6th October 2014.

List of Algorithms

1 Generic algorithm to retrieve all applicable instances of an aspect. . . 30
2 The bind function of the *pertarget* policy. 31
3 The bind function of the `NotificationLog` aspect 32
4 Generic implementation of isExact 35

List of Figures

1 Aspect-instance retrieval as part of the pointcut evaluation. 21
2 Products of the thesis. Arrows represent order of production. 25
3 The relation between the semantic properties and responsibilities of instantiation policies. 29
4 Number of operations (y-axis) required by A and B for a given input size n (x-axis) 44
5 Number of operations (y-axis) required by C and D for a given input size n (x-axis) 45
6 Array of length 5. Boxes with dotted outlines refer to unused array positions. Arrows indicate references to objects. 54
7 A linked list with three elements. Arrows depict references. Dashed arrows mean $null$ 57
8 Objects of a BST. Arrows depict references. 61
9 Nodes forming a trie. Arrow labels show the element in the string that is used to discriminate children. Objects with thin borders depict data referenced by leaf nodes. 64
10 UML diagram showing a part of the design of the benchmark. Scenarios and storages can be combined freely, allowing new scenarios or storages to be added easily. 75
11 Mean execution time of exact queries for different N with fixed query-key length $M = 1$. 89
12 Mean execution time of exact queries for different N with fixed query-key length $M = 5$. 90
13 Mean execution time of exact queries for different N with fixed query-key length $M = 2$. 91
14 Mean execution time of exact queries for different M with fixed number of elements $N = 10$. 92
15 Mean execution time of exact queries for different M with fixed number of elements $N = 10000$. 93

16 Singleton scenario. $M = 0$. Comparing mean look-up times a) when no instance has been stored yet (“Not found”) and b) when an instance has been stored (“Found”) 94
17 Mean execution time of partial-range queries for different N with fixed query-key length $M = 2$ and one wildcard. 97
18 Mean execution time of partial-range queries for different number of wildcards W with fixed query-key length $M = 5$ and $N = 10$. 98
19 Mean execution time of partial-range queries for different number of wildcards W with fixed query-key length $M = 5$ and $N = 100$. 99
20 Mean execution time of partial-range queries for different number of wildcards W with fixed query-key length $M = 5$ and $N = 100$. 100
21 Mean execution time of partial-range queries for different number of matched elements Q with fixed query-key length $M = 5$ and $N = 10000$. 101
22 Mean execution time of full-range queries for different key-tuple sizes and a fixed number of elements $N = 1$. 104
23 Mean execution time of full-range queries for different key-tuple sizes and a fixed number of elements $N = 10000$. 105
24 Mean execution time of full-range queries for different numbers of elements and a fixed key-tuple length $M = 1$. 106
25 Mean execution time of full-range queries for different numbers of elements and a fixed key-tuple length $M = 5$. 107
26 Choosing a data structure when only one type of query is required. . . 110

List of Tables

1 Research questions and references to section in which they are answered. 27
2 Examples of complexity classes 41
3 Average asymptotic time complexity of basic operations on an array. . 52
4 Asymptotic computational complexity of different operations on data structures with n elements and key-tuples of length k. Full range queries are omitted, because those queries simply mean “full traversal” and is $O(n)$ in all cases.
[†1] using hash tables as secondary storage.
[†2] if prefix query 69

Listings

1 A simple association aspect [10] 9
2 The association aspect `NotificationLog` 10

3 Modified object layout for association aspects 11
4 Instantiation of `Equality` [10] 12
5 Examples for the **perthis** and **pertarget** instantiation policies. 14
6 Implementation of **pertarget**: an additional field is added to target classes. 15
7 **perthis** and **pertarget** expressed as association aspects 16
8 Conceptual example of singleton aspects as association aspects 17
9 Setting up a *NotificationLog* 34
10 A generic Java class that represents an association between a key tuple and an aspect instance of the provided type $Aspect$. 53
11 A generic single-linked list node to store associations of aspect-instances and key tuples 55
12 A simple single-linked list. 56
13 A binary search-tree node for a tree that stores associations between aspect instances and key tuples. 59
14 A sortable `Tuple` implementation for use in a BST 60
15 A node used in a trie that stores strings of Booleans. 61
16 A node for a trie that uses key-tuples with elements of type *Object.* . . 62
17 A node for a trie that associates key tuples with aspects of type *Aspect* . 64
18 Parts of a hash table to store $Association$s 68
19 Association class that relates a key tuple to an aspect instance. 76
20 Collecting all elements in the custom `SimpleHashMap` without allocating an `Iterator` 78
21 Executing a partial-range query on a `SimpleHashMap`. 78
22 `findNode` finds the node for a specific key prefix. 79
23 Method to add an aspect to the trie. 80
24 Finding a value by a key in a trie. 80
25 Implementation of a prefix query in the fixed depth prefix tree implementation 81
26 Exact query on a `SingletonStorage` 81
27 Parts of `KeyItem` that are relevant for `PerObjectStorage` 82
28 Storing and retrieving aspect instances in `PerObjectStorage` 82
30 `AssociationAspectStorage`: storing aspect instances 83
31 AssociationAspectStorage: exact query 83
29 Parts of `KeyItem` that are relevant to `AssociationAspectStorage` 84
32 `AssociationAspectStorage`: performing partial-range queries 85