i3QL: Language-Integrated Live Data Views

Ralf Mitschke[1] Sebastian Erdweg[1] Mirko Köhler[1] Mira Mezini[1,2] Guido Salvaneschi[1]

[1]Technische Universität Darmstadt, Germany
[2]Lancaster University, UK
< $lastname$ >@cs.tu-darmstadt.de

Abstract

An incremental computation updates its result based on a change to its input, which is often an order of magnitude faster than a recomputation from scratch. In particular, incrementalization can make expensive computations feasible for settings that require short feedback cycles, such as interactive systems, IDEs, or (soft) real-time systems.

This paper presents i3QL, a general-purpose programming language for specifying incremental computations. i3QL provides a declarative SQL-like syntax and is based on incremental versions of operators from relational algebra, enriched with support for general recursion. We integrated i3QL into Scala as a library, which enables programmers to use regular Scala code for non-incremental subcomputations of an i3QL query and to easily integrate incremental computations into larger software projects. To improve performance, i3QL optimizes user-defined queries by applying algebraic laws and partial evaluation. We describe the design and implementation of i3QL and its optimizations, demonstrate its applicability, and evaluate its performance.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Incremental Computation, Scala, Reactive Programming

1. Introduction

Incremental computation is a technique that updates an algorithm's results in the event of changes to the respective inputs and is often faster than a re-computation from scratch (i.e., over the full input). The fundamental concept that makes incremental computations appealing is the *principle of inertia* [16], which means that small changes in the input have only small impacts on the output.

Computations in many domains are *embarrassingly incrementalizable*, i.e., conceptually the potential for speed-up is immediately obvious. As a result, incrementalization has been researched and successfully applied in a broad variety of systems, e.g., parsing, compilation, truth maintenance or constraint solving (cf. [36] for a survey) and applications continue to emerge, e.g., machine learning [47], static program analysis [10], dynamic program invariant checking [45] and distributed parallel processing of big-data [2]. Yet, even today the actual development of an incremental computation is a complex task that involves tedious manual labor.

To develop incremental computations many of the earlier research works have produced designated incremental algorithms for specific problems, e.g., parsing or compilation (cf. [36]). These algorithms utilize highly specialized data structures and update semantics for the problem at hand to achieve optimal performance. However, this form of incrementalization requires manual work to devise and implement an incremental version of the respective algorithm.

In this paper we present a general-purpose solution — termed i3QL (*incremental integrated in-memory query language*) — that (i) allows developers to express incremental computations declaratively inside a programming language and (ii) has an efficient back-end for executing incremental computations in-memory. Incremental computations formulated in i3QL are easy to use, without developing and debugging specialized data structures and update semantics. Developers express the algorithm in a declarative *embedded domain-specific language* (EDSL) [20] that is inspired by SQL and features incrementalization and optimizations. Both the EDSL and the back-end of i3QL are influenced by research from incrementalization of relational algebra performed in the database community. The latter has produced efficient techniques for incrementalization under the term *incremental view maintenance* [16].

Traditionally, relational algebra is restricted to database systems. Yet, integrating the semantics of relational algebra into a programming language has several advantages for incremental computations both for the front-end in which users express computations and the back-end, i.e., the run-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '14, October 20–24 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660242

time semantics. From the perspective of the front-end: (i) the SQL-inspired EDSL is a declarative language and thus facilitates formulation of incremental computations as a data transformation rather than a series of imperative statements that update and retrieve the state of the computation, (ii) SQL is an industry standard and, thus, we enhance the accessibility to our solution for end-users with a database background, and (iii) end-users familiar with SQL can reason about the runtime semantics since queries have a predictable semantics, which can also be reviewed as an i3QL query plan[1] that shows how a query is executed. From the perspective of the back-end: (i) relational algebra provides a well-defined and well-understood semantics as the basis for incremental computations and is also suitable for optimizations, (ii) the operators of the algebra are compositional and results can be reused over different incremental computations, (iii) for each operator we implemented a principal incrementalization semantics that provides optimal performance, and, (iv) new operators can be easily added to extend the back-end.

We have implemented i3QL in the programming language Scala as an EDSL that can be integrated as a library by end-users.[2] The EDSL is statically typed to help users in defining their queries in a type-safe manner. In addition, we used *light-weight modular staging* (LMS) [37] to conduct query optimizations based on the reification of first-class functions.

In comparison, previous approaches to incremental computations from the programming-languages community are confined to certain algorithms and/or data types as well as specialized optimizations [26, 49, 52]. Other solutions are generally applicable [1, 45], but are based on *generic* optimizations like function memoization and dependency tracking to selectively trigger recomputations upon change. In contrast, i3QL is based on relational algebra and provides a unique combination of features: i3QL is generally applicable to a wide range of incremental computations, easy to use as an embedded DSL, features relational optimizations and partial evaluation, has low memory overhead due to event propagation, and is modularly extensibility with new operators and optimizations.

The contributions of this paper are as follows:

- We present the design and implementation of i3QL, an integrated and optimized query language for incremental computations.
- i3QL is embedded in the Scala programming language and incremental computations are easy to formulate.
- While i3QL is inspired by SQL and the semantics of relational algebra, it is generally applicable due to support for general recursion.
- We present incremental operators based on relational algebra that are compositional and reusable.

[1] Query plans are often used in database systems to guide engineers towards a design that has better performance.

[2] Source code is available online: `http://github.com/seba--/i3QL`

```
1 val students: Table[Student] = new Table[Student]()
2
3 val sallies: View[Student] =
4  (SELECT (*) FROM students
5   WHERE (s => s.firstName == "Sally")).asMaterialized
6
7 students.add(new Student("Sally", "Fields"))
8 students.add(new Student("George", "Tailor"))
9
10 sallies.foreach(s => println(s.lastName))
11 // prints: "Fields"
12
13 students.add(new Student("Sally", "Joel"))
14 // incremental update of sallies
15
16 sallies.foreach(s => println(s.lastName))
17 // prints: "Fields" and "Joel"
```

Figure 1: The relation sallies is incrementally maintained.

- We rewrite i3QL queries to optimize performance by applying algebraic laws and using staging.
- We demonstrate the applicability of i3QL by developing an incremental parser and evaluate its performance by a benchmark of 15 FindBugs [19] analyses.

2. i3QL: An Overview

i3QL is an embedded DSL for expressing incremental computations with an SQL-like syntax in Scala. Figure 1 provides an illustrating example that we use in the following to introduce the central concepts of i3QL.

A **table** represents a multiset and provides a standard collections interface permitting addition, removal, and update of elements. For example, line 1 of Figure 1 declares a table students containing Student objects. Line 7, 8, and 13 each add a student to the students table.

i3QL supports SQL-like queries on tables, resulting in a **view** of the data. In the example query sallies, we filter students to select only those students whose first name is Sally. The result can be traversed by the user or processed by further queries, since both tables and views are relations.

Tables allow direct manipulation by the user, whereas views are **incrementally maintained**: i3QL updates a view whenever any of the queried relations changes. To this end, i3QL registers queries with all involved relations to receive change notifications. For example, sallies receives a change notification whenever students is changed; its result is updated incrementally. In particular, the predicated (s => s.firstName == "Sally") is executed exactly once for each student that is consecutively added in lines 7, 8, and 13. Without support for incremental maintenance, the first two students would be tested twice, once in line 10 and again in line 16.

An i3QL relation occurs either as a virtual relation or as a materialized relation. A **virtual relation** does not store any of its elements and is only used to notify dependent

queries about changes. For example, the students relation is declared as a virtual relation. As a consequence, the Student object "George Tailor" is not persisted (in memory) in our example: When adding "George Tailor" in line 8, students does not store the object, but only notifies sallies, whose result does not contain "George Tailor" either. Thus, the usage of a virtual relation has allowed us to forego persistence of objects that are irrelevant to the user that defined the query. A **materialized relation** does store the elements of the relation for later use. For example, we declared that the query sallies should produce a materialized relation, because we want to use the elements of the resulting relation outside of i3QL queries; namely in the foreach traversal in lines 10 and 16.

i3QL queries can be **composed** from existing queries, since tables and views are both relations. For example, the following query computes the average age of students whose first name is Sally:

```
val averageAge = SELECT AVG((_:Student).age) FROM sallies
println(averageAge.head)
```

i3QL incrementally maintains the resulting relation averageAge by updating the result whenever the relation sallies changes. Note that changes to the relation students do not directly affect age; only students that pass the filter in sallies lead to an update of age. This allows the modular specification and composition of incrementally maintained queries in i3QL.

i3QL provides **incremental versions of all standard operators** from relational algebra. In the above examples we already saw selection, projection, and average aggregation. The following example demonstrates joins.

```
val council: Table[Member] = new Table[Member]()
def councilStudAge(sm: (Student, Member)) = sm._1.age
val membersAge =
  SELECT (councilStudAge) FROM (students, council)
  WHERE (firstName === firstName) AND
         (lastName === lastName)
```

The query selects the age of all students that participate in the university council. The result of the join is a relation with pairs of Student and Member objects, for which the join condition holds. Our incremental version of the join operator exploits the fact that a change in the students relation only affects membersAge if the student participates in the council.

In addition to the standard operators, i3QL also features **general recursion**, which makes it a Turing-complete language for incremental computations. For illustration, consider a relation tutors between a tutor and her students. We want to transitively compute all students tutored by a tutor's students:

```
 1 val tutors = Table[(Tutor, Student)]
 2 def tutorId(ts: (Tutor, Student)) = ts._1.id
 3 def studentId(ts: (Tutor, Student)) = ts._2.id
 4 def indirectTutor(ttss: ((Tutor, Student), (Tutor, Student))) =
 5   (ttss._1._1, ttss._2._2)
 6 val indirectStudents =
 7   SELECT (indirectTutor) FROM (tutors, tutors)
 8     WHERE (studentId === tutorId)
 9 val transitiveStudents = WITH RECURSIVE(
10   tutors UNION ALL (indirectStudents))
```

Query indirectStudents performs a join to find all students of a tutor that are tutors themselves. The results are new tuples of the original tutor and her students' students (generated by the function indirectTutor). However, this query can only find one level of indirection between a tutor and her students. To find all transitive students of a tutor a recursive query is necessary that feeds back results of indirectStudents into the join, as if they were part of the original tutors relation. The WITH RECURSIVE operator implements this behavior.

This concludes the overview of i3QL from a user's perspective. In the following section, we present the technical realization of i3QL.

3. Technical Realization

In this section, we present the run-time system of i3QL and, in particular, how incremental changes are propagated through algebraic operators. For space reasons, we explain only a representative subset of the operators we implemented. A list with the most relevant operators appears in Table 1.

3.1 Change Propagation

A change always originates in a Table, which allows users to add, update, and remove elements. i3QL transitively propagates changes from tables to operators that (transitively) dependent on the table. To this end, we compile a query into an operator graph where nodes represent tables and operators involved in the query. We propagate changes along the edges of operator graphs.

For illustration, consider the initial example of filtering students for the name Sally from Sec. 2. Figure 2 depicts the operator graph of this example. The figure shows the propagation of adding a single student (named "Sally Joel") and starts at the table, which pushes changes to the selection operator σ. Since the condition of the filter s.firstName == "Sally" evaluates to **true** for the added element, the change is further propagated to the resulting view sallies. If the condition evaluated to **false**, the change would not have been further propagated.

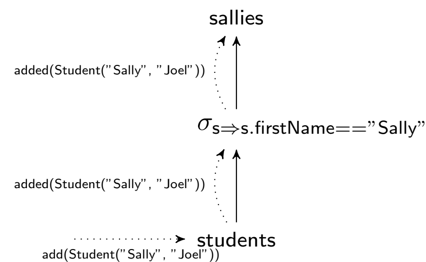


Figure 2: Filtering a table of students

Operator	Use
Selection	SELECT ... FROM t1 WHERE pred(t1.f1)
Projection	SELECT f1 FROM t1
Unnesting	SELECT ... FROM UNNEST (t1, _.f1)
Aggregation	SELECT ... FROM t1 GROUP_BY t1.f1
Dupl. elimination	SELECT DISTINCT f1 FROM t1
Recursion	WITH RECURSIVE(...)
Cartesian product	SELECT ... FROM (t1, t2)
Join	SELECT ... FROM (t1, t2) WHERE t1.f1 === t2.f2
Semi-join	SELECT ... FROM t1 WHERE EXISTS (SELECT ... FROM t2 WHERE t1.f1 === t2.f2)
Anti semi-join	SELECT ... FROM t1 WHERE NOT EXISTS (SELECT ... FROM t2 WHERE t1.f1 === t2.f2)
Difference	SELECT ... FROM t1 EXCEPT SELECT ... FROM t2
Union	SELECT ... FROM t1 UNION SELECT ... FROM t2
Union all	SELECT ... FROM t1 UNION ALL SELECT ... FROM t2
Intersection	SELECT ... FROM t1 INTERSECT SELECT ... FROM t2

Table 1: i3QL operators.

The basis of the change propagation is the Observer interface depicted in Figure 3. The interface declares three atomic change events for addition, removal and update.

```
1 trait Observer[V] {
2   def added(v: V)
3   def removed(v: V)
4   def updated(oldV: V, newV: V)
5 }
```

Figure 3: The observer interface for change propagation.

Every i3QL operator implements the Observer trait and a corresponding Observable trait. The implementation of the Observer defines incremental semantics of an operator. An operator registers itself as an observer of its predecessors in the graph and receives atomic change events through any of the three methods of Observer.

Selection Operator To illustrate the implementation of operators, we show the implementation of the selection operator in Figure 4. The selection operator observes an underlying relation containing elements of type V and filters the elements using a predicate function. Each operator has a type for resulting elements, defined by the type argument passed to Observable. In case of selections,V is the input and output type of the operator. The change propagation of selection is comparatively simple: Selection only propagates a change to its observers if the involved value satisfies the predicate. Finally, selection registers itself as an observer of the underlying relation.

```
 1 class Selection[V](rel: Relation[V], predicate: V => Boolean)
 2   extends Observer[V] with Observable[V] {
 3   def added(v: V) { if(predicate(v)) notify_added(v) }
 4   def removed(v: V) { if(predicate(v)) notify_removed(v) }
 5   def updated(oldV: V, newV: V) {
 6     if(predicate(oldV)) {
 7       if(predicate(newV)) notify_updated(oldV, newV)
 8       else notify_removed(oldV) }
 9     else if(predicate(newV)) notify_added(newV) }
10   rel addObserver this
11 }
```

Figure 4: The incremental selection operator.

Join Operator The join operator is a binary operator that correlates elements of a left-hand and right-hand side relation. The results are pairs of elements from the two relations. Instead of a binary predicate expressing the join condition, our join operator uses two functions that map elements to key values; the operator joins two elements if they are mapped to the same key. For example, in the join example shown in Sec. 2, we compared the properties firstName and lastName of elements from two relations. i3QL automatically translates this condition into two key functions of the form (s:Student) => (s.firstName, s.lastName), where the key is a tuple of all values that have to be equal.

```
 1 class Join[V1, V2, Key](
 2     left: Relation[V1], right: Relation[V2],
 3     leftKeyFun: V1 => Key, rightKeyFun: V2 => Key)
 4   extends Observable[(V1,V2)] {
 5     val leftIndex : Map[Key, List[V1]] = ...
 6     val rightIndex : Map[Key, List[V2]] = ...
 7     object LeftObserver extends Observer[V1] {
 8       def added(v: V1) {
 9         val key = leftKeyFun(v)
10         rightIndex.get(key) match {
11           case Some(list) =>
12               list.foreach( u => notify_added ((u,v)) )
13           case _ => // no changes propagated
14     }}...}
15     object RightObserver extends Observer[V2] { ... }
16 }
```

Figure 5: Excerpt of the incremental join operator.

Figure 5 shows an excerpt of the implementation of the incremental join operator. The join—and any binary operator in general—observes the input relations via two differently typed observers (line 7 and 15). To correlate elements, a join

takes two functions leftKeyFun and rightKeyFun that map elements of the two relations to keys. To quickly access all elements that are mapped to the same key value, a join indexes both input relations (line 5 and 6). Upon adding a new element to either underlying relation, we use the corresponding key function to obtain a key of the added element. Next, we use the index to look up all elements from the other relation that have a matching key (line 10). For each of the elements found in the other relation a new result pair is propagated (line 12). Thus, a join can propagate multiple changes due to the addition of a single element.

Aggregation Operator The aggregation operator in i3QL works like its counterpart in SQL: It can be used to aggregate elements of a relation into a single value and to group a relation into smaller subrelations. Either feature is optional. For example, in Sec. 2 we used AVG((_:Student).age) to aggregate values without grouping.

```
 1 class Aggregation[V, Key, Out](
 2    rel: Relation[V],
 3    groupingFun: V => Key,
 4    aggregateFun: (Out, Out) => Out,
 5    inverseAggregateFun: (Out, Out) => Out,
 6    conversionFun: V => Out)
 7  extends Observer[V] with Observable[(Key, Out)] {
 8  val groups = Map[Key, Out]()
 9  def added(v: V) {
10    val key = groupingFun(v)
11    if (!groups.contains(key)) {
12      val out = conversionFun(v)
13      groups.put(key, out)
14      notify_added((key, out))
15    }
16    else {
17      val oldOut = groups(key)
18      val newOut = aggregateFun(oldOut, conversionFun(v))
19      if (newOut != oldOut) {
20        groups.put(key, newOut)
21        notify_updated((key, oldOut), (key, newOut))
22      }
23    }
24  }
25
26  def removed(v: V) {
27    ...
28    val newOut = inverseAggregateFun(oldOut, conversionFun(v))
29    ...
30  }
```

Figure 6: Excerpt of the incremental aggregation operator.

Figure 6 shows an excerpt of the implementation of the aggregation operator and, specifically, of its most general case aggregation with grouping.[3] The operator takes four functions as parameters. The grouping function maps input elements of type V to the type of the grouping attribute Key. The aggregate function is used for adding new elements to the previous result by taking two values and computing a single aggregated value. The operator also requires an inverse aggregation function, which we use to incrementally remove values from the aggregated result.

Typical aggregation functions work over integers, e.g., COUNT, SUM, MIN, MAX. i3QL's aggregation operator works with aggregation functions over any type by using a conversion function from elements of the underlying relation to the type of result values Out. The results of the operator are tuples of type (Key, Out), i.e., for each group we obtain the grouping attribute and the value aggregated from all elements in the group.

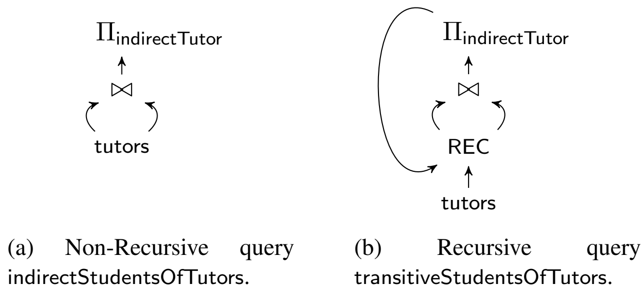
To enable incrementalization, we store the previously computed values for each group (line 8). When a new element is added to the underlying relation, we compute the key of the added element (line 10). If no value was previously computed for the group, we obtain the initial value by applying the conversion function to the added element (line 12) and notify observers that a new key value pair was added (line 14). If a previously computed value exists, we apply the aggregation function to the old value and the converted new element (line 18). We notify observers only if the aggregation yields a new result newOut that is different from the old result oldOut.

Users of i3QL do not define aggregate functions directly when implementing a query. Instead, they use factory objects, such as AVG, that serve a dual purpose: First, the factory encapsulates both the aggregation function and its inverse, so that users of i3QL do not need to supply two function in a query. Second, they distinguish *distributive* from *non-distributive* aggregate functions [33]. This is important for the correct incrementalization of the aggregation operator, because only distributive aggregate functions such as AVG can be updated based on the old and new value alone. Non-distributive functions such as MIN require more data. For example, MIN requires the entire relation to compute the new minimum when the current minimum is removed. The implementation of the aggregate operator in Figure 6 is only correct for distributive aggregate functions; we have a separate implementation for non-distributive functions. The factory objects used by users of i3QL must select the appropriate implementation.

Recursion Operator Recursion introduces a loop into the operator graph that allows to receive change propagations from operators that depend on the output of the recursion (directly or transitively). i3QL provides a fixpoint recursion operator REC that eliminates duplicate derivations.

For illustration, consider the recursive query that transitively finds all students of a tutor, shown in Sec. 2. Figure 7a depicts the operators graph for the non-recursive query indirectStudents that computes the students of a tutor's students. Figure 7b shows the recursive query transitiveStudents. The declaration WITH RECURSIVE substitutes tutors in the non-recursive query by the recursion operator REC, which

[3] The other cases use specialized classes based on which clauses, e.g., GROUP_BY, are present in a query.



(a) Non-Recursive query indirectStudentsOfTutors.

(b) Recursive query transitiveStudentsOfTutors.

Figure 7: Substitution performed by WITH RECURSIVE.

observers tutors as well as the result of the non-recursive query.

The REC operator implements a standard counting *delete and re-derive algorithm* [18]. Each element that is added to REC increases a counter for this element. The counter represents how many times an element was (recursively) added. If an element is encountered the first time it is propagated to the operator graph, otherwise only the counter is increased.

Deletions are performed in a two-step algorithm *delete* and *re-derive*. The deletion step propagates each deletion once and internally decreases the counter for every recursive deletion encountered. All elements with a count of zero are removed from the operator. The re-derivation step propagates additions for all elements that still have a positive count. This step is necessary because the deletion phase is an over-approximation. This treatment of recursion is generally applicable regardless of the underlying elements of a relation.

3.2 Materialization and Self-Maintainability

Materialization refers to data that is permanently required in memory for correct incremental computations. The amount of data that must be stored depends on the operator. For example, the join operator (cf. Figure 5) requires two data structures leftIndex and rightIndex that are used during the propagation of element additions and removals. These data structures are indispensable to the correctness of the propagation semantics of joins and are permanently kept in memory, since i3QL assumes that a new addition or removal can happen at any time.

Self-Maintainability refers to operators that require no materialization at all. For these operators the result of a change propagation can be determined solely based on the data associated to the change. For example, selections are self-maintainable, since the change propagation only has to test the element via the predicate function and notify observers if the predicate evaluates to true.

The concepts of materialization and self-maintainability stem from traditional view maintenance approaches in databases where materialization is linked to persisting the data on a hard-disk. For live data views, like in the case of i3QL, materialization is important for efficiently processing large quantities of input with a limited amount of main memory. Relational algebra allows reasoning over memory consumption of a query on a per-operator basis. To give an intuition of the amount of materialization that i3QL operators require, we categorize operators in increasing order of memory usage:

- Selection, projection, union all, and unnesting are self-maintainable.
- The materialization of aggregation varies depending on the provided functions (distributive/non-distributive) and whether grouping is present. For example, for distributive non-grouped aggregations only a single value needs to be materialized.
- Duplicate elimination and recursion need to materialize a single map (dictionary) with a size corresponding to the number of added elements.
- Difference, union, and intersection require to retrieve any element from each operand during addition and retrieval. In practice they can perform better than materializing all elements, since their change propagation can be reduced to omit elements present in both operands. Thus, they perform well for inputs with a large overlap.
- Cartesian product and join must materialize two maps (dictionaries) with a size corresponding to the number of added elements of both operands.

Finally, interaction among operators influences materialization. For example, many queries contain operators with materialization, but filter out a large number of elements via selections before materialization is needed. Such queries consume minimal amount of memory because only elements in input relations relevant to the computation are materialized. Since, early filtering is a deciding factor for these queries, algebraic optimizations (next section) play a major role for materialization, since they can move the selections to the earliest possible stage in the operator graph.

4. Query Optimizations

i3QL uses *algebraic optimizations* and *partial evaluation* to improve the performance of queries. The algebraic optimizations we implemented are mostly standard database optimizations that reorder the application of relational operators (cf. [48]). For example, we want to apply selections as soon as possible, so that less changes are propagated to the rest of the operator graph. Due to i3QL's integration into a programming language, and more specifically due to its use of higher-order abstract syntax [34], these optimizations frequently require analysis (and/or transformation) of Scala functions that users supply to the algebraic operators.

For example, a selection on top of a join of students and council members uses a predicate function of type (Student, Member) => Boolean. To apply the selection be-

fore the join we need to analyze the predicate and split it into two predicates applying to students and members. The pushed-down selections then have predicates of type Student => Boolean and Member => Boolean, respectively.

To implement these kind of optimizations, we use *lightweight modular staging* (LMS) [38], a Scala framework that supports the inspection and transformation of functions. LMS provides a reified expression tree for Scala functions. LMS users write their programs almost as if using plain Scala. The only difference is that types must be wrapped in the special type constructor Rep (short for representation). For example, with LMS, the query for filtering students with the name "Sally" looks as follows:

```
SELECT (*) FROM students
  WHERE ((s: Rep[Student]) => s.firstName == "Sally")
```

By using the Rep type the body (s.firstName == "Sally") is compiled into a function that returns the expression tree of its own body, instead of performing the actual computation. We use these expression trees to analyze and optimize i3QL queries. i3QL applies algebraic optimizations and partial evaluation bottom-up in the operator graph until no more optimizations are applicable. After optimization, the optimized predicates are compiled to perform the actual computation.

In the following, we summarize the optimizations supported by i3QL. All optimizations preserve semantics given that all functions used in the query are pure. For impure functions, algebraic optimizations would be invalid since they change the order of side-effects when reordering operators. Since i3QL adopts the modular approach of LMS, it is easily extensible with new optimizations.

i3QL supports the following *algebraic optimizations*:

Pushing down selections in the operator graph allows early filtering of elements not needed in the computation.

Operator fusion of multiple operators of the same type, e.g., the fusion of two selections over students. Operator fusion reduces the size of the operator graph.

Nested sub-query elimination derives a single operator tree in the presence of sub-queries nested in a WHERE clause. For example, consider the query below that selects students such that no student council member with the same name exists.

```
1 SELECT (*) FROM students WHERE (
2   (s: Rep[Student]) => NOT EXISTS (
3     SELECT (*) FROM council WHERE
4       ((m: Rep[Member]) => m.name == s.name)
5 ))
```

The existential quantification (line 2) contains an inner sub query (line 3) that depends on the elements obtained in the outer query m.name == s.name (line 4). Without this optimization we must instantiate a new operator tree for the inner query, whenever an element s is added to the outer query in order to bind the expression s.name to a concrete value.

Sub-query sharing compiles equivalent parts of the query to the same operator graph. The compiled operator graph serves as input to multiple observers, thus removing redundant computations and reducing the overall query execution time.

Various simplifications that (i) remove superfluous operators or (ii) replace costly operators with cheaper alternatives. For example, (i) if the expression of a selection predicate has reduced to **true** after some other optimization, the selection is superfluous, or (ii) we can replace the intersection over two selections on the same relation by $\sigma_1(a) \cap \sigma_2(a) = \sigma_1(\sigma_2(a))$.

Automated Indexing is performed for join operators as late as possible. Indexing is a memory intensive operation. In relational algebra, indices are defined over entire input tables. Yet, the design goal of i3QL is to materialize as little as possible, i.e., not to materialize indices for all input elements. Thus, indexing is performed only after filtering. Such indices are referred to as *filtered indices* in the database context, where they are specified by the database designer. In i3QL the filtered indices are created automatically for all contained joins based on the optimization of a query.

i3QL supports the following optimizations known from *partial evaluation*.

Function inlining replaces calls to a function by an expression tree corresponding to the function's body. This enables further algebraic optimizations and partial evaluation.

Constant propagation specializes computations based on constants occurring in the computation. Constant propagation is particularly useful after inlining of user functions.

Common subexpression elimination performs computations for equivalent subexpression only once; equivalent subexpressions share their result. For example, multiple calls to a method firstName within a selection predicate only lead to a single execution of the method. Common subexpression elimination is built into LMS.

5. Case study: An Incrementalized Parser

To demonstrate the broad applicability of i3QL and to illustrate how to model an algorithm using relational algebra, we developed an incremental parser as a case study of i3QL. Based on a context-free grammar, the parser incrementally adapts to changes in the input sentence. Moreover, our parser also incrementally adapts to changes of the grammar, such as the introduction of new terminal symbols or the modification of a production.

We selected incremental parsing as a case-study because it is a non trivial application. Existing incremental parsers use highly specialized incrementalization engines. For example, a common approach is to incrementalize the push-down automata of the parser (cf. [50] for an overview). In contrast, i3QL uses a general-purpose incrementalization engine. Clearly, our goal is not to compete with state-of-the art

algorithms for incremental parsing, but to demonstrate that our approach is general enough to be applicable to such specialized domains. In particular, a performance comparison of our incremental parser with existing parsers is not in scope of this paper.

Context-Free Grammars. We use a standard notion of a context-free grammar $G = \langle N, T, P, s\rangle$, where N is the set of *nonterminals* and T is the set of *terminals*. N and T must be disjunct and together form the set $V = N \cup T$ of *symbols*. P is the set of *productions* of the form *head* $\longrightarrow$ *body*, where *head* $\in V$ and *body* $\in V^*$ is a sequence of symbols. $s \in N$ is the *start symbol*.

Figure 8 shows an example grammar for English sentences that we use as a running example throughout this section. The nonterminals are *S* (*sentence*), *NP* (*noun phrase*), *VP* (*verb phrase*), *AdjP* (*adjective phrase*), *Noun*, *Verb*, *Adj* (adjective) and *Adv* (adverb). Nonterminal *S* is the start symbol. As terminals we use the words that occur in the sentence borrowed from Chomsky [7] and depicted in Figure 9.

$$S \longrightarrow \mathit{NP}\ \mathit{VP} \tag{1}$$
$$\mathit{NP} \longrightarrow \mathit{Noun} \mid \mathit{AdjP}\ \mathit{Noun} \tag{2}$$
$$\mathit{VP} \longrightarrow \mathit{Verb} \mid \mathit{Verb}\ \mathit{Adv} \tag{3}$$
$$\mathit{AdjP} \longrightarrow \mathit{Adj} \mid \mathit{Adj}\ \mathit{AdjP} \tag{4}$$
$$\mathit{Noun} \longrightarrow \text{green} \mid \text{ideas} \mid \text{sleep} \tag{5}$$
$$\mathit{Verb} \longrightarrow \text{green} \mid \text{sleep} \tag{6}$$
$$\mathit{Adj} \longrightarrow \text{colorless} \mid \text{green} \tag{7}$$
$$\mathit{Adv} \longrightarrow \text{furiously} \tag{8}$$

Figure 8: Grammar for simple English sentences.

colorless	green	ideas	sleep	furiously
Adj	*Adj*	*Noun*	*Verb*	*Adv*

Figure 9: A grammatically correct sentence.

We implement a bottom-up parser that starts at the sequence of terminals and tries to find nonterminals that can derive subsequences of the input sentence. The parser accepts an input sentence if the start symbol can derive it. For example, for the sentence in Figure 9, the parser determines that the sentence starts with an adjective phrase (*AdjP*) followed by a noun yielding a noun phrase (*NP*). The last two terminals can be derived as a verb phrase (*VP*). Together, the noun phrase and verb phrase can be derived as a sentence (*S*) that covers the full input. The parsing process is complicated by ambiguities in the language, e.g., the word "green" can also be used as a noun or a verb, and the word "sleep" can also be used as a noun. To this end, our parser derives all possible parse trees. We elaborate more on the parsing process in Section 5.2.

5.1 Data Definitions for an Incrementalized Parser

The incrementalized parser uses three data types:

```
case class Token(word: String, position: Int)
case class Terminal(word: String, nonterminal: String)
case class Rule(head: String, body: Seq[String])
```

A `Token` denotes one word in the input sentence together with its (zero-based) position, e.g., `Token("colorless", 0)` represents the first word in the sentence in Figure 9. To simplify our parser, we split productions into those exclusively dealing with terminals (productions (5)–(8) in Figure 8) and those exclusively dealing with nonterminals (productions (1)–(4)). For each nonterminal a terminal belongs to, there is one `Terminal` production such as `Terminal("sleep", "Noun")` and `Terminal("sleep", "Verb")` for the word `sleep`. For the other productions, we get one `Rule` for each alternative, where the body represents a sequence of nonterminals. For example, `Rule("S", Seq("NP", "VP"))` represents the first production in Figure 8.

We represent the input sentence and productions as tables in i3QL:

```
val sentence : Table[Token]
val terminals : Table[Terminal]
val rules : Table[Rule]
```

Users can add, remove, and update any of the three tables. In particular, they can trigger updates for changes to the input sentence:

```
sentence += Token("ideas", 5)
sentence.update(Token("sleep", 3), Token("green", 3))
sentence -= Token("colorless", 0)
```

5.2 Chart Parsing

Before we define the incremental parser, we briefly describe *chart parsing* [21], which is the underlying approach our parser uses. A chart parser infers a set of edges—organized in a matrix called *chart*—from the input, where edges denote what nonterminal a certain subsequence of the input sentence can be. The parser uses several inference rules to combine existing edges into new ones, until the whole sentence is covered. Several variations of chart parsing exist (cf. [21]); the concepts presented here serve to illustrate the general principles.

The key idea is to organize the sentence in a *graph* with *nodes* denoting the position of words and *edges* denoting the nonterminals found for the word(s) between the nodes. For example, Figure 10 shows one possible derivation of edges for the sentence in Figure 9. A sentence is correct according to the grammar, if there exists an edge spanning from the first node to the last node and the nonterminal on the edge is the start symbol, e.g., the edge *S* spanning from node 0 to 5. The chart parser distinguishes *passive* and *active* edges. Passive edges denote the full application of a production rule.

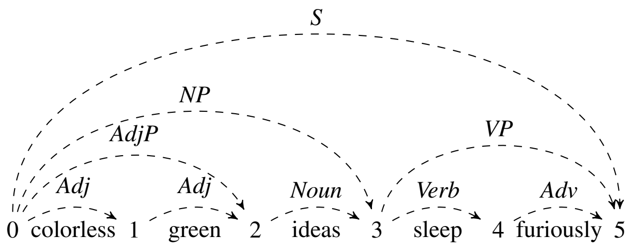


Figure 10: Parsed nonterminals as edges in a graph.

For example, all edges shown in Figure 10 are passive. In contrast, active edges denote intermediate steps that arise after parsing parts of the body of a production rule. Figure 11 depicts an active edge for the rule $S \longrightarrow NP\ VP$ after parsing the noun phrase (*NP*) in the sentence. The dots (. . .) indicate that we have already parsed part of the body. *VP* is the next nonterminal that needs to be parsed in the rule.

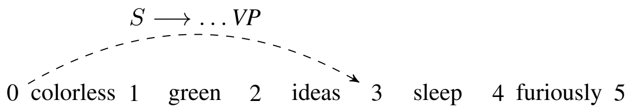


Figure 11: An active edge after parsing the noun phrase.

To continue on an active edge, we need to find a passive edge that starts at the target node of the active edge and derives the required nonterminal. For example, in Figure 11, we need a passive edge starting at node 3 for nonterminal *VP*. The combined edge then spans from the start node of the active edge to the target node of the passive edge. If there are no more nonterminals in the body of the rule, the combined edge becomes passive itself, otherwise it remains active.

5.3 Incrementalized Chart Parser

The parser defined in this section infers edges (passive and active) and combines active edges with passive edges until a fixpoint is reached, i.e., no new edges can be inferred. We define a class Edge as follows:

```
case class Edge(start: Int,
                end: Int,
                nonterminal: String,
                body: List[String]) {
  def isActive = body != Nil
}
```

All edges have indices for start/end nodes and the name of the nonterminal they derive. The body of an edge represents those nonterminals that still have to be parsed. Thus, an edge is passive if the body is empty.

We divide the parsing algorithm into three steps.

1. We *construct passive edges for tokens* from the input sentence by matching tokens with terminal productions defined in the grammar.
2. We *construct new edges* by applying rules to existing passive edges if the first nonterminal in the rule's body matches the passive edge.
3. We *combine edges* (an active with a passive edge) as explained in Section 5.2 above.

We perform the latter two steps repeatedly until no new edges can be derived either via construction or combination.

Deriving Terminal Edges. Query terminalEdges below encodes the first step by correlating elements from the tables sentence and terminals. The correlation is a join on the attribute word of the input token and the terminal, e.g., the elements Token("green", 1) and Terminal("green", "Verb") have matching words and, thus, contribute to the result of the join. The results of the query are projected using function terminalEdge, which constructs a passive edge for each token and terminal. The edges start at the node with index of the token (in.position) and end at the next node. Their nonterminal is that of the matched terminal production and they are passive, i.e., have an empty body.

```
val terminalEdges: Relation[Edge] =
  SELECT (terminalEdge) FROM (sentence, terminals)
  WHERE ((in: Rep[Token], t: Rep[Terminal]) =>
          in.word == t.word)

def terminalEdge(in: Rep[Token], t: Rep[Terminal]) =
  Edge (in.position, in.position+1, t.nonterminal, Nil)
```

Constructing and Combining Edges. Since step two and three are mutually recursive, we encode them in a single recursive query derivedEdges shown at the end of this subsection. The encoding uses the two helper functions constructedEdge and combinedEdge:

```
def constructedEdge(r: Rep[Rule], e: Rep[Edge]) =
  Edge (e.start, e.end, r.head, r.body.tail)

def combinedEdge(act: Rep[Edge], pas: Rep[Edge]) =
  Edge (act.start, pas.end, act.nonterminal, act.body.tail)
```

Function constructedEdge constructs a new edge by applying a production rule from the grammar to a passive edge that derives the first nonterminal required in the body of the production. The applicability of the rule is not checked in constructedEdge but in the query. The constructed edge spans the same indices as the passive edge from which it is constructed and has the nonterminal of the grammar rule, i.e., the part of speech that is parsed once all nonterminals in the body of the rule have been derived. The body of the new edge initially contains all nonterminals of the body, except the first one, since this was the nonterminal derived by the passive edge.

Function combinedEdge constructs a new combined edge from an active and a passive edge, where the passive edge starts at the end of the active edge and derives the next nonterminal required by the body of the active edge. Again, the applicability is not checked in combinedEdge but in the query. The constructed edge spans the indices from the start of the

active edge to the end of the passive edge. The resulting nonterminal is that of the active edge and the remaining nonterminal are all those of the active edge except the first nonterminal, which was derived by the passive edge.

```
1 val derivedEdges: Relation[Edge] =
2   WITH RECURSIVE (
3     (edges: Relation[Edge]) =>
4          terminalEdges
5       UNION ALL (
6          SELECT (constructedEdge)
7          FROM (rules, edges)
8          WHERE ((r: Rep[Rule], e: Rep[Edge]) =>
9            NOT(e.isActive) AND r.body.first == e.nonterminal)
10      UNION ALL (
11         SELECT (combinedEdge)
12         FROM (edges, edges)
13         WHERE ((l: Rep[Edge], r: Rep[Edge]) =>
14           l.isActive AND NOT(r.isActive)
15           AND l.end == r.start
16           AND l.body.first == r.nonterminal))))
```

Figure 12: The query for incremental parsing.

A Query for Incremental Parsing. The query in Figure 12 combines all three steps described above to implement the incremental parser. The query recursively derives new edges until a fixed point is reached. All derived edges are added as results to a temporary result relation named edges (line 3). The terminal edges (line 4) serve as the starting input of the recursion. They are combined via UNION ALL with newly constructed edges and newly combined edges, defined in the respective sub-queries. Newly constructed edges are derived via a join of the rules relation with all recursively derived edges, where the edge e must be passive and derive the nonterminal expected first in the body of the rule. Similarly, we combine edges, checking that the first edge is active, the second edge is passive, they end respectively start at the same index, and the second edge derives the nonterminal next required by the first edge.

We can use query derivedEdges to determine whether an input is accepted by the grammar, which is the case if an edge exists that spans the whole input, is not active, and derives the start symbol. We can also easily extend our parser to compute parse trees by additionally storing (i) which rule lead to the creation of a new edge in constructedEdge and (ii) which passive edges have been combined with an edge in combinedEdge. This extension does not influence the incrementalization or the main query derivedEdges.

Discussion. We have shown how to incrementalize the non-trivial problem of parsing with i3QL. The incremental parser adapts to changes of the input sentence as well as to changes of the grammar. In contrast to most previous work on incremental parsing, we used our general-purpose incrementalization engine. This case study demonstrates the broad applicability of i3QL.

i3QL computations are inherently bottom-up because an i3QL query inductively derives new facts from ground facts and already derived facts. For example, our parser incrementally derives facts about which substring can be generated from which nonterminal. Such bottom-up computations are natural in many contexts and fit well to the relational programming style of i3QL.

Nevertheless, many computations are more naturally expressed as top-down computations traditional to functional programming. For example, it is not obvious how to represent a recursive descent parser or a syntax-directed interpreter function as an equivalent i3QL computation. In ongoing work, we are investigating how to automatically translate top-down computations and the used data types to i3QL queries and relations.

6. Performance Evaluation

This section presents the results of our empirical evaluation that was designed to answer the following questions:

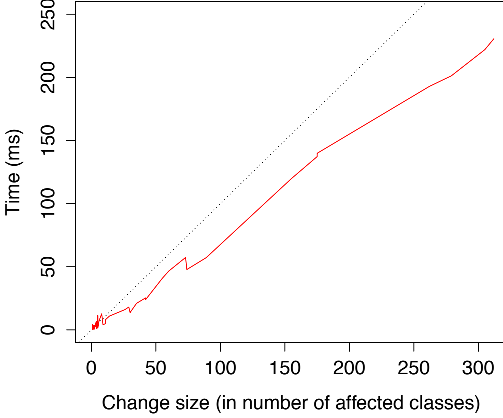
1. Does i3QL make the execution time needed to update the result of a computation to an input change increase in a piecemeal way with the size of the changes?
2. Do computations encoded with i3QL execute significantly faster than non-incremental executions of the same computations in the presence of input data changes?
3. Is the memory price to pay for enabling incrementalization of computations reasonable?
4. What is the effect of the optimizations on performance?

Evaluation Setup. The evaluation benchmark consists of 15 static analyses from FindBugs [19] implemented as i3QL queries (a list of the 15 analyses appears in Table 2). The analyses take Java bytecode as input and produce a list of identified bugs. It is desirable to continuously update the list of bugs as the code base changes to make FindBug analyses behave like analyses built into the compiler and scale with the size of analyzed projects.

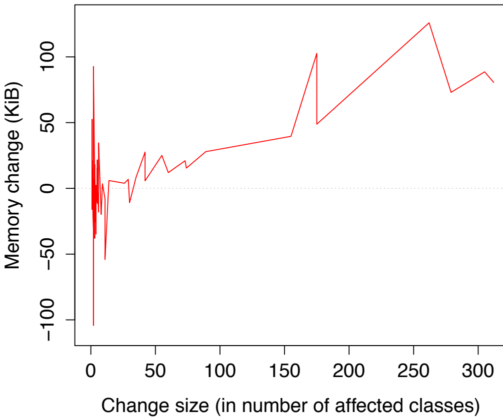
As test data for the analyses we used the revision history of Vespucci[4], a project for modeling and continuously checking a program's structural dependencies [29]. The initial revision of Vespucci consists of 381 class files, followed by a total of 242 changes (additions, removals, updates), each affecting between 1 and 312 class files. We benchmarked each analysis by running it on the initial 381 class files, and then replaying the 242 changes in order. We ran the benchmarks on a 64-bit Windows 8.1 machine with an Intel Core i7 3.4GHz processor and 8GB memory, using JVM version 1.7 and Scala 2.10.2.

Overall performance. For each optimized analysis, we measured the run time and variation in memory consumption for each change. We discuss the mean time and mean mem-

[4] http://www.opal-project.de/vespucci_project



(a) Update time per change.



(b) Variation in memory consumption per change.

Figure 13: Mean time and memory consumption per change for all 15 Findbugs analyses implemented with i3QL.

ory consumption averaged over all 15 analyses (the standard deviation was low). The benchmarks have been performed 5 times for each analysis with an additional 4 warmup iterations. The end result for each analysis is the average of those 5 measurements.

Figure 13a shows the *time* it takes to update the analysis result for each change, that is all classes that are changed simultaneously. The x-axis shows the size of a change and the y-axis shows the time that is needed to update the analysis result. We ordered data points according to the change size. The plot shows that the time it takes to update the overall analysis result is roughly linear in the change size; it does not seem to depend on the history of changes already processed or on the size of the initial revision.

A comparison with the non-incremental analyses implemented in FindBugs unsurprisingly shows an order-of-magnitude improvement in run-time performance. Running an analysis in FindBugs for the initial revision takes 534 ms on average, whereas i3QL only takes 129 ms on average. Since the FindBugs implementation will take roughly the same time for each revision, checking all revisions amounts to about 130 seconds, whereas i3QL only requires little more than 2 seconds. This means that on our test data i3QL provides a speedup of 65x compared to the non-incremental FindBugs analyses.

Figure 13b shows the variation in *memory* for each change. Like in the graphic above, the x-axis displays the size of one change. The y-axis shows how much more (or less) memory is needed after the change has been processed. We run the garbage collector after the replay and propagation of each change to get accurate measurements. The plot shows that the memory impact of a change is moderate (at most 120 KiB for a change affecting more than 250 classes), which means that auxiliary data stored to enable incrementalization does not have a significant effect. For comparison, the class files of the initial revision account for 1.39 MiB.

Effect of Optimizations. To measure the effect of optimizations in our implementation, we run the analyses in two configurations of i3QL: with and without the optimizations presented in Section 4. The results are shown in Table 2. The rows of the table are indexed by the analyses (A through O). The execution times of the analyses with i3QL optimizations turned off respectively on are shown in the first respectively the second column of the table. The last column shows the speedup achieved by turning optimization on.

By comparing the results, we observe that in 6 analyses, the optimizations improved performance significantly ($> 20\%$), most prominently for analyses A, C, D, and E. In the other analyses, the effect of optimization is less significant. In case of analysis B, the optimizations even had a small negative effect. The reason for such a difference is twofold.

First, analyses A, C, D, E, F, G, K, N contain queries over the byte-code instruction of the codebase—a much larger table than the table of method declarations, for example. Thus, the optimizations have a greater effect. Second, A, C and D use equi-joins, which further amplifies this effect. Equi-joins are only created with optimizations; without optimizations those relations are simply cross products and selections. Equi-joins are very benefiting for large relations (e.g. instructions), since the join operator hashes its values—opposed to cross product and selection, which have to iterate over all elements in order to find equal elements. The difference between analysis A and C, D is that analysis A joins two large instruction relations while C and D join an instruction relation with other relations that are not as large.

Summary. Our performance evaluation shows that i3QL queries run an order of magnitude faster than their non-incremental counterparts, without requiring much memory. The run time for a change is linear in the change size and independent of size of the initial revision of the input. The evaluation clearly shows that optimizations bring significant performance improvements; especially, they ensure that the analyses execution time scales well for big sets of input data.

	Time (no opt.)	Time (with opt.)	Speedup
A	1329.936	2.312	575.22
B	2.084	2.162	0.96
C	10.889	2.299	4.74
D	39.452	2.254	17.50
E	4.842	2.491	1.94
F	2.464	2.152	1.15
G	2.345	2.083	1.13
H	2.257	2.121	1.06
I	2.198	2.096	1.05
J	2.109	2.055	1.03
K	3.244	2.345	1.38
L	2.264	2.100	1.08
M	2.185	2.109	1.04
N	3.785	2.317	1.63
O	2.532	2.140	1.18

FindBugs patterns from http://findbugs.sourceforge.net/bugDescriptions.html: A=BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION, B=CI_CONFUSED_INHERITANCE, C=CN_IDIOM, D=CN_IDIOM_NO_SUPER_CALL, E=DM_GC, F=DM_RUN_FINALIZERS_ON_EXIT, G=DP_DO_INSIDE_DO_PRIVILEGED, H=EQ_ABSTRACT_SELF, I=FI_PUBLIC_SHOULD_BE_PROTECTED, J=IMSE_DONT_CATCH_IMSE, K=MS_PKGPROTECT, L=MS_SHOULD_BE_FINAL, M=SE_BAD_FIELD_INNER_CLASS, N=SW_SWING_METHODS_INVOKED_IN_SWING_THREAD, O=UG_SYNC_SET_UNSYNC_GET

Table 2: Total time in seconds and optimization speedup for analyzing the initial revision and replaying all 242 changes.

7. Related Work

Related work spans over different fields such as language integration of queries, in-memory collection, incrementalization, automatic recomputation of query results and combinations thereof.

Embedding of SQL-like Queries In recent years an increasing amount of works have provided integrations of database queries into programming languages. Existing language-integrated query systems include LINQ [27], Ferry [15], and ScalaQL [12]. These systems have deep roots in functional programming and are influenced by earlier works on functional query languages, e.g., HaskellDB [22] or Kleisli [53]. The goal of these solutions is to integrate standard database technologies with general purpose programming languages to provide a better handling of persistence related code. The main issues addressed are (i) the type-correctness for database queries w.r.t. schemas in persistent databases and (ii) query optimization in the programming language (i.e., on the client side) to reduce the amount of persistent data moved between client and database.

In general, these language integrations are based on (monad) comprehensions (cf. Gray et al. [14]), which can be understood as succinct notation for collection operations. The simplicity of the notation lies in the fact that many SQL-operators, such as selections and projections have immediate translations to comprehension operations, such as filter and map.

In-memory Collections A major motivation for the success of LINQ is that the same SQL-like query language applies to heterogeneous data sources. These include databases, as already discussed, but also in-memory collections, which exploit syntax tree rewriting to introduce optimizations. The Java Query Language (JQL) [51] provides declarative queries over Java collections. Yet, JQL is very specialized w.r.t. the supported relational operators – mainly joins. The approach uses standard optimization techniques, primarily join ordering, to improve runtime performance. Giarrusso *et al.* [13] describe SQuOpt, an optimized implementation of Scala collections that, similar to LINQ, reifies the query syntax tree for optimization. Differently from i3QL these solutions are not incremental.

To speed up queries that are repeatedly run on the same dataset with minor modifications, in-memory collections have been extended with incrementalization techniques. The commercial library LiveLINQ [25] provides caching of result collections and indexing to optimize joins. All cached results, and indices, as well as the underlying base relations, are then fully materialized in main-memory. The range of incrementalized operations provided by LiveLINQ is not well documented. However, the basic and set-theoretic operators are declared in the standard LINQ API. Nevertheless, there is no concrete documentation on how exactly individual operators are incrementally maintained. Similar to LiveLINQ, the work in [52] extends JQL with caching and incremental maintenance of query results. The approach integrates with Object-oriented programming in that query results are cached and incrementally maintained whenever the collections and the (mutable) objects involved in the query are updated. Nerella et al. [31] extended JQL to use information from earlier executions to optimize the construction of the query plans. Scala.react's incremental collections [26] are designed to speed up reactive computations that involve data structures.

All these solutions feature completely materialized collections and caching (i.e., materialization) of results. This approach is conceptually equivalent to materialized views in databases: In-memory collections for base relations and for results are materialized alongside any auxiliary data structures required for view maintenance. i3QL, instead, materializes as little data as possible to allow scalability to large(r) amounts of data. In i3QL, the base relations and the view results are transient data and only the auxiliary data structures for incremental view maintenance are materialized.

Liu et al. [24] propose a systematic method to analyze a program and generate code for caching and incrementalize set comprehensions. The approach is further automated in a follow-up work by Rothamel and Liu [39]. Compared to JQL, the solution proposed by Rothamel and Liu better

integrates in the Object-oriented model, and supports aliasing (e.g., updates to fields referred to by other object fields), which results in a broader set of queries that can be expressed. A major difference with our work is that, in contrast to these works, which statically generate more efficient code, i3QL works at runtime and, thanks to LMS, can perform optimizations leveraging dynamic information.

Constraints and Functional Dependencies Maintenance of query results in i3QL can be seen as providing support for *constraints*. A number of approaches share the concept of specifying a functional dependency that is automatically enforced by the runtime.

Constraint-based languages, like Kaleidoscope [11] integrate constraints into OO languages. Constraints are enforced according to a priority ranking. The Garnet and Amulet [30] graphical toolkits support automatic constraint resolution to relieve the programmer from manual updates of graphical interfaces. Demetrescu et al. [9] implemented dataflow constraints based on a runtime environment which natively supports reactive memory.

Reactive programming supports dedicated abstractions for functional dependencies – referred to as *signals* or *behaviors*. Researcher proposed extensions to existing languages that support these features, such as Flapjax [28] (Javascript), FrTime [8] (Scheme), REScala [44] and Scala.react [26] (Scala). Similar to i3QL's operator graph, these solutions maintain in memory a tree of dependent values. When leaves are updated, the change is propagated thorough the tree. A major limitation of these approaches is to recompute dependent values from scratch every time an input (partially) changes, which severely limits performances.

Self-adjusting computation [1] is about automatic derivation of the incremental version of a program from a traditional one. The language runtime builds a graph of the dependencies among values in the program and incrementally recomputes only the parts that are affected by a change in the input. This technique has been recently applied in the context of big data to obtain an incremental version of MapReduce [2]. DITTO [45] is an automatic incrementalizer for invariants check of datastructures. DITTO combines memoization and dependency tracking to speedup invariants evaluation by reusing previous executions. In contrast to DITTO and Self-adjusting computation, which target generic computations that can be optimized only by generically tracking dependencies and recomputing changed values, i3QL is based on a set of relational operators for which well-known specific optimizations (e.g., push-down of selections) exists.

View Maintenance in Database Systems Several approaches have been developed for the incremental maintenance of materialized views.

Counting algorithms keep a multiplicity counter, i.e., the number of derivations for a tuple, as extra information. Upon insertion or deletion of a tuple the counter is incremented or decremented accordingly and tuples are deleted when their counter reaches zero. Similar variants were introduced by Shmueli et al. [46] and Blakeley et al. [4]. i3QL employs counting as a technique for several operators, e.g., duplicate elimination and recursion. *Algebraic query rewriting* defines different change propagation expressions that can be simplified for insertions and deletions from the base relations in different ways. The simplified relational expressions compute the change to the view without doing redundant computation. The idea is first introduced in [32] under the term *finite differencing* and was used subsequently in [35] for view maintenance of select-project-join (SPJ) views with set semantics. *Logic query rewriting* is a similar technique proposed in the area of deductive databases [18]. i3QL's optimizations also employ a query rewriting technique, however queries are optimized uniformly without considering insertions and deletions separately. a separate treatment requires adaptations to the optimizer, which must include a time and memory trade-off, since many operators, e.g., join, union or difference, share intermediate results for insertions and deletions. *Active rules* have been proposed by Ceri and Widom [5] in the context of extending an existing active database system with rules to support incremental view maintenance. They define an algorithm to generate active rules that manipulate the view by inserting or deleting appropriate tuples, e.g., using SQL insert or delete statements. This work is complementary to i3QL, which directly translates views to incrementally maintained relational operators. *Memoing* originated from logic programming languages that use the SLD[5], a backtracking search through all rules that make up a logic program. Memoing stores intermediate results for already computed procedure calls [41]. The term *tabling* [42] is commonly used for memoing in logic languages, as the already computed results are stored and retrieved from a result table. Memoing is an indispensable technique for the speed of top-down evaluation in logic programs, e.g., to store the results of a filtering predicate and not test the same elements multiple times. i3QL employs a bottom-up technique and changes are propagated only once. Intermediate results are stored also in i3QL, but only if necessary for the incremental evaluation of the operators.

Optimization Techniques We provide an overview of optimization techniques that have been proposed to in the context of materialized view maintenance. *Identifying irrelevant modifications* tests whether a particular modification affects a given view (i.e., it is relevant). If the test is negative, no maintenance operations are necessary. In [3] a proposal is made to test SPJ views in relational algebra. This work was motivated by the fact that maintenance in persistent databases is costly. i3QL propagates data through the operator graph without prior testing and relies on pushing down selections for early filtering. Since computations

[5] SLD resolution is termed after the initial letters in "**L**inear resolution with **S**election function for **D**efinite programs"; cf. [23] for a treatment of the SLD calculus.

are in-memory, the benefits of additional tests over early filtering are unclear. *Efficient storage of views* minimizes memory requirements for view maintenance. For example, Roussopoulos [40] proposes *view caches*, a data structure that stores only pointers to tuples in the underlying relations that contribute to a view, instead of storing the concrete values of the tuples. Similarly, [43] defines a compression scheme that relies on sharing supporting facts in logic databases. Since i3QL is embedded into an OO programming language we have a pointer semantics and already benefit from the reduced memory. Further compression techniques are complementary, but have not yet been evaluated for i3QL. *Adapting views after redefinitions* maximizes the reuse of results and auxiliary data when the definition of a view is redefined with slight changes. The basic technique is discussed for relational algebra in [17]. i3QL's sub-query sharing reuses all operators and results from the bottom up, until the point where the deviation from the original query occurred. Results further up in the operator graph are completely re-computed and i3QL could benefit from such an optimization in an environment with frequently changing queries. *Optimizing queries using maintained views* was proposed to speed up query processing time for arbitrary queries and not only the queries of the incrementally maintained views [6]. The technique finds alternative formulations of queries that incorporate already existing views maintained by the database system. This technique is not incorporated into i3QL, which optimizes single queries and shares subqueries. Due to i3QL's optimizations queries are in essence normalized, hence some alternatives will be reused. Yet, additional analysis on the expression level is required for a broader treatment, e.g., filters with selection expressions $i - 1 > 0$ and $i > 1$ are not treated as alternatives in i3QL.

8. Conclusion

We presented i3QL, a general-purpose query language for incremental computation embedded in Scala. i3QL provides general recursion for expressiveness and combines optimizations from relational algebra with partial evaluation to achieve high performance. We have demonstrated the applicability of i3QL by developing an incremental parser and 15 FindBugs analyses, and our performance evaluation shows an order-of-magnitude improvement compared to non-incremental computations.

Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 01IC12S01V (SINNODIUM) and by the European Research Council, grant No. 321217.

References

[1] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):3:1–3:53, Nov. 2009.

[2] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[3] J. A. Blakeley, N. Coburn, and P.-V. Larson. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, Sept. 1989.

[4] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 190 –200, mar 1995.

[7] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, September 1956.

[8] G. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In P. Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin Heidelberg, 2006.

[9] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 407–426, New York, NY, USA, 2011. ACM.

[10] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic incrementalization of prolog based static analyses. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, pages 109–123, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 77–88, New York, NY, USA, 1990. ACM.

[12] M. Garcia, A. Izmaylova, and S. Schupp. Extending Scala with database query capability. *Journal of Object Technology*, 9(4):45–68, 2010.

[13] P. G. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, and C. Kästner. Reify your collection queries for modularity and speed! In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Develop-*

ment, AOSD '13, pages 1–12, New York, NY, USA, 2013. ACM.

[14] P. M. D. Gray, L. Kerschberg, P. J. H. King, and A. Poulovassilis. *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[15] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 1063–1066, New York, NY, USA, 2009. ACM.

[16] A. Gupta and I. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[17] A. Gupta, I. S. Mumick, and K. A. Ross. Adapting materialized views after redefinitions. *SIGMOD Rec.*, 24(2):211–222, May 1995.

[18] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM.

[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[20] P. Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.

[21] M. Kay. Readings in natural language processing. chapter Algorithm schemata and data structures in syntactic processing, pages 35–70. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[22] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[23] V. Lifschitz. Principles of knowledge representation. chapter Foundations of Logic Programming, pages 69–127. Center for the Study of Language and Information, Stanford, CA, USA, 1996.

[24] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 473–486, New York, NY, USA, 2005. ACM.

[25] LiveLinq Web site. `http://www.componentone.com/SuperProducts/LiveLinq/`.

[26] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *Proceedings of the 27th European conference on Object-Oriented Programming*, ECOOP'13, pages 707–731, Berlin, Heidelberg, 2013. Springer-Verlag.

[27] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):22:1–22:50, Dec. 2008.

[28] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. *SIGPLAN Not.*, 44(10):1–20, Oct. 2009.

[29] R. Mitschke, M. Eichberg, M. Mezini, A. Garcia, and I. Macia. Modular specification and checking of structural dependencies. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 85–96, New York, NY, USA, 2013. ACM.

[30] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *IEEE Trans. Softw. Eng.*, 23(6):347–365, 1997.

[31] V. Nerella, S. Surapaneni, S. Madria, and T. Weigert. Exploring optimization and caching for efficient collection operations. *Automated Software Engineering*, 21(1):3–40, 2014.

[32] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In *Advances in Data Base Theory*, pages 171–209, 1982.

[33] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 802–813. VLDB Endowment, 2002.

[34] F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, June 1988.

[35] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[36] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[37] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *SIGPLAN Not.*, 46(2):127–136, Oct. 2010.

[38] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[39] T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 55–66, New York, NY, USA, 2008. ACM.

[40] N. Roussopoulos. An incremental access method for viewcache: concept, algorithms, and cost analysis. *ACM Trans. Database Syst.*, 16(3):535–563, Sept. 1991.

[41] K. Sagonas, T. Swift, and D. S. Warren. Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2):442–453, May 1994.

[42] D. Saha and C. Ramakrishnan. Incremental evaluation of tabled logic programs. In C. Palamidessi, editor, *Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 392–406. Springer Berlin Heidelberg, 2003.

[43] D. Saha and C. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evalua-

tion. In M. Gabbrielli and G. Gupta, editors, *Logic Programming*, volume 3668 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin Heidelberg, 2005.

[44] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[45] A. Shankar and R. Bodík. Ditto: automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 310–319, New York, NY, USA, 2007. ACM.

[46] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 240–255, New York, NY, USA, 1984. ACM.

[47] O. Sumer, U. Acar, A. T. Ihler, and R. R. Mettu. Efficient bayesian inference for dynamically changing graphs. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1441–1448. MIT Press, Cambridge, MA, 2008.

[48] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[49] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In *SLE*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013.

[50] T. A. Wagner and S. L. Graham. Efficient and flexible incremental parsing. *ACM Trans. Program. Lang. Syst.*, 20(5):980–1013, Sept. 1998.

[51] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for Java. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 28–49, Berlin, Heidelberg, 2006. Springer-Verlag.

[52] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the Java query language. *SIGPLAN Not.*, 43(10):1–18, Oct. 2008.

[53] L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10:19–56, 0 2000.