A Backend Extension Mechanism for PQL/Java with Free Run-Time Optimisation

Hilmar Ackermann[1], Christoph Reichenbach[1], Christian Müller[1], and Yannis Smaragdakis[2]

[1] Goethe University Frankfurt
{hilmar.ackermann,creichen}@gmail.com
[2] University of Athens
yannis@smaragd.org

Abstract. In many data processing tasks, declarative query programming offers substantial benefit over manual data analysis: the query processors found in declarative systems can use powerful algorithms such as query planning to choose high-level execution strategies during compilation. However, the principal downside of such languages is that their primitives must be carefully curated, to allow the query planner to correctly estimate their overhead. In this paper, we examine this challenge in one such system, PQL/Java. PQL/Java adds a powerful declarative query language to Java to enable and automatically parallelise queries over the Java heap. In the past, the language has not provided any support for custom user-designed datatypes, as such support requires complex interactions with its query planner and backend.
We examine PQL/Java and its intermediate language in detail and describe a new system that simplifies PQL/Java extensions. This system provides a language that permits users to add new primitives with arbitrary Java computations, and new rewriting rules for optimisation. Our system automatically stages compilation and exploits constant information for dead code elimination and type specialisation. We have re-written our PQL/Java backend in our extension language, enabling dynamic and staged compilation.
We demonstrate the effectiveness of our extension language in several case studies, including the efficient integration of SQL queries, and by analysing the run-time performance of our rewritten prototype backend.

1 Introduction

Modern CPUs are equipped with increasingly many CPU cores. Consequently, parallel execution is becoming more and more important as a means for higher software performance. However, traditional general-purpose mechanisms for parallel programming (such as threads and locks) come with complex semantics [17] and may increase code size substantially, raising the risk of program bugs. Hence, modern languages are beginning to provide language facilities that simplify common parallel patterns.

Our past work, PQL/Java [14], presented one such system, implemented as a language extension that adds a Parallel Query Language (PQL) on top of Java. PQL/Java provides easy access to embarrassingly parallel computations (computations that can be completed in constant time with enough CPU cores) and *fold*-like reductions, both constructing and querying Java containers. Unlike other parallel language extensions such as Java 8 Streams and Parallel LINQ, PQL/Java *automatically* decides which part of a query to parallelise, and how.

Consider a simple example: a user wishes to compute the intersection of two hash sets, `s1` and `s2`, and at the same time eliminate all set elements `e` that have some property, such as `e.x < 0`. In Java 8 streams, such a computation can be parallelised with a parallel stream and a filter, as follows:

```
s1.parallelStream()
  .filter(e -> s2.contains(e) && e.x >= 0).collect(Collectors.toSet());
```

This prescribes the following execution: Java will iterate over `s1`, possibly in parallel (after partitioning the set), and for each set element (a) check if it is also contained in `s2`, and, if so, (b) also check if it satisfies the filtering condition. All elements that pass are then collected in an output set.

The same intent can be expressed in PQL as the following:

```
query(Set.contains(e)):
     s1.contains(e) && s2.contains(e) && e.x >= 0;
```

PQL, unlike Java Streams, *does not prescribe an execution strategy* for the above. The PQL/Java system may choose to evaluate the above query exactly as in the Java Streams example, or it may choose to test `e.x >= 0` before `s2.contains(e)` (since the former will typically be much faster and will eliminate some of the latter checks), it may choose to iterate over `s2` instead of `s1`, and it may even choose to execute the query sequentially if it predicts that the overhead for parallel execution would be too high.

PQL can thus perform complex strategic optimisations over parallel queries that are beyond the scope of other parallel language extensions. Partly this is due to more restricted semantics of operations (e.g., guarantee of no side-effects, which allows the order of two conditions to be exchanged).

However, even within a more restricted semantic framework, the ability to optimise comes at a price:

– PQL must be aware of different *execution strategies* for language constructs. For our example above, it must be aware both of the *iteration* strategy and of the *is-contained-check* strategy.
– PQL requires a *cost model* for each execution strategy in order to be able to choose between different implementation alternatives.

For example, if a user wishes to query a custom matrix datatype, PQL will not be able to help, as PQL has no built-in knowledge about this type. The user can at best convert the datatype, but support for custom queries (such as ‘sum up all matrix elements for all rows at column 2’) or for parallelism would require changes to the compiler.

We have therefore designed an *extension specification language* PQL-ESL that simplifies the task of extending PQL, to allow programmers to easily add support for their own datatypes and computations. In addition to providing PQL-ESL for language extensions, we have re-implemented the PQL/Java backend in PQL-ESL. This has allowed us to perform additional optimisations, particularly *run-time query optimisation*.

Run-time query optimisation applies common ideas from staged execution to query processing: we re-optimise queries as new information becomes available. In our earlier example, the dynamic execution system may determine the sizes of `s1` and `s2` before deciding which execution strategy to apply. This is critical for efficient execution: recall that we iterate over one set and check for containment in the other. Iteration is $O(n)$ over the size of the set, but containment checks are $O(1)$ for hash sets. Iterating over the smaller set therefore provides substantial performance benefits.

Our contributions in this paper are as follows:

– We describe PQL-ESL, an extension specification language for our parallel declarative query programming language PQL/Java. PQL-ESL allows programmers to provide compact, human-readable implementations of execution strategies, while exploiting static and dynamic information about the program (static and dynamic types of query parameters).
– We show how PQL-ESL can be used to facilitate run-time query optimisation in PQL/Java.
– We describe an implementation of PQL mostly in PQL-ESL, and compare the performance of the PQL/Java system with and without PQL-ESL.
– We provide five case studies to show how PQL-ESL allows users to quickly extend PQL with new primitives, including an SQL connector.

Section 2 provides background to our PQL/Java system and summarises relevant aspects, with a focus on our intermediate language. Section 3 then describes our extension specification language, and Section 4 describes how we process and compile the language. Section 5 evaluates our new backend. Section 6 discusses related work, and Section 7 concludes.

2 Background

PQL/Java consists of several layers: The query language PQL itself, which we summarise in Section 2.1, the intermediate language PQIL, basis for our optimisations, which we describe in section 2.2, and the PQL runtime, which supports all of the above. We expand the description of our pre-existing work from [14] in this section before discussing our new extension mechanism in Section 3.

2.1 PQL

With parallelism becoming increasingly important, we designed the Parallel Query Language (PQL) to guarantee that everything written in the language can be parallelised effectively. As we learned from the research area of "Descriptive Complexity" [12], the complexity class that most closely corresponds to our notion of 'embarrassingly parallel' problems matches first-order logic over finite structures. We thus designed PQL on top of first-order logic, with an extension to support *fold*-like reductions (which are not embarrassingly parallel, but are present in most practical parallel frameworks, such as Map-Reduce).

Our goal is to guarantee that any computation expressed in PQL is either trivial or highly parallelisable. This is certainly the case with our original set of primitives. While user extensions and convenience support for user data types may violate this guarantee, our compiler can be set up to issue suitable warnings.

Below is a brief PQL example:

```
exists int x: s1.contains(x) && x > 0;
```

Here, `x` is a logically quantified variable of type **int**. This query tests whether there exists any element in set `s1` that is greater than zero and returns **true** or **false** accordingly. Analogously, we provide universal queries via **forall**.

PQL queries may include Java expressions as logical constants, as long as these expressions do not depend on logically quantified variables. For example:

```
exists int x: s1.contains(x) && x > arbitraryMethod();
```

Here, we treat the subexpression `arbitraryMethod()` as a logical constant—we execute it precisely once, and supply its constant result to the query. Note that we do not allow PQL programs to use `arbitraryMethod(x)`, as `x` is a logically quantified variable. This is a design decision to ensure that programmers don't have to worry about the order of side effects if `arbitraryMethod` is not pure.

We further provide **query**, which constructs a container; we have already seen it used to construct sets, but it can also construct arrays and maps:

```
query(Map.get(int x) = int y): s1.contains(x) && y==x*x;
```

constructs a map from all values in `s1` to their square values.

reduce signifies a value reduction:

```
reduce(addInt) int x: s1.contains(x);
```

sums up all values from set `s1`. Here, `addInt` can be a user-defined static binary method that we require to be associative, and annotated with a neutral element (0 in this case, to be returned in case `s1` is empty). We require the method to be associative, to permit parallel computation with subsequent merging. Note that in the above we will reduce all viable `x` values, meaning that `x` determines both the set of possibilities we consider and the bag of values we combine. In some cases, these are not the same and `x` may occur more than once; to support this scenario, we provide the following syntax:

```
reduce(addInt) int x over int y: x = a[y] * b[y];
```

which computes a vector dot product over all vector indices `y`.
Our query constructions may be nested freely. For example,

```
query(Map.get(int x) = int y):
       s1.contains(x) && y = reduce(mulInt) int z: range(0, x, z);
```

would compute a map from all values in `s1` to their factorial values. Here, `range` is a method that constructs a set of values from `0` to `x` and tests whether `z` is contained within; our system has special support for evaluating this method efficiently (i.e., without generating an intermediate set).

In addition, our query expressions may contain any Java expressions, though only some of them may contain logical variables. Specifically, logical variables may occur in any primitive computation or expression (i.e., we model all unary and binary operators, including **instanceof** and the `?:` ternary operator), as well as `.contains()`, `get()`, and array index accesses on sets, maps, and arrays. We further simplify syntax so that map and array accesses can use the same notation.

2.2 The PQL Intermediate Language, PQIL

Our focus in this paper is on our system's backend, which operates on our relational intermediate language, PQIL. Every operator in the language can be viewed as a predicate, i.e., a virtual (not necessarily physically materialised) database table. Our system performs computations as a generalised version of a database join. We use the term 'join operators' for the PQIL primitives, and each operator takes a number of parameter variables. For example, the PQL expression `s1.contains(x) && s2.contains(x)` may be represented by the following PQIL program, consisting of two primitive operators in sequence within a block:

$$\{\quad \textsc{Contains}(s_1, x);\quad \textsc{Contains}(s_2, x);\quad \}$$

The above will compute a join over s_1 and s_2, with $\textsc{Contains}(s_1, x)$ iterating over all possible values for x, and $\textsc{Contains}(s_2, x)$ filtering out all values that are not also in s_2. We thus translate logical expressions that would have a **boolean** value in Java into queries that search for the exact values that will make the expression come true.

In the above example, $\textsc{Contains}(s_1, x)$ *writes* x and $\textsc{Contains}(s_2, x)$ *reads* x. PQIL can make this distinction explicit by marking the variables as '$?x$' for reading and '$!x$' for writing, i.e., $\{\textsc{Contains}(?s_1, !x); \textsc{Contains}(?s_2, ?x); \}$. Variables being in 'read mode' or 'write mode' thus describes the operational behaviour of each join operator: $\textsc{Contains}(?s_1, !x)$ must write $!x$ and thus iterates over $?s_1$, while $\textsc{Contains}(?s_2, ?x)$ only reads its parameters and thus performs a containment check. For optimisation purposes, we support a further variable mode, '_', which stands for 'ignore': this can be useful e.g. in the PQL query

```
query(Set.contains(x)): exists y: a[x] = y;
```

where `y` is immaterial and we only care about the index values of array `a`. Here, our PQIL representation is $\textsc{ArrayLookup}\langle int\rangle(?a, !x, \_)$ which our backend can exploit to generate efficient code that never dereferences any array elements.

Each PQIL program has a *program context* that assigns each variable v a type $\tau(v)$ and a value binding $val(v)$. Whenever the value is not a known constant, we set $val(v) = \top$. For example, when performing a range check `range(1, 10, x)`, the numbers `1` and `10` will be represented by variables with such a known value.

Join operators are typed; for example, $\textsc{Range}(x : int, y : int, z : int)$ joins over three integer variables. PQIL allows mismatching actual parameter types; our backend generates implicit conversion code, including (un)boxing, as needed.

PQL provides join operators to support all of Java's unary, binary, and ternary operators, as well as operators to interface with sets, maps, and arrays, as well as to support conversions, field accesses, and container accesses. Each operator is monomorphic, so we use different variants for all viable types. Figure 1 lists some of our operators.

$\textsc{Field}\langle\tau, f\rangle(o_1, o_2)$	$o_2 = ((\tau)o_1).f$
$\textsc{Contains}(s, v)$	$s.\mathsf{contains}(v)$
$\textsc{RangeContains}\langle\tau\rangle(s, e, v)$	for τ in int, long: $s \leq v \leq e$
$\textsc{ArrayLookup}\langle\tau\rangle(a, k, v)$	$a[k] = v$
$\textsc{Type}\langle\tau\rangle(v)$	Checks that v has type τ; checks bounds for integral types

Fig. 1: List of five of our primitive PQIL predicates ("join operators"). The remaining operators are analogous.

The semantics of each operator is that it will attempt to produce all viable bindings for all parameter variables $!x$ that match any given, previously bound variables $?y$ and then *proceed*. We may later backtrack to the same operator, at which point it may proceed again. If no more bindings are available, the operator *abort*s. For example, $\textsc{Contains}(?s, !x)$ on a set with three elements will succeed three times, then abort. We treat (_) as write-mode variables with fresh names.

We permit duplicate variable bindings where appropriate, so that PQIL observes a Bag semantics. For most primitive operators, such as $\textsc{Add}\langle int\rangle(?x, ?y, !z)$, variables can only be bound once (there is only one z for any given x and y such that integer addition yields $z = x + y$), but for other operators, especially container accesses, multiple bindings are possible (as in $\textsc{Contains}$).

The alternative $\textsc{Add}\langle int\rangle(?x, ?y, ?z)$ reads and compares z and produces one binding (succeeds, if $z = x + y$) or zero bindings (aborts, otherwise).

In addition to the above primitive operators, PQIL provides a number of control structures: boolean materialisation (translating successful/failed bindings into **true**/**false** values), disjunctive and conjunctive blocks, and reductions.

We have already used conjunctive blocks in our earlier examples (denoted by curly braces, `{` j_0 `, ... ,` j_k `}`). The semantics of such a block are $j_0 \bowtie \ldots \bowtie j_k$, i.e., we join each primitive with its neighbour (again with bag semantics). PQIL also supports a disjunctive block to model the semantics of the 'or' operator.

Finally, we use the reduction operator $\textsc{Reduce}$ to express generalised reductions, which include map, set, and array construction. For example, we express the PQL query `reduce(Map.get(x) == y) : s1.contains(x) && y == x+1` as:

$$\textsc{Reduce}[\mathrm{MAP}(?x, ?y, !m)]\ \{\textsc{Contains}(?s_1, !x); \textsc{Add}\langle int\rangle(?x, 1, !y);\}$$

Here, $\mathrm{MAP}(?x, ?y, !m)$ is a *reductor* that specifies that for each viable binding produced by the body of the reduction (the block containing CONTAINS and ADD), the fresh map m obtains a mapping from x to y. Multiple mismatching bindings for x cause an exception. The reduction provides a singular binding of variable m. Reductions can construct sets, maps, maps with default values, and arrays, or *fold* (**reduce**) values through a user-supplied method.

2.3 Optimising PQIL

Before compiling PQIL to Java bytecode, we perform high-level optimisations:

- *Nested Block Flattening* splices conjunctive blocks into their parent conjunctive block, if it exists (analogously with disjunctive blocks).
- *Common Sub-Join Elimination* combines redundant primitive join operators within the same conjunctive block, if one is equal to *or generalises* the other.
- *Type Bound Elimination* eliminates unnecessary occurrences of $\mathrm{TYPE}\langle\tau\rangle(x)$.
- *Access Path Selection* re-orders join primitives (see below).
- *Map Reduction Nesting* merges nested reductions that are part of a map/array construction into a single reduction [14].
- *Read/Write Assignment* assigns each variable occurrence a flag to determine whether the variable is read, written, or ignored (Section 2.2). The accompanying must-define flow analysis is an important subroutine in Access Path Selection, and it enables Common Sub-Join Elimination.

Access path selection (or 'query planning') is a standard database optimisation [15] that we apply to each conjunctive block. This technique searches for the most efficient strategy for satisfying a sequence of constraints (as expressed in our join operators). PQL uses a single-phase access path selector that re-orders individual join operators within a conjunctive block. This optimisation is mandatory in our compilation process, as it assigns variables' read/write modes.

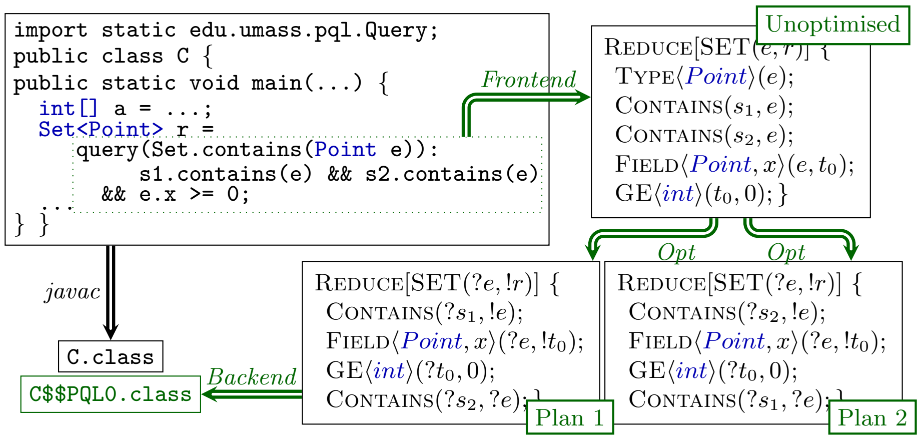
Consider the intermediate code in Figure 2 ('Unoptimised'). The reduction here contains five primitive join operators, of which Type Bound Elimination can eliminate one ($\mathrm{TYPE}\langle \mathit{Point}\rangle(e)$). Access path selection can re-order the remaining four. Our access path selection employs a beam search strategy (retaining the best partial access paths found so far) to limit the search space.

We model the cost for executing each join operator in four *cost attributes*:

- *size*: how many bindings do we expect the operator to generate?
- *cost*: how much does generating one binding cost?
- *selectivity*: what fraction of past bindings will our current join not filter out?
- *parallel*: is this join operator parallelisable?

Not all of this information is available at compile time, so we estimate it when necessary. This can lead to sub-optimal decisions (Section 5), highlighting the need for staged compilation.

We use the *parallel* flag to discount *size*, but only if the join operator occurs in the head of its block, which is where our backend can parallelise the operator.



```
import static edu.umass.pql.Query;
public class C {
public static void main(...) {
  int[] a = ...;
  Set<Point> r =
    query(Set.contains(Point e)):
         s1.contains(e) && s2.contains(e)
      && e.x >= 0;
  ...
} }
```

$\textsc{Reduce}[\mathrm{SET}(e, r)]\ \{$
$\quad\textsc{Type}\langle \mathit{Point}\rangle(e);$
$\quad\textsc{Contains}(s_1, e);$
$\quad\textsc{Contains}(s_2, e);$
$\quad\textsc{Field}\langle \mathit{Point}, x\rangle(e, t_0);$
$\quad\mathrm{GE}\langle \mathit{int}\rangle(t_0, 0);\ \}$

$\textsc{Reduce}[\mathrm{SET}(?e, !r)]\ \{$
$\quad\textsc{Contains}(?s_1, !e);$
$\quad\textsc{Field}\langle \mathit{Point}, x\rangle(?e, !t_0);$
$\quad\mathrm{GE}\langle \mathit{int}\rangle(?t_0, 0);$
$\quad\textsc{Contains}(?s_2, ?e);\ \}$

$\textsc{Reduce}[\mathrm{SET}(?e, !r)]\ \{$
$\quad\textsc{Contains}(?s_2, !e);$
$\quad\textsc{Field}\langle \mathit{Point}, x\rangle(?e, !t_0);$
$\quad\mathrm{GE}\langle \mathit{int}\rangle(?t_0, 0);$
$\quad\textsc{Contains}(?s_1, ?e);\ \}$

Fig. 2: An example of PQL compilation. Here, GE is the join operator for greater-than-or-equal. The frontend emits PQIL (Section 2.2), which we then optimise (Section 2.3). Access path selection identifies multiple query plans (Plan 1, Plan 2) and chooses the most efficient one. The backend generates Java bytecode either to disk (depicted) or into memory at runtime (Section 4).

Each join operator can have different sets of cost attributes depending on its variables' access modes. For example, the *size* of $\textsc{Contains}(?s, ?v)$ is always 1, whereas the *size* of $\textsc{Contains}(?s, !v)$ is exactly the size of the set s. Some combinations of access modes are disallowed, if our system lacks an implementation.

Each variable x is marked as 'write' ($!x$) precisely the first time it appears (or never, if it is a logical constant and $\mathit{val}(x) \neq \top$), so our search algorithm can unambiguously determine access modes and the correct cost from our model. In our example, the possible solutions cannot start with $\textsc{Field}\langle \mathit{Point}, x\rangle(e, t_0)$ because e is not bound yet. Similarly, we cannot start with $\mathrm{GE}\langle \mathit{int}\rangle(t_0, 0)$ because t_0 is not bound yet. Our algorithm will only consider $\textsc{Contains}(?s_i, !e)$ (for $i \in \{1, 2\}$) as initial join operation in the block. Filtering by set is slower yet equally selective to loading and comparing an integer field, so our access path selector will generate either Plan 1 or Plan 2 here.

3 PQL Extension Specification Language

Reflecting PQL, our intermediate language PQIL is a powerful language with a large degree of variability in how each of its join operators might be implemented. This poses a challenge for implementing new join operators or extending existing ones. We thus opted for an extension specification language, PQL-ESL, which allows us to compactly (re-)implement old and new operators.

We designed PQL-ESL so that it should be (a) easy to compile to efficient Java bytecode, (b) simplify the implementation of PQIL and future extensions, and (c) be compact and expressive. To that end, we based our language syntax and semantics on Java's, for expressions, statements, and method definitions, and borrowed from Java's annotation syntax for special attributes. The language includes a number of changes and adds several features:

– Reference semantics for operator parameters
– Type inference (parameters and locals need not be explicitly typed)
– Access mode specifications and tests
– *sections*
– Control operators to signal successful variable binding, or binding failure
– PQIL property tests for parameters
– Templates
– Explicit parallelism support
– Cost model attribute computations

Figure 3 summarises the most salient features of our grammar. In the following, we discuss some of the more interesting features from the above list by looking at examples from our specifications.

3.1 Access mode specifications

Consider the following example:

```
1 @accessModes{rr}
2 lt(val1, val2) {
3     local:
4         if ( @type{int} val1 < val2) proceed;
5         else abort;
6 }
```

This program describes the implementation of the PQIL operator for 'less than'. The first line describes the possible access modes for all parameter variables, which we here restrict to be read mode (`r`) for both. We permit listing multiple access modes, with read, write (`w`), and wildcard mode (`_`) annotations, plus a meta-wildcard (`.`) operator for any of `r`, `w`, `_`.

Line 2 specifies the operator and its parameters. PQL-ESL infers variable types automatically in most cases, so users need not specify them here.

3.2 Sections

Line 3 is a *section* specifier. Each PQL-ESL program can describe up to four sections: *global*, which marks one-time initialisation code (which we don't need here), *local*, which marks code that must be executed each time we start evaluating the PQIL operator, *iterate*, which marks code that we must evaluate every time we backtrack to this operator due to the failure of a subsequent operator, and *model*, which computes cost model attributes (Section 3.7). Section markers may be conditional: we use this to permit conditionally moving computations from the local to the global section.

opdef	::= $\langle$*generic*$\rangle$∗ ($\langle$*init*$\rangle$ '{' $\langle$*stmt*$\rangle$∗ '}')?
init	::= $\langle$*accessm*$\rangle$ ID '(' (ID (',' ID)∗)? ')'
generic	::= '`@generic`' '{' ID '}' '{' STR (',' STR)∗ '}'
accessm	::= '`@accessModes`' '{' (ACC (',' ACC)∗)? '}'
stmt	::= $\langle$*if*$\rangle$ | $\langle$*section*$\rangle$ | $\langle$*goto*$\rangle$ | $\langle$*return*$\rangle$ | $\langle$*while*$\rangle$ | $\langle$*do*$\rangle$ | $\langle$*assign*$\rangle$ | $\langle$*call*$\rangle$
return	::= '`abort`' '`;`' | '`proceed`' '`;`' | '`proceed`' '`on`' ID '`?=`' $\langle$*expr*$\rangle$ '`;`'
assign	::= $\langle$*typeinfo*$\rangle$? ID '=' $\langle$*expr*$\rangle$ '`;`'
typeinfo	::= '`@type`' '{' TYPE '}'

Fig. 3: Partial EBNF grammar for PQL-ESL, eliding more standard language constructions (nested expressions, **while** loops, new object constructions, method calls, etc.) that are syntactically similar or identical to Java.

Lines 4 and 5 contain a standard Java conditional. The only noteworthy feature is the use of `@type{int}`, which can resolve type ambiguity in overloaded operators. This is *optional* (Section 3.5); omitting the specification would permit our implementation to compare any numeric type.

3.3 Special control operators

Finally, lines 4 and 5 also describe what we should do if the comparison succeeds (**proceed**) and what we should do if it fails (**abort**). Here, **proceed** signals a successful evaluation of the PQIL operator, proceeding e.g., to the next nested operator in a conjunctive block or to a reductor that aggregates a successful conclusion of a reduction body. **abort**, meanwhile, signals that the reductor cannot produce any (or any more) bindings and must backtrack. It then backtracks to the most closely nested operator that provides an `iterate` section. All control flow must end with an explicit **abort** or **proceed**; we do not permit leaving the body of an PQL-ESL program implicitly.

3.4 Property tests on parameter variables

Since each operator may supply multiple access modes, PQL-ESL code provides a means for testing these access modes, using the operator `isMode( (`a_1`, ...,` a_n`), (`m_1 `|| ... ||` m_k`) )`, which evaluates to 'true' iff the access modes for variables a_1 to a_n match one of the access modes m_i $(1 \leq i \leq k)$.

As an example, consider the following (slightly simplified) fragment from our original implementation of negation, Neg(a, b):

```
if (isMode( (b), (r) )) {
    tmp = !a;
    if (b == tmp) proceed;
    else abort;
} else {
    b = !a;
    proceed; }
```

This shows the variability when dealing with two access modes (read vs. write) for the second parameter: if `b` is in read mode, we must compare to determine if we should proceed or abort, if it is in write mode, we assign to it. Wildcard mode would require another **isMode** check.

We have found the above to be a very common pattern for expression and function operators so we provide a short form, **proceed on** v ?= expr, that expands to the above and also handles wildcard mode. The proceed-on operator simplifies the above example to `proceed on b ?= !a;`

We provide a second test on parameters, `isConst(x)`, which evaluates to 'true' iff $val(x) \neq \top$; we use it to move initialisations related to x from the *local* section to the *global* section to reduce initialisation overhead to constant time.

3.5 Templates

Our earlier definition of `lt` is not the actual code that we use, as that would require a substantial amount of repetition both to support PQL's primitive numeric types and to support similar operations. Instead, PQL-ESL provides a template programming mechanism that allows us to re-use such specifications, as in the example below:

```
@generic{operator}{"<=", "<"}
@generic{type}{"int", "long", "double"}
@accessModes{rr}
lt_lte(val1, val2) {
        local:
        if ( @type{#type#} val1 #operator# val2)
             proceed;
        else abort;
}
```

This program has two template parameters: `operator`, which can be 'less-than or equal' (`<=`) or 'less-than' (`<`), and `type`, which can be int, long, and double. The PQL-ESL template processor generates all 6 possible pairs of substitutions for the two parameters. Such instantiation is critical for performance, as each of the 6 different operations uses different bytecode operations. Note that we could omit `@type{#type#}` and the explicit template specification for `type`, as type inference will implicitly introduce suitable template parameters as needed.

3.6 Explicit parallelism

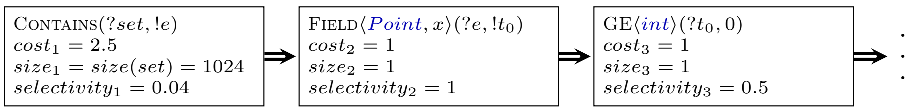
Support for parallel execution is a central PQL feature, so PQL-ESL provides a special interface for parallel access. Operator specifications call **isParallelMode()** to detect whether the operator should run in parallel. In parallel mode, two predefined variables are available: `__thread_index` (indicating the current operator's thread ID) and `__threads_nr` (indicating the total number of threads in use).

For example, an implementation of ArrayLookup$(?a, !i, !v)$ may explore the array a in parallel on multiple cores. It reads `__thread_index` and `__threads_nr` and computes the beginning and end of the array indices it should explore; the perspective of the individual operator is that it will explore only that fraction of the index space. Each operator implementation provides its own solution for parallel execution.

3.7 Cost attributes

Effective language extensions for PQL must also be able to provide cost models to our access path selection mechanism; otherwise we may use them ineffectively (i.e., pick a less efficient access path over a more efficient one). We therefore provide a special section, *model*, that only serves to compute cost attributes. This *model* section is divided into three subsections: *size*, *cost* and *selectivity* (cf. Section 2.3). Let's have a look at the formulas to calculate the cost of an entire given conjunctive block:

$$\boxed{\begin{array}{l} c_0 = 0,\ c_n = (c_{n-1} + cost_n) \cdot s_{n-1} \\ s_0 = 1,\ s_n = s_{n-1} \cdot size_n \cdot selectivity_n \end{array}}$$


$\text{CONTAINS}(?set, !e)$
$cost_1 = 2.5$
$size_1 = size(set) = 1024$
$selectivity_1 = 0.04$
$\text{FIELD}\langle Point, x\rangle(?e, !t_0)$
$cost_2 = 1$
$size_2 = 1$
$selectivity_2 = 1$
$\text{GE}\langle int\rangle(?t_0, 0)$
$cost_3 = 1$
$size_3 = 1$
$selectivity_3 = 0.5$

$c_1 = (c_0 + cost_1) \cdot s_0 = (0 + 2.5) \cdot 1 = 2.5$
$s_1 = s_0 \cdot size_1 \cdot selectivity_1 = 1 \cdot 1024 \cdot 0.04 = 40.96$

$c_2 = (c_1 + cost_2) \cdot s_1 = (2.5 + 1) \cdot 40.96 = 143.36$
$s_2 = s_1 \cdot size_2 \cdot selectivity_2 = 40.96 \cdot 1 \cdot 1 = 40.96$

$c_3 = (c_2 + cost_3) \cdot s_2 = (143.36{+}1) \cdot 40.96 = 5872.0256$
$s_3 = s_2 \cdot size_3 \cdot selectivity_3 = 40.96 \cdot 1 \cdot 0.5 = 20.48$

In this example we calculated the cost for the earlier example Plan 1 of figure 2. The cost till operator i is respectively in c_i. Assignments to distinguished variables in these sections (e.g., `size`), are handed to the access path selector. Note that the *parallel* cost attribute need not be specified, as we can infer it from uses of **isParallelMode()**.

Since this section is part of the specification body, it can take advantage of access mode, constantness, and parallelism information, as well as dynamic properties (such as actual container sizes).

4 Translation with PQL-ESL

PQL-ESL allows us to specify what bytecode we should generate for which join operator. However, the exact bytecode can vary substantially based on (a) access modes for each parameter x, (b) type information $\tau(x)$ and value bindings $val(x)$, as provided by the program context (where known), and (c) whether we are generating code for parallel execution or for sequential execution.

We first *precompile* all PQL-ESL specifications into Java code; this step is only required to add or change PQIL operators. Precompilation reads PQL-ESL specifications, performs name and type analysis as well as template expansion, and generates the PQL-ESL static compiler backend.

At static compile time (invoking our extended `javac`), we process any PQL source code as per Figure 2, then feed the resultant PQIL into this PQL-ESL backend. Static compilation determines all compilation possibilities for each of our PQIL operators in the given context. For each viable configuration of each operator, it generates a *snippet*, consisting of bytecode (via ASM [3]) and metadata for linking. The dynamic compiler later chooses between snippets.

4.1 Static compilation and snippets

The static compiler backend takes in a PQIL program (passed down from the compiler frontend, Figure 2) and compiles it to a dynamic code generator. Since each PQIL operator may be compiled in one of several alternative ways (depending on access modes, etc.), the static compiler performs various checks to determine which compilation patterns to apply. We derive these checks directly from each PQL-ESL specification. Consider compiling CONTAINS(s, v). To provide a correct translation, we must check for numerous alternatives:

(a) What is the mode of v? Read, write, or wildcard? (b) What is the type of s? An unknown/user-defined set type that we cannot parallelise, or a known set type whose internal representation we know how to parallelise? (c) Are we performing parallel access in write or wildcard mode?

As we discussed in the previous section, PQL-ESL can capture all possibilities concisely. Consider the following example:

```
1 local:
2     if (isConst(s))
3         global:
4     if (s instanceof PSet) ...
```

First, consider line 4. Here, we check whether s is of type `PSet` or a general set, where `PSet` is PQL's set implementation, specially optimised for parallel evaluation. Additional parallelisable set types can be supported easily.

The default interpretation of the above **instanceof** check is that we should perform it locally, i.e., every time we enter the operator for the first time (possibly with a new s). This correctly captures the possibility that s might change frequently. Lines 2 and 3 capture a first optimisation: if we know that s is going to be constant during the evaluation of the query, we only need to execute this code once (which we accomplish by moving it into the *global* section).

Thus, the code explicitly handles the distinction between evaluation at operator execution vs. evaluation at query execution start time. Our system handles remaining distinctions automatically: if the dynamic compiler knows whether s is constant and/or whether its dynamic type will be a subtype of `PSet`, it will perform constant folding/dead code elimination, though we precompute this optimisation at *static* compile time. That is, the static compiler compiles the same operator multiple times under different assumptions, such as *known PSet* (only compiling the **true** branch), *known not PSet* (only compiling the **false** branch), and *unknown whether PSet* (compiling both branches and a dynamic check).

The result of static compilation is a multitude of different bytecode sequences that contains all of the variations that are plausible from the static compiler's perspective. We further separate this bytecode into the *static* and *local+iterate* sections, as these need to be executed at different times. We accompany the resultant bytecode sections with a brief relocation table, to resolve back-tracking jumps to the *iterate* section start. Each such combination we refer to as a *snippet*. Simultaneously, our static compiler generates a *composition scaffold* for each operator, which is effectively a nested **switch** table to pick the optimal snippet.

4.2 Dynamic compilation

Of our PQIL optimisations (Section 2.3), two (Type Bound Elimination and Access Path Selection) potentially benefit from run-time information. Since access path selection can have a substantial impact on the execution time of a query, our dynamic compiler re-runs access path selection before dynamic code generation, factoring in newly available information (e.g., run-time container sizes).

The result is a re-ordered PQIL specification, which we then pass into the composition scaffold. The scaffold examines each operator parameter's access modes and may further examine constantness, dynamic types, and whether the operator is to be compiled for parallel execution. It then picks the optimally specialised snippet for each operator and configuration, and links it against its neighbouring snippets, emitting bytecode that is ready for execution.

5 Evaluation

To test the utility of our PQL-ESL language, we used it to re-implement our PQL/Java bytecode backend. We found the language to be entirely suitable to this task, though implementing the FIELD operator's field access operation required the addition of a single PQL-ESL feature.

To further evaluate our language, we performed three forms of evaluation: We evaluated the usability of PQL-ESL by implementing four new extensions, we examined in detail the performance of our dynamic compiler with four pre-existing PQL benchmarks [14] and two new synthetic benchmarks and we implemented support for communicating with SQL data sources via JDBC.

5.1 Case Studies

To evaluate the generality of our language, we selected four language extensions that we did not previously support in PQL and added them to our system.

sqrt: Our first extension was a square-root function on doubles. This addition permits two access modes, one for computing and one for testing the square root.

primes: We further added support for prime numbers. We added two extensions, PRIMECHECK, which determines whether a number is prime by trying to divide by all smaller non-even numbers up to the number's square root,

Extension	**LOC**	**Snippets**
Sqrt	5	4
PrimeCheck	20	2
PrimeRange	42	4
StreamContains	21	4
Modulo	8→61	16→32

Fig. 4: Sizes of our new operators, counting global, local, and iterate sections. The arrows for the pre-existing Modulo operator indicate changes due to our extensions.

Benchmark	**Snippets**	**Bytes**
threegrep	16	1155
wordcount	8	1747
bonus	18	1773
webgraph	16	1608
setnested	8	1407
arraynested	10	993

Fig. 5: Snippet statistics for our benchmarks. Here, **total bytes** is the size of all bytecodes in the final linked bytecode for the query, in bytes.

and PrimeRange, which computes all prime numbers in a given range, using the Sieve of Eratosthenes. Our rewriting engine automatically introduces PrimeRange when PrimeCheck and Range affect the same variable.

Java 8 streams: To bridge the gap between Java 8 streams and PQL, we added a StreamContains(s, v) operator analogous to our Contains(s, v) operator. In access mode Contains$(?s, ?v)$, it uses a Java 8 `EqualityPredicate` to text whether v is contained in the stream s. For access mode Contains$(?s, !v)$, it iterates over all stream elements and binds them to v, again unifying two mechanisms into a single interface.

modulo: For our last extension, we modified the existing Modulo(x, y, z) operator, for int and long parameters, to permit access modes Modulo$(?x, !y, ?z)$ and Modulo$(!x, ?y, ?z)$, allowing users to find all x for a given y and z such that $x \mod y = z$ (analogously for y).

Figure 4 summarises the sizes of our extensions, counting their lines of code and number of snippets. Despite the large amount of variability in many of the extensions, we found the code sizes to be very manageable.

5.2 JDBC Link

Our JDBC link adds three new user-facing operators: one that represents a database, one that represents a table in the database, and one that represents access to a field in the database. In an approach comparable to that of Cheung et al.'s QBS [7] we then automatically promote PQL operators to database operators where possible, using our rewriting engine. We further filter expected/required result fields automatically through a custom PQIL analysis. In total, this link uses ten custom operators with 9–53 lines of code. Operators include a single highly-polymorphic operator that represents all numeric comparisons between fields and constants, operators for simplified access to three particular database systems, and the SQL `LIKE` operator. Automatic promotion to joins between tables is not supported yet, though the PQL-ESL-specified *operators* are as expressive as JDBC permits them to be.

While database access via JDBC inherently cannot be parallelised, our JDBC link allows users to take advantage of our query language and to optimise interactions with data sources on the Java heap.

5.3 Performance Evaluation

We examined our system with the following benchmarks:

- *threegrep*, which finds all strings (in a set of 100-character strings) that contain the substring "012" [8].
- *wordcount*, which computes the absolute numbers of occurrences of words in documents. We represent words by unique integer IDs. The result is a map from word IDs to the number of times they occur in a set of documents.
- *bonus*, which is a well-known example from the databases literature [19] that computes employee salary boni, given each employee's department, the department's bonus policy, and the employee's accumulated bonus. The result is again a map, from each employee to their aggregate bonus.
- *webgraph*, as defined by Yang et al. [19], in which we compute the set of all documents in a graph structure that point to themselves via one point of indirection.
- *setnested*, which computes the intersection of two random sets of integers, plus a size bound (similar to our initial motivating example, but without a field access). One of the sets contains only a single element, while the other contains 500,000.
- *arraynested*, which computes the set intersection of two random arrays, again with a size bound. The sizes of the arrays are identical to *setnested*.

Figure 5 summarises the number of snippets for each benchmark. Even for our four more complex benchmarks, our static compiler is effective at keeping the number of relevant alternatives small. While some important snippets can reach a substantial size (up to around 650 bytes for a snippet in wordcount that represents a nested reduction into a default map, to aggregate the total counts), the size of the ultimately generated bytecode remains below $2kB$, within the size of what we might expect for such a computation.

We took each benchmark's PQL implementation and compiled it both with our original backend [14] and with the new PQL-ESL backend. For comparison, we also ran best-effort manual implementations. We ran all benchmarks 13 times, discarding the first 3 runs as warm-up runs. Compared to our earlier backend, we configured our benchmarks to '50% mode', which reduces the workload for each benchmark by at least 50%, thereby making the dynamic compilation overhead more easily visible and putting our original system at a deliberate advantage.

We ran all benchmarks on the Oracle JDK 1.8.0_05, on a Sun SPARC64 (Sparc v9) Enterprise-T5120 system, with 8 cores at 8 SMT threads each. We left all system configuration at its defaults, other than increasing the default heap size to 13200 MiB. Access path selection used a search window size of 16.

Our current PQL-ESL backend does not yet serialise snippets to class files. For our experiments, we therefore ran the static and dynamic compilation phases

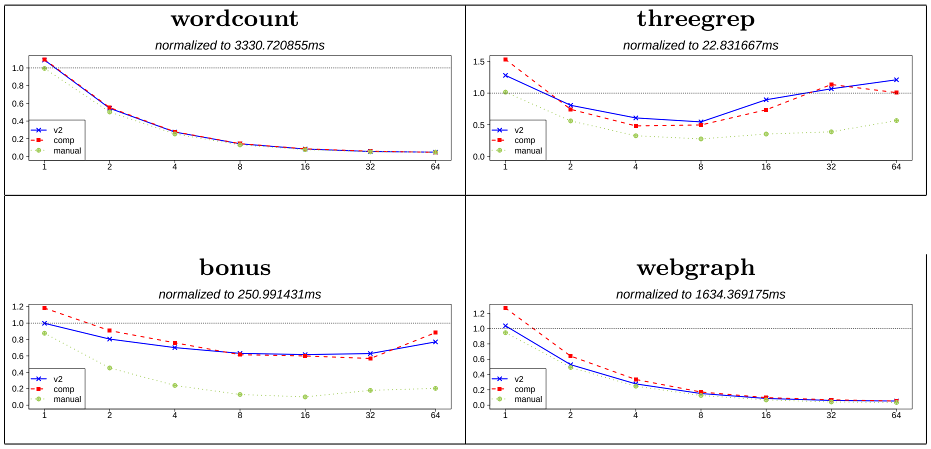


Fig. 6: Benchmark execution times: Execution time (y axis) by number of threads (x axis). The figure shows manual java implementation (*manual*), the new backend (*v2*) and the original backend (*comp*).

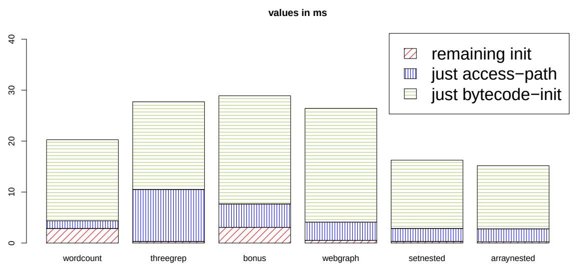


Fig. 7: Dynamic compilation overhead, split into snippet-based code generation, access path selection, and remaining initialisation (allocating supporting data structures).

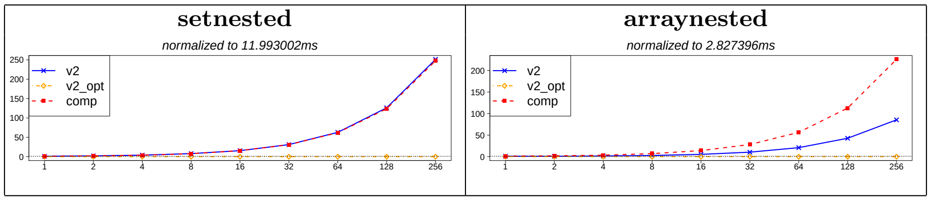


Fig. 8: Execution times (y axis) for queries with increasing workloads (x axis), in benchmarks that rely on dynamic access path selection, running single-threaded.

in the same JVM, taking care to separate the execution phases. We avoided including the rest of the PQL/Java compiler by separately compiling all PQL source code into PQIL and feeding the result directly into our backend.

Figure 6 shows our total execution time, *excluding* dynamic recompilation time. The quality of the code generated by our new backend is competitive with our existing backend for all existing benchmarks, outperforming it in *webgraph* and, for small and large numbers of threads, in *bonus*.

Figure 7 separately shows dynamic recompilation time, which is currently in the millisecond range, meaning that dynamic recompilation is only effective for large data sets. We can leverage this insight to use suitable heuristics that determine whether dynamic optimisation is warranted from the size of any input containers, and default to execution without dynamic compilation for small workloads, where sub-optimal access paths are not critical.

Dynamic compilation overhead is the price we pay for dynamic optimisation. We see the value that we gain for this price in Figure 8, which again shows our benchmarks *setnested* and *arraynested*. Both benchmarks operate over two containers with substantially different sizes. Performance depends on picking the larger container to iterate over, and the smaller container to check containment in; this is only possible with run-time information.

For each benchmark, we scale the sizes of both containers by the factor on the x axis. In our graphs, *comp* and *v2* show our old and new backend, respectively. As we can see, our new backend outperforms the old backend even without recompilation. However, execution time increases exponentially for both implementations if they choose the less optimal container for iteration. *v2_opt* shows our new backend with dynamic re-compilation, with near constant-time performance. Initialisation overheads are again shown separately, in Figure 7.

Overall, we found performance of PQL-ESL to be on par with our current system when not factoring in dynamic optimisation. With dynamic optimisation, our PQL-ESL backend can greatly outperform our existing system. At the same time, re-writing our backend in PQL-ESL has given us the flexibility to re-target compilation to static or dynamic compile time, to add new language features quickly, and to apply further language-based optimisations in the future.

6 Related Work

Extensible query languages are widely known in the literature. For example, the Meteor language for the Stratosphere platform [11] can be extended with new operators, as can PigLatin [13]. FlumeJava [5], PLINQ [9] and Java 8 Streams, which act as internal DSLs, can be extended by implementing predefined interfaces. All of the above systems thus permit users to add new operators.

However, to the best of our knowledge none of these systems utilise a special-purpose extension language to simplify optimisation, multi-stage or otherwise. Similarly, none of the above systems will automatically select between different modes of user-defined operators, as permitted by our read/write mode distinction. The only optimisations they can apply to language extensions are therefore optimisations provided by the host language compiler. As we have shown, this suggests that these systems may be missing optimisation opportunities.

Delite, a framework for highly-optimised domain-specific languages, includes a query language, OptiQL [16] that can conceptually integrate with other domain-specific languages, performing parallelisation and other optimisations across DSLs. However, we are not aware of OptiQL supporting custom operators. StreamJIT [2], a commensal compiler framework, treats IRs as libraries and permits IR-level optimisation. Instead of code generation, commensal compilation relies on compiler optimisations; however, it is unclear that complex control flow between operators such as ours could be inlined automatically (and without inline guards) by standard JVM JIT inlining.

Meanwhile, general-purpose Turing-complete languages geared at easing access to parallelism, such as Chapel [4], Fortress [1], and X10 [6] allow user-defined extensions through standard abstractions (functions, classes). However, these languages provide control over parallel primitives (task distribution, atomic regions) rather than automating parallelisation, and being Turing-complete, they may be too powerful for effective automatic parallelisation. By contrast, PQL/-Java strives first to be easy to parallelise, foregoing expressivity to achieve this goal, with PQL-ESL bridging the gap to our Turing-complete host language.

Our focus in this paper has been on extending the PQL/Java backend. Complementary frontend extensions could be based on techniques from the literature, such as those found in Sugar [10] or Silver [18].

7 Conclusion

PQL-ESL is a mechanism for extending the PQL/Java backend with support for new primitives and for more effectively optimising existing primitives. Leveraging PQL-ESL with a two-stage compiler permits us to compile PQL queries statically and dynamically re-optimising as new information becomes available.

We have shown that PQL-ESL is effective at describing PQL primitives by re-implementing our compiler backend in it, and effective at describing new primitives by adding four extensions to PQL. Furthermore, our experiments demonstrate that our execution performance is competitive with our previous backend. For some queries, dynamic compilation enables optimisations that permit the PQL-ESL backend to outperform our previous backend by reducing the algorithmic overhead from $O(n)$ to $O(1)$. Our implementation is publicly available[3].

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Steele, G., Tobin-Hochstadt, S.: The Fortress Language Specification. Tech. rep., Sun Microsystems (2008)

[3] `http://sepl.cs.uni-frankfurt.de/pql.html`

2. Bosboom, J., Rajadurai, S., Wong, W.F., Amarasinghe, S.: StreamJIT: A Commensal Compiler for High-performance Stream Programming. In: OOPSLA '14. pp. 177–195. ACM, New York, NY, USA (2014)
3. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. Adaptable and Extensible Component Systems (2002)
4. Callahan, D., Chamberlain, B., Zima, H.: The cascade high productivity language. In: High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on. pp. 52–60 (April 2004)
5. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. In: Programming Language Design and Implementation (PLDI). pp. 363–375. ACM, New York, NY, USA (2010)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing
7. Cheung, A., Solar-Lezama, A., Madden, S.: Optimizing database-backed applications with query synthesis. SIGPLAN Not. 48(6), 3–14 (Jun 2013)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Operating Systems Design & Implementation (OSDI). USENIX Association (2004)
9. Duffy, J., Essey, E.: Parallel LINQ: Running queries on multi-core processors. MSDN Magazine (October 2007)
10. Erdweg, S., Rieger, F.: A framework for extensible languages. In: Järvi, J., Kästner, C. (eds.) GPCE. pp. 3–12. ACM (2013)
11. Heise, A., Rheinländer, A., Leich, M., Leser, U., Naumann, F.: Meteor/sopremo: An extensible query language and operator model. In: Proceedings of the International Workshop on End-to-end Management of Big Data (BigData) in conjunction with VLDB 2012. Istanbul, Turkey (0 2012)
12. Immerman, N.: Descriptive Complexity. Springer (1998)
13. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: A not-so-foreign language for data processing. In: Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD). pp. 1099–1110. ACM (2008)
14. Reichenbach, C., Smaragdakis, Y., Immerman, N.: PQL: A purely-declarative Java extension for parallel programming. In: ECOOP. pp. 53–78 (2012)
15. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: ACM SIGMOD Int. Conf. on Management of Data. pp. 23–34. SIGMOD '79, ACM, New York, NY, USA (1979), `http://doi.acm.org/10.1145/582095.582099`
16. Sujeeth, A., Rompf, T., Brown, K., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and reuse with compiled domain-specific languages. In: Castagna, G. (ed.) ECOOP 2013 Object-Oriented Programming, Lecture Notes in Computer Science, vol. 7920, pp. 52–78. Springer Berlin Heidelberg (2013)
17. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) OOPSLA. pp. 867–884. ACM (2013)
18. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming 75(1–2), 39–54 (January 2010)
19. Yang, H.c., Dasdan, A., Hsiao, R.L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: ACM SIGMOD Int. Conf. on Management of Data. pp. 1029–1040. SIGMOD '07, ACM, New York, NY, USA (2007)