JavaSet – extending Java by persistent sets

Markus Schordan, Harald Kosch and Laszló Böszörményi
Institute of Information Technology, University of Klagenfurt,
Klagenfurt, Austria
email: markuss,harald,laszlo@ifi.uni-klu.ac.at
tel: 0043-0463-2700-513 ; fax: 0043-0463-2700-505

Abstract

We introduce an extension of the programming language Java (called JavaSet) by polymorph sets of objects, which can be both transient or persistent. JavaSet provides transparent persistence, i.e. the handling of persistence is purely declarative. The extension is type-safe and upwards compatible. Due to the integration into the language the compilation system can take advantage of database optimization and parallelization techniques. The compilation process incorporates the mapping to an efficient object algebra, the optimization of set expressions and the generation of an execution plan.

1 Introduction

General-purpose programming languages still support the "empty-computer model", i.e. they make the (rather unconscious) assumption that programs are loaded into an empty computer, in order to make there some computations. After having loaded the program, the code is available, however, the program has to make major efforts to get its data into main memory at proper place in proper structure. In reality, most of our computations are done not in an empty, but rather in a huge context, which is typically persistent. The more surprising is the fact that general-purpose programming languages typically have no notion of persistence.

One of the reasons for the lack of persistence is the lack of proper abstractions. Actually all programming languages support arrays, which are excellent abstractions for matrix-based mathematical algorithms but definitely bad for huge amount of persistent data. For latter we need a kind of associative store in which we can search efficiently driven by content and not by index (as in an array). For this, the set model has always been used by databases. The concept of sets is a well-understood mathematical notion. Beside the basic set operations, query languages define powerful search operations based on sets. Sets are per definition unordered, therefore, if we apply an operation on all elements of a set, we are not allowed to make any assumption about the order of the execution. This feature make sets inherently parallel data structures.

These considerations led us to extend Java[1] - as a clean and type-safe general-purpose programming language - with the notion of polymorphic object sets. They can be both transient or persistent. The extended language (called JavaSet) is upwards compatible with Java. Due to the integration into the language object sets are type safe and set expressions (including complex queries) can be automatically optimized and parallelized in a relatively easy way (compared e.g. to nested-loop parallelization in parallel programming languages).

[1] Java^{TM}

JavaSet is based on previous work [1] in the course of which Modula-3 was extended by persistent and distributed polymorphic object sets. JavaSet has much more functionality regarding persistence and optimization. Distribution is subject for further study.

Next section 2 discusses related works. Section 3 introduces the specification of the language. In section 4 the compiler and its strong cooperation with the optimizer will be shown. The connection between persistent variables and the corresponding persistent stores will be also described. Section 5 concludes the paper and points to future works.

2 Related works

Persistence in object-oriented programming languages gains more and more interest (for persistence in Java there exists even a special workshop series [2]). Persistence is in most cases understood as file persistence, i.e. objects are stored in files with the help of a serialization function [3]. File persistence can be easily implemented in Java, because the serialization function is a standard component from the starting of JDK. However, such simple persistence allows obviously only very restrictive object operations and is not appropriate for the management of large amounts of persistent objects. Recently, more sophisticated approaches to persistent Java object stores have been proposed. Some approaches as the PSE [4] obtain a faster access to the persistent objects by using smaller serialization unites, another prototype (PJama [5]) extends the Interpreter JVM in the scope to allow a more flexible access to the persistent objects. None of these prototypes provide querying concepts.

Persistence and more general full database functionality (inclusive query optimization) in an object-oriented programming language can be achieved by the means of a combination of a database- with a programming language. The most popular example is the OQL C++ binding [6] (designed by the Object Data Management Group ODMG) implemented e.g. in O_2 [7] or Poet [8]. This combination supplies to the programmer full database functionality, however the resulting language volume is quite large [3]. Furthermore, the programmer needs good database knowledges to work efficiently with this combined database- and programming languages. Recently, the ODMG published a first design proposal for a OQL Java binding (see chapter 7 of the book [6]). This binding supports the class hierarchies, objects and references. Up-to date this proposal is not maturated, but growing interest in such kind of binding is signalized by industrials. However, as in the case of the OQL C++ binding, we doubt if the complex binding concepts can be used by non-database specialist.

Database functionality in programming language can also be provided by the JDBC/ODBC Application Interface (API) [9]. Querying a database is done through the passing of a string (contains the query command literally) to a Java method. This obvious mismatch between programming- and database language renders difficult the programming of persistent sets.

In this context, some authors have recognized that the concept of sets offer a high abstraction to process persistent objects without augmenting the volume of a language. Several approaches to integrate persistent sets into a programming language have been yet undertaken. First propositions are the (parallel) database programming languages FAD [10] and SVP [11]. These languages are very restricted compared to modern object-oriented programming languages. Later approaches, like ParSet [12] implemented on the top of the Shore Object Store do not offer direct language support.

A part from the attempt to integrate persistence in object-oriented programming languages set-oriented languages have been designed : SetL [13] operations on sets are similar to those proposed in JavaSet, however they operate only on transient sets. The StarSet [14] programming language supports persistent sets, but the set access is however not transparent to the programmer, as for loading and writing sets the programmer has to employ special functions.

3 Language definition

The integration of sets in JDK1.2 shows that this programming concept is important. With JavaSet we introduce a more sophisticated concept than that proposed in JDK1.2. JavaSet contains a powerful select-expression which allows the access to multiple sets in a database query like way based on an elegant mathematical formalism. Furthermore, a complete palette of set operations are introduced allowing a more flexible dealing with the notion of sets than in JDK. Sets in JDK1.2 are only transient, JavaSet extends to persistent sets At the same time, we render transparent the set reading and writing, as the connection to the concrete persistent store. Querying persistent sets is made very effective through an algebraic optimization of select-expressions. Therefore, the efficient utilization of persistent data in an object-oriented programming language like Java is made possible.

Set Type. All the elements of a set have the same type, called the element type of the set. If the element type of a set is T, then the type of the set itself is written $T\{\}$. We call a variable of type set a set variable. A set's size is not part of its type. All the members from class `Object` are inherited and additionally the methods `size()` and `isEmpty()` are provided for transient and persistent sets. The element type must be of type reference type. Sets are dynamically created objects and can be assigned variables of type `Object`.

Set Variables. A variable of set type holds a reference to an object. Declaring a variable of set type does not create a set object or allocate any space for set elements.

Set Creation. A set is created by a set creation expression, a set initializer or a select expression.

A set initializer may be specified in a declaration, creating a set and providing some initial values. In JavaSet an additional information can be given to define whether the set is bound to a persistent store or is kept transient.

(1) $\{\ Variableinitializers_{opt}\texttt{,}_{opt}\ \}$ `@` $PSIdentifier$
(2) $\{\ Variableinitializers_{opt}\texttt{,}_{opt}\ \}$

$PSIdentifier$ `(1)` is a string which uniquely identifies the external representation of the persistent set. When specified, the set is bound to a persistent store. This information is used during compilation to optimize set expressions. By the object creation, the connection to the persistent store is established if it has not yet been connected. If the set does not exist in the persistent store, it is created and initialized with the given `Variableinitializers`. If the set already exists in the database it is overwritten with the new initialization. Existing sets in the persistent store, i.e. sets created by a previous program run or more general by other programs, can be accessed by a variable declaration without initialization. In this case the persistent set is loaded and the set variable is bound to this set.

```
Student stud=new Student("John");
Person{} persons={new Person("Mary"), stud} @ "db1:pers1";
```

In the above example the variable `persons` is bound to the set `db1:pers1` in the persistent store. If the set has not yet been created in the persistent store, a new polymorph set is created consisting of two elements. For all elements of a set holds that the type is assignment compatible to the element type of the set (see figure 1). If another program want to access this created set (identified by the string "db1:pers1") in the persistent store, it should declare a variable without an initialization (otherwise it would overwrite the set) as in the following example declaration : `Person{} persons2= @ "db1:pers1";`.

Assignment. When a set is assigned to a set variable, the reference to the set is overwritten. But the information regarding the location of the set is not overwritten because it is bound to the variable. Reference semantics hold for sets but not for persistence. Thus, new sets can be easily created and made persistent. Also the modification of persistent sets can be achieved by expressions and assignments.

```
Student{} stud3={} @ "db1:students1";
Student{} stud4;
stud4=stud3;
```

A set with the element type `Student` is created and initialized with an empty set. The connection to the persistent store `db1` is established. This object is assigned to the variable `stud3`. If another set is assigned to `stud3` then this set becomes persistent and overwrites the previously bound set. When the assignement `stud4=stud3` has been executed, then the variable `stud4` can be used to access all objects that are referenced by `stud3`. Thus, using an assignment, sets can be made transient or persistent.

A set variable stands for more than just a simple reference. More formally we can define a tuple $< L, R >$ which represents the reference of a set variable to the persistent store and to its root, where :

L is either a reference to a persistent store or it is left null and the set is transient.

R is the root of the set. This reference can be modified and provides reference semantics for the life time of the variable.

Using this notation, the assignment and its semantics in JavaSet are defined as follows :

Variable Initialization. When a set variable is initialized, then the L (location) is set to the persistent store and the root R refers to the elements of the set. If no `PSIdentifier` is specified, then L is left null and the set is transient.

Assignment. An assignment only effects R, the reference to the elements of a set. Let us assume two variables v_1 and v_2 have been declared and initialized. We have two tuples $< L_1, R_1 >$ and $< L_2, R_2 >$. After an assignment $v1 = v2$ has been executed, R_1 has been overwritten by R_2 and v_1 is $< L_1, R_2 >$.

Set Operations and the Extended Set Creation Expression. Additionally to the set creation expression using `new` we provide an extended set creation expression. With this expression we can add and remove elements easily from a set. The expression is similar to the syntax of the set initialization expression, but must involve at least one element. The result type is the type of the most general type of the given elements. The operations set union, intersection and difference are binary expressions that only operate on exactly two set values or as for '+', '*' and '-' semantics are already defined in Java.

```
persons=persons+{var}+stud3;
persons+={var}+stud3;
```

`{var}` creates a set consisting of one element. A new set union consisting of the element referenced by `var`, `stud3` and `persons` is created and assigned to `persons`. Set operations are optimized to ensure that a minimum of temporary intermediate sets are created to evaluate set expressions. In the second version of the example above the object `var` and the set `stud3` are added to the set `persons`. It is not necessary to create any temporary sets. Hence, set operations can be written as assignments and simple set operations but optimizations avoid the inefficient creation of unnecessary intermediate sets.

The type of the result for set union is the more general one, for intersection the more specific one, and for difference the type of the first operand.

```
class Lecture {
   public String title; ...
}
class Person {
   protected String name;
   protected Address address;
   Person(String name, String adr){...}
   public Address adr() {...}...
}

Person{} persons = {} @ "db1:persons1";
Student{} stud1 = {} @ "db1:students1";
Student{} stud2 = {} @ "db1:students2";
```

```
class Student extends Person {
   public Lecture lect;
   Student(String name){...} ...
}
class Pair {
   private Person p1,p2;
   Pair(Person p1, Person p2) {
   ...} ...
}
```

Figure 1: Class Definition and Program Fragment.

Selection. The central operation in JavaSet is the select-expression. It allows one to express queries similar to OQL queries [6]. The select-expression takes a number of sets and a boolean expression as input and produces a new set. The new set can be generated in two ways : object-creating or object-conserving. Sets are written similar to the mathematical formulation. The general form is :

$$\{\mathit{Selector} \mid id_1 \texttt{ in } set_1, \ldots, id_n \texttt{ in } set_n \texttt{ :: } \mathit{expression}\}$$

The identifiers B_i $(1 \leq i \leq n)$ must be distinct, while the same set may occur more than once. `Selector` can be an identifier, a constructor of a class or a path expression. An object is added to the result set when the boolean condition *expression* holds for any objects of the given sets. The type of *Selector* is the element type of the result set of the select-expression. The expressions set_i are arbitrary set expressions. Hence, select-expressions can be nested.

```
(1) {p|p in persons :: p.adr()==null};
(2) {st1| st1 in stud1, st2 in stud2 :: st1.adr().equals(st2.adr())};
(3) {Pair(st1,st2)| st1 in stud1, st2 in stud2 :: st1.adr().equals(st2.adr())};
```

In example 1 all persons without a valid address are selected. The semi-join (2) selects all students of the set `stud1` that have the same address as any of the students from `stud2`. The result is of type `Student{}`. The join-expression (3) creates all pairs of students from two input sets having the same address. The selector is the constructor of the class `Pair`. The result is a set of type `Pair{}`.

Foreach.

We extend Java by an additional loop statement which applies a statement (sequence) on each member of a set.

`foreach (` *Identifier* `in` *SetExpression* `)` *Statement*

Each element of the result set of *SetExpression* is bound to the identifier and *Statement*[2] is executed. The order of these bindings is not fixed as a set is an unordered collection of elements. The scope of *Identifier* is the foreach statement itself.

Miscellaneous Expressions. Additionally we provide universal and existential quantifiers and an expression for testing the membership of an object. Although these expressions can be formulated by other JavaSet expressions as well, we integrated them into the language definition because their explicit appearing allows the application of more efficient optimizations.

(1) `all` *Identifier* `in` *SetExpression* `::` *BoolExpression*
(2) `any` *Identifier* `in` *SetExpression* `::` *BoolExpression*
(3) *Expression* `in` *SetExpression*

SetExpression is an expression that must be of type set type, *BoolExpression* must be of type boolean. The result type of all three expressions is boolean. The universal quantifier expression `(1)` checks whether all objects of the result set of *SetExpression* satisfy a condition specified by *BoolExpression*; analogous for the existential quantifier and the condition must hold for at least one element. The test for the membership[3] `(3)` of an object is true if the object that is the result of *Expression* is contained in the set specified by *SetExpression*.

4 Compilation

JavaSet is a superset of Java. Pure Java expressions, set operations and the select-expressions will be compiled separately from each other (see figure 2). The reason is to achieve a compile module granularity which is the best for optimization. The set operations are transformed to a normal-form which is optimized according to the well-known mathematical properties. The select-expression is translated into an object-algebra expression and then optimized with the support of a cost model (see section 4.1 on the next page).

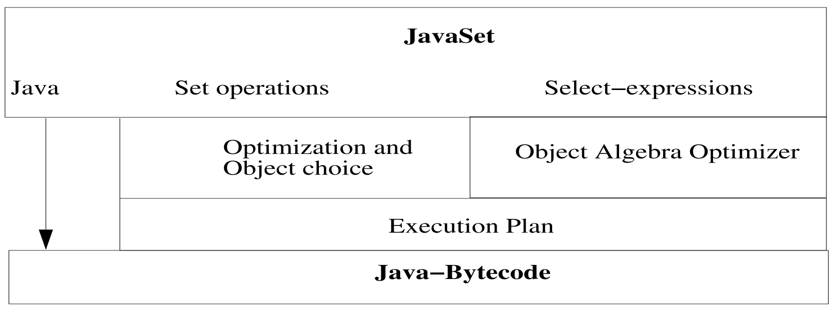

Figure 2: Compilation of a JavaSet program.

The optimized set operations and select-expressions are integrated in the generation of an execution plan. This execution plan determines the order in which the different optimized expressions should be processed. The order is optimized according to a cost model which relies on the sizes of the processed sets. However, if the ratio of the sizes differ at run-time from those estimated at compile-time (especially over a long period of time), then it is necessary to adapt the operator ordering at run-time (see 4.2 on page 8).

[2] A *Statement* can be a *Block*; analogous to the definition of `for` in Java.
[3] The expression is a shorthand for `!({`*Expression*`} *` *SetExpression*`).empty()`

4.1 Optimization

Our optimization approach builds on an algebraic optimization technique [15]. The select-expression is translated to an algebraic expression over a well-defined object-algebra which operators represent atomic execution units on the sets. The most important operators are : the *Join* and the *Selection* (similar to relational database operators), the implicit Object-Join *OJoin* which operates on unique OIDs, moreover the *Flatten* which flattens set of sets and finally the *Mat* which materializes one component of a path expression.

The select-expression is translated by the compiler to a processing tree : *The leaves of a processing tree represent the sets that participate in the select-expression and intermediate nodes model operators. These latters receive their input intermediate sets via the incoming edges and send the result sets through the outgoing edge to the next operation. The root of the tree produces the result of the whole select-expression.*

The processing tree is an ideal abstraction to represent the order of the object-algebra operators. The optimizers task is now to find the best order according to a cost model. The cost model we chose bases on the sizes of the intermediate sets, i.e. on the selectivity of the different operators. The lowest-cost processing tree is that which processes the lessest set elements. The optimizer utilizes a search strategy to find the lowest-cost processing tree. Several approaches have been proposed in related works. For a small number of participating sets, the dynamic programming approach [16] works the best, it delivers under our special requirements always the lowest-cost processing tree. For more complex select-expression, heuristics or randomizes search [17] must be applied. These methods always seek for a good quality, instead of the best processing tree which would be too time-consuming.

Consider as example the following select-expression which returns all students `st1` from `stud1` such that there exists students `st2` in the set `stud2` that have the same address as `st1` students and are registrated in a lecture with the title "Hardware" (see the class definition of figure 1 on page 5) :

```
{st1 | st1 in stud1, st2 in stud2, v in st2.lect ::
st1.adr().equals(st2.adr()) && v.title.equals("Hardware") }
```

Figure 3 shows two possible processing trees for the example selection-expression. Left hand displays an execution strategy which requires the Ojoin of the result of the Flatten operator (it flatten the path expression `st2.lect()`) and the Join operator. The processing tree on the right hand displays a more efficient execution strategy which leaves out the OJoin.

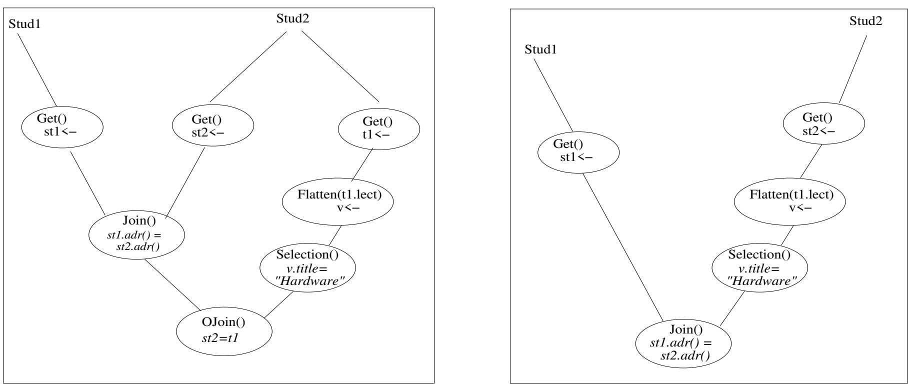

Figure 3: Two possible processing trees for the example select-expression.

The processing tree is integrated together with the optimized set operations into an execution plan (see again the figure 2 on the preceding page). The execution plan is implemented as a data-structure containing all relevant informations of the sets in order to be interpreted at run-time. Relevant informations are beside the name, type, operator orderings. . . , also the estimated size of the generated intermediate sets. The latter is verified at run-time. If significant difference to the real sizes is detected, a new operator ordering must be computed.

4.2 Connection to the persistent store

The interpretation of the execution plan bases on a physical set interface consisting of operators working on the physical implementation of the sets. The interface definition is independent from the underlying persistent store. It contains methods for the management of all existing sets and methods for their manipulation and querying.

Before and during the interpretation of the execution plan at run-time, a control of its validity is done. This control compares the ratio of the estimated set sizes against the real ones. In the case of significant differences, a re-optimization of the operator ordering is triggered. This re-optimization will be carried out by a rule-based transformation in order to get a better plan within a short time. This approach justifies as the processing tree computed at compile-time is probably still a good-quality execution strategy.

The interpretation of the execution plan could be simplified regarded as step-by-step method calls to the physical set interface. Typical employed methods are the opening of the connection to the persistent store, the insertion of elements in a set, the manipulation operators of the sets and the closing of the connection.

Until now two persistent object stores have been utilized to implement the physical operator interface. First, Pjama [5] a persistent object store developed in a cooperation of the university Glasgow with SUN and second PSE [4] from ObjectStore. Moreover a simple implementation based on the JDK serialization is made available.

5 Conclusions and further work

It was shown that with the extension of a general-purpose programming language (such as Java) by proper abstractions (such as sets of objects and the related operations) the access to persistent data can be made entirely transparent, and still efficient. It was also demonstrated that by generating appropriate, dynamically adaptable data-structures during the compilation, we can take advantage of optimization techniques known from the database technology.

The maintenance of the connection between the persistent variables and the corresponding stores must be further investigated. Our last goal is to free the programmer from all concerns of persistence, except the declaration of a variable as persistent. This needs the introduction of a transaction concept, which is still subject of further study.

Sets should be also distributable over a number of nodes. At the time being we hope that we can achieve this by a general interpretation of the *PSIdentifier*. This must be, however, still investigated in detail.

References

[1] L. Böszörményi and K.-H. Eder. M3set – a language for handling of distributed and persistent sets of objects. *Parallel Computing*, 22(1):1913–1925, January 1997.

[2] M. Jordan and M. Atkinson. First International Workshop on Persistence and Java. Technical Report TR 96-58, Sun Microsystems Laboratories, September 1996.

[3] N. Paton, R. Cooper, H. Williams, and P. Trinder. *Database Programming Languages*. Prentice Hall, London, GB, 1996.

[4] G. Landis, C. Lamb, T. Blackman, S. Haradhvala, M. Noyes, and D. Weinreb. Objectstore PSE: A Persistent Storage Engine for Java. In *Proceedings of the 1st International Workshop on Persistence for Java*, Glasgow, Scotland, September 1996. Sun Microsystems, TR 96-58.

[5] M.P Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *Sigmod Records*, 25(4):68–75, December 1996.

[6] R. G. G. Cattell. *The Object Database Standard : ODMG 2.0*. Morgan Kaufmann, 1997.

[7] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System, the Story of O_2*. Morgan Kaufmann, 1992.

[8] Poet Software cooperation. Poet object oriented database. http://www.poet.com/, 1998.

[9] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java: A Tutorial and Annotated Reference Manual*. Addison-Wesley, 1997.

[10] Patrick Valduriez. Parallel database systems: open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165, 1993.

[11] D.S. Parker, E. Simon, and P. Valduriez. SVP - a Model Capturing Sets, Streams, and Parallelism. In *Proceedings of the International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, August 1992.

[12] D. DeWitt, J. Naughton, J. Shafer, and Sh. Venkataram. Parallelizing OODBMS traversals : A performance evaluation. *Very Large Databases Journal*, 5(1):3–18, 1996.

[13] J. Schwartz, R.B.K Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets - An Introduction to SetL*. Springer, 1986.

[14] M. Gilula. *The Set Model for Database and Information Systems*. Addison-Wesley, 1994.

[15] L. Brunie, H. Kosch, and W. Wohner. From the modeling of parallel relational query processing to query optimization and simulation. *Parallel Processing Letters*, 8(1):2–14, March 1998.

[16] R.S.G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proceedings of the International Conference on Very Large Data Bases*, pages 429–445, Dublin, Ireland, August 1993.

[17] L. Brunie and H. Kosch. Optimizing complex decision support queries for parallel execution. In *International Conference of PDPTA 97*, pages 1123–1133, Las Vegas, USA, July 1997. CSREA Press.