MASTER THESIS

EFFICIENT POINTCUT PROJECTION

Remko Bijker

COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

EXAMINATION COMMITTEE
Dr.-Ing. Christoph Bockisch
Dr. Pim van den Broek

UNIVERSITY OF TWENTE.

29-10-2010

Software Engineering Group
University of Twente

Master thesis

Efficient pointcut projection

Author:
Remko Bijker

Supervisors:
Dr.-Ing. Christoph Bockisch
Dr. Pim van den Broek

October 29, 2010

Abstract

Pointcuts in aspect-oriented programming languages specify run-time events which cause execution of additional functionality. Hereby, pointcuts typically have a pattern-based static component selecting instructions whose execution triggers an event, e.g. a pattern that selects method-call instructions based on the target method's name. Current implementations realise identification of matching instructions by examining all instructions in the executed program and matching them against all patterns found in the program's pointcuts. But such an implementation is slow. An optimised implementation is therefore highly desirable in run-time environments which support the dynamic deployment of aspects; slow pattern evaluation invariably causes a slowdown of the entire application.

The patterns used in pointcuts supported by current languages, i.e. method, constructor, and field signatures, are well structured. In order to speed-up pattern matching, this structure can be exploited, both to create indexes over the relevant instructions and to optimise the order in which the sub-patterns are evaluated. In this thesis we present background on pattern matching and outline two optimisation strategies. We furthermore present two case studies that survey signatures and patterns that actually occur in the real-world. This results in several heuristics that can improve the performance of matching patterns to instructions.

The information gathered in the survey and the two outlines optimisation strategies are used to determine how efficient the different optimisation strategies work at run-time. We present multiple implementations of these optimisation strategies and compare their run-time performance. This results in several efficient optimisation strategies which can be chosen based on variables such as the number of attachments that are going to be applied, application size and memory requirements.

Contents

Chapter 1

Introduction

Aspect-orientation languages are means to reduce the maintenance costs for application components that cross-cut large parts of an application, e.g. logging.

This section will first define the terminology used in this thesis, after that the motivation and problem statement will be discussed.

The main contributions of Chapter 2, 4, and 5 are published in the proceedings of the VMIL workshop [4].

1.1 Terminology

In aspect-orientation there are different, sometimes opposite, terminologies. This section describes the terminology used in this report that is derived from [18]. The terminology is based on the one used for the ALIA4J framework [6].

Generic-function The logical grouping of functionality implementations which may all be executed as the result of executing a call-site. They apply to the same static context (have the same signature) but are applicable in different dynamic situations.

Pattern Description of a generic-function's signature, e.g. a MethodPattern describes the signature of a method.

Context Run-time information about the program's current call context, e.g. from which method it is called.

Atomic predicate A single test to execute at run-time.

Predicate A boolean formula of atomic predicates to execute at run-time.

Action Functionality implementation that needs to be executed on a dispatch. This is the code part of an AspectJ advice.

Specialisation Association of a pattern, predicate and contexts.

Attachment A collection of specialisations associated with an action.

Projection of specialisation All call-sites of generic-functions where the signature of the called generic-function matches the pattern of the specialization.

1.2 Context and motivation

In aspect-orientation implementations there are different times in the deployment of an application at which the needed support for can be applied [19].

Compile-time While compiling the application the compiler inserts the predicate evaluations and actions at every possible projection of the specialisations. This generally requires no modifications to virtual machines or loaders; all needed code is part of the application.

Load-time While loading the application an overridden loader inserts the predicate evaluations and actions at every possible projection of the specialisations. The modifications to the loader mean that there is a dependency on the workings of the virtual machine.

Run-time While running the application a hook is set at every possibly projection of the specialisations. This hook calls the predicate evaluations and actions. The break points are implemented by either a modified virtual machine or via debugging hooks.

In all implementations one needs to match call-sites to attachments. One common optimisation is calculating what specialisation may match at any given call-site and insert only code to execute the actions for the matching attachments like AspectJ does. In this case adding the actions at compile-time is a disadvantage because even if an attachment is not activated the code has to be added. For load and run-time deployment one can omit the code to execute an action if the attachment the action belongs to is not activated, thus reducing the impact of the attachment.

Even if it is possible, there are many concepts in aspect orientation that cannot be determined at compile/load-time, which is where matching call-sites to an attachment has to be done at e.g. run-time. The trivial solution to this matching is using a list of specialisations that is searched for matches with the call-site the application's execution is currently at. This can be optimised by using the non run-time dependent to determine which specialisations could possibly match at a given call-site.

This study evaluates different concepts from database query optimisation to determine whether they can have a beneficial performance effect on matching call-sites to a specialisation. This study will put focus on a survey of used patterns and call-sites as well as an implementation to test the actual merits of the proposed improvements in the ALIA4J framework.

1.3 Contents

The rest of this report is structured as follows: Chapter 2 describes the data model of the ALIA4J framework, which is the model that will be used for the examples. Chapter 3 explores the problem statement and sets the boundaries for this study. Chapter 4 will discuss different possible optimisations using techniques from database query optimisation.

Chapter 5 discusses a survey of real world applications and patterns to determine how to apply the theory from the first chapters to improve the performance of implementation described in Chapter 6.

Chapter 7 compares this research with other studies after which Chapter 8 evaluates the results and Chapter 9 discusses what follow-up research can be done.

Chapter 2

Data model of ALIA4J

In ALIA4J's high level design an attachment is associated with an action, schedule information which specifies relative ordering of actions, and a number of specialisations. A specialisation is associated with a pattern, contexts and a predicate. This is visualised in Figure 2.1.

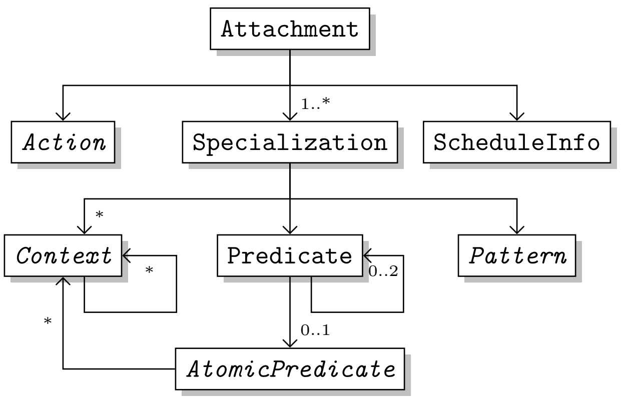

Figure 2.1: UML class diagram of an attachment in ALIA4J [6]

As mentioned earlier the main focus of this research lies on the patterns. The patterns are split into five categories which can be used to describe all the possible call-sites:

- Method pattern
- Constructor pattern
- Static initialiser pattern
- Field read pattern
- Field write pattern

These patterns have all been split up in sub-patterns that are finally used for matching, e.g. the constructor pattern only matches the (enclosing) class type, modifiers, parameters and exceptions sub-patterns. The sub-patterns are:

- **Modifiers pattern** Used to match the modifiers of a method, constructor or field that is accessed. Examples are `private`, `public`, but also `static`, `final`, and many more. It is possible to specify an inverse modifier, e.g. not `final`.
- **Type pattern** Used to match the return, parameter or exception type of methods and exceptions or the type of a field. The type is matched by either its fully qualified class/primitive name or a wild card, e.g. `java.*` for all classes in the `java` package.
- **Class type pattern** Used to match the enclosing type of accessed methods, constructors, static initialisers and field patterns. It works the same as the type pattern, although primitive and array types are not allowed.
- **Name pattern** Used to match the name of a method or field. Matching can be done with exact matches or by means of regular expressions.
- **Parameters pattern** Used to match the parameters of a method or constructor. Matching is done by means of matching type patterns for the parameters including a wild card type that matches any number of parameters.
- **Exceptions pattern** Used to match the exceptions that a method or constructor have declared to be thrown. Matching is done by means of matching type patterns for the exceptions.

All sub-patterns have the possibility of performing logical algebra on a number of sub-patterns, i.e. to negate the whole pattern, requiring matching multiple patterns or requiring one of a set of patterns to match.

2.1 Examples

To make the concepts of ALIA4J more concrete this section will show a number of examples, which will be used throughout this report to explain the different concepts important to the individual parts of this study.

Listing 2.1 shows a number of snippets from a hypothetical transport simulation game/model where cargo is moved by Vehicles, e.g. Trains and Busses, between stations. Their relationship is shown in Figure 2.2.

Table 2.1 shows a number of patterns to apply on the source code. In the table “*” means a wild card so everything matches that, “Vehicle+”

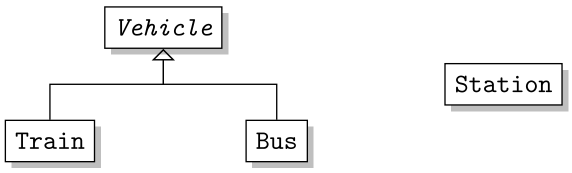

Figure 2.2: UML class diagram of the example application used in this study

```
interface Vehicle {
  Cargo GetTransportedCargo();
  int GetAmount();
  int GetCapacity();
  void ChangeAmount(int delta_amount)
}
static int Vehicle::AfterCrash(Train t, Bus b) {
  /* Busses implictly transport passengers */
  int casualties = b.GetAmount();
  if (t.GetTransportedCargo() == CARGO_PASSENGER)
    casualties += t.GetAmount();
  return casualties;
}
void Station::MoveCargoToVehicle(Vehicle v) {
  Cargo c = v.GetTransportedCargo();
  int waiting = this.waiting_cargo[c];
  int capacity = v.GetCapacity() - v.GetAmount();
  int to_move = min(waiting, capacity);
  v.ChangeAmount(to_move);
  this.waiting_cargo[c] = waiting - to_move
}
void Station::MoveCargoToStation(Vehicle v) {
  Cargo c = v.GetTransportedCargo();
  int to_move = v.GetAmount();
  v.ChangeAmount(-to_move);
  this.waiting_cargo[c] += to_move;
}
static void Vehicle::CollsionDetect() {
  ..
  int casualties = AfterCrash(t, b);
  ..
}
```

Listing 2.1: Code snippets from example application.

	Kind	Type	Class	Name	Parameter
1	call	*	Vehicle+	Get*	*
2	call	Cargo	*	Get*	-
3	call	void	Vehicle+	ChangeAmount	*
4	call	int	Vehicle	AfterCrash	Train, Bus
5	set	int[]	Station	waiting_cargo	*

Table 2.1: The patterns to match in the example application.

	Kind	Type	Class	Name	Parameter	Line
1	call	int	Bus	GetAmount	-	9
2	call	Cargo	Train	GetTransportedCargo	-	10
3	call	int	Train	GetAmount	-	11
4	call	Cargo	Vehicle	GetTransportedCargo	-	15
5	get	int[]	Station	waiting_cargo	-	16
6	call	int	Vehicle	GetCapacity	-	17
7	call	int	Vehicle	GetAmount	-	17
8	call	void	Vehicle	ChangeAmount	int	19
9	set	int[]	Station	waiting_cargo	-	20
10	call	Cargo	Vehicle	GetTransportedCargo	-	23
11	call	int	Vehicle	GetAmount	-	24
12	call	void	Vehicle	ChangeAmount	int	25
13	set	int[]	Station	waiting_cargo	-	26
14	get	int[]	Station	waiting_cargo	-	26
15	call	void	Vehicle	AfterCrash	Train, Bus	30

Table 2.2: The generic-function call-sites in the snippets from Listing 2.1.

		Generic-function call-sites														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Patterns	1	✓	✓	✓	✓		✓	✓			✓	✓				
	2				✓						✓					
	3								✓				✓			
	4															✓
	5									✓				✓		

Table 2.3: Patterns from Table 2.1 mapped to call-sites from Table 2.2.

means Vehicle or its sub classes, “-” means no parameters. For the “call” patterns the “Type” is the return type of the called function whereas it is the type of the (instance) variable for “set” patterns.

Table 2.2 lists the call-sites in the snippets. Some call-sites, e.g. call-sites 7 and 11, are the same when looking at only the pattern, however due to the a call-site being the place where a function is called, and not the (unique) function that is called, they are different. For brevity the modifiers information is omitted.

Finally Table 2.3 shows how the generic call-sites from Table 2.2 match to the patterns of Table 2.1.

Chapter 3

Problem statement

The aim of this study is to determine what methods from different fields of software engineering can be used to improve the performance of matching call-sites to patterns. The focus lies on the algorithms and data structures that are used for efficiently matching call-sites to patterns. This matching should take the structure of the executed application into account, i.e. it should use matching heuristics based on the executed application.

On one hand the characteristics, including theoretical complexities and trade-offs, of the algorithms and data structures have to be evaluated. On the other hand, the actual performance effects of the best algorithms have to be measured.

3.1 Sub problems

Even though the problem statement seems to imply that this research studies only the matching of call-sites to patterns the inverse is important for ALIA4J as well. With ALIA4J there are two distinct points in the 'execution' time of an application where the matching takes place. The first is when the specialisations, and thus patterns, are loaded and ALIA4J needs to find the call-sites that match the new patterns. The second is when new classes are loaded and ALIA4J needs to find the already loaded specialisations, and thus patterns, that match the call-sites in these classes.

To be more precise, "one call-site to many patterns"-look-ups and "one pattern to many call-sites"-look-ups are needed by ALIA4J. This study, however, focusses on the latter as that is the field where the benefits will likely be the biggest as the number of call-sites is likely magnitudes bigger than the number of patterns.

As such this study's first sub problem is: *what is the theoretically most efficient method of matching call-sites to patterns?*

To determine what the actual performance effects of the algorithms one has to take a look at the structure of applications and patterns, and apply that knowledge to the implementation. This results in the following two further sub problems: *what are the general characteristics of call-sites and generic-functions in real world applications?* and *what are the general characteristics of patterns in real world aspects?*.

The information gathered by solving these sub problems will then be used to implement improvements that can and will be measured.

3.2 Limitations

As said before the "one pattern to many call-sites"-look-ups are a secondary interest. The information from the survey and database query theory can be used give an idea on how to improve these look-ups, but these improvements are not implemented and as such their performance effects are not measured.

There might be cases where doing "call-sites of class to many patterns"-look-up, although conceptually the right one, might be slower than a "pattern to many call-sites"-look-up. For example one has two patterns that match a specific function of a specific class. In this case checking whether a call-site matches one of the patterns would mean going through all the call-sites of the class whereas when matching the patterns to the call-sites one can quickly determine that the (newly loaded) class does not have any call-sites that match the pattern. Determining when this might be the case is outside of the scope of this study.

An important observation that can be made is that, conceptually, the signature of call-sites is a pattern without wild cards. This means that their conceptual 'design' can be the same, i.e. when creating a database they could have the same database model. The following design is assumed:

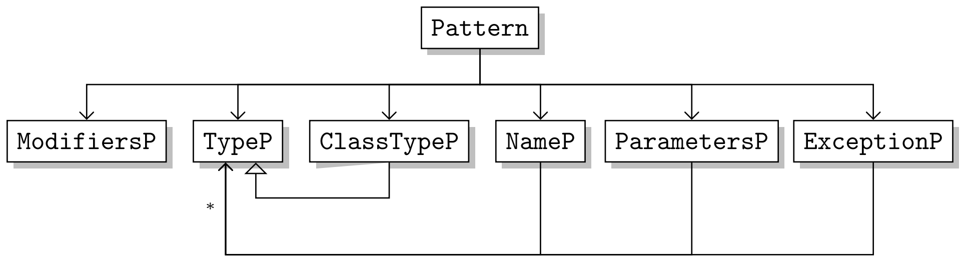

Figure 3.1: UML class diagram of a pattern in ALIA4J

Chapter 4

Database query optimisation

There are several database optimisation techniques that might be used to improve the performance of finding the projection of the specialisations. These methods will be explained and evaluated in this chapter. The reader is assumed to be familiar with basic SQL queries as described in e.g. [3, 10, 14, 15, 22].

4.1 Indices

The principle behind indices is that one performs an expensive operation while changing data, including addition and removal, while making searching less expensive. If applied properly this reduces the overall amount of time spent.

4.1.1 Theory

There are many different ways to implement indices, but the most common ones are based on a B+-tree and hashing [3]. The execution cost of the different methods for indexing, including no indexing, are given in Table 4.1. As can be seen, using hashing is fastest in all common cases except range look-ups, e.g. finding generic functions whose name starts with a given character.

Table 4.1 also lists the extra space requirement in “pointers” and, for the B+-trees, “keys”. The pointers are the abstraction of the links between the different nodes of B+-trees and hashes. The keys are needed for B+-trees because its design uses the search key in the nodes to facilitate the searching/sorting. For simplicity it is assumed that keys have the same size as pointers.

An unsorted array does not add any data for indexing, however an array is, at least in Java, an array of primitives or an array of references, i.e. pointers as references to objects are placed in the array. This means that even an

Action	Unindexed (Unsorted)	Indexed B+-tree	Hash
Initial sort	$O(0)$	$O(n\,log\,n)$	$O(n)$
Insertion	$O(1)$	$O(log\,n)$	$O(1)$
Deletion	$O(1)$	$O(log\,n)$	$O(1)$
Look-up	$O(n)$	$O(log\,n)$	$O(1)$
Look-up (range)	$O(n)$	$O(log\,n)$	$O(3n)$ (0.5 load)
Extra space	n	$5n$ (2-order)	$4n$ (0.5 load)

Table 4.1: Complexity of operations [3, 20]

unsorted simple array uses 1 pointer per row, resulting in n pointers being used. With other programming languages, e.g. C++, it might be possible to have an array of objects and then there would be no space overhead for the array but insertion gets more overhead.

For B+-trees it is important to be aware of the fact that each node has the same size which depends on the "fanout" of the tree, i.e. how many children a node has. This is called the order of a B+-tree, e.g. a 2-order B+-tree has two children. Nodes also have a pointer to the next node and a key for each of the children, resulting in a size of $2m + 1$ for a m-order B+-tree. Given the fact that each (n) rows of a table is a child of a node, there must be at least n/m "leaf" nodes and each m nodes have a parent. A 2-order B+-tree will be the deepest and as such be the upper limit of space usage. A 2-order B+-tree will create $n - 1$ nodes for n rows. Multiply this with the size of a node results in $(2m+1)n = 5n$. Higher order B+-trees will be less deep, but they will waste more space for unused places for children, however due to the lower depth space will be saved as less inter-node links are needed.

For example a 5-order B+-tree with 25 leafs will have 6 nodes, resulting in $(2m + 1)n = 11 \times 6 = 66$ pointers whereas, for a 2-order B+-tree, there would be 24 nodes, resulting in $(2m + 1)n = 5 \times 24 = 120$ pointers for an 2-order B+-tree. For a B+-tree with 26 leafs a 5-order B+-tree would have 8 nodes and thus 88 pointers compared to 125 pointers for a 2-order B+-tree. So, overall, higher order B+-trees are slightly more space efficient.

The space usage considerations for hash tables assume a closed addressing hash table, i.e. a hash table using a "collection" to resolve collisions instead of using the "next", determined by either $n + 1$ or another hash function, free element in the hash table. Furthermore a "load factor" of 0.5 (50%) for the hash table is assumed. This collection that the hash table links to can be a linked list, or a binary tree structure. For the space usage a linked list is assumed as that is the easiest and most used variant [2]. A linked list's node stores two pointers; one for the "next" node and one for the "leaf". As each row in the (database) table is linked by a node, there must be n nodes with 2 pointers and as such $2n$ pointers. The load factor of 0.5 implies that for every node there are two rows in the database table, thus there are $2n$ rows, i.e. pointers, in the hash table. These 2 pointers per node and $2n$ pointers in the hash table result in $4n$ pointers being used by the hashing. The $3n$ steps for "range look-ups" is directly related to the assumed load factor of 0.5; for the range look-ups the whole table has to be scanned, which is $2n$ entries long and the linked lists need to be traversed adding another n steps.

When deciding what indexing method to use for optimising the pattern to generic-function look-up one has to know how the index is going to be used and thus what the patterns match generally on. As such statistics about the used values must be gathered, e.g. a histogram of the values a variable can have can be made [10].

4.1.2 Application of theory

To determine what indexing method to use it is necessary to know what signature parts of generic-functions are going to be matched by the patterns and more importantly how often and how, i.e. as a direct look-up like `name = 'GetCapacity'` or a range look-up like `name = 'Get*'`.

When looking at the patterns from Table 2.1 and the call-sites from Table 2.2 one can count the occurrence of matches of each (name) pattern in the call-sites. This results in the data gathered in Table 4.2 and it can be seen as a histogram of the occurrences of a pattern. This data can then be used to determine what kind of index is best suited.

	Name	Occurrences	Percentage
1	Get*	8	53%
2	ChangeAmount	2	13%
3	AfterCrash	1	7%
4	waiting_cargo	4	27%

Table 4.2: The patterns from Table 2.1 and their occurrences in the call-sites from Table 2.2 out of 15 call-sites.

To determine which index is best suited, one first has to determine how often the different actions are performed. Given the patterns from Table 2.1 that means one range look-up and three exact look-ups. However, one must not forget the initial costs of sorting the table in the comparisons.

Assume n is the number of call-sites, m is the number of patterns and o_i is the number of call-sites that match a particular name pattern.

For an unsorted collection, i.e. an unindex collection, one has to loop the collection once per pattern regardless whether it is a range or exact lookup. This means $n \times m = 5 \times 15 = 75$ steps.

For a hash table one has the initial sort, two iterations of the whole table and three look-ups with o steps to iterate over the data in the bucket of the hash table. Assuming there are no hash collisions this means:

$$\begin{aligned}
\textit{initial cost} + \textit{range look-up cost} + \textit{look-up cost} &= \\
n + 2 \times 3n + \sum_{i=1}^{3}(1 + o_i) &= \\
15 + \big[6 \times 15\big] + \big[(1+2) + (1+1) + (1+4)\big] &= 15 + 90 + 10 \\
&= 115 \text{ steps}
\end{aligned}$$

The B+-tree sorted index has the same steps as a hash table, but there is no need for a full scan of the table but only o scans for range look-ups. For this example a 2-order B+-tree is assumed:

$$\begin{aligned}
\textit{initial cost} + \textit{range look-up cost} + \textit{look-up cost} &= \\
n\, log_2\, n + \sum_{i=4}^{5}(log_2\, n + o_i) + \sum_{i=1}^{3}(log_2\, n + o_i) &= \\
15 \times 4 + \big[(4+8) + (4+8)\big] + \big[(4+2) + (4+1) + (4+4)\big] &= 60 + 24 + 19 \\
&= 103 \text{ steps}
\end{aligned}$$

The example clearly shows that the weak spot of hash indices is range look-ups, whereas the initial sorting is the biggest cost factor for B+-tree indices. However the $log\, n$ look-up costs should not be ignored either as with many patterns and call-sites that number becomes significant too.

To determine which indexing method is the best in a given circumstance one has to look at all factors; the initial sorting, cost for simple look-ups and the cost for range look-ups. In the following formula n is the number of call-sites, p the number of patterns and l is the percentage of patterns that is a range look-up. The cost equations ignore the actual iteration of the data that matches.

$$\begin{aligned}
\textit{hash cost} &= \textit{initial cost} + \textit{look-up cost} + \textit{range look-up cost} \\
&= n + (1 - l)p + l3pn \\
&= (1 + l3p)\, n + (1 - l)p
\end{aligned} \tag{4.1}$$

Equation 4.1 shows the simple cost estimation for hash tables. This consists of the initial sorting cost, a single loop over the hash table ($O(n)$), the simple look-up cost which involves a single hash look-up ($O(1)$), and the range look-ups which involve scanning the whole hash table ($O(3n)$).

$$
\begin{aligned}
\textit{B+-tree cost} &= \textit{initial cost} + \textit{look-up cost} + \textit{range look-up cost} \\
&= n\, log_q\, n + (1-l)p\, log_q\, n + lp\, log_q\, n \\
&= n\, log_q\, n + p\, log_q\, n \\
&= (n+p)\, log_q\, n
\end{aligned}
\tag{4.2}
$$

Equation 4.2 shows the simple cost estimation for q-order B+-trees, a generalisation to the 2-order trees used in the previous examples. This consists of the initial sorting cost ($O(n\, log_q\, n)$), the simple look-ups ($O(log_q\, n)$) and range look-ups ($O(log_q\, n)$). As can be seen the simple look-ups and range look-ups have the same access time.

These two formulas can be used to get an initial idea of the costs, but they ignore how often a pattern matches. This means that when comparing the estimates directly the B+-tree indices' estimation will be too low with respect to the hash table cost. This due to the fact that the hash table's cost estimation includes iterating over the matches of the range look-ups whereas the B+-tree estimation does not.

$$
\begin{aligned}
\textit{B+-tree cost} &= \textit{hash cost} \\
(n+p) log_q n &= (1 + l3p)\, n + p(1-l) \\
l &= -\frac{-ln\, n - p\, ln\, n + n\, ln\, q + p\, ln\, q}{(3n-1)\, p\, ln\, q}
\end{aligned}
\tag{4.3}
$$

Equation 4.3 shows the break even point between between a hash table and q-order B+-tree. The result of the equation is the minimal percentage of range look-ups from where a q-order B+-tree is more efficient. The final equation was derived using Maple's equation solver.

Figure 4.1 shows the break even point for range look-ups (l) given a number of call-sites and patterns. As the break even point is in percentages all cases bigger than 100% mean that the 2-order B+-tree implementation would be more expensive than a hash table. Figure 4.1 shows that when applying one or two patterns the hash table is always the fastest solution. The more patterns are applied the lower the break even point becomes. What is notable is that with an increasing number of patterns all break even points eventually level out. This can be seen by the "500 call-sites" case that levels out around 0.53% but is also visible in the figure for the 5000 and 50 000 cases.

The number of call-sites are chosen to reflect a range of projects in size. For these numbers the assumption is made that there is one call-site per line of code. 500 lines of code represents very small, usually helper, applications, 50 thousand to 500 thousand lines of code represent stand alone applications

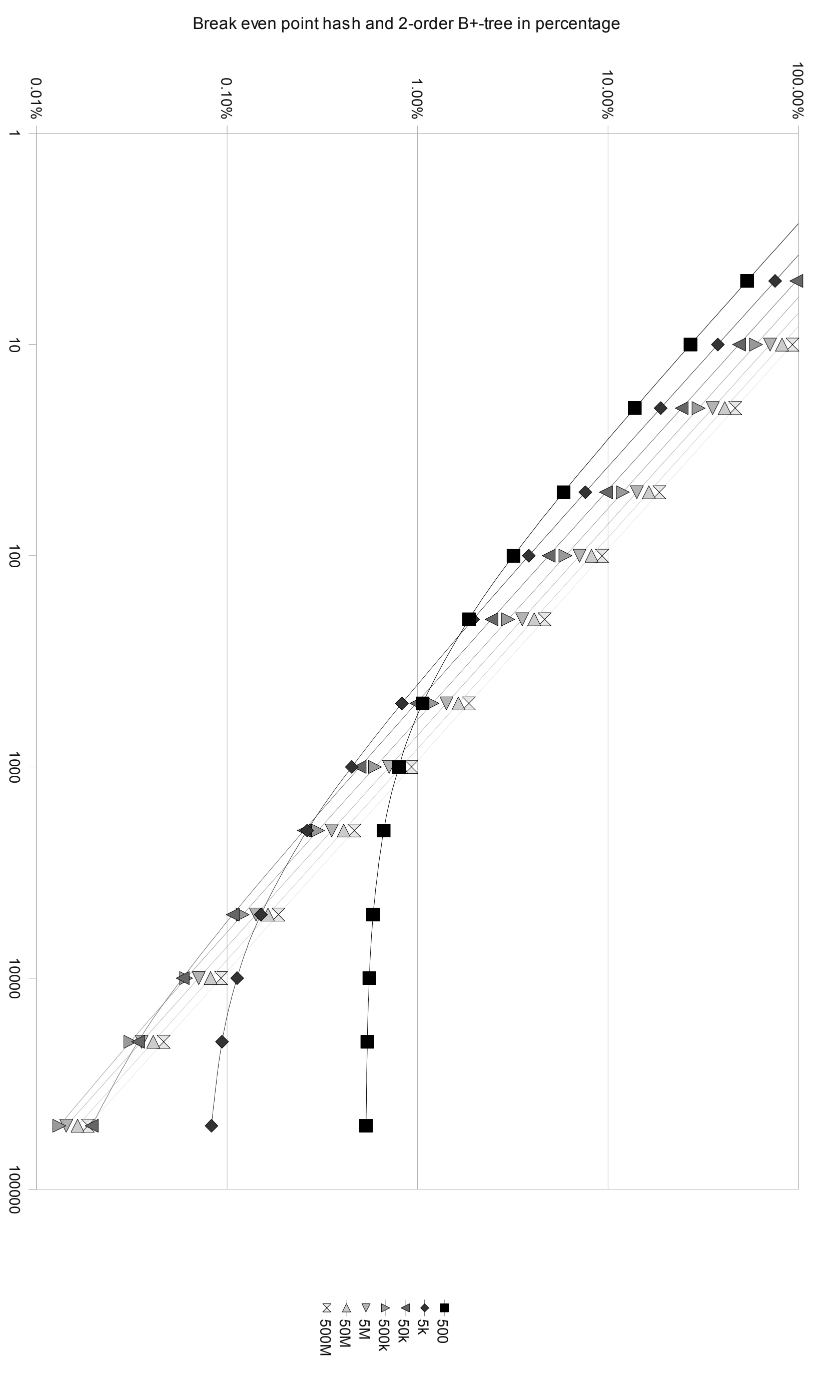

Figure 4.1: Break even point for range look-ups (l) for hash and 2-order B+-tree by number of call-sites and patterns. If for a given number of patterns the percentage of range look-ups is higher than at intersection with the line for the number of call-sites, then using a 2-order B+-tree will be more efficient.

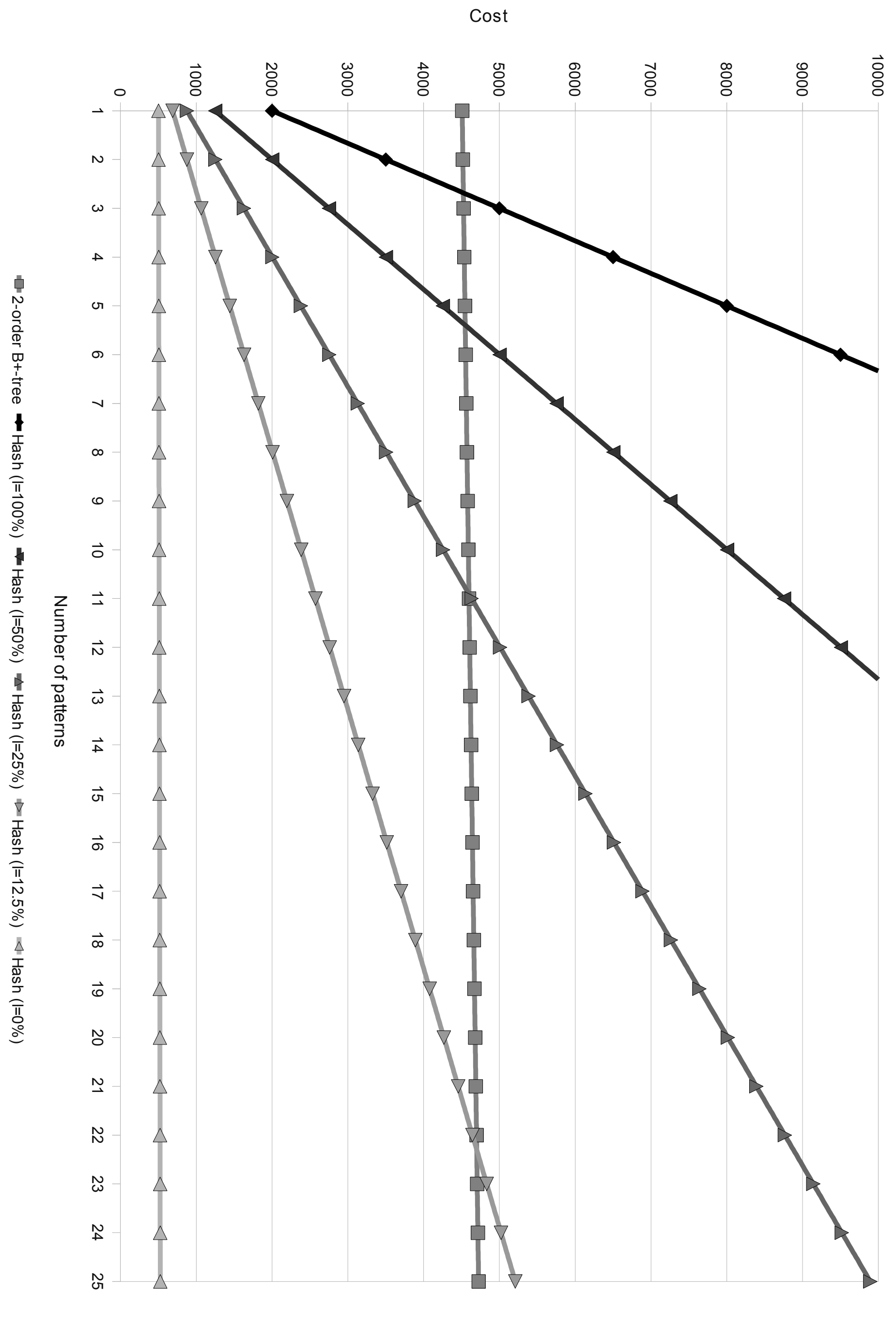

Figure 4.2: Comparison of cost for applying a number of patterns for the different indices and range look-ups (l) given 500 call-sites

such as Subversion, 5 million lines of code represent suites such as OpenOffice whereas 50 to 500 million lines of code represents main stream operating systems and distributions such as Windows and Debian [1].

Figure 4.2 shows the influence of the percentage of range look-ups in case of 500 call-sites. As the percentage of range look-ups does not matter for a B+-tree there is only one data set for B+-trees and a number of hash data sets for different range look-up percentages (l). What clearly shows is the bigger initial cost for setting up the B+-tree, roughly 9 ($log_2(500)$) times higher than for the hash table.

This figure can be compared with a small part of Figure 4.1. Both figures show that for one or two patterns hashes are faster regardless of the percentage of range look-ups and that the higher the number of patterns becomes, the lower the percentage of range look-ups becomes for which a hash table is faster than a B+-tree.

It can be concluded that different indexing mechanisms have different characteristics and that determining which mechanism to use must be a run-time decision. This means that the complexity considerations must be done at run-time, making it both more complex and flexible.

4.2 Selectivity

The principle behind selectivity is that the most efficient way to search a single table is determined. This is done by determining how complex, computational-wise, the different selections are. A selection is a filter that selects a subset of the whole table or the `WHERE` clause of SQL statements.

This section assumes a table "call_sites" that contains all call-sites, similar to Table 2.2, instead of the design described in Chapter 3.

4.2.1 Theory

The primary look-up is from pattern to call-sites and not the other way around. The call-sites are stored in a (single) table which is scanned, i.e. there is no need for optimising the aggregation of data from multiple tables. There are (database) normalisations that would make the storage of call-site signatures spread over multiple tables, but those would not be much more than a table of types which are then linked to the call-sites table by means of a foreign key which is usually associated with an index. For the reverse look-up, i.e. from call-site to patterns, there seems to be use for a join optimiser as there are (conceptually) several classes of sub-patterns that can be queried independently.

However, the pattern to call-sites look-up is no more than doing a selection on the call-sites table, like in Listing 4.1 which is a pattern that matches all `int` with `GetCapacity` as name. This is where statistics about the different (sub-)parts of call-sites become useful. These statistics can be

```
SELECT *
FROM call_sites
WHERE
    return_type = 'int'    AND
    name        = 'GetCapacity'
```

Listing 4.1: Example SQL query for pattern

used to estimate the "selectivity factor" (F) for each of the predicates of the `WHERE` clause of a SQL statement [22]. The F for a predicate is determined based on the estimated uniqueness of the value in the table.

This uniqueness can already be known because of the creation of histograms which were needed for indices. Otherwise, one has to use another heuristic. For such a heuristic a study of the actual call-sites in an application needs to be performed.

A simple heuristic would be to determine the average number of call-sites per generic-functions, but this assumes that they are evenly spread which might not be the case.

The following gives the selectivity factors for different kinds of predicates [22]. The selectivity factor (F) is in this case a number between 0 and 1. These selectivity factors are all independent best-guess values.

Exact pattern : $pred = \text{“}GetName''$

$F(pred) = \#Occurrences(\text{“}GetName'')/\#call\_sites$

$F(pred) = 1/\#generic\_functions$

As can be seen there are two approaches to calculate the selectivity a pattern. The first one is when one has information about how often the pattern occurs, i.e. a histogram, whereas the latter has no specific information.

Wild card pattern : $pred = \text{“}Get{*}''$

$F(pred) = \#Occurrences(\text{“}Get{*}'')/\#generic\_functions$

A wild card pattern is much trickier; a full wild card matches everything and thus has a selectivity of 1. However, for partial wild card patterns, like "Get*", one needs to know how often it occurs or make an educated guess based on the content of the non-wild card part of the pattern. The educated guess needs research into how function names are generally constructed and thus how unique the non-wild card part will roughly be. See Chapter 5 for more information.

Inverse predicate : $pred =!(pred')$

$F(pred) = 1 - F(pred')$

Having an inversions might give a low selectivity, but be aware that it might mean much more costly searches.

Conjunction of predicates : $pred = pred'$ *and* $pred''$:
$F(pred) = F(pred') \times F(pred'')$
Under the assumption that the predicates are independent, the chance of row in the set matching pred' to exist in the set matching pred" is as big as the chance of a random row existing in the set of matching pred"s. As such their selectivity factors can be multiplied.

Disjunction of predicates : $pred = pred'$ *or* $pred''$
$F(pred) = F(pred') + F(pred'') - F(pred') \times F(pred'')$
For the or predicates we can sum the factor of both predicates and then exclude the intersection of the rows matching both predicates, i.e. subtract the selectivity factor of the conjunction of pred' and pred".

The selectivity factor makes assumptions about even distribution and independency between predicates. With applications this will generally not be the case; some generic-functions will be referenced much more often than others. Therefore optimisation based on selectivity factors can actually be (way) worse than the best optimisation. However, getting the exact selectivity factors means doing the actual selection which is the thing we want to optimise, not execute more often.

4.2.2 Application of theory

Given Listing 4.1, what would be the selectivity? The first step is figuring out what the selectivity of the different parts of the predicate. First the `return_type = 'int'`; this is an exact pattern, however it occurs 5 times in the 15 call-sites of Table 2.2. This means that $F = 5/15 = 0.33$. When there is no histogram of the occurrences, it is impossible to tell it occurs 5 times and one has to estimate. The generic-functions have four different return-types, so given the mean that would result in a $F = 4/15 = 0.27$. The median amount of call-sites per return-type is 3.5, resulting in a $F = 3.5/15 = 0.23$.

The next step is determining `name = 'GetCapacity'`'s selectivity factor. It occurs one time in the 15 call-sites giving it a $F = 1/15 = 0.07$. There are 5 different names, giving a $F = 3/15 = 0.2$ (the mean) which overestimates the real number by a factor 3.

The final step is determining the `AND`'s selectivity factor. This is simply $F = 0.33 \times 0.07 = 0.02$, however with the estimations this would become $F = 0.27 \times 0.2 = 0.06$ which is closer to the actual selectivity of $F = 1/15 = 0.07$ than the version without estimates because the `return_type = 'int'` does not add any restrictions once `name = 'GetCapacity'` is evaluated.

When looking at wild card patterns, e.g. `name = 'Get*'`, the selectivity factor calculation gets trickier. For example, making a histogram of the first three letters would give a data set like in Table 4.3. This shows that

Name	Occurrences
Get...	8
wai...	4
Cha...	2
Aft...	1

Table 4.3: Histogram of first three letters of name from the generic-function call-sites in the snippets from Listing 2.1.

the pattern matches 8 in 15, or $F = 0.53$, of the generic-function call-sites, however the mean and median would be respectively 3.75 and 3.

This means that there is possibly a need for histograms, but the real question is *how much better do histograms make the estimations?* and *how much does making and querying histograms cost?*. This requires extra studying, but falls outside the scope of this research as more empirical information is needed.

4.3 Query reordering

The principle behind query reordering is that an order of the individual parts of the query is determined to yield the minimum amount of processing required. This is done by determining how complex, computational wise, the individual parts are as well as the size of the resulting search space. The reordering results in a "search-plan".

This section assumes the design described in Chapter 3, however the theoretical explanation uses a different example as the design from Chapter 3 makes the example unnecessarily complex.

4.3.1 Theory

The major problem with determining the best order is that there are many different orders and that the computational complexity and resulting search space of the individual parts (n), e.g. sub-patterns, depends on the already executed parts, basically meaning that there are in worst case $n!$ different orderings [3, 15].

However, the search space for the best search-plan can be drastically reduced. For example when getting information from multiple tables by making sure that the intermediate results stay relatively small. For example imagine the SQL query in Listing 4.2 and the data as given in Table 4.4. In this example `a.ssn` is a unique key, i.e. the values are unique and there is a 1-on-1 relation between `a.id` and `b.id`.

```
SELECT b.name
FROM a JOIN b ON a.id = b.id
WHERE a.ssn = '1234' AND b.gender = 'male'
```

Listing 4.2: Initial SQL query

a.ID	a.SSN
1	2341
2	1234
3	3412
4	4123

b.ID	b.Name	b.Gender
1	Jane	Female
2	John	Male
3	Pete	Male
4	Penny	Female

Table 4.4: Example tables for Listing 4.2.

Table 4.5 shows what happens when first the Cartesian product of the two tables is calculated before continuing with filtering that data. In total the table would be 16 records long, the product of the size of both table a and table b.

a.ID	a.SSN	b.ID	b.Name	b.Gender
1	2341	1	Jane	Female
1	2341	2	John	Male
1	2341	3	Pete	Male
1	2341	4	Penny	Female
2	1234	1	Jane	Female
..	..	..	..	..

Table 4.5: Example of the Cartesian product of Table 4.4.

When first getting the record from `a` with `ssn = '1234'` as intermediate result the Cartesian product would be significantly smaller as there can at most one record in `a` with that `ssn` due to the unique key. The resulting Cartesian product can be seen in Table 4.6. When taking indices into account one can optimise the scanning of the tables away, but this is explained in Section 4.1. In these examples it is assumed that half of the people is male.

The search space for the search-plan is reduced in three steps. First the 'query tree', i.e. the tree representation of the initial query, is simplified into a "master plan". The master plan groups consecutive commutative and associative operators of the same kind. For example `(a JOIN b) JOIN c` and `a JOIN (b JOIN c)` are equivalent and can be stored as one `JOIN` operation in the master plan.

The second step generates "logical query execution plans" from the mas-

a.ID	a.SSN	b.ID	b.Name	b.Gender
2	1234	1	Jane	Female
2	1234	2	John	Male
2	1234	3	Pete	Male
2	1234	4	Penny	Female

Table 4.6: Example of the Cartesian product of Table 4.4 using a reduced intermediate result for `a`.

ter plan. At this stage the "selections", e.g. `a.ssn = '1234'` are pushed down in the 'query tree'. For example Listing 4.2 would be transformed into Listing 4.3. The rationale behind this step is that reducing intermediate results speeds up execution, although there are cases where not fully pushing down the selections results in a faster execution. For example when joining two tables an index on the "join selection" means faster look-up of records in the joined table. What would happen to the example tables can be seen in Table 4.7.

```
SELECT d.name
FROM (
    SELECT a.id
    FROM a
    WHERE a.ssn = '1234'
  ) AS c JOIN (
    SELECT b.id, b.name
    FROM b
    WHERE b.gender = 'male'
  ) AS d ON c.id = d.id
```

Listing 4.3: SQL query with fully pushed selections

a.ID	a.SSN
2	1234

b.ID	b.Name	b.Gender
2	John	Male
3	Pete	Male

Table 4.7: Example tables for pushed down selections.

Doing the (pushed) selection before the join might remove this index and just make the actual joining slower. An example of this is Listing 4.3 which would be slower to execute than Listing 4.4 when there is an index on `b.id` because the intermediate results `c` and `d` do not have an index. In the example the reduced `a` would be one record and the reduced `b` would contain

2 records, or half of `b`. On the join the whole reduced `b` would need to be iterated, but what is worse is that there is no index on `b.gender` causing, in effect, the whole of `b` to be iterated to get the intermediate set.

With the partial push, visualised in Listing 4.4, the join would use the index on `b.id` and do a join with only one record in the table followed by the `b.gender = 'male'` selection. Because of this several plans are generated with different levels of pushed down selections.

```
SELECT b.name
FROM (
    SELECT a.id
    FROM a
    WHERE a.ssn = '1234'
  ) AS c JOIN b ON c.id = b.id
  WHERE b.gender = 'male'
```

Listing 4.4: SQL query with partially pushed selections

The third step simplifies the “logical query execution plans” into ‘left-deep query trees’ which are the final output of the reordering. A left-deep query tree is a binary tree of joins where the right node is always a leaf node containing a single operation to do. Operations in this case would be selections and joins. The advantage of such a tree is that implementing “pipe-lining” is possible. With pipe-lining the results of a join are piped into the next join which means that there is no need to store intermediate results. The most important benefit is that this reduces the join search space (joining n tables) from $O(n!)$ to $O(2^n)$ [14].

To create an optimal left-deep tree all permutations of the joins must be considered. As this number of permutation grows exponentially some heuristics must be used to cut corners to create a good, although not necessarily the best, query plan quickly. One of the methods that can be used is dynamic programming as described by [22]. This method works as follows: first several “1-relation plans” are made, at least one per tuple of to-be-joined tables although there can be more depending on existing indices. These plans are then evaluated, i.e. a value is given to the plan based on their estimated execution cost. Then the best of these plans are chosen to be joined with another table, i.e. a tuple for each chosen 1-relation plan with a table that is not in the plan. This results in several “2-relation plans” that are evaluated and the cycle continues until the n-relation plan has expanded to contain all tables. This n-relation plan is the result, i.e. search-plan, of the query optimiser [14].

4.4 Summary

With query reordering all of the theory from this Chapter comes together. The existence and type of indices have great influence on the cost of execution, whereas the selectivity factor tells what might be the better way to reduce the size of the intermediate data and then the query reordering helps us to decide how to order the query to quickly determine a good plan for executing the query.

The following example will show how all the theory can be applied. In the example pattern 2 from Table 2.1 will be matched against the call-sites. The first step is determining the selectivity of the different components of the pattern that needs to be matched. The number of occurrences and their selectivity can be seen in Table 4.8.

Sub-pattern	To match	Occurrences	Selectivity	Index	Look-up cost	Cost estimate
Kind	call	11	0.73	Hash	$O(1)$	12
Modifiers	public	11	0.73	B+-tree	$O(log\, n)$	15
Type	Cargo	3	0.20	Hash	$O(1)$	4
Class	*	15	1.00	Hash	$O(3n)$	35
Name	Get*	8	0.53	B+-tree	$O(log\, n)$	12
Parameters	-	12	0.80	None	$O(n)$	15

Table 4.8: The selectivity information for pattern 2 and the used index per sub-pattern.

The table clearly shows that the class sub-pattern selects everything and as such is not useful for the result and that selecting on Cargo is the most selective. However, the selectivity is not what the focus should be on, but the expected cost which is shown in the last column. This cost is the sum of the look-up cost and the cost for looping through linked lists and/or range look-up, i.e. the number of occurrences. The expected costs are important because doing a slightly less selective range look-up on a B+-tree indexed variable is faster than doing a range look-up on a hash table indexed variable. In the former case one would only need to scan the resulting, usually significantly smaller, sub set on the most selective variable which means doing less comparisons in total.

As query reordering is in essence a divide-and-conquer strategy, one can see the steps to determining the best order as “rounds” in a match. Each time the best solutions, i.e. order of a sub set of sub-patterns, go for a next round that adds another sub-pattern until all sub-patterns of the pattern are in the order.

When applying the methods from query reordering, the best or most promising options are chosen. In this case the type sub-pattern is most selective, i.e. should be evaluated first. The kind pattern and name pattern are equally promising so both go to the next round.

In this second round a design difference between the modeling of the data in databases and Java will be encountered. In databases each table is linked to another by means of an identifier, which is usually indexed. However, in Java connections are made by means of references. This means that the joins in Java are essentially free, i.e. one just needs to loop over the intermediate result set and query the sub elements directly. This means that reducing the result set becomes most important, i.e. using the selectivity factors.

First pattern	Second pattern	Estimated. sel.	Actual. sel	Cost
Type	Base	0.15	0.20	6
Type	Modifiers	0.15	0.20	6
Type	Name	0.11	0.20	6
Type	Parameters	0.16	0.20	6
Base	Modifiers	0.53	0.73	20
Base	Type	0.15	0.20	14
Base	Name	0.39	0.53	18
Base	Parameters	0.58	0.53	21
Name	Base	0.39	0.53	18
Name	Modifiers	0.39	0.53	18
Name	Type	0.11	0.20	14
Name	Parameters	0.42	0.53	18

Table 4.9: The selectivity information for pattern 2 and the used index per sub-pattern.

The next step is determining the cost and size of the result set. This can be done by using the selectivity factors from Table 4.8 and applying that to three call-sites selected in the first step. This results in Table 4.9. Note that the class pattern is not in the table. This is because it is not selective at all and therefore only makes the number of possible orderings bigger when it is already known that it does not reduce the search set. The table shows the estimated selectivity, using an histogram telling how often a pattern occurs, which is the information the query optimiser has access to. The table also shows the actual selectivity, i.e. how many call-sites actually match the two sub-patterns, to show how far off the estimate is. The actual selectivity will not be known at run-time as mentioned before. The last column the table shows the estimated cost which is the cost from the previous round, i.e. how many comparisons were done to get the first intermediate result, plus the size of the expected result set, i.e. the size of the second intermediate result.

It has to be noted that after the first round the histogram data becomes unreliable because there are usually correlations between sub-patterns. For example the name pattern “Get*” and its return type non-void. Thus when the “Get*” pattern is matched matching the return type for being non-void is a waste of time. However, making the query optimizer aware of that fact in a generic manner is far from trivial. Especially as it is unknown what dependencies between sub-patterns or parts of a signature exist.

4.5 Conclusion

Three major facets from the database field have been studied: indexing data sets, determining selectivity of a sub-pattern, i.e. how many results will matching a sub-pattern give, and query reordering, i.e. ordering the sub-pattern matches in the most effective way. All these facets can be used to improve the performance of matching call-sites to patterns and vice versa. However, the “join” optimisation of the latter method will not be needed.

Determining selectivity of a sub-pattern can be used to quickly reduce the result set and thus reduce the number of evaluations, however it still requires a full scan of the table when there is no index. As such selectivity measurements can only reduce the number of sub-pattern matches to determine whether a particular pattern matches; it cannot skip patterns completely when it is known they cannot match. However, as it is possible to determine what index is best at run-time, the most selective sub-pattern is a good, if not the best, candidate to add the index to.

Having indices on certain sub-patterns can be used to get a small sub set of the initial data set by partly removing the need to scan the data set. However, there are different indexing methods that have their own characteristics. A (sorted) B+-tree can quickly scan a range of the data set due to logarithmic access, however the costs for creating and maintaining a B+-tree are high. A hash table has an extremely fast look-up for a specific value, but for a range look-up a full scan of the table is needed. As a hash table is bigger than an the actual data set, such range look-ups are more costly than in an unsorted unindexed array.

As such the index to choose highly depends on the actual structure of the application and the patterns that have to be matched. When the most selective sub-pattern has lots of range look-ups a B+-tree will be the fastest and a hash table will be slower than a normal table, but when the most selective sub-pattern has only exact look-ups a hash table will outperform a B+-tree, although the B+-tree's look-up will outperform the normal table.

The high initial cost, logarithmic to the number of items in the data set, of B+-trees are a major factor. It can be better to choose a slightly less selective sub-pattern that has mostly exact look-ups and thus can use hash look-ups. This will decrease the performance of the look-ups slightly, but an application will most likely not keep loading specialisations, thus patterns, or classes, thus call-sites, infinitely and therefore the extra look-up costs can dwarf the cost for building the B+-tree.

As such the decision of what indexing method to use exactly depends highly on the selectivity data which is extracted from the executed application.

Chapter 5

Call-site and pattern survey

In the previous chapter we have presented indices and search-plan optimization as possible optimizations for pattern evaluations. In order to apply these mechanisms some knowledge about the structure of the data on which queries are performed is needed. For instance, it must be known for which sections of signatures the effort of keeping an index may pay off; for search-plan optimization, heuristics are needed to estimate the selectivity of certain sub-patterns.

In this chapter a survey of real-world applications and aspects is done to be able to answer what the best strategy for optimizing pattern to call-site look-up is. To answer this question the following sub problems have to be addressed first: *what are the general characteristics of call-sites and generic-functions in real world applications?* and *what are the general characteristics of patterns in real world aspects?*.

These sub problems are solved in respectively Section 5.1 and Section 5.2. Finally Section 5.3 formulates optimization strategies based on the findings of both sections.

5.1 Call-site characteristics

For determining what performance optimizations are suitable for signature pattern matching it is necessary to analyse real-world applications. From these real-world applications the call-sites need to be extracted after which one can determine the data that is significant for the optimizations. Significant in this context are selective parts of a signature. For example, when all functions return the same type the selectivity of that return type is low. For any other type, however, the selectivity is very high as it would never select anything.

5.1.1 Methodology

To acquire the call-site information a small tool, called Extract, has been written to extract the call-site information from Java class files. Extract uses ASM [8], a Java byte code framework, to read the Java class files and parts of the SiRIn core, a reference implementation of ALIA4J, to determine what generic-function a call-site belongs to.

The input to Extract is a list of Java class files or Java archives with Java class files of a particular application. All Java class files and all classes they reference are analysed recursively. Examples of referenced classes are the super class, classes used as return or parameter type, classes of instance variables and classes used in the implementation of a method. The final result of Extract is a list with all generic-functions and the number of times they were referenced.

The recursive behaviour of the analysis means that a large part of the Java API's implementation will be part of the output. However, the implementation of the Java API should be seen as a black box and one would generally not want to add aspects that modify the API's implementation. AspectJ, e.g. thus applies aspects only to the files that it compiles itself; this excludes the class files of the Java API. These internal calls are therefore ignored by this survey. Nevertheless, calls to the Java API from the application are of importance. After the extraction the data is analysed using common data mining techniques to find correlations and significant information.

The applications that have been analysed in this context are the following four:

- **ANTLR** A tool to construct parsers, compilers and translators.
- **FreeCol** A turn-based strategy game.
- **LIAM** A major part of the implementation of ALIA4J.
- **TightVNC** An application to remotely take over a desktop.

These applications are varied in nature in an attempt to get a general view of the call-site characteristics instead of only getting a view of non-graphical applications that do not use a network connection, as is commonly the case for benchmark suites used in performance evaluation [5].

Larger applications such as Eclipse, an integrated development environment, and OpenOffice, an office suite, have been considered. But these applications have a complex system of plug-ins and dependencies which makes it hard for Extract to cover all referenced classes. Tools like TamiFlex [7] can alleviate this problem, but were not yet available when the survey was conducted.

5.1.2 Acquired information

Statistics The analysed applications have 2 432 classes containing a total of 28 065 generic-functions and 150 432 call-sites. The following sections will discuss the most salient features separately for each generic-function kind; only the breakdown on class names is presented in a single section as it is almost the same in all cases.

Storage Per generic-function kind there is also a discussion on the best storage technique for quickly retrieving call-sites per sub-pattern kind on the basis of the information gathered. The techniques considered here are "bucket arrays", like hash tables, and "sorted collections", like B+-trees.

Section 5.1.8 summarises with a comparison of the storage techniques over the different sub-patterns looking at differences and similarities in the use of patterns between the different sub-patterns.

5.1.3 Class names

Statistics The combined test input consists of 2 432 different classes that are spread over 147 packages, on average placing roughly 16.5 classes in each package.

Depth	Amount	Percentage
0	20	0.8%
1	135	5.6%
2	414	17.0%
3	479	19.7%
4	333	13.7%
5	525	21.6%
6	526	21.6%

Table 5.1: Package depth for class names

Table 5.1 shows the classes' package depth, i.e. the number of super packages a class has before reaching the "unnamed" default package. The 20 classes with depth 0, i.e. the ones placed in the unnamed package, and 135 classes with depth 1 do not comply with the standard practice of using a reverse DNS name. The classes at depth 2 are primarily from the Java API, whereas classes at depths of 4 and more are almost exclusively used by applications.

Due to the practice of using a reverse DNS name the first few levels of package naming do not help in quickly reducing the search space. The way package names are generally constructed, however, makes it reasonably easy to determine a start and end point in a sorted set and thus reduce the amount of checks: Assume the classes of the Java API are lexically sorted by their fully qualified name and are put in a sorted set. Then all classes that match the `java.lang..*` pattern can be found by calculating a subset: the starting point of the range is constructed by removing `.*` from the pattern, the end point is constructed by replacing `..*` with `/` (U+002F), whose codepoint the Unicode character encoding places immediately after the `.` (U+002E). Calculating the subset only requires two $O(log\ n)$ comparisons. When a pattern starts with `.*` then the whole sorted set is returned.

Storage When looking at the storage techniques the first option is using a sorted collection. This has the benefit of ordering all classes lexically by name; thus the aforementioned technique can be used. However, storing all classes into buckets by package can be used to efficiently look up all classes in a given package, but due to the use of sub-packages one has to determine how to find all classes that are in a particular package or sub-package. Either the buckets also contain references to sub-packages which are then recursively searched or each class gets inserted into its package and all ancestor packages. A major drawback of the latter technique is the fact that the number of times the class is referenced is equivalent to the package depth plus one. The former technique resembles the behaviour of a sorted collection.

5.1.4 Static initialisers

Statistics A total of 571 static initialiser were found. The static initialisers do not have a call-site, i.e. they are called implicitly, and the name, modifiers, return type, parameters, and exceptions are the same for every static initialiser. As such the only way to distinguish between static initialisers is their containing class which means there is at most one per class.

Furthermore none of the static initialisers are of the Java API and as such there are no class names starting with `java`.

Storage Static initializers only have one sub-pattern: the declaring-class pattern. Thus, the same considerations as in Section 5.1.3 apply.

5.1.5 Constructors

Statistics A total of 3 034 constructors were found; on average about 1.25 constructors have been found per class.

Table 5.2 shows how often a particular modifier is used for a constructor. The first set of four modifiers, the ones that govern access to the constructor, are mutually exclusive and cover all constructors. As a result the total of the first four modifiers is always exactly 100%. The other modifiers are optional and multiple of them can be used per constructor.

Modifier	Amount	Percentage
package visible	759	25.0%
`public`	1 971	65.0%
`private`	151	5.0%
`protected`	153	5.0%
`transient`	14	0.5%
annotation	53	1.7%
deprecated	7	0.2%

Table 5.2: Modifier usage for constructors

# Parameters	Amount	Percentage
0	669	22.1%
1	1 171	38.6%
2	720	23.7%
3	241	7.9%
4	144	4.7%
5+	89	2.9%

Table 5.3: Parameter usage for constructors

Table 5.3 relates the number of parameters to the amount of constructors having such a parameter count. The 3 034 constructors declare a total of 4 420 parameters, giving about 1.5 parameters per constructor. Of all these parameters 588 are of type `java.lang.String`, 564 are `int` and 158 are `boolean`, from a total of 518 parameters types.

Of the 3 034 constructors 2 855 do not throw an exception, leaving 179 that do throw at least one. A total of 188 exceptions is declared; a few constructors declare more than one exception. The exceptions most frequently thrown are `javax.xml.stream.XMLSteamException` (84 times) and `java.io.IOException` (40 times), which are both part of the Java API.

A total of 14 526 call-sites of constructors were found. Of these 64% call a constructor from the Java API; only 36% of the calls are for creating application specific classes. Classes from the `java.lang` package are constructed 39% of the time, with the `StringBuffer/StringBuilder` being responsible for over 50% of the calls. Most of these string building calls are generated automatically by the compiler when it encounters the concatenation of strings using the `+` operator. This means that in theory these call-sites should not be affected by the aspects, however it is impossible to distinguish between there implicitly generated calls and explicit calls.

Storage The name of a constructor can be best stored in a sorted collection as it is similar to the class names; it has many unique names as well.

For the modifiers a bucket array is best as there are only a few valid different buckets to consider. However, there are sometimes multiple buckets a constructor would match with. In that case they have to be put in all, but this is not a big problem as a relatively small amount would be placed in multiple buckets. It has to be considered whether physically storing the `public` modifiers bucket is needed at all, as it matches the vast majority of constructors.

We consider three techniques to store parameters: First, the number of parameters can be used to determine the bucket. Second, the first parameter type can be used. (If a function has no parameters, `void` is used as first parameter.) Third, the concatenation of all parameter-type names can be stored lexically sorted.

The benefit of the first technique is that searching for a particular amount of parameters is extremely efficient, whereas the second is efficient in finding constructors that have a particular type as first parameter.

The third technique is well suited for finding constructors that start with particular parameters, but finding constructors with a given length requires looking through the whole collection.

Finally the exceptions can best be stored in a bucket as well. Here, each declared exception is put into a bucket. Given the low amount of actually declared exceptions and the low amount of constructors with more than one declared exception this does not impact storage much. The constructors that do not throw an exception are not stored specially.

5.1.6 Field reads and writes

Statistics In total, 3 884 fields have been found, yielding 1.6 fields per class. Of these fields 3 874 are read and 3 863 are written. Some fields, primarily with a `protected` access modifier, are not written to or not read. As these fields are all part of the Java API this can be explained by the application extending such a framework class and only then writing or reading said fields.

An example of this can be seen with `javax.swing.JComponent.ui`. This field of `JComponent` has a `protected` access modifier and a method to set it. It does not have a function to read the value, though. Thus if a custom, non Java API, sub lass of `JComponent` requires information from that field it needs to access it directly. As a result there is a field read on that field, but the application does not (directly) write to it, so there is no field write.

All `static final` reads are ignored because the majority of call-sites can be silently destroyed by the compiler optimising them away.

In total, there are 2 697 different field names. The most frequently used field name `logger` is used 38 times which means that the field names themselves are all quite selective. The majority, 95.9%, of the field names start with a lower case letter. Of the 3.2% of field names that start with an upper case letter 55.6% are `static`. About 0.9% of all names start with either a `$` (U+0024) or a `_`(U+005F).

The field names have an average length of 9.7 characters. Around 99% of these characters are letters. Looking at the first three characters does not show any discernible patterns; the most prefix, `can`, is used in less than 1.75% of the field names. This means that the field names diverge relatively fast in all cases.

The length and divergence of the field names can be used to estimate the time required for one comparison of two strings. If, e.g., at most 1.75% of the field names start with the same three letters one knows that in three comparisons there is at most a 1.75% chance that further characters have to be examined.

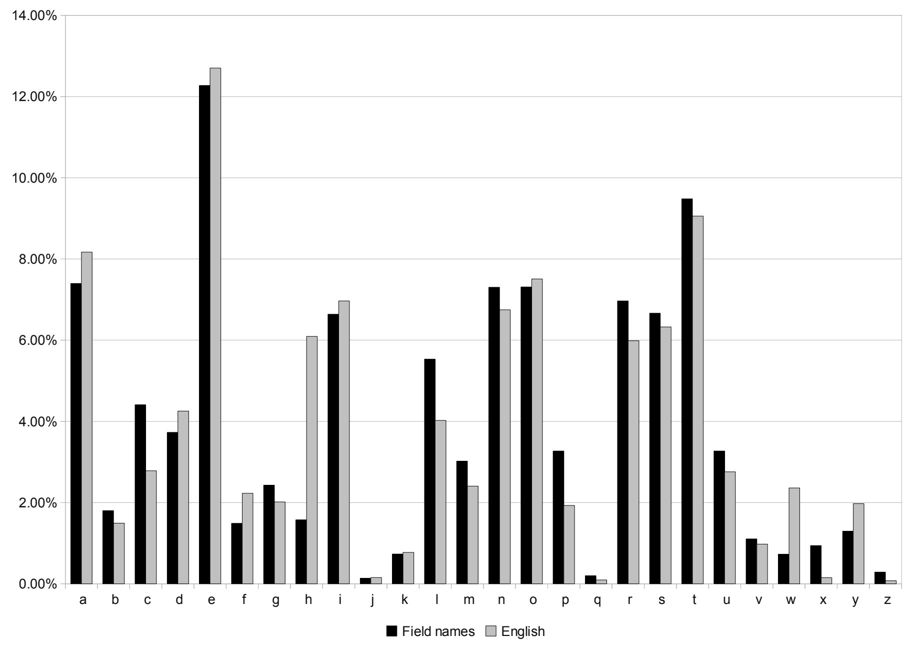

Figure 5.1: Letter frequencies of field names compared to English

Figure 5.1 shows the letter frequencies in the field names which are fairly similar to the letter frequencies in English, although there are a few letters whose frequency differs significantly.

Table 5.4 shows how often a particular modifier is used for a field. This table is almost identical to Table 5.2 with the exception that the percentages for `public` and `private` modifiers are switched; fields are about ten times

Modifier	Amount	Percentage
package visible	862	22.1%
`public`	236	6.1%
`private`	2 189	56.4%
`protected`	597	15.4%
`static`	395	10.2%
`volatile`	8	0.2%
`transient`	50	1.3%
annotation	22	0.6%
deprecated	9	0.2%

Table 5.4: Modifier usage for fields

more often `private` than constructors are. This situation is reversed for for `public`.

Modifier	Amount	Percentage
package visible	40	10.1%
`public`	52	13.2%
`private`	291	73.7%
`protected`	12	3.0%
`volatile`	3	0.8%
annotation	22	5.6%

Table 5.5: Modifier usage for `static` fields

Table 5.5 shows the modifiers, but only for the `static` fields. What stands out is that all package visible fields are also `static` and that there are much fewer `protected` fields and fields without modifiers than for non-`static` fields.

For field writes the (hypothetical) return type is always `void` whereas for field reads there is no parameter type. As the return type of a field read is the same as the parameter type of a field write for the same field, these types are the same. Of these types 53% is either a primitive or comes from the `java.lang` package, with `int`, `boolean` and `java.lang.String` taking respectively 22%, 12% and 10% of the total.

Field reads and writes cannot throw checked exceptions.

A total of 27 979 field reads and 9 752 field writes, respectively 7.2 and 2.5 per field. Roughly 16.4% are reads from and 12.4% are writes to classes from the Java API.

Storage For storing the name a sorted collection makes finding a particular name or a range easy. Using a bucket array is possible, but either the whole name has to be hashed or only the first character is taken into account. In the former case doing a name range look-up becomes expensive, whereas in the latter case one still has to go through a long list of items after the first bucket. If one were to chain the buckets per character one would in effect be building sorted collection.

The modifiers can only be stored in a bucket array due to the limited amount of options. It can be considered to not create a physical bucket for the `private` fields as they match more than half of the fields and as such are not very selective.

The type of the field can best be stored in a sorted collection. There are quite a number of types, although it is very conceivable that a set of types from one package is considered. In that case having a collection sorted on the type name would make getting those ranges work in the same way as for class names. However, it is conceivable to store the data in a bucket array if there can be, e.g. due to a less powerful point cut languages, are no range look-ups on the type of a field.

5.1.7 Methods

Statistics A total of 17 294 methods were found, which means about 7.1 methods have been found per class. There are 6 812 different names for the methods, yielding 2.5 methods with the same name.

The average length of a method name is 12.3 characters; almost 3 characters more than field names. Around 99% of the characters are a letter.

Figure 5.2 shows the letter frequency in the method names which is less similar to the letter frequency in English. Letters "c", "e", and "t" are used significantly more often whereas the usage of letters such as "h", "w", and "y" have dropped by up to 75%.

Name	Amount	Percentage
get	4 596	25.6%
set	1 346	7.8%
cre	541	3.1%
acc	499	2.9%
add	413	2.4%

Table 5.6: Frequency of first three letters in method name

Contrary to the insignificance of the first three letters of field names the first three letters of method names are significant as can be seen in Table 5.6. What has to be noted is that 417 of the 499 methods that start with "acc"

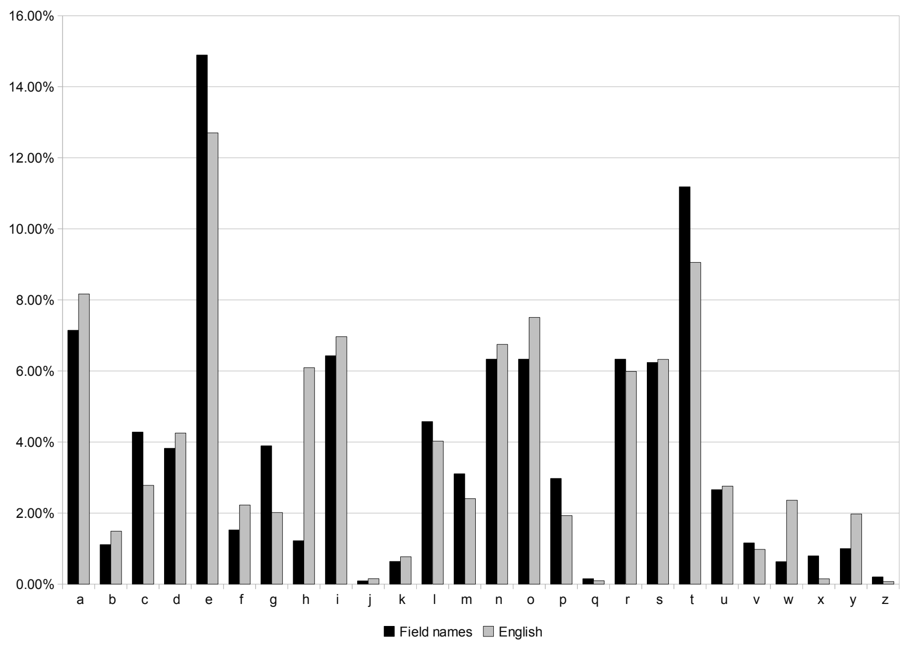

Figure 5.2: Letter frequencies of method names against English

are `access$n` methods created by the compiler for inner classes that try to access outer classes.

Table 5.7 shows how often a particular modifier is used for a method. As multiple modifiers can be used at the same time the total is more than 100%. What immediately catches one's eye is that almost 80% of the methods are `public` and that furthermore a large portion is `static`.

Table 5.8 shows the modifiers, but only for `static` methods. What stands out is the majority of package visible methods are also `static` and that there are almost no `protected static` methods. Note that `abstract static` methods are impossible as one cannot override static methods.

Of 37.7% of the methods the return type is `void`. About 36.4% is a primitive type or comes from the `java.lang` package, with `int`, `boolean` and `java.lang.String` respectively taking 10.4%, 10.1%" and 7.7%. When ignoring the `void` return type, these return types are similar to the type of fields.

When considering the methods starting with `get` and `set` more closely, one finds that there are 6 (0.01%) of the former methods have a return type of `void` and 49 (3.6%) of the latter have a return type different from `void`. This means that when encountering a pattern matching methods that start with `get` or `set` in an aspect one can be fairly sure the method respectively returns or does not return something. Consequently, the return-type check

Modifier	Amount	Percentage
package visible	1 022	5.9%
`public`	13 399	77.5%
`private`	1 450	8.4%
`protected`	1 423	8.2%
`static`	2 178	12.6%
`final`	1 527	8.8%
`synchronized`	315	1.8%
`volatile`	127	0.7%
`transient`	62	0.4%
`native`	58	0.3%
`abstract`	1 623	9.4%
annotation	558	3.2%
deprecated	24	0.1%

Table 5.7: Modifier usage for methods

Modifier	Amount	Percentage
package visible	545	25.1%
`public`	1 331	61.1%
`private`	301	13.8%
`protected`	1	0.0%
`final`	353	16.2%
`synchronized`	23	1.1%
`volatile`	1	0.0%
`transient`	8	0.4%
`native`	25	1.1%
annotation	432	19.8%
deprecated	7	0.3%

Table 5.8: Modifier usage for `static` methods

should be done in the last step.

# Parameters	Amount	Percentage
0	6 568	38.0%
1	7 115	41.1%
2	2 247	13.0%
3	728	4.2%
4	342	2.0%
5+	293	1.7%

Table 5.9: Parameter usage for methods

Table 5.9 relates the number of parameters to the amount of methods having such a parameter amount. With 16 838 parameters there are roughly 0.94 parameters per method. That is about 0.5 (or 35%) less parameters per method than parameters per constructor.

Of all passed parameters 11 595, or 68.9% have a type stemming from the Java API, so 5 243 (31.1%) have a custom type. Of the parameters stemming from the Java API 8 335 (49.5%) are either a primitive or from the `java.lang` package. Looking at the specific types of parameters the following can be gathered: 3 218 (19.1%) are `int`, 1 859 (11.0%) are of the `java.lang.String` type, 970 (5.7%) are `boolean` and 734 (4.4%) are `java.lang.Object`.

About 7.6% of the methods declare an exception. Of these 1 317 methods 992 declare one exception, 129 declare two, 191 declare three, and 5 declare four. Of the declared exceptions 44% are application-specific and the remaining 56% are exceptions taken for the Java API. The majority, about 70%, of the methods that declare an exception are application methods.

In this survey two applications declared each 35% of those 70% leaving about 15% for the other two (smaller) applications. When the number of methods per application is taken into account then for each applications about 7.1% of the application-specific methods declare an exception.

A total of 98 175 call-sites of methods have been found. Of these, 46.7% call a method from the Java API; the other 53.3% of the calls are directed at application-specific classes. The `java.lang` package is called in 21.8% of the cases. Hereby, the pair `StringBuffer/StringBuilder` are responsible for over 50% of those calls. Most of these string-building calls are generated automatically by the compiler when it encounters the concatenation of strings using the `+` operator. This means that in theory these call-sites should not be affected by the aspect. In practice, however, it is impossible to distinguish between such implicitly generated and explicit calls.

Other Java API packages that are commonly called are GUI related, such as `javax.swing` and `java.awt` at 13.9%, and `java.util` for 9.9%.

Storage The storage techniques suitable for the sub-patterns of methods are mostly described above for other generic-function kinds. The name pattern is described in the section about fields, whereas the modifier, parameter and exception pattern are described in the constructor section. The return type is the exception. The return type, however, should be handled in a unique fashion: while it follows the same technique as the type pattern of fields, it also has a `void` type which the type pattern does not have. As this matches a large part of the methods it should be considered whether actually storing the bucket for `void` types is of use.

5.1.8 Summary

For most sub-patterns there are use cases for both bucket arrays and sorted collections. As such information about the actually looked for patterns has to be gathered before it is possible to determine which technique makes sense for a particular sub-pattern.

Table 5.10 shows the sub-patterns, their technique and some remarks. Some of the sub-patterns are not supported by all generic-function kinds. In that case one has to ignore the sub-pattern information. In most cases the data applies to all generic-function kinds that support the sub-pattern, however with the modifier sub-patterns there is a difference between constructors and methods on one hand and fields on the other hand; for the former the `public` access modifier is not selective whereas for the latter the `private` access modifier is not selective.

Sub-pattern	Storage	Remarks
class name	bucket	when ranges are not used
class name	sorted	when ranges are used
name	bucket	when ranges are not used
name	sorted	when ranges are used
modifiers	bucket	for constructors and methods, except public
modifiers	bucket	for fields, except private
(return) type	bucket	when ranges are not used, except void
(return) type	sorted	when ranges are used
parameters	bucket	looks at first type
parameters	bucket	looks at parameter count
parameters	sorted	looks at all types
exceptions	bucket	

Table 5.10: Proposed storage techniques per sub-pattern

5.2 Pattern characteristics

To be able to tell how often a particular part of a pointcut pattern is used one needs to look at real-world aspects and the patterns they use in their pointcuts. For example, if none of the aspects specify a class-name pattern it would be of little use to build a sorted list of class names. As such having this information can help avoiding mistakes made based solely on the characteristics of the generic-functions.

5.2.1 Methodology

To acquire the characteristics of aspects a number of applications and aspect libraries have been analysed. This analysis was mostly done by hand on the source code of the aspects (all written in AspectJ). The manual work was needed because in most of the cases the package name of classes was not part of the join point as the class was imported or in the local package scope.

Aspect mining techniques, such as described in [9], that extract aspects from real world applications has not been used as this would result in application specific patterns that possibly would favour the larger applications as they have more places where aspects could be found.

The following AspectJ applications and libraries and the patterns used in them are analysed:

- ajlib-incubator; a library with aspects that are meant to be reusable.
- Contract4J5; aspects to support "design by contract" in Java.
- Glassbox; an application for monitoring other applications by using aspects.
- NVersion, RecoveryCache; generic aspects for fault tolerance.
- Sable Benchmarks; set generic aspects used for benchmark purposes.

5.2.2 Acquired information

In total, 170 different patterns have been found in 242 different aspects. What is noteworthy about the found patterns is that many of them were found within code that limits the call-sites for a particular pattern to a specific class or package. In these cases doing a full search of all call-sites would be a waste of time; only looking at an unsorted list of call-sites within a particular class or package would suffice.

5.2.3 Static initialisers

Two patterns have been found that look at static initialisers. One matches for all static initialisers, whereas the other matches exactly one class.

5.2.4 Constructors

In total, ten patterns matching a constructor have been found. Of these, four have an "any class" pattern, one has a "any class in package" pattern and the remaining five match a particular class. Two patterns match the `native` modifier and one matches the `public` modifier. It has to be noted, however, that the former two patterns do not make sense, as the `native` modifier is prohibited for constructors by the Java language specification [11, Chapter 8.8.3]; thus these patterns can never match. Furthermore there is one pattern that requires an exception to be declared.

The ten patterns are used by sixteen different pointcuts; six pointcuts use the "match any constructor" pattern, two times a "match any public constructor" pattern. All other patterns have been found only once.

5.2.5 Field reads and writes

We found five field read and four field write patterns. Of these, four matched "any class", three "any class in package", and two patterns matched a specific class. The latter two patterns furthermore matched a specific "name" pattern. Two modifiers patterns matched `public` methods, while two others did match on the modifiers respectively being `static` or not being `static`.

Only the "match any field read/write" patterns were found multiple times; three and two times for field reads and field writes, respectively.

5.2.6 Methods

The majority of patterns are method patterns; 149 of the found patterns match a method call. 36 of these match "any class," 2 match "any class in package", and the remaining 111 match a specific class. Of these, 75 classes match a class from the Java API.

18 patterns match a method with "any name", 23 match the "first few characters of a name", one matches the "last characters of a name", and the remaining 107 match a specific name. A small number of the same name patterns were found in different patterns, however, none in more than three patterns. In total there are 103 unique patterns.

10 patterns match on the `public`, 2 on the `native`, and 1 on the `static` modifier. Also, 3 patterns match methods without the `static` modifier. One of the methods matches both the `public` and the `static` modifier.

13 return type patterns match `void`, 13 match a specific type and one matches all non-`void` return types. The remaining patterns match any return type.

41 parameter patterns specify that there may be no parameters, 4 specify that there must be exactly one parameter regardless of type, 12 parameters have a specific pattern for the parameters, 3 describe the first parameter

but allow more parameters, and the remaining 89 patterns match any parameters.

In total, 4 patterns match declared exceptions; in 2 of these cases any exception suffices whereas in the others a specific exception must be declared to be thrown.

Of the 212 found aspects 67 match any class. Of these 23 match any name, resulting in 44 cases that describe at least a partial name. The former all match on either `public`, non-`static` or “any” modifiers. Furthermore the matched return types of these 23 are `void`, non-`void` and “any”. This means that in the case no class and no name is given the pattern can be considered to be a “match any” pattern in view of using sorted data sets for look-up.

For patterns that match “any class in a package” the other sub-patterns are generally match ”any” sub-patterns with the exception of an occasional match `public` or match throws `java.lang.Throwable`. This means that in that case the package could be used as initial data set.

Of the 212 found patterns 143 match a specific class and two match a class within a specific package. Of these, 143 only six have a match “any method name” pattern. 44 of the remaining 67 patterns match on the method name. This leaves 23 patterns that do not match on the class or method name. Of these 23 patterns, four are fairly selective as they specify selective parameter types, modifiers or declared exceptions.

5.3 Optimisation strategies

This section describes the default optimization strategies that can be extracted from the Sections 5.1 and 5.2. As in those sections the optimization strategy will be discussed per generic-function kind.

5.3.1 Static initialisers

Static-initialisers patterns can only be evaluated in one way: looking at the class name. To optimise this, the generic-functions could be sorted by their full class name to aid the look-up speed.

Due to the small amount of patterns matching on static initializers found in the case study it is unclear whether an index data structure can pay off. If such patterns occur sufficiently often, a sorted collection can offer quick access.

5.3.2 Constructors

For constructors there are basically two sub-patterns that can be used for optimization: A sorted collection with the full class names when a class name is known and a bucket array with the call-sites per modifier otherwise.

Hereby, the sorted list of class names would be the primary way of searching; only if there is no class name to match, one can consider using the bucket array.

In cases where modifiers other than `public` are used in method patterns, matching that modifier using the bucket array should be considered due to the high selectivity of those modifiers. However, when the full class name is very selective, i.e. the class name has no wild cards in it, that would still take precedence.

There are no aspects referring to declared parameters and exceptions of a constructor. Therefore, we cannot deduce an optimal look-up and storage strategy. Until sufficient data is available, we assume that carrying over the results for methods is a reasonable starting point.

5.3.3 Field reads and writes

For field reads and writes the main optimization point is the class name as well. Nevertheless, the modifiers are also useful, as the ones found in the survey are also very selective with the exception of the `private` modifier. In contrast, the field name and the type are not worth considering, even though the name is actually very selective in this case. This is due to the fact that it is rarely used and thus the overhead of building the index outweighs the small extra selectivity over the class name which can be used much more often. However, a majority of the patterns found in the survey match everything; thus, in these cases there is little to optimize.

5.3.4 Methods

Most methods can be matched by their name. They are matching a specific pattern making them not that selective, e.g., `get*` and `set*`. So in reality they match more cases, first checking the class name will yield better initial selectivity unless the class name can be anything.

Parameter and return types are the next most selective, however, in the majority of the cases the class or name were already matched meaning that as initial search parameter they are not very useful, especially because they are matching very common types, such as `void` for return types (or being parameter-less).

The modifiers of a method are in most cases not selective at all; most of them match more than 75% of the methods. This actually makes the return type more selective than the modifiers for the case where there are no class or method names to match.

Exceptions are rarely used in patterns. Effectively, patterns that do refer to declared exceptions just require that at least one exception is thrown but they do not further specify patterns for the exception types. In the two patterns we found in the case study that do specify a required exception type, the sub-pattern for the declaring class is more selective than the exceptions sub-pattern. As such, the exceptions sub-pattern should not be evaluated early.

5.3.5 Conclusion

Taking the above analysis into account, we now give a summary of the best storage techniques for improving the performance of the pattern matching in Table 5.11. Hereby, the numbers represent the order in which to evaluate the sub-patterns.

Sub-pattern	Technique	Static init.	Constr.	Field	Method
class name	sorted	1	1	1	1
name	sorted	-	-	3	2
modifiers	bucket	-	2	2	4
type	bucket	-	-	4	-
return type	sorted	-	-	-	3
parameters	sorted	-	3	-	5
exceptions	bucket	-	4	-	6

Table 5.11: Best storage techniques per sub-pattern and order of use by kind

As can be seen, the class name is the best sub-pattern to start with. This is usually followed by the modifiers, except for the case of methods; here, the name of the method is a more selective secondary search parameter. Methods also have a third search parameter, the return type.

Note, however, that this data depends on the generic-functions used by the actual applications and on the aspects used to search for a subset of these. If an application, for example, declares only a few checked exceptions and all aspects look for a particular exception, then a simple bucket array could be the most selective and thus best way to start the search.

Another example would be a pattern matching all `public static` methods starting with `get`. In that case the name sub-pattern would match around 25% of the methods whereas the modifiers sub-pattern matches only 8% and thus the modifiers pattern would be the most selective.

Chapter 6

Implementation

6.1 Methodology

It is necessary to have a performance measuring methodology to determine the performance effects of the proposed improvements. To compare the actual performance effects one has to use one of the benchmarking techniques. The areas and the data used for benchmarking are described in this section as well.

6.1.1 Benchmarking techniques

On a high level performance can be measured by looking at the execution time whole application, or by looking at a small piece of an application which is called micro benchmarking.

The advantage of the former technique is that it is easy to set up; just run an application and time how long it takes. For better accuracy and precision this is done multiple times in a row and then the mean of all runs is taken. The major disadvantage is that the part this research is interested in is a very small part of the running of an application.

Micro benchmarking does not have that problem as it looks at only very specific parts of an application. It requires adding bits of code to determine the time that passes between entering the area one is interested in and leaving that same area. As the pieces of code that are evaluated are relatively small this needs a high precision method of getting the time, e.g. some measurements had an average length of less than 750 nanoseconds. At this time-scale the disadvantage of micro benchmarking comes to light; anything that interrupts the execution while in a time measured area of the execution will have an enormous effect on the average.

In this study this disadvantage is mostly negated by explicitly trying to remove as much of the interruptions as possible. This is done by, amongst others, manually calling the garbage collector until the garbage collector does not free more objects, manually yielding the execution so the chance the operating system's scheduler does schedule another task is reduced, fixing the frequency at which the CPU runs so its frequency does not fluctuate and disabling as many things that generate interrupts as possible such as network or background services. Even though this helps giving a more consistent result there is, especially for the smaller cases, a lot of fluctuation between runs. To reduce the "noise" from this fluctuation the methods are called multiple times and then the lowest and highest 5% of the data point values are removed. Of the remaining 90% the mean is calculated and used.

6.1.2 Benchmarked areas

The major area of interest is the actual matching of patterns against the call-sites including the iteration of the call-sites. However, another area of interest is the "sorting" of the call-sites or rather the insertion of the call-sites in the collection of all found call-sites. Therefore this study will look at two areas of interest: the actual matching and the insertion. It will not look at the costs incurred with finding the call-sites in an application of applying the attachment to the call-sites that match the associated pattern.

Though, for determining what the best order of sub-pattern matching it is needed to know how effective the sub-patterns are. For this each sub-pattern's matching is benchmarked and compared to the "match"-to-"no match" ratio of the single sub-patterns. In this case all sub-patterns are evaluated and the lazy evaluation, returning "no match" on this first non-matching sub-pattern, of the whole pattern is not performed.

6.1.3 Setting up the benchmark data

To set up the benchmark data it is necessary to have call-sites and patterns to match against. The easiest way for this is by using NOIRIn or SiRIn, two of ALIA4J's reference implementations. However, NOIRIn matches a call-site to a pattern, i.e. the matching happens in the inverse direction. Furthermore both NOIRIn and SiRIn load call-sites whenever they come across them. This means that one has to go through a large number of code paths within an application to load all the call-sites.

To overcome this a small extension to "Extract" was written called "Benchmark". This tool extracts call-sites in the same way as "Extract" and loads patterns from a comma separated file. These call-sites are inserted into the FIAL framework while being timed. After the time it takes to get all call-sites matching the patterns, one lookup per pattern, is measured.

To efficiently perform these tests only the FIAL framework is reset between each measurement run, i.e. the loaded call-sites within FIAL are cleared. The full benchmark returns the measurements of many runs.

The sub-pattern matching measurements are done using the same measurement, but it only returns the results of the sub-pattern matchings.

The FIAL framework does not support constructor and static initialisers thus these are excluded from the benchmarks.

6.1.4 Benchmark data

For the benchmarking the same applications as for the survey in Chapter 5 are taken together with two larger "applications" to determine how the different benchmarked algorithms scale in large applications. The benchmarked "applications" are:

- **ANTLR** A tool to construct parsers, compilers and translators.
- **FreeCol** A turn-based strategy game.
- **Java's runtime** Java platform's core runtime API.
- **Java's tools** Non-core Java platform classes like the Java compiler.
- **LIAM** A major part of the implementation of ALIA4J.
- **TightVNC** An application to remotely take over a desktop.

Application	Methods	Field reads	Field writes	Total
ANTLR	23 468	8 936	2 661	35 065
FreeCol	58 843	11 010	4 716	74 569
Java's runtime	471 368	204 995	79 796	756 159
Java's tools	120 633	41 528	14 242	176 403
LIAM	3 098	558	236	3 922
TightVNC	1 321	1 656	474	3 451
Total	678 731	268 713	102 125	1 049 569

Table 6.1: Amount of call-sites per application

The amount of benchmarked call-sites per application can be seen in Table 6.1. It shows that, when looking at the size of these applications in a logarithmic scale, there is a reasonable spread of application sizes given the small set a benchmarked applications.

For Java's runtime the call-sites in the classes of the API are evaluated whereas for the other applications the call-sites in Java's API are excluded. Call-sites to Java's API are always included.

The patterns that are used for this benchmark are as well from the survey in Chapter 5. The only difference is that where some patterns were found multiple times, all patterns are applied only once. This as FIAL caches the result of getting the call-sites of a pattern and as such getting it multiple times would mean getting them from the cache which is relatively cheap. As mentioned before the constructor and static initialiser patterns are ignored as well resulting in 158 patterns that are looked up.

The benchmarks resulted in a lot of data. The most interesting results will be discussed in this chapter and the full data and graphs can be found in Appendix A. The small summary tables in this chapter refer to a table in the appendix where more information about the range and accuracy of the measurements can be found.

6.2 Base

The base benchmark, i.e. the benchmark using the original data structure and algorithms, consists of two parts. A benchmark of the insertion of call-sites and a benchmark of the matching of a pattern against the call-sites. All further benchmarks will be normalised to these two when comparing algorithms and implementations.

The base algorithm puts all call-sites in a hash table and iterates over the complete table trying to match a pattern against all call-sites.

Insertion The base insertion is insertion of call-sites into a hash set. As expected, the larger the application the longer a single benchmark run takes.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
73ns	82ns	171ns	167ns	178ns	188ns

Table 6.2: Average insertion time per call-site (Table A.1)

Table 6.2 shows the average insertion cost per call-site. The table shows that the insertion for the two small applications are roughly equal and the the same holds for the four larger applications. This might be explained by the in-memory size of the hash tables, as described in Section 6.5. It it conceivable that the smaller hash tables fit entirely in the CPU's cache and thus the CPU has to wait less for the requested data to be available.

Matching The base matching method, i.e. the original matching method, is determined by matching all 158 method, field read and field write patterns found in Chapter 5. Finally the average of that number is taken as measurement for matching the average pattern.

Table 6.3 shows the average matching cost per call-site. The variation in the matching times can be explained by the package names of the different packages. Table 6.4 shows the "primary root package name", which is the

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
275ns	686ns	580ns	781ns	692ns	642ns

Table 6.3: Average matching time per call-site (Table A.2)

Application	Primary root package	Class name length
TightVNC	.	12.1
LIAM	org.alia4j.liam.	27.8
ANTLR	antlr.	19.8
FreeCol	net.sf.freecol.	30.7
Java's tools	com.sun.	37.2
Java's runtime	java.	33.6

Table 6.4: Statistics about the application's package and class names

part of the longest package name that is the same for the majority of the classes, and the length of the average full class name.

As can be seen in the tables: the application with the shortest primary root package name, i.e. TightVNC, matches fastest, whereas the applications with the longest primary root package names match slowest. Furthermore is the class name of influence, but a smaller influence than the length of primary root package's name.

However, the insertion of LIAM was much faster than the larger applications. If all call-sites were kept in memory, as suggested in the previous section, the LIAM case should be much faster. As can be seen in Section 6.5 the memory usage for LIAM is significantly higher than any other application. This higher memory usage suggests that the load factor for LIAM is lower than for the other applications. This lower load factor means that iterating the whole table consists of a relative large amount of empty entries. That causes more empty entries need to be skipped per actual call-site and thus the average matching time increases.

6.3 Sub-pattern matching order

The first few optimisations that are benchmarked are based upon the data gathered from the aspects as described in Section 5.2 and 5.3, and the selectivity principle. In other words, this section shows and evaluates the performance of improvements to the matching without changing the data structure where the call-sites are stored in.

Table 6.5 shows the sub-pattern matching order of the different methods described in this section. The original method is seen as the "base" method and the others are compared to that.

	Original		Class-first		Name-first	
Sub-pattern	Field	Method	Field	Method	Field	Method
class name	3	3	1	1	2	2
name	4	4	3	2	1	1
modifiers	1	1	2	4	3	3
type	2	-	4	-	4	-
return type	-	2	-	3	-	4
parameters	-	5	-	5	-	6
exceptions	-	6	-	6	-	5

Table 6.5: Sub-pattern order in different matching orders

Class-first The first match optimisation strategy is to use the order defined in Section 5.3. In short the class name sub-pattern seems to be the most selective when looking at the aspects and as such they should be checked first, followed by the other sub-patterns. See Table 5.11 for more information.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
1.07	1.08	1.05	0.99	1.19	1.12

Table 6.6: Relative time of class-first matching (Table A.3)

A comparison of the class-first matching to the original matching method can be seen in Table 6.6. The numbers in the latter table show the relative time the matching took compared to the original matching method. A number larger than 1 means it took more time, thus its performance is worse, whereas a number smaller than 1 means a speed-up. Overall it can be said that the performance when matching the class-first is worse than when the original method is used.

Sub-pattern efficiency As can be seen the results from Section 5.3 did not improve the performance of the matching. This is primarily due to the fact that that section did not take the "cost" of the sub-pattern matches into account; a sub-pattern can be most selective, but if it costs a lot to evaluate it might be more efficient to evaluate another sub-pattern first.

To determine what the best sub-pattern to start with is one has to know both the time it takes to evaluate a sub-pattern and what the selectivity of that sub-pattern is.

To determine this all sub-patterns have been benchmarked and their selectivity was calculated based on the 158 method and field read/write patterns of the survey and the call-sites of the benchmarked applications.

These numbers are then normalised to respectively the time all sub-pattern benchmarks took and the number of matchings. This means that for each application and sub-pattern there are two numbers: the share of the matching cost and the percentage of call-sites that did match. The efficiency is then calculated using the following formula:

$$\mathit{efficiency} = \frac{1 - \mathit{matching\ share}}{\mathit{share\ in\ matching\ cost}}$$

The resulting efficiency for the different sub-patterns can be seen in Table A.4. For the methods the efficiency factors are relatively conclusive; the different sub-patterns are ranked in the same order for all applications.

For fields the data is less conclusive; the ranking of the sub-patterns differs per application and the type always has an efficiency of 0. The latter has to do with the fact that the found patterns did never match on a field's type. The inconclusive ranking is primarily caused by the cost of the class name matching for TightVNC; the cost is roughly 10% lower meaning that the relative costs of name and modifier matching increase. This results in an increase of the class sub-pattern's efficiency factor and a decrease in the efficiency factor for the others. Besides that the ranking of the class and modifiers sub-patterns are occasionally reversed.

Nevertheless, the final ranking is the same regardless of whether the individual rankings or the average efficiency factor is used to determine the ranking. Table 6.5 shows the optimal ranking according to the efficiency factor. This "name-first" ranking is primarily due to the fact that matching the name sub-pattern is significantly cheaper than matching the class sub-pattern.

Sub-pattern efficiency based matching The benchmark results using the ranking of the aforementioned efficiency factors to order the sub-pattern for matching are shown in Table 6.7.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.29	0.38	0.28	0.27	0.33	0.36

Table 6.7: Relative time of name-first matching (Table A.5)

As can be seen, the improvements are a significant threefold over the original algorithm. This has mainly to do with the much higher efficiency factor of the name sub-pattern than the class sub-pattern.

The relatively small improvement for LIAM furthermore enforces the assumption that the load factor of the hash table for LIAM was significantly lower than for the others. If more time is spent in iterating empty entries in the hash table the improvements in matching will have a smaller effect on the whole.

Pattern optimised matching With this optimisation strategy the pattern matching method is optimised on a per-pattern basis. The optimisations primarily include not doing pattern matches when it is certain they do not match, e.g. matching an "any name" sub-pattern is not needed. It uses the name-first order as basis.

This is implemented by creating a class with the optimised match method at run-time taking the different sub-patterns of the to-be-matched pattern into account.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.29	0.38	0.29	0.28	0.32	0.35

Table 6.8: Relative time of pattern optimised matching (Table A.6)

Table 6.8 shows the improvements against the base matching method. The differences between pattern optimised matching and name-first matching are small; for the smaller applications there is a small slowdown, whereas the matching larger applications became slightly faster. This can all be explained by the overhead for creating the class.

Evaluation Of the three matching proposed optimisations the class-first matching is worse than the original implementation in all but one case, i.e. it is not an optimisation in practice.

The two other optimisations show roughly the same result; a threefold performance improvement in matching a single pattern. The pattern-optimised method seems to scale better for large applications, although even there the difference is almost negligible. Given the larger complexity of the pattern optimised method the name-first method seems to be better. However, the pattern optimised method can be improved by looking at more parts of the sub-patterns in an effort to improve its performance, i.e. to reduce the amount of effort needed to determine a call-site does not match.

Another improvement to the pattern optimised method that can be considered is taking more selectivity information into account. For example when it is known that a `void get*` method is matched one can perform the return type after the name as that would be very selective. However, taking more selectivity information into account means a larger overhead at the class construction time. The relatively small improvement by not matching certain sub-patterns does suggest that most time is actually spent in iterating the hash table and as such further improvements to the matching performance will be relatively insignificant.

6.4 Sorting

The main idea behind storing the call-sites in a sorted structure is to trade insertion performance for matching performance and ideally improve the performance in general. In total three different sorting "keys" were implemented:

- Class; using the full class name as sorting key.
- Name; using the full method/field name as sorting key.
- Name prefix; using the first three characters of the method/field name as sorting key.

Implementation The basis of the implementation is a mapping between the key and the call-sites where each key refers to multiple call-sites. This map is sorted on the key.

The implementation uses a `TreeMap` which is comparable to a 2-order B+-tree. As a `TreeMap` cannot map one key to multiple items it is needed to let it map to another collection. In this case that is the same hash table as the used in the original unsorted system. This makes the size of the sorted map, and thus the lookup, related to the number of unique keys.

Lookups are done as described in Section 5.1.3. This means that in case of a class name with wild card a sub map of the sorted map is requested. In case of pattern matching a specific class a single lookup is done and when any class is matched the whole map is iterated.

6.4.1 Sorting by class

Section 5.3 concludes that class names are best as initial sub-pattern to search on.

Insertion Benchmark results of inserting the call-sites into a collection that is sorted by class name are shown in Table 6.9.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
2.86	3.33	2.14	3.44	4.49	5.64

Table 6.9: Relative time for insertions when sorted by class name (Table A.7)

The larger applications are relatively slower than the smaller ones as could be expected by the $O(log\, n)$ complexity of insertion into a B+-tree. The low relative value for ANTLR is due to the relatively high cost of insertion into the original data structure. ANTLR furthermore has an extraordinarily large number of methods and fields in some classes causing the sorted tree to be relatively shallow.

Original matching Table 6.10 shows the benchmark result of the original matching method using the collection that is sorted by class name.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.07	0.09	0.08	0.08	0.09	0.09

Table 6.10: Relative time for original matching with data sorted by class name (Table A.8)

The table shows the relative time taken for the matching. What is immediately clear is that this matching method is at least 10 times faster for all the applications than the original method. It also looks like the performance improvements decrease in relation to the size of the application. This might well be related to the depth of the sorted tree and its lookup cost.

Sub-pattern efficiency As the range lookup already filters out a lot of unmatchable call-sites the sub-pattern efficiency factors are different. Table A.9 shows the efficiency factors given that the collection is sorted by class name and that a range lookup can be done.

Unlike the base case the efficiency ranking for methods is less clear; the name sub-pattern is definitely the most efficient, the class sub-pattern is the worst as almost all call-sites match and the modifiers sub-pattern is second worst. The return type, parameters and exceptions sub-pattern are very close with their average ranking being equal and their average efficiency factor differing by less than one percent.

	Class		Name		Name prefix	
Sub-pattern	Field	Method	Field	Method	Field	Method
class name	4	6	2	1	2	1
name	2	1	3	6	3	3
modifiers	1	5	1	2	1	2
type	3	-	4	-	4	-
return type	-	4	-	3	-	6
parameters	-	3	-	5	-	5
exceptions	-	2	-	4	-	4

Table 6.11: Best sub-pattern order for different sorting keys

The efficiency ranking for methods is even less clear. As in Table A.4 the type sub-pattern matches everything, but now the class sub-pattern is matched in all cases as well. The name sub-pattern did not match in 0.0024% of the time leaving the modifiers sub-pattern as the best ranked sub-pattern. Table 6.11 shows the best sub-pattern order for the different sorting methods.

Sub-pattern efficiency based matching Table 6.12 shows the results when the matching takes the aforementioned order of sub-pattern matches into account.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.06	0.08	0.07	0.08	0.09	0.09

Table 6.12: Relative time for matching using efficiency factor with data sorted by class name (Table A.10)

This implementation's sub-pattern order is always around 10% faster than the original sub-pattern matching order.

Pattern optimised matching Table 6.13 shows the benchmark results when optimising the pattern matching by not performing the sub-pattern matches that are certain to match. In this case the fact that a range lookup is done is taken into account as well; when it is certain the class sub-pattern is already matched that is not evaluated either. As more sub-patterns could be removed it now shows a slight improvement over the matching order using the efficiency factor.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.07	0.07	0.08	0.07	0.09	0.08

Table 6.13: Relative time for pattern-optimised matching with data sorted by class name (Table A.11)

This implementation is slightly slower (less than 5%) for TightVNC and ANTLR, which have short package names, whereas there is an improvement of up to 20% LIAM. Overall the improvement is roughly 2.5% compared to the efficiency ranking ordered matching.

Evaluation The matching when doing range lookups based on the class name cannot be improved as impressively as the original storage method did. When determining whether to go for the sub-pattern matching order based on efficiency factors or for the optimised pattern matching one has to look at the length of the primary root package. If that is short the former method should be taken and if the primary root package is long the optimised pattern matching has to be considered.

6.4.2 Sorting by name

As seen in Section 6.3 the name has a significantly higher efficiency factor when matching than the class name has. This section evaluates whether this is also the case when the B+-tree is sorted by the name.

Insertion Table 6.14 shows the results for inserting the call-sites into a collection sorted by the name of the fields and methods.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
4.53	4.37	2.27	3.61	4.16	5.99

Table 6.14: Relative time for insertions in a collection sorted by name (Table A.12)

As can be seen in the table, sorting with this key is slower for all applications except the Java tools.

Original matching Table 6.15 shows the result of the original matching method using the collection that is sorted by name.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.12	0.17	0.11	0.12	0.14	0.17

Table 6.15: Relative time for original matching with data sorted by name (Table A.13)

The four largest applications show a gradual increase in time as can be expected by the larger sorted collection. The large numbers for the two smallest applications are likely due to favourable CPU cache effects fast in the base benchmark and as such the improvements cannot be in line with expectations.

The larger number for LIAM can, again, be explained by a lower load factor for the secondary hash tables as was the case for the original case. More time is spent in iterating the hash tables and as such the improvements are smaller.

Sub-pattern efficiency Table A.14 shows the efficiency factors for the different applications and sub-patterns and Table 6.11 shows the order for sub-pattern matching that is most efficient based on that ranking.

Contrary to the efficiency factors for class names there are no cases where there is no name to match. This is due to the fact that there are patterns that match the last few characters of a name and as such still need to iterate all call-sites.

The efficiency of the name sub-pattern for methods is ranked lowest, the class sub-pattern highest and the four others are pretty close although their average ranking and ranking of their averages result in the same order. For fields the type sub-pattern is again the lowest ranking pattern due to the fact that no pattern matches it closely followed by the name sub-pattern. The modifiers sub-pattern is ranked highest.

Sub-pattern efficiency based matching Table 6.16 shows the results of matching using the efficiency ranking to order the sub-pattern matching.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.13	0.19	0.11	0.13	0.14	0.15

Table 6.16: Relative time for matching using efficiency factor with data sorted by name (Table A.15)

When comparing Table 6.16 with the relative time taken for the original matching order in Table 6.15 one can see that the matching takes longer for all applications except the two largest. There is no clear explanation for this behaviour besides the patterns that match the end of a name. In that case all call-sites have to be iterated and evaluated and the order using the efficiency factor moved the sub-pattern matching to the end, thus more sub-patterns have to be evaluated in that case.

Pattern optimised matching Table 6.17 shows the results when optimising the pattern matching by not performing the sub-pattern matches that are certain to match. In this case the fact that a range lookup is done is taken into account as well; when it is certain the name sub-pattern is already matched that is not evaluated either. As more sub-patterns could be removed it now shows a slight improvement over the matching order using the efficiency factor.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.13	0.18	0.11	0.12	0.13	0.16

Table 6.17: Relative time for pattern-optimised matching with data sorted by class name (Table A.16)

As with the optimised matcher when the call-sites are sorted by class name the smallest applications' matching gets slightly slower whereas the matching of the larger applications gets faster. However, the benefit for the larger applications is smaller than when sorted by class.

Evaluation Of the matching methods benchmarked the original seems to be the best overall. The sub-pattern matching order based on the efficiency factor is clearly slower and the optimised pattern matcher only seems to have effect on applications with a large number of call-sites.

6.4.3 Sorting by name prefix

As concluded in Section 6.4.2 the initial sorting costs for sorting by name are significantly higher than those for classes. Together with the notion that the number of method/field names outweighs the number of classes and thus the B+-tree gets quite deep it has to be considered whether sorting by only the first three characters as mentioned in Section 5.1.

Insertion Table 6.18 shows the results for inserting the call-sites into a collection sorted by the prefix, the first three characters, of the method and field names.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
3.72	6.64	1.98	2.61	3.29	4.07

Table 6.18: Relative time for insertions in a collection sorted by name prefix (Table A.17)

Sorting with this key is slower for small applications but faster for larger applications when comparing it to sorting by class name. This can be explained by the fact that there is a limited number of keys for the name prefix, but in small applications the number of methods and fields that use the same prefix is small and thus relatively a lot of time is spent in the B+-tree insertion and lookup instead of the relatively cheap insertion into a hash table.

Original matching Table 6.19 shows the result of the original matching method using the collection that is sorted by the first three characters of the method or field name.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.14	0.19	0.12	0.14	0.14	0.17

Table 6.19: Relative time for original matching with data sorted by name prefix (Table A.18)

The high values for the two smallest applications are explained by the fast benchmark results of the original matching method and the low load factor for LIAM's hash tables.

Sub-pattern efficiency Table A.19 shows the efficiency factors for the different applications and sub-patterns and Table 6.11 shows the order for sub-pattern matching that is most efficient based on that ranking.

The field matching efficiency factors are the same as the factors from when the data is sorted by name. This is due to the fact that the patterns only matched three character long names, or any name.

For the method the sub-patterns are more interesting; the class sub-pattern is clearly the most efficient, but all others are roughly equally well. The average of the rankings does not even match the ranking of the averages. As a result of this there is no crystal clear order in which to evaluate the sub-pattern, thus the ranking of the averages is taken.

Sub-pattern efficiency based matching Table 6.20 shows the benchmark results for matching using the sub-pattern matching order derived from the efficiency ranking.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.13	0.18	0.12	0.11	0.13	0.16

Table 6.20: Relative time for matching using efficiency factor with data sorted by name prefix (Table A.20)

This method is in all cases faster than the original matching sub-pattern order.

Pattern optimised matching Table 6.21 shows the benchmark results when optimising the pattern matching by not performing the sub-pattern matches that are certain to match. In this case the range lookup is taken into account, but only the if the pattern did match three or less characters at the start; otherwise it will still do a name sub-pattern match.

TightVNC	LIAM	ANTLR	FreeCol	Tools	Runtime
0.12	0.17	0.11	0.11	0.13	0.16

Table 6.21: Relative time for matching using efficiency factor with data sorted by name prefix (Table A.21)

The performance of the optimised matcher is better than the performance using only the efficiency factors.

Evaluation It is clear that the reordering the sub-pattern matches and optimised sub-pattern matching has a beneficial effect. However, the differences are relatively small and as such the extra costs of construction the optimised sub-pattern matchers might not be worth it.

6.4.4 Evaluation

In all cases sorting by the method and field name is not the fastest; its insertion and matching are reliably slower than when sorting by name. This leaves the sorting by class and name prefix.

On one hand the name prefix's insertion is faster for all but the smallest applications. On the other hand the matchings using the prefix name are slower than when sorted by the class name. This means that the decision on which algorithm to use depends on both the amount of call-sites and the number of patterns that are going to be applied, e.g. when 5 patterns are applied the method sorted by class name wins for all but the two largest applications and when 10 patterns are applied the method sorted by class name is best for all applications.

6.5 Memory usage

Besides performance implications related to the different optimisation strategies there are memory implications as well. This section compares the memory requirements of the different base strategies, i.e. the memory requirements for the collection of call-sites. As can be seen in Table A.22 the extra memory requirements depend heavily on the amount of call-sites in an application, e.g. only 50% extra memory on top of the base is needed for a small application whereas a 200% extra memory is needed for a large application.

Sorting by name always uses the most memory, but relatively to memory usage when sorting by name it decreases. Furthermore for the smaller half of the applications sorting by class name uses less memory than when sorting by the prefix name and vice versa for the larger half. However, the difference between the class and prefix sorted approaches never differs more than 10% except for the smallest application.

The fact that the prefix sorted's memory usage grows slower than for class names can be found in the fact the the number of prefixes is more or less limited whereas the number of classes is unlimited. As a result the hash tables of the prefix approach will grow for larger applications whereas the tree structure will primarily grow for the sort-by-class approach.

The reason the name sorted's memory usage is this large in the small applications is the fact that the initial hash table size is 16 items and when adding only a few there will be a lot of overhead. However, the larger the application the more a name will be reused and as such the smaller the overhead of the initial hash table size gets.

When ignoring the data for LIAM, which is the outlier in almost all of the above benchmarks, the memory behaviour of the different storage methods becomes more visible. The average size per call-site for classes remains constant around 70 bytes, whereas all others are becoming smaller when more call-sites are loaded. The largest relative improvement can be seen in the original hash table which becomes 55% smaller, the two others become about 30% smaller.

6.6 Evaluation

As seen reordering the sub-pattern matches can improve the performance of the matching threefold. The matching's performance can be improved another four times by sorting the call-sites and using that order to reduce the amount of data that has to be scanned. However, this improvement is offset by an factor four slowdown of the initial insertion of call-sites.

Only reordering the sub-pattern matches has no influence on the insertion and as such going for the name-first ordering of the sub-pattern matches is a good first step into improving the performance of the matching. A pattern optimised sub-pattern matcher can be considered when the patterns are to be applied to large applications although the improvement over the simple name-first reordering is small and the complexity of creating a sub-pattern matcher is big due to the creation of classes at run-time.

The improvements that can be made by sorting depend on the size of the application due to the increasing insertion cost for larger applications as well as the extra memory requirements, and the number of patterns that are applied due to the break-even point between insertion slowdown and matching speed-up.

Figure B.6 shows the relative time taken for the different insertion methods against. The high values for the first two, small, applications can be explained by the fact that the base insertion was unexpectedly fast. The rest shows that the larger the application gets the larger the slowdown in insertion as expected by the $O(log\, n)$ complexity of the B+-tree over $O(1)$ complexity for insertion into the hash table.

Figure B.7 shows the relative time taken for the different sub-pattern efficiency based matching methods on differently sorted collections compared to the original matching method. From this figure it is clear to see that when the data is sorted by class the matching is fastest in all cases. The difference between name and prefix matchings does not differ much, though it is still faster than the sub-pattern efficiency based, i.e. name-first, matching on the original data.

When combining the information shown in Figure B.6 and B.7 one can deduce that given a large enough number of patterns to match the method that sorts the data by the class is the fastest. The relative time for insertion and matching one, two, five and ten average patterns can be seen in, respectively, Figures B.2, B.3, B.4 and B.5.

It is clear that when one pattern is to be applied the simple sub-pattern match, name-first, reordered method is fastest, however all the sorted methods are faster than the original method except for the largest application. When applying two patterns the class sorted method is equally fast or faster than the name-first reordered method for the smaller half of the applications, whereas the prefix sorted method is faster than class sorted for the larger half.

From five applied patterns the class and prefix sorted methods are always faster than the name-first reordered method and the class sorted method becomes fastest for the four smallest applications. Finally with ten applied patterns all sorted methods are faster than the name-first reordered method with the class sorted method winning in all cases.

Chapter 7

Related work

Masuhara et al. have discussed the implementation of the weaver component in the AspectJ compiler [17]. In particular, they also discuss the algorithm used for finding join-point shadows for pointcuts. As mentioned in the introduction, this entails an evaluation at all join-point shadows in the program. The authors also suggest a mechanism they call *fast match* to rule out the matching on a per-class-basis. This mechanism builds on the fact that each Java class is represented in one bytecode file which contains the so-called *constant pool*, i.e., a table of symbols used in the class. The symbols include all signatures of methods and fields accessed by instructions in this class.

In the fast match approach, patterns are first evaluated against the signatures in the constant pool. If no match was found, the class cannot contain matching join-point shadows. Thus, no expensive parsing of the method bodies' instructions is required in order to find the location of join-point shadows.

This approach does not consider the structure of signatures and patterns themselves, but groups join-point shadows according to the locations in which they occur. The constant pool is a summary of the occurring join-point shadows and is used to exclude some locations from the search as shortcut. Our approach is orthogonal to that and exploits common structures in signatures and patterns themselves.

The Steamloom virtual machine [12] supports dynamic aspect deployment and therefore also seeks to improve performance of partial pointcut evaluation. Steamloom implements an indexing mechanism that allows to quickly map from matched signatures to the locations of the corresponding join-point shadows. This is similar to the fast match in the AspectJ compiler, but also does not consider heuristics of common signature and pattern structures.

In [13] Haupt and Mezini introduce a method for benchmarking the cost of dynamic aspect deployment. This is a superset of what has researched for this study as this study ignores the actual weaving, i.e. insertion of code at the join-point shadows.

Chapter 8

Conclusion

To determine what the most efficient method for finding call-sites that match a pattern several sub problems have to be solved. These are shortly summarised before getting to the final conclusion.

What is the theoretically most efficient method of matching call-sites to patterns? The theoretically most efficient method of matching depends on the type of lookup that a sub-pattern needs to do. If the sub-pattern always wants to look-up a specific value then using a hash table is the preferred way, but when there are a lot of range look-ups then a sorted tree can be more efficient. Whether this is more efficient depends on the extra cost of constructing the sorted tree and the amount of range look-ups that will be done.

What are the general characteristics of call-sites and generic-functions in real world applications? Based on the call-sites and generic-functions in a number of real world applications both a sorted tree or a hash table are applicable for each sub-pattern, except that a sorted tree is not applicable for the exception sub-pattern. The selectiveness of the different sub-patterns depends a lot on the actual patterns that are used.

What are the general characteristics of patterns in real world aspects? For static initialisers and constructors not enough generic patterns have been found in the real world to give a conclusive answer to this question. For field reads and writes the class name seems the most selective initial sub-pattern followed by the modifiers, name and type sub-patterns. For methods the class name is the most selective initial sub-pattern, too, followed by the name, return type, modifiers, parameters and finally exceptions sub-pattern. However the latter modifiers, parameters and exceptions sub-patterns are rarely used in the patterns and as such usually not very selective.

Of the most selective sub-patterns the class name and name sub-patterns are best suited with a sorted data set, i.e. they use a reasonable number of range lookups, whereas the modifiers and type sub-pattern are better suited by a hash table.

What is the most efficient method of matching call-sites to patterns? The most efficient method of matching depends on the number of patterns that need to be matched and the size of the application. The easiest optimisation, reordering the sub-pattern matches, that was made, reduces the matching cost by about 67%, and when including the cost for constructing the set of call-sites matching one pattern is about 55% faster.

When introducing changes to the data structure of the call-sites, i.e. changing it to a set sorted by class, the matching cost can be reduced by around 92.5%. However, the costs for constructing the data set are around four times higher and the data structure requires between two to four times more memory. Setting up the data set and matching one pattern is on average only 20% faster, but when applying five patterns instead of one it is faster than the simple "sub-pattern match reordering" optimisation.

Sorting the data structure by the first three characters of method and field names requires slightly less memory than when sorted by class name for large applications and its insertion costs are lower as well. However, the actual matching is slower. This means that for large applications with a small amount of patterns this method is more efficient than when the data structure is sorted by the class name, but it is slower than the easiest optimisation.

Therefore it can be conclude that the reordering of sub-pattern matches is the most efficient method up to around five patterns or when memory usage is important. For the other cases the method with the data structure sorted by class name is the most efficient.

Chapter 9

Future work

As the implementation of this study used a hybrid of the proposed techniques from the theoretical section it would be interesting to see whether going for a fully sorted tree of call-sites has any further beneficial effects. The ranking of the sub-patterns using the efficiency factors could be used, but even then there are multiple call-sites referring to a single generic-function. It is therefore likely that such a study shows that the possible gains, like range lookups for the second sub-patterns as well, would outweigh the extra costs of sorting.

Furthermore one can study the optimal initial size and load factors for the hash tables. For example for small applications using the data sorted by name there will be a lot of hash tables that are fairly empty. In case of a search all those mostly empty hash tables need to be iterated which wastes time as well as memory. However, reducing the initial size will mean that more rehashes need to be done.

Study other fields in software engineering that depend heavily on matching to find more optimisations for the matching process. An example of such a field is functional programming in which "decision trees" are constructed that are either fast or small [16, 21]. From these trees code is generated that performs the actual matching.

As described in Section 3.2, doing an inverse look-up might be faster in some cases. It should be studied whether this is actually the case and if so, how bit the benefits of doing the inverse look-up are.

As suggested in Section 6.2, the small applications' performance might be extremely quick due to the CPU's cache. This should be confirmed by performing the benchmarks on a machine a much smaller amount of cache so it is impossible that all data fits in the cache.

As discussed in Chapter 6, the pattern optimised matching can possibly be improved by putting more run-time knowledge of the application and the patterns into consideration when determining the order in which the sub-patterns are being matched and which sub-patterns not to match at all.

Finally the benchmarks and surveys performed in this study should be expanded to cover a much larger set of applications and aspects, possibly generated using aspect mining techniques [9], to confirm that results of these small scale benchmarks and surveys apply in general as well.

Bibliography

[1] J.-J. Amor-Iglesias, et al. From pigs to stripes: A travel through Debian. In *Proceedings of the 6th Debian Conference*. Helsinki, FI, 2005.

[2] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] A. J. Bernstein and M. Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[4] R. Bijker, C. Bockisch, and A. Sewe. Optimizing the evaluation of patterns in pointcuts. In *Proceedings of VMIL*. 2010.

[5] S. M. Blackburn, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of OOPSLA*. 2006.

[6] C. Bockisch. An effcient and flexible implementation of aspect-oriented languages. 2009.

[7] E. Bodden, et al. Taming Reflection (Extended version). Technical Report TUD-CS-2010-0066, CASED, 2010.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*. 2002.

[9] M. Ceccato, et al. A qualitative comparison of three aspect mining techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. IEEE Computer Society, Washington, DC, USA, 2005.

[10] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 1998.

[11] J. Gosling, et al. *Java^{TM} Language Specification.* Addison-Wesley, 3rd edition, 2005.

[12] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages.* Ph.D. thesis, Technische Universität Darmstadt, 2006.

[13] M. Haupt and M. Mezini. Micro-measurements for dynamic aspect-oriented systems. In M. Weske and P. Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies*, volume 3263 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2004.

[14] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1), 1996.

[15] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2), 1984.

[16] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming.* ACM, New York, NY, USA, 2001.

[17] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of CC.* 2003.

[18] R. Muschevici, et al. Multiple dispatch in practice. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications.* ACM, New York, NY, USA, 2008.

[19] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development.* ACM, New York, NY, USA, 2002.

[20] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, Inc., New York, NY, USA, 1999.

[21] R. L. Rivest and L. Hyafil. Constructing optimal binary decision trees is NP-complete. In *Information processing letters.* 1975.

[22] P. G. Selinger, et al. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data.* ACM, New York, NY, USA, 1979.

Appendix A

Benchmark data

All benchmarks are made with an Intel® Core^{TM}2 Duo CPU T9400 at 2.53GHz. On a 64 bits Linux (2.6.32 kernel) with OpenJDK 6b18-1.8.1-2.

The sections in the captions of the tables refer to a section in Chapter 6 which corresponds with the data in the table.

Tables that show the aggregation of raw data show the relative minimum, maximum and standard deviation for easier comparison of the stability of the measurements. The applications in tables are sorted by their amount of call-sites, thus TightVNC (abbreviated as TVNC) is the smallest and the Java runtime is the biggest.

For matching benchmarks the average for matching a pattern is taken from the total time it takes to match all 158 patterns. Thus the actual run-time of these benchmarks was 158 times larger, which is why the deviations are generally smaller than the deviations for insertion.

When listing sizes in bytes (B) the "K" stands for 1 024 bytes and "M" for 1 048 576 bytes.

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	241μs	308μs	5.85ms	12.1ms	31.2ms	142ms
Mean	253μs	322μs	5.99ms	12.4ms	31.3ms	142ms
Maximum	299μs	380μs	6.61ms	14.1ms	31.8ms	145ms
Std. dev.	10.4μs	12.4μs	138μs	359μs	103μs	526μs
Norm. min.	0.95	0.96	0.98	0.97	0.99	1.00
Norm. max.	1.18	1.18	1.10	1.14	1.02	1.02
Norm. dev.	0.04	0.04	0.02	0.03	0.00	0.00

Table A.1: Original insertion benchmark (Section 6.2)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	941μs	2.67ms	20.2ms	58.1ms	121ms	479ms
Mean	948μs	2.69ms	20.3ms	58.2ms	122ms	485ms
Maximum	961μs	2.75ms	20.4ms	58.4ms	124ms	491ms
Std. dev.	3.91μs	15.9μs	49.1μs	75.0μs	643μs	3.05ms
Norm. min.	0.99	0.99	1.00	1.00	0.99	1.00
Norm. max.	1.02	1.02	1.01	1.00	1.02	1.01
Norm. dev.	0.01	0.01	0.00	0.00	0.01	0.01

Table A.2: Original matching with original sorting benchmark (Section 6.2)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	1.01ms	2.87ms	21.2ms	57.6ms	144ms	541ms
Mean	1.01ms	2.90ms	21.3ms	57.7ms	145ms	545ms
Maximum	1.03ms	2.96ms	21.4ms	57.9ms	149ms	553ms
Std. dev.	4.80μs	17.1μs	43.5μs	62.0μs	967μs	3.80ms
Norm. min.	0.99	0.99	1.00	1.00	0.99	0.99
Norm. max.	1.02	1.02	1.00	1.00	1.02	1.01
Norm. dev.	0.01	0.01	0.00	0.00	0.01	0.01

Table A.3: Class-first matching with original sorting benchmark (Section 6.3)

Sub-pattern	TNVC	LIAM	ANTLR	FC	Tools	RT	Mean
Methods							
Class	2.68	2.45	2.54	2.42	2.37	2.37	2.47
Name	4.71	4.88	5.70	5.63	5.47	5.63	5.34
Modifiers	0.25	0.23	0.24	0.21	0.27	0.28	0.25
Return type	0.97	1.26	1.08	1.14	1.16	1.13	1.12
Parameters	2.16	1.95	2.07	1.87	2.09	2.11	2.04
Exceptions	2.17	1.97	2.13	1.91	2.14	2.16	2.08
Fields							
Class	1.64	1.36	1.50	1.40	1.39	1.33	1.44
Name	1.38	1.45	1.53	1.43	1.49	1.49	1.46
Modifiers	1.22	1.45	1.23	1.35	1.14	1.37	1.29
Type	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.4: Efficiency factors for sub-patterns (Section 6.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	271μs	1.00ms	5.74ms	15.5ms	39.6ms	175ms
Mean	285μs	1.01ms	5.77ms	15.6ms	39.9ms	176ms
Maximum	290μs	1.04ms	5.81ms	16.0ms	41.2ms	178ms
Std. dev.	3.39μs	7.00μs	15.7μs	98.6μs	403μs	394μs
Norm. min.	0.99	0.99	1.00	0.99	0.99	1.00
Norm. max.	1.05	1.03	1.01	1.03	1.03	1.01
Norm. dev.	0.01	0.01	0.00	0.01	0.01	0.00

Table A.5: Sub-pattern efficiency based matching with original sorting benchmark (Section 6.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	269μs	1.01ms	5.85ms	16.0ms	39.0ms	169ms
Mean	271μs	1.02ms	5.87ms	16.0ms	39.1ms	169ms
Maximum	276μs	1.04ms	5.90ms	16.1ms	39.2ms	170ms
Std. dev.	1.25μs	4.75μs	14.0μs	21.2μs	38.5μs	207μs
Norm. min.	0.99	0.99	1.00	1.00	1.00	0.99
Norm. max.	1.02	1.02	1.01	1.00	1.00	1.02
Norm. dev.	0.00	0.00	0.00	0.00	0.00	0.00

Table A.6: Pattern optimised matching with original sorting benchmark (Section 6.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	691μs	1.24ms	12.6ms	42.3ms	139ms	801ms
Mean	724μs	1.31ms	12.8ms	42.8ms	141ms	803ms
Maximum	774μs	1.42ms	13.3ms	44.0ms	152ms	807ms
Std. dev.	19.0μs	38.2μs	127μs	334μs	2.47ms	1.54ms
Norm. min.	0.95	0.95	0.98	0.99	0.99	1.00
Norm. max.	1.07	1.08	1.04	1.03	1.08	1.01
Norm. dev.	0.03	0.03	0.01	0.01	0.02	0.00

Table A.7: Sorting by class insertion benchmark (Section 6.4.1)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	63.4μs	238μs	1.67ms	5.05ms	11.5ms	44.0ms
Mean	64.8μs	240μs	1.68ms	5.06ms	11.5ms	44.1ms
Maximum	68.3μs	245μs	1.71ms	5.09ms	11.6ms	44.3ms
Std. dev.	0.96μs	1.55μs	9.66μs	10.4μs	16.4μs	65.4μs
Norm. min.	0.98	0.99	0.99	1.00	1.00	1.00
Norm. max.	1.05	1.02	1.02	1.01	1.00	1.00
Norm. dev.	0.01	0.01	0.01	0.00	0.00	0.00

Table A.8: Original matching with data sorted by class name benchmark (Section 6.4.1)

Sub-pattern	TNVC	LIAM	ANTLR	FC	Tools	RT	Mean
Methods							
Class	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Name	3.38	3.42	3.41	4.05	4.00	4.25	3.75
Modifiers	0.52	0.50	0.51	0.44	0.52	0.56	0.51
Return type	1.05	1.32	1.18	1.16	1.17	1.10	1.16
Parameters	1.26	1.13	1.21	1.04	1.16	1.17	1.16
Exceptions	1.26	1.14	1.23	1.05	1.16	1.16	1.17
Fields							
Class	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Name	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Modifiers	1.93	1.93	1.90	1.83	1.52	1.78	1.82
Type	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.9: Efficiency factors for sub-patterns for data sorted by class name (Section 6.4.1)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	59.0μs	215μs	1.48ms	4.46ms	10.6ms	40.9ms
Mean	60.2μs	222μs	1.50ms	4.48ms	10.7ms	41.7ms
Maximum	63.5μs	251μs	1.63ms	4.57ms	10.8ms	43.4ms
Std. dev.	0.91μs	7.96μs	25.3μs	25.4μs	47.9μs	661μs
Norm. min.	0.98	0.97	0.98	0.99	0.99	0.98
Norm. max.	1.05	1.13	1.09	1.02	1.01	1.04
Norm. dev.	0.02	0.04	0.02	0.01	0.00	0.02

Table A.10: Sub-pattern efficiency based matching with data sorted by class name benchmark (Section 6.4.1)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	61.0μs	180μs	1.51ms	4.26ms	10.6ms	40.0ms
Mean	62.7μs	184μs	1.54ms	4.29ms	10.7ms	40.5ms
Maximum	66.3μs	200μs	1.61ms	4.62ms	10.9ms	41.6ms
Std. dev.	0.97μs	3.47μs	22.2μs	44.7μs	68.0μs	384μs
Norm. min.	0.98	0.99	0.99	1.00	1.00	1.00
Norm. max.	1.05	1.02	1.02	1.01	1.00	1.00
Norm. dev.	0.01	0.01	0.01	0.00	0.00	0.00

Table A.11: Pattern optimised matching with data sorted by class name benchmark (Section 6.4.1)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	1.12ms	1.46ms	13.5ms	44.6ms	130ms	851ms
Mean	1.15ms	1.41ms	13.6ms	44.8ms	130ms	853ms
Maximum	1.22ms	1.45ms	14.1ms	45.5ms	132ms	858ms
Std. dev.	25.2μs	27.8μs	13.6μs	204μs	415μs	1.71ms
Norm. min.	0.97	0.96	0.99	0.99	1.00	1.00
Norm. max.	1.06	1.03	1.03	1.02	1.01	1.01
Norm. dev.	0.02	0.02	0.01	0.00	0.00	0.00

Table A.12: Sorting by name insertion benchmark (Section 6.4.2)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	113μs	555μs	2.13ms	6.99ms	16.3ms	80.1ms
Mean	114μs	558μs	2.15ms	7.00ms	16.6ms	81.0ms
Maximum	118μs	570μs	2.20ms	7.03ms	17.1ms	82.4ms
Std. dev.	1.02μs	2.94μs	14.7μs	11.3μs	213μs	586μs
Norm. min.	0.99	0.99	0.99	1.00	0.99	0.99
Norm. max.	1.04	1.03	1.02	1.00	1.03	1.02
Norm. dev.	0.01	0.01	0.01	0.00	0.01	0.01

Table A.13: Original matching with data sorted by name benchmark (Section 6.4.2)

Sub-pattern	TNVC	LIAM	ANTLR	FC	Tools	RT	Mean
Methods							
Class	1.90	1.76	1.93	1.83	1.72	1.67	1.80
Name	0.50	0.45	0.45	0.50	0.46	0.51	0.48
Modifiers	1.01	1.04	1.02	0.96	1.15	1.29	1.08
Return type	1.04	0.98	1.01	0.98	1.03	1.00	1.00
Parameters	0.95	1.00	0.85	0.92	0.94	0.95	0.94
Exceptions	0.96	1.01	0.86	0.93	0.96	0.96	0.95
Fields							
Class	1.13	1.01	1.11	1.06	1.01	0.94	1.04
Name	0.00	0.00	0.00	0.00	0.01	0.00	0.00
Modifiers	1.67	1.86	1.67	1.76	1.51	1.82	1.72
Type	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.14: Efficiency factors for sub-patterns using a data sorted by name (Section 6.4.2)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	125μs	495μs	2.28ms	7.21ms	16.6ms	74.9ms
Mean	128μs	500μs	2.31ms	7.28ms	16.7ms	75.3ms
Maximum	134μs	527μs	2.44ms	7.60ms	16.9ms	75.9ms
Std. dev.	1.78μs	5.67μs	27.6μs	72.8μs	44.2μs	260μs
Norm. min.	0.98	0.99	0.99	0.99	1.00	1.00
Norm. max.	1.05	1.05	1.05	1.04	1.01	1.00
Norm. dev.	0.01	0.01	0.01	0.01	0.00	0.00

Table A.15: Sub-pattern efficiency based matching with data sorted by name benchmark (Section 6.4.2)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	117μs	482μs	2.30ms	7.11ms	16.0ms	75.8ms
Mean	119μs	487μs	2.32ms	7.17ms	16.1ms	77.1ms
Maximum	130μs	501μs	2.38ms	7.37ms	16.2ms	79.3ms
Std. dev.	2.19μs	4.50μs	18.5μs	58.1μs	46.0μs	829μs
Norm. min.	0.98	0.99	0.99	0.99	1.00	0.98
Norm. max.	1.09	1.02	1.03	1.03	1.00	1.03
Norm. dev.	0.02	0.01	0.01	0.01	0.00	0.01

Table A.16: Pattern optimised matching with data sorted by name benchmark (Section 6.4.2)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	910μs	2.07ms	11.7ms	32.2ms	103ms	579ms
Mean	942μs	2.14ms	11.8ms	32.4ms	103ms	580ms
Maximum	980μs	2.19ms	12.2ms	33.1ms	105ms	584ms
Std. dev.	18.3μs	34.5μs	75.2μs	182μs	506μs	1.35ms
Norm. min.	0.97	0.97	0.99	0.99	0.99	1.00
Norm. max.	1.04	1.03	1.03	1.02	1.01	1.01
Norm. dev.	0.02	0.02	0.01	0.01	0.09	0.00

Table A.17: Sorting by name prefix insertion benchmark (Section 6.4.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	131μs	517μs	2.40ms	8.18ms	16.6ms	84.0ms
Mean	132μs	519μs	2.41ms	8.21ms	16.7ms	84.2ms
Maximum	136μs	524μs	2.43ms	8.25ms	17.7ms	84.4ms
Std. dev.	0.87μs	1.76μs	6.32μs	16.1μs	24.8μs	92.3μs
Norm. min.	0.99	1.00	1.00	1.00	1.00	1.00
Norm. max.	1.02	1.01	1.01	1.00	1.00	1.00
Norm. dev.	0.01	0.00	0.00	0.00	0.00	0.00

Table A.18: Original matching with data sorted by name prefix benchmark (Section 6.4.3)

Sub-pattern	TNVC	LIAM	ANTLR	FC	Tools	RT	Mean
Methods							
Class	2.06	1.88	2.00	1.99	1.72	1.80	1.91
Name	1.02	0.88	0.84	1.27	0.87	1.09	1.00
Modifiers	0.92	1.05	0.97	0.85	1.10	1.17	1.01
Return type	0.95	0.94	0.96	0.89	1.00	0.96	0.95
Parameters	0.92	1.04	0.89	0.96	0.97	0.98	0.96
Exceptions	0.93	1.06	0.90	0.97	0.98	0.97	0.97
Fields							
Class	1.13	1.01	1.11	1.06	1.01	0.94	1.04
Name	0.00	0.00	0.00	0.00	0.01	0.00	0.00
Modifiers	1.67	1.86	1.67	1.76	1.51	1.82	1.72
Type	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.19: Efficiency factors for sub-patterns for data sorted by name prefix (Section 6.4.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	120μs	486μs	2.42ms	7.58ms	15.5ms	75.7ms
Mean	124μs	488μs	2.43ms	7.59ms	15.5ms	75.9ms
Maximum	134μs	493μs	2.45ms	7.61ms	15.6ms	76.1ms
Std. dev.	0.77μs	1.72μs	6.17μs	11.4μs	22.1μs	83.7μs
Norm. min.	0.99	1.00	1.00	1.00	1.00	1.00
Norm. max.	1.02	1.01	1.01	1.00	1.00	1.00
Norm. dev.	0.01	0.00	0.00	0.00	0.00	0.00

Table A.20: Sub-pattern efficiency based matching with data sorted by name prefix benchmark (Section 6.4.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Minimum	117μs	453μs	2.26ms	7.09ms	15.3ms	77.3ms
Mean	117μs	455μs	2.27ms	7.11ms	15.3ms	77.5ms
Maximum	120μs	462μs	2.29ms	7.14ms	15.3ms	77.7ms
Std. dev.	0.74μs	2.02μs	6.11μs	11.9μs	20.2μs	829μs
Norm. min.	0.99	1.00	1.00	1.00	1.00	1.00
Norm. max.	1.02	1.02	1.01	1.00	1.00	1.00
Norm. dev.	0.01	0.00	0.00	0.00	0.00	0.00

Table A.21: Pattern optimised matching with data sorted by name prefix benchmark (Section 6.4.3)

	TNVC	LIAM	ANTLR	FreeCol	Tools	Runtime
Total size						
Base	162 KB	320 KB	1.47 MB	2.24 MB	5.72 MB	15.6 MB
Class	233 KB	482 KB	2.19 MB	4.87 MB	12.0 MB	50.6 MB
Name	396 KB	624 KB	2.46 MB	6.15 MB	14.5 MB	60.7 MB
Prefix	301 KB	507 KB	2.25 MB	4.80 MB	11.3 MB	47.3 MB
Size per call-site						
Base	48.0 B	83.7 B	44.1 B	31.5 B	34.0 B	21.6 B
Class	69.0 B	126 B	65.6 B	68.5 B	71.1 B	70.1 B
Name	118 B	163 B	73.5 B	86.5 B	86.5 B	84.2 B
Prefix	89.4 B	132 B	67.4 B	67.5 B	67.1 B	65.6 B

Table A.22: Memory requirements (Section 6.5)

Appendix B

Benchmark result visualisation

The figures in this appendix visualise the performance of the different methods as described in Chapter 6, specifically Section 6.6.

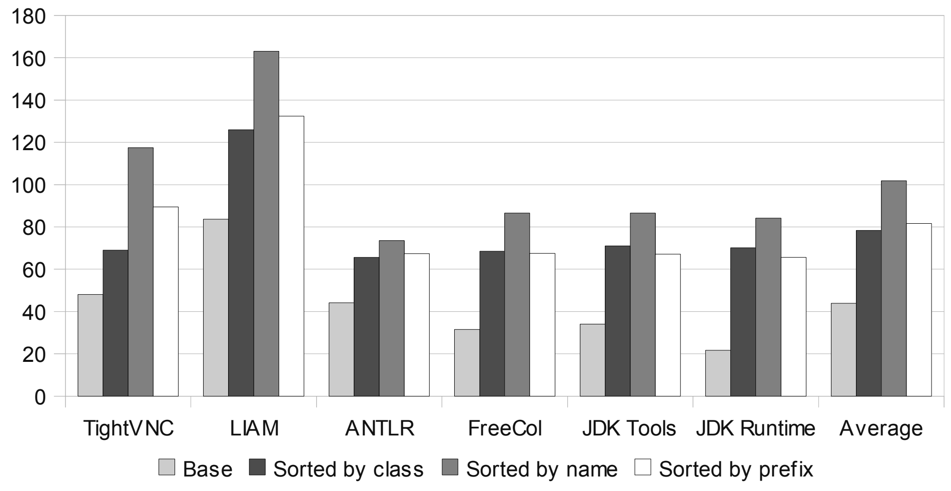

Figure B.1: Memory requirements per call-site in bytes

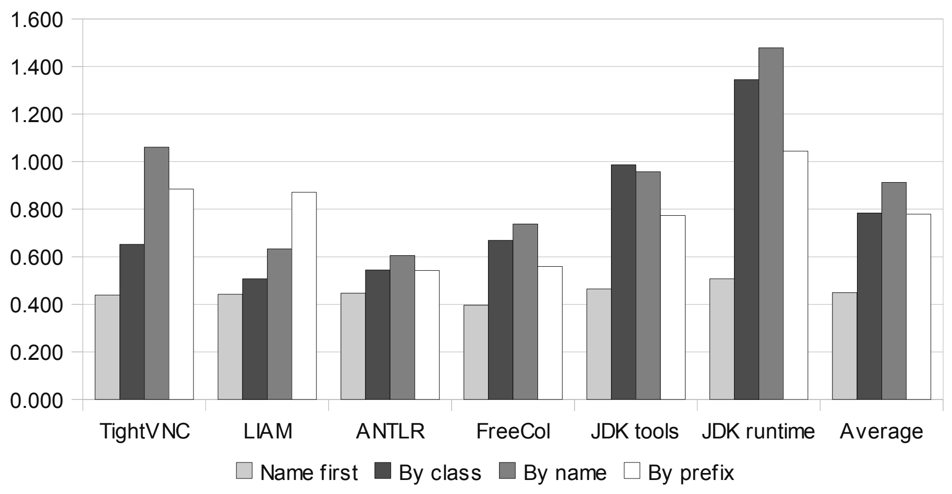

Figure B.2: Relative times for insertion and one sub-pattern efficiency based pattern matching for differently sorted collections

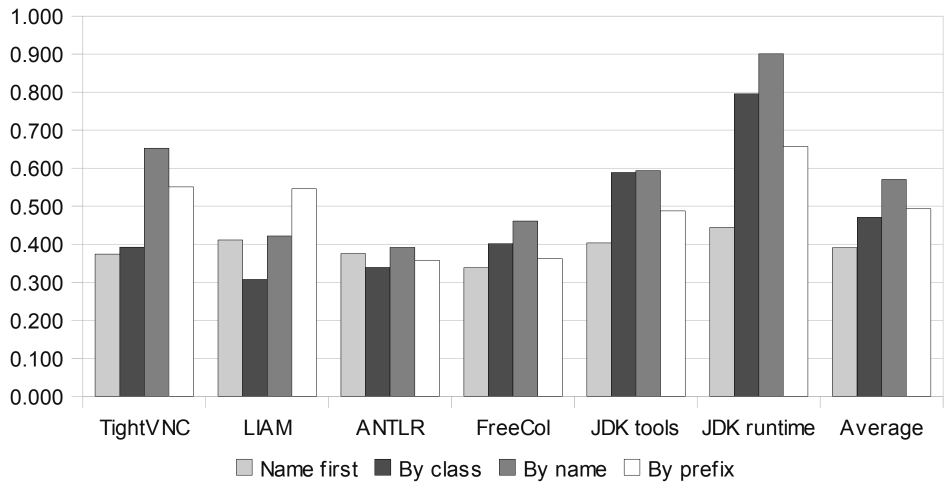

Figure B.3: Relative times for insertion and two sub-pattern efficiency based pattern matchings for differently sorted collections

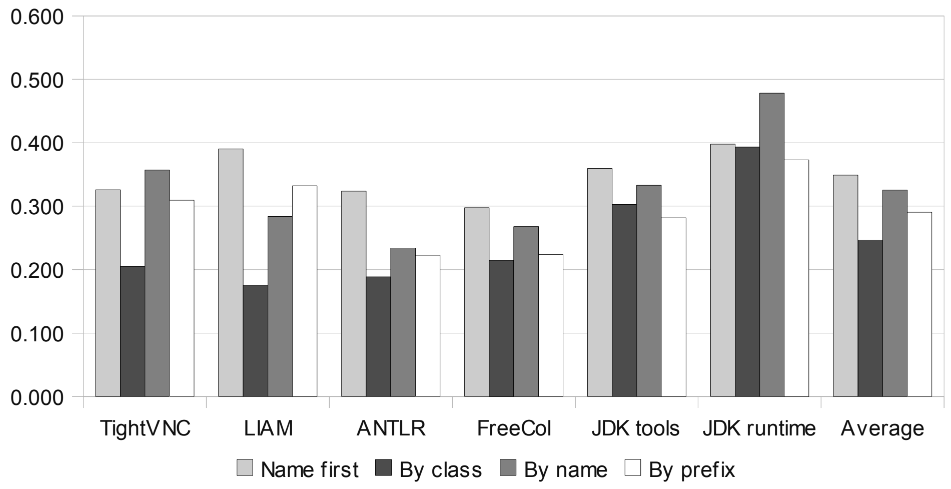

Figure B.4: Relative times for insertion and five sub-pattern efficiency based pattern matchings for differently sorted collections

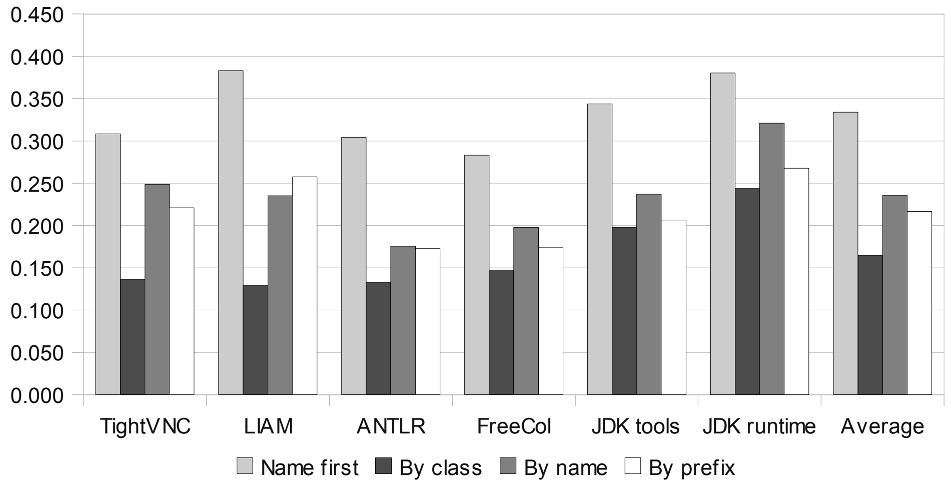

Figure B.5: Relative times for insertion and ten sub-pattern efficiency based pattern matchings for differently sorted collections

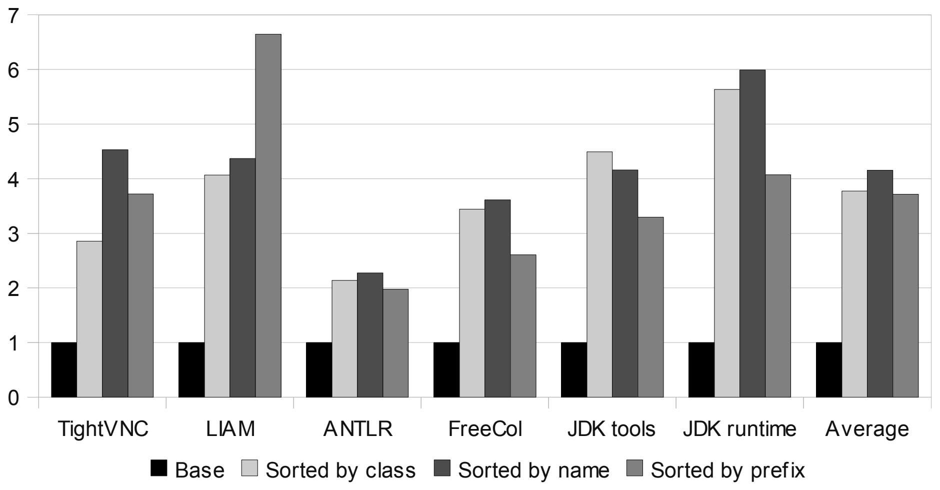

Figure B.6: Relative times for insertion into differently sorted collections

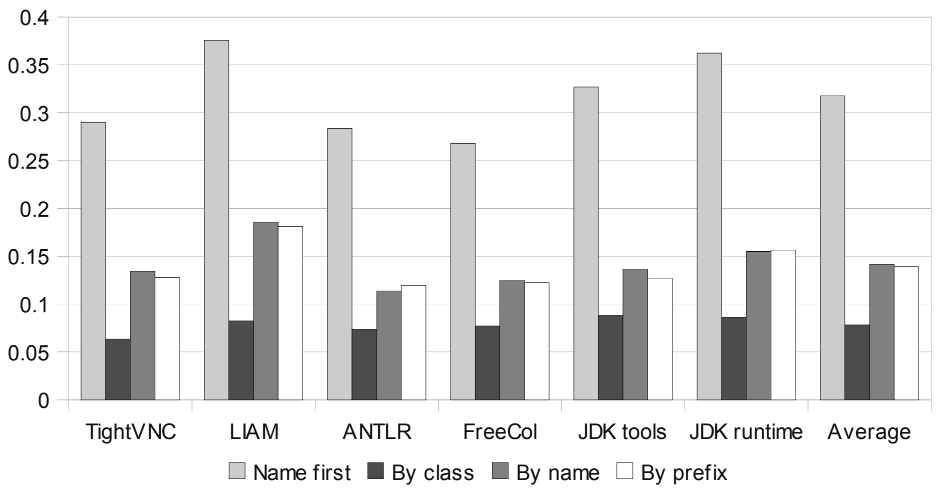

Figure B.7: Relative times for sub-pattern efficiency based matching for differently sorted collections