Stream Fusion using Reactive Programming, LINQ and Magic Updates

Gereon Schueller
Fraunhofer FKIE
Fraunhoferstr. 20
53343 Wachtberg, Germany
Email: gereon.schueller@fkie.fraunhofer.de

Andreas Behrend
University of Bonn
Römerstr. 164
53117 Bonn, Germany
Email: behrend@cs.uni-bonn.de

Abstract—**The paradigm of reactive programming has attained more and more interest in recent time. In this paper, we show why reactive programming is a well-suited paradigm for sensor fusion algorithms. As a new contribution, we show how the efficiency of stream-oriented reactive programs can be enhanced by applying the magic update method that originally stems from the field of update propagation in deductive databases.**

I. Introduction

In recent papers [1]–[3] we investigated the suitability of relational database systems for the processing of sensor data streams with the goal of enhanced and descriptive programming. While it turned out that relational database management systems can be used well for the task of sensor data fusion, there are many applications where the usage of such a system may be immoderate or improper as no data has to be stored persistently. In our research, we pointed out that the descriptive paradigm of relational database systems is a great advantage in compare to the commonly used imperative paradigm.

In recent times, the techniques of *reactive programming* in combination with programming language integrated queries have emerged. With these new paradigms, it is possible for sub-programs to *subscribe* to occurrence of certain events. The events can be filtered by means of queries that are in general comparable to relational queries in relational database management systems (RDBMs).

The advantage of this approach is, that it is flexible, leads to easily understandable source code and offers new possibilities for optimization. On the one hand, many optimizations that are already made in relational database can be directly transferred to the program integrated query languages. For instance, technical aspects, like parallel task execution, can be integrated by the designers of the programming language, as it is the case in DBMS. On the other hand, relational optimization methods can be done by the programmer to avoid unnecessary processing steps. As reactive programming focuses on changing data after events, one of the most important techniques stems from the area of *update propagation*, where the most general case is the method of magic updates [4]. In this paper, we show how this technique can be used in integrated queries and how this can be used in combination with reactive programming. We also give a short evaluation of a concise example that shows the benefit of this method.

The rest of the paper is organized as follows: In the next section, we give a short summary of related work. In Section III we introduce the paradigm of integrated queries by means of Microsoft's LINQ. In Section IV we present the paradigm of reactive programming. In Section V we show that the relational algebra and LINQ expressions are idempotent. An introduction to incremental update propagation and magic updates is given in Section VI and VII, respectively. The paper closes with an evaluation the proposed methods for an aircraft monitoring scenario.

II. Related Work

In [5], the problem of optimization for reactive computing in combination with sensors is discussed with respect to energy efficiency.

The Language Integrated Query LINQ was introduced by Microsoft for C# and Visual Basic 9 in 2007 [6]. Beside programming-oriented books [7], LINQ did not receive much attention in the scientific literature. One of the few publications that copes with LINQ optimizing is "Avalanche-Safe LINQ Compilation" [8]. In a recent paper [9], the idea of integrating heterogeneous data with LINQ was discussed.

Koenig and Paige were one of the first authors who suggested thirty years ago to avoid costly calculation of derived expressions but to use incremental counterparts instead [10]. Since then, many methods for incremental recomputation of relational expressions have been proposed in the literature. Although most approaches essentially apply the same techniques, they mainly differ in the way they have been implemented and in the granularity of the computed induced updates. With respect to implementation, e.g. authors used the database languages SQL (including triggers) [11], Datalog [12]–[14] and relational algebra [15]–[17] for specifying their delta rules – just to mention a few.

The Magic Set method was first proposed with respect to the optimization of logic programming and database queries [18]. It was later extended to the optimization of database updates [4]. The applicability of the method to data streams was proposed in [1]–[3]. A proposal for higher-level sensor fusion by means of relational, descriptive rules was given in [19].

III. LINQ

LINQ (contrived acronym for *Language Integrated Query*) was introduced by Microsoft for the .NET based languages in order to fill the mismatch between object oriented languages and the handling of structured data. Briefly said, LINQ is a descriptive, functional extension of the mostly imperative programming languages of the .NET family. Meanwhile, similar constructs exist for other programming languages like Java or Python. In the following, we present LINQ by means of its C# implementation.

While the first implementations of LINQ referred to relational databases, it is possible to write LINQ mappers to virtually any kind of structured data. As a concise example, we present the problem of joining two arrays. Let `employees` be an array of employee names together with their grade and `salaries` a list of grades with the actual salary. If we want to have the salary of each employee, an imperative program would look like:

```
foreach (e in employees) {
  foreach (s in salaries) {
    if (e.grade == s.grade) {
      Console.WriteLine("{0} earns {1} $",
        e.name, s.salary);
    }
  }
}
```

With LINQ, we can write:

```
var res = from s in salaries join e in employees
          on s.grade equals e.grade
          select new {e.name, e.grade, s.salary};

foreach (r in res) {
 Console.WriteLine("{0} earns {1} $",r.name,r.salary);
}
```

While the example using LINQ is not necessarily shorter than the imperative way, it is often easier to read. Another advantage is, that the C# compiler will automatically decide how to evaluate the LINQ expression. For example, it may decide on multi-core machines to parallelize the expression. We can also use aggregate functions, as known from relational databases. For instance, if we want the salary sum over all employees with the same grade, we can write

```
var sum = from r in res
          group by grade into g
          select new {Grade=g.Key, Sum=g.Sum()};
```

What makes LINQ interesting in the field of data fusion is the fact that virtually any data source can be connected to LINQ by implementing appropriate providers. Besides database source, network streams and arrays, numerous providers to web services, like *Google, Twitter* or *Wikipedia* exist. Also text or XML files can be directly queried. The integration of sensor data streams is straight ahead.

In [19], we presented a way of expressing criteria for anomaly detection using relational algebra. For instance, off road vehicles could be detected using the two relations *cars*, containing the position of all vehicles measured by a sensor and *roads*, containing geographical information about the road net:

$$\begin{aligned}&\text{rareSituation} = \sigma_{\text{weight}>3500\wedge(t<6{:}00\vee t>22{:}00)}(\text{Cars})\\ &\bowtie_{\text{onRoad(Cars.x, Cars.y, r.x1, r.y1, r.x2, r.y2)}}(\sigma_{\text{r.quality='Dirt'}}(\text{Roads})))\end{aligned} \tag{1}$$

where the symbol $\bowtie$ denotes the join. Using LINQ, we can transform this to the following expression:

```
var rare = from c in Cars join r in Roads
           on c.t<6 OR c.t>22
           where r.quality = "Dirt" and
           c.onRoad(r);
```

with an appropriate implementation of the function `onRoad`. In the following section, we will see how this static example can be applied to a stream model using the *Reactive Programming Extensions* Rx of .NET

IV. Reactive Programming

The Reactive Programming Framework Extension (Rx) is a paradigm for implementing programs that can *react* to some events, called *Observables*. In many stream scenarios (e.g. for sensor data processing), programs run in an infinite loop processing data streams:

```
while(true) {
  r = data.read();
  process(r);
}
```

This classical approach has the disadvantage, that the data read can only be processed by one consumer; in this case the routine `process(r)`. All other functions that would like to process the data have to be added to the loop or to the function. In addition, it has to be checked whether the data is interesting for being processed or not.

The Rx framework solves this problem: The data read can be "thrown" into an event queue. Instead of "pulling" the data from the queue, the queue itself invokes its processing. To this end, the processing function can *subscribe* observables. Let `r` be an Observable, then we can transform the code above to Rx like this:

```
r.subscribe(value => process);
```

The expression can be understood in the following way: After an instance of `r` occurs, the lambda-function `value => process` is invoked, where `value` automatically is initialized with the instance that triggered the lambda-function. The lambda function can also consist of a direct implementation:

```
r.subscribe(value =>
  {Console.WriteLine(value.id);
   long a = v.length/2048;
   Console.WriteLine(a);});
```

There exist many ways to generate Observables. One way is to call the generator

```
Observable.Generator(initialState, condition,
                     iterate, resultSelector)
```

For instance, we can "loop" through an array and return an Observable for each element of the array:

```
IObservable<string> o = Observable.Generate(
    data.GetEnumerator(),
    e => e.MoveNext(),
    e => e,
    e => e.Current);
```

Another straight forward example is listening to a TCP/IP port: After receiving a TCP packet, the data from that packet is thrown as an Observable. An overview of Rx can be found in [20].

Another interesting fact is that Observables are "first-class-objects" in .NET, i.e. they can be treated as any other object (passed to functions, redefined, extended etc.) and that they also implement the `IEnumerable` interface, which means that they can be queried with LINQ. For instance, if we want only to react on observable that fulfill a certain condition, we can select them out:

```
var r2 = from   myR in r
         where  myR.size > 1024
         select myR;
r2.subscribe(value => process);
```

which means that only observable from `r` whose attribute `size` is larger than 1024 are processed. As Observables can also be joined to Enumerables, this offers a wide spectrum in sensor data fusion applications. We will take this up in Section VIII.

V. Relational Algebra vs. LINQ

In order to apply techniques from the world of relational algebra (RA) to the world of LINQ, we have to show that LINQ is *relational complete*, i.e. that we can find an equivalent LINQ expression for every RA expression. We do this by giving explicit transformation rules between RA and LINQ.

The RA has been proposed by Codd [21] in 1970 for handling relational data. To this end, all data stored inside a relational database are assumed to be stored in *relations*. A relation R is a subset of the cross-product of some *domains* $D_1, \dots, D_n$:

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n \tag{2}$$

so each element (tuple) of R has the form $(d_1, d_2, \dots d_n)$ with $d_i \in D_i$ $(1 \leq i \leq n)$. The d_i are called *attributes* of the tuple. For simplicity, each attribute is given a name, with the notation *t.attributeName*. We can operate on relations using the set functions union ($\cup$), intersection ($\cap$) and set difference ($\setminus$), with the constraint that both operands have the same domain structure. Additionally, the cross product $\times$ on relations is defined as:

$$\begin{aligned} A \times B := \{\{(a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m)\} \\ |\{(a_1, a_2, \dots, a_n,) \in A \wedge (b_1, b_2, \dots, b_m) \in B\}\} \end{aligned} \tag{3}$$

Additionally, we define the operation selection σ_c:

$$\sigma_{c(r)}(R) := \{r \in R : c(r) \text{ is true}\} \tag{4}$$

where c is a boolean expression. The projection $\pi_{a_{i_1}, a_{i_2}, \dots a_{i_m}}(R)$ removes attributes from a relation:

$$\pi_{a_{i_1}, a_{i_2}, \dots a_{i_m}}(R) := \{(a_{i_1}, a_{i_2}, \dots a_{i_m}) | a \in R\} \tag{5}$$

The rename operator $\rho_{b \leftarrow a}$ renames attribute a to b. The often used join operator $\bowtie$ is a combination of cross-product and selection:

$$A \bowtie_{c(a,b)} B := \sigma_{c(a,b)}(A \times B) \tag{6}$$

We can extend the RA by "defining equations" of the form $C \leftarrow A \otimes B$. As C does not correspond directly to a stored relation in the database, but is generated, C is called a *derived* relation, while A and B are stored relations.

LINQ supports each operation of the relational algebra as well as defining equations. The transformation rules for all basic RA operators are shown in Table I.

Please note that there are some constraints on the `join` operation in LINQ. However, this does not pose a problem as the join can be simulated by the cross-product and selection. It has also to be mentioned that LINQ supports functions that are similar to the aggregate functions of SQL, as well as the Boolean check `.any` and the grouping via `.groupBy`, and thus is more powerful than RA. In the following, we will not consider these operators in the context of incremental update propagation. A solution for incremental updates on these functions has been given e.g. in [22]

VI. Incremental Update Propagation

Query optimization is a well-established database research topic which is divided into logical and physical optimization. Physical optimization copes strategies for storing data efficiently in structures, e.g. trees or hash tables. Physical optimization techniques used in the area of relational databases cannot be transferred directly to LINQ due to the generic nature of the queried Enumerables.

In contrast, logical optimization optimizes queries or update statements on databases by rewriting relational expressions into another, equivalent form. As we have seen in the previous chapter, LINQ is relational complete. Thus, we can transfer logical optimization techniques from the world of relational databases to LINQ. In this section, we present a technique for decomposing update statements into equivalent counterparts that are (in general) more efficient.

The problem of update propagation occurs when the data of derived views are stored in memory. When the underlying data of the derived relation changes, the question arises how the derived data has to be aligned when the source data change. In LINQ, this corresponds to the problem that the results of a LINQ expression have been assigned to a variable and the right-hand variables change. A complete recomputation may often be costly and unnecessary. Taking a real world example, no one would print a new phone book if only one phone number changes. To this end, the method of *incremental update propagation* has emerged.

The idea behind incremental update propagation is to identify all data in derived relations that must be changed upon changes in the underlying base relations. The relations that contain the essential changes are called *delta relations*, referring to the symbol Δ in mathematics.

In the following, we will recall well-known transformations for deriving specialized delta rules from a given set of LINQ rules. In order to achieve completeness, so-called transition rules are considered afterward which allow to simulate the old and new state of a relation.

RA	LINQ
$C = A \cup B$	`var C = A.union(B);`
$C = A \cap B$	`var C = A.intersect(B);`
$C = A \backslash B$	`var C = A.except(B);`
$C = A \times B$	`var C = from a in A from b in B select new {a, b};`
$C = \sigma_c(A)$	`var C = from a in A where (c) select a;`
$C = \pi_{x,y,z}(A)$	`var C = from a in A select new {a.x, a.y, a.z};`
$C = \rho_{x \leftarrow z}(A)$	`var C = from a in A select new {x=a.z};`
$C = A \bowtie B$	`var C = from a in A join b in B select new {a, b};`

TABLE I: The operators of relational algebra (RA) and their counterparts in LINQ

Various authors introduced delta views, but not always in an identical manner. In this paper, we will use a style similar to that of [14], [23] where delta views are directly defined. Other authors define laws for iteratively transforming a complex expression into its delta version (as [24] does). We assume that all elements are unique, so no bags need to be considered, which facilitates the incremental expressions. Approaches for incremental updates with duplicates have been considered e.g. in [11], [17]. (Uniqueness for LINQ expressions can be achieved by using the `.distinct()` method or by adding unique key attributes).

In the following, for each relation name `P`, two delta relations `P_ins`, `P_del` are used for representing the insertions and deletions induced on `P` by an update of the underlying Enumerables. The delta relations defined for a relation `P` have the same schema and type as `P`. The state of a relation `P` before the changes is denoted by `P_old`, whereas the new state is represented by `P_new`. The delta sets can be defined as follows:

```
var P_ins = P_new.except(P_old);
var P_del = P_old.except(P_new);
```

The most common expressions in LINQ are of the simple form

```
var p = from q in Q
        join r in R on a
        where c
        select {q,r};
```

We can distinguish three different subsets of inserted data upon insertions into the underling Enumerable variables:

1) An insertion into `Q` that finds a matching partner in the old state of `R`
2) An insertion into `R` that finds a matching partner in the old state of `Q`
3) An insertion into `Q` that finds a matching partner in the new inserted data of `R`

Thus, the join expression has the incremental version

```
var p_ins = (
from q_ins in Q_ins join r_0 in R_0 on a where c)
.union(
from r_ins in R_ins join q_0 in Q_0 on a where c)
.union(
from r_ins in R_ins join q_ins in Q_ins on a where c);
```

A similar consideration holds for deletions:

```
var p_del = (
from q_del in Q_del join r_0   in R_0   on a where c)
.union(
from r_del in R_del join q_0   in Q_0   on a where c)
.union(
from r_del in R_del join q_del in Q_del on a where c);
```

Usually a residue expression has to be included in both rules in order to eliminate induced insertions for which a different derivation already existed in the old state, or to eliminate induced deletions for which another derivation will exist in the new state, respectively [14]. This kind of "effectiveness test" is normally also required for the union operator because of duplicate derivations generated. Since we assume that the elements are unique, this effectiveness test is not necessary and we can use the more simpler incremental expression instead.

The most complicated version is that for the set difference, as the second argument is a "negative" one and can lead to deletions upon insertion, so the transformation of

```
var p = Q.except(R);
```

is defined as

```
var p_ins = (from q_ins in Q_ins
             select q_ins)
            .union(R_ins.intersect(Q_0))
            .union(R_del.intersect(Q_ins)
            .except(R_0)
            .except(R_ins));
var p_del = Q_del.union(R_ins.intersect(Q_0));
```

A. Transition Rules

Generally, for computing safe updates, it is necessary to refer to the old, the new or the preserved state of the LINQ sets. In [25] it is assumed that all derived relations are materialized which simplifies the state evaluation process but seems to be unrealistic in practice. Therefore, the possibility has been investigated of dropping the explicit references to one of the states by deriving it from the other one and the given updates. The benefit of such a state simulation is that it is not required to store intermediate results explicitly but one may work on one state only. Rules for state simulation will be called *transition rules* according to the naming in [26].

We have defined the delta rules in a way that only references to preserved relation states occur. Therefore, we will concentrate on their simulation given the changes of the underlying data sources although new and old relation states can be simulated in a similar way. In the following, we assume that the preserved state of a source relation is provided by the data source (together with the respective delta relations). Consequently, the relation states `Q_0` and `R_0` in the expression above are and can be seen as base tables. The remaining preserved states of the derived views, e.g. `P_0`, however, have to be simulated. A very straightforward approach is to use so-called *naive transition rules* which employ the preserved relation state of the data source for simulating derived ones:

```
var P_0 = from q_0 in Q_0
          join r_0 in R_0
          select {q_0, r_0};
```

The disadvantage of these transition rules, however, is that each derivation with respect to a derived state of a preserved relation has to go back to the preserved base relations and hence makes no use of the implicit updates already derived during the course of propagation. What is more, the expensive projections – implementing data transformation and cleansing operations – are again applied to very large relations and no optimization effect is achieved for them.

In the Internal Events Method [26] as well as in [14] it has been proposed to improve state simulation by employing not only the extensional delta relations but the derived ones as well, leading to so-called *incremental transition rules*. However, this does not help to avoid the costly projections. Therefore, we refrain from using incremental transition rules rather the naive one such that the optimization effects of Magic Sets becomes more apparent.

Stratification Briefly said, a set of LINQ expression is *stratifiable* if it does not contain negative cycles (cf. [27]). An example of a negative cycles would be:

```
var a = x.except(b);
var b = a;
```

Naive Transition Rules $\tau(\mathcal{R})$ Let $\mathcal{R}$ be a set of stratifiable view definitions and $\mathcal{R}^{nf} := \texttt{norm}_{\mathcal{R}}(\mathcal{R})$ the corresponding set in normal form. The set of naive transition rules for safe updates and preserved state simulation with respect to $\mathcal{R}^{\text{nf}}$ is denoted $\tau(\mathcal{R})$ and is defined as follows:

1) For each auxiliary relation H introduced by the mapping $\texttt{norm}_{\mathcal{R}}(\mathcal{R})$ and each rule rule of the form `var r = `E with H=`rel`(R) and E representing an arbitrary LINQ expression a transition rule of the form
```
var H_0 =
...E[Pi|Pi_0]
```
is in $\tau(\mathcal{R})$ where `E[Pi|Pi_0]` denotes E with every relation name P_i in E substituted by P_i^o.
2) No other rules are in $\tau(\mathcal{R})$.

VII. Magic Updates

Logical optimization rules include a selection pushing strategy which allows for shifting selection condition to lower-level operands within a given operator tree such that irrelevant answer tuples are eliminated as early as possible. As soon as recursive expressions are allowed, however, the classical selection pushing strategy cannot deal with new selection constants introduced during the course of evaluation. For solving this problem, the rewriting technique Magic Sets has been proposed in [18] which uses auxiliary relations (the "Magic Sets") in order to store and to dynamically apply those generated constants.

The introduction of auxiliary Magic Sets containing dynamically generated selection constants can also be used for improving the focus within non-recursive expressions as shown in [28]–[30]. As an example, consider the following algebra expression for defining the view $P(x)$ based on the relations $Q(x)$, $R(x)$, $S(x,y)$ and $T(x,z)$:

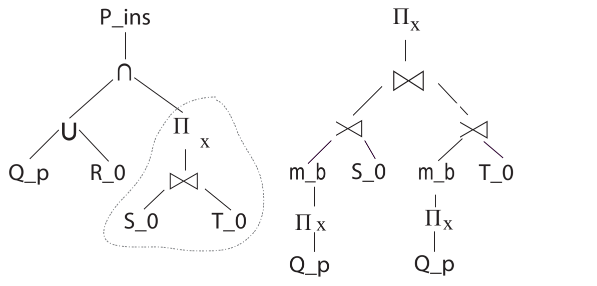

Fig. 1: An evaluation tree for the incremental evaluation and its magic-optimized sub-tree. In the left tree, no optimization effect for $S \bowtie T$ is achieved, while the magic relation `m_b` pre-selects `S_0` and `T_0`, leading to a smaller sub-expression.

```
var P = (Q.union(R))
        .except(from s in S
                join t in T
                on s.ID equals t.ID
                select s.x);
```

Applying our transformation would yield the following rule for computing induced insertions `P_ins` of `P` resulting from insertions `Q_ins` into `Q`:

```
var P_ins = (Q_ins.union(R_0))
            .except(from s_0 in S_0
                    join t_0 in T_0
                    on s_0.ID equals t_0.ID
                    select s_0.x);
```

Despite of the focus on changes with respect to Q, no optimization effect is achieved with respect to the evaluation of the right-hand argument of the set difference operator. Magic Sets, however, allows to use the small number of tuples in `Q_ins` already for determining all matching join partners by introducing the Magic Set `m_aux_b` applied in two semi-joins:

```
var m_aux_b = Q_ins;

var m_s_0 = from m in M_AUX_B_P
            join s_0 in S_0
            select s_0;
var m_t_0 = from m in M_AUX_B
            join t_0 in T_0
            select t_0;

var P_ins = (Q_P.except(R))
            .except(from m_s_0 join m_t_0
                    on ...
                    select s.x);
```

Under the assumption that S^o and T^o are quite large in comparison to the size of Q^+ and there is a low selectivity of the tuples in Q^+, the argument sizes of the join and difference operator are considerably reduced. Thus, the resulting incremental algebra expression provides a much better focus on the changes to Q.

Note that Magic Sets does not always lead to an improved evaluation. Generally, its optimization effects strongly depend on relation sizes, selectivities and the chosen sideways information passing strategy. Under the assumption that delta relations are considerably smaller than state relations, however, the focus on selection constants in delta sets by using MS provides notable optimization effects. This is especially the case if the increased focus allows to reduce the input size for expensive operations that perform data cleansing or transformation.

A. Magic Updates Rewriting

We will now present the general approach for Magic Update Rewriting. As the Magic Update relies on the detection of bound and free variables in the query, define

Bound/Free attributes An attribute a_i of a relation $R(a_1, a_2, \ldots, a_i, \ldots, a_n)$ is *bound* if it is used in a select or join condition. All other attributes are *unbound*.

Adornment An adornment string $\mathtt{ad}(C)$ for an n-ary relation $C \leftarrow A$ is defined as

$$\mathtt{ad}(C) = \gamma_1 \circ \gamma_2 \circ \gamma_3 \circ \cdots \circ \gamma_n$$

where $\gamma_i := \mathtt{'b'}$ if the attribute on position i is bound and $\gamma_i := \mathtt{'f'}$ if it is unbound (free).

We can now define the Magic Rewriting transformation for queries:

Magic Predicates Let $A \equiv P_{ad}(\vec{x})$ be a positive literal with relation name P, adornment $\mathtt{ad}$ and $\mathtt{bd}(\vec{x})$ the sequence of attribute names within $\vec{x}$ indicated as bound in the adornment $\mathtt{ad}$. Then the magic predicate of A is defined as

$$\mathtt{magic}(A) := \mathtt{m\_ins\_ad}(\mathtt{bd}(\vec{x})).$$

Given a set of view definitions $\mathcal{R}$ and an adorned query $Q \equiv P_{ad}(\vec{x})$ with $P \in \mathtt{rel}(\mathcal{R})$, the Magic Sets transformed rules with respect to Q are denoted by $\mathtt{ms}_Q(\mathcal{R})$.

The Magic Update Set is obtained by applying the magic rewriting to the set of delta views for a database (or, in our case, for a LINQ program):

Magic Updates Rewriting Let $\mathcal{R}$ be a set of stratifiable view definitions, $\mathcal{R}^{\text{nf}} := \mathtt{norm}_{\mathcal{R}}(\mathcal{R})$ the corresponding set in normal form, and $\varphi(R)$ the set of delta rules with respect to $\mathcal{R}^{\text{nf}}$. The MU rewriting of $\mathcal{R}^{\text{nf}}$ yields the magic rule set $\mathcal{R}^{\text{magic}} := \varphi(\mathcal{R}) \cup \mathcal{R}^{\text{query}} \cup \mathcal{R}^{\text{state}}$ where $\mathcal{R}^{\text{query}}$ and $\mathcal{R}^{\text{state}}$ are defined as follows:

For each delta view of the form

```
var P = R_{p,m}.{union|except(S_0)};
```

or of the form

```
var P = from R_{p,m}
        join S_0
        select ....;
```

we define the following subquery rule:

```
var magic_S_0_ad = from R_{p,m}
                   select bd(x);
```

where `bd(x)` denotes the sequence of attributes within the schema of `R` indicated as bound by the adornment of relation $\mathtt{magic}(S^o_{ad})$.

From the set $\tau(\mathcal{R})$ we derive the rule set $\mathcal{R}^{\text{state}}$: For each relation symbol $\mathtt{magic}(L^o_{ad}) \in \mathtt{rel}(\mathcal{R}^{\text{query}})$ the corresponding MS transformed rule set $\mathtt{ms}_Q(\tau(\mathcal{R}))$ is in $\mathcal{R}^{\text{state}}$ where $Q \equiv L^o_{ad}$ represents an adorned query with $\mathtt{rel}(L^o) \in \mathtt{rel}(\tau(\mathcal{R}))$. No other rules are in $\mathcal{R}^{\text{state}}$.

The MU rewriting of the original rule set $\mathcal{R}$ is denoted $\mu(\mathcal{R})$ and is defined as $\mathtt{norm}^{-1}_{\mathcal{R}}(\mathcal{R}^{\text{magic}})$.

VIII. Example: Flight Delay Calculation

As an example for our discussed transformation, we present a problem in which we combine a stream of data with static master data and slowly changing context data. In our example, we calculate the delays for a stream of flight track data. Let
`trackdata` An Observable of flight track data that changes rapidly (e.q. every second)
`flightplans` An Enumerable of flight plans that changes slowly (e.q. each minute)
`airports` An Enumerable that contains the position of airports that does not change while the system is running.
In order to calculate the delay, we first calculate an "ideal flight", i.e. a flight that starts at the scheduled airport as defined in the flight plan and flies on a straight towards its destination. Then, we calculate the remaining times for the ideal flight and the real flight to reach the destination. The difference gives us a lower bound of the delay:

Listing 1: Ideal Flight Calculation

```
var idealFlight =
     from a in airports
     join f in flightplans
     on a.ICAO equals f.DEST_AIRPORT
     select new {f, a, fLon = idealLON(a,f),
                       fLat = idealLAT(a,f)};
```

where `idealLon/Lat` is a simple function that calculates the ideal position in longitude and latitude. The raw data for the delay can then be calculated by joining the static `idealFlight` with the observable `trackdata`:

Listing 2: Delay Calculation

```
var delay = from t in trackdata
            from idf in idealFlight
            where (t.TRACK_ID == idf.TRACK_ID
                   && t.SPEED_HORI_KT > 0)
            select new
            {idf.TRACK_ID, idf.a.
                ScheduledLandingTime,
            idf.a.Lat, idf.a.Lon,
            idf.fLat, idf.fLon, t.SPEED_HORI_KT
                };
```

The delay itself can then be calculated by subscribing the IObservable `delay`:

Listing 3: Delay Subscription

```
delay.Subscribe(value =>
    Console.WriteLine("Delay: {0} for Track No.
        {1}",
    Delay.calculateDelay(value.
        ScheduledLandingTime,
     value.aLat, value.aLon, System.DateTime.Now
        ,
     value.fLat, value.fLon, value.SPEED_HORI_KT
        ),
     value.TRACK_ID));
```

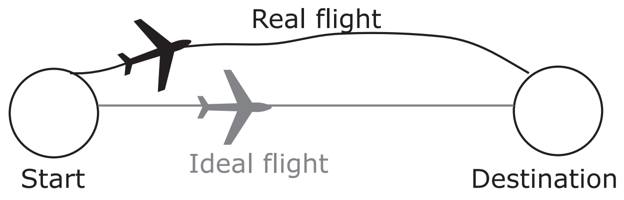

Fig. 2: Calculation of delay using an "ideal flight". The difference between both remaining times to landing gives a lower estimation of the delay.

However, this approach (which we call "the static approach" in the following) does not work correctly if the data in `flightplan` are changed while the system is running. To this end, we have to re-calculate the `idealFlight` for each new IObservable:

Listing 4: Delay Calculation with Naive Incremental Join

```
var delay = from t in trackdata
            from a in airports
            from f in flightplans
            where ((t.TRACK_ID == f.TRACK_ID
            && (t.SPEED_HORI_KT > 0)
            && (a.ICAO == f.DEST_AIRPORT)))
            select new {t,a,f,
                        fLon = idealLON(a,f),
                        fLat = idealLAT(a,f)};
```

This solution (called the "naive incremental join" in the following) has the drawback that it queries all airport and flightplan data without focusing with respect to the new trackdata. So we do a magic set transformation of the naive join with the `trackdata` as insert relation:

Listing 5: Delay Calculation with Magic Update

```
var delay =
from t in trackdata
from idf in
    (from a in airports
     join f in flightplans
     on a.ICAO equals f.DEST_AIRPORT
     where t.TRACK_ID == f.TRACK_ID
     select new
     {f.TRACK_ID, f.ScheduledLandingTime,
         aLon = a.LON, aLat = a.LAT,
         fLon = f.LON, fLat = f.LAT,
         f.SPEED_HORI_KT}) //Magic subquery
where (t.TRACK_ID == idf.TRACK_ID
                  && t.SPEED_HORI_KT > 0)
select new {idf.TRACK_ID, idf.
    ScheduledLandingTime,
    idf.aLat, idf.aLon, idf.fLat,idf.fLon,
    t.SPEED_HORI_KT};
```

Please note that we incorporated the magic subquery directly into the main expression. This way, the size of the flight plan relation is already shrunk before the join takes places.

	size of trackdata		
Method	1 500	50 000	1 500 000
static	4	109	2 732
naive	213	2 739	197 561
magic	3	92	2503
opt. array	1.5	48	1 444

TABLE II: Run time (results (in seconds) for three different approaches and varying size of track data

	size of flight plan		
Method	28	280	2 800
static	11	20	109
naive	78	711	2739
magic	10	20	92

TABLE III: Run time (results (in seconds) for three different approach and varying size of context data (flight plan). The track data size was fixed to 50,000

IX. Evaluation

For evaluation our approach, we implemented a test framework that read all data from textfiles.[1] The program reads data from the file into an array and generates an observable for each entry:

Listing 6: Observable from Array

```
IObservable<track> o = Observable.Generate(
    readTracks.GetEnumerator(),
    e => e.MoveNext(),
    e => e,
    e => e.Current);
```

The data is then subscribed as described in the previous section. The runtimes for the complete evaluation of the test set where measured using the `stopWatch` function of C#. We varied the size of track data while we left the size of the other tables unchanged. The results can be found in Table II. It can clearly be seen that the static approach performs fastest, while the naive approach performs rather poorly. From the data, it can be seen that the naive approach has a super-linear run-time. The magic set approach, in contrast, performs with a linear runtime behavior and allows for handling changing context data. In Table III, the run times for a changing size of the flight plan data can be found. Also here, it can be seen that the magic approach is always faster than the dynamic approach. The static approach is only faster if the size of the context data is small, but still has the disadvantage of possibly incorrect results.

X. Conclusion

In this paper, we presented a way of fusing data using the Reactive Extension Framework and LINQ. We have shown how these techniques can lead to easy understandable, flexible descriptive programming. We have also shown the drawback of this solutions when joining streams with large context data. To solve this problem, we proposed the usage of the Magic

[1]The source code of the test implementation is available at http://code.google.com/p/magic-linq/

Update method that is already established in the context of deductive databases. We have shown how magic updates can be applied to LINQ queries. We evaluated the approach for a real life example and have shown the improvement in runtime behavior.

In a future work, we will show the effects of an automatic magic update compiler for LINQ expressions. It is also of interest to explore the possibilities of indexes in LINQ with respect to rule transformations.

References

[1] A. Behrend, G. Schüller, and M. Wieneke, "Efficient tracking of moving objects using a relational database," *Information Systems* (accepted), 2012. Available Online: http://dx.doi.org/10.1016/j.is.2012.01.001

[2] G. Schüller, A. Behrend, and R. Manthey, "AIMS: an SQL-based system for airspace monitoring," in *ACM SIGSPATIAL IWGS 2010*. New York, USA: ACM, pp. 31–38.

[3] A. Behrend, R. Manthey, G. Schüller, and M. Wieneke, "Detecting moving objects in noisy radar data." in *ADBIS 2009*, Springer LNCS vol. 5739, pp. 286–300.

[4] A. Behrend and R. Manthey, "Update propagation in deductive databases using soft stratification," in *ADBIS 2004*, Springer LNCS vol. 3255, pp. 22–36.

[5] C. Chen, Y. Xu, K. Li, and S. Helal, "Reactive programming optimizations in pervasive computing," in *SAINT 2010*, pp. 96 –104.

[6] E. Meijer, "The world according to LINQ," *Commun. ACM*, vol. 54, no. 10, pp. 45–51, Oct. 2011.

[7] S. Klein, *Professional LINQ (Programmer to Programmer)*. Wrox, 2008.

[8] T. Grust, J. Rittinger, and T. Schreiber, "Avalanche-safe LINQ compilation," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 162–172, Sep. 2010.

[9] Y. Wang and X. Zhang, "The research of multi-source heterogeneous data integration based on LINQ," in *ICCSEE 2012*, vol. 1, pp. 147–150.

[10] S. Koenig and R. Paige, "A transformational framework for the automatic control of derived data," in *VLDB 1981*. IEEE Computer Society, pp. 306–318.

[11] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," in *VLDB 1991*, pp. 577–589.

[12] U. Griefahn, T. Lemke, and R. Manthey, "Tools for chimera: An environment for designing and prototyping advanced applications in an active DOOD model," in *ADBIS 1997*, Nevsky Dialect, pp. 346–355.

[13] V. Küchenhoff, "On the efficient computation of the difference between concecutive database states" in *DOOD 1991*, pp. 478–502.

[14] R. Manthey, "Beyond data dictionaries: Towards a reflective architecture of intelligent database systems," in *DOOD 1993*, pp. 328–339.

[15] R. Agrawal, "Alpha: An extension of relational algebra to express a class of recursive queries," *IEEE Trans. Software Eng.*, vol. 14, no. 7, pp. 879–885, 1988.

[16] G. Dong and J. Su, "Incremental maintenance of recursive views using relational calculus/SQL," *SIGMOD Rec.*, vol. 29, no. 1, pp. 44–51, 2000.

[17] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," in *ACM SIGMOD 1995*, pp. 328–339.

[18] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs (extended abstract)," in *PODS 1986*. New York, NY, USA: ACM, pp. 1–15.

[19] G. Schüller, W. Koch, and J. Biermann, "Pattern recognition using queries in relational tracking data bases," in *SWIFT 2009*, Skövde, pp. 17–21.

[20] L. Campbell, *Introduction to Rx*. Online: www.introtorx.com, 2012.

[21] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[22] A. Behrend, C. Dorau, R. Manthey, and G. Schüller, "Incremental view-based analysis of stock market data streams," in *IDEAS 2008*, pp. 269–275.

[23] M. Sköld and T. Risch, "Using partial differencing for efficient monitoring of deferred complex rule conditions," in *ICDE 1996*, pp. 392–401.

[24] X. Qian and G. Wiederhold, "Incremental computations of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 3, pp. 337–341, 1991.

[25] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *ACM SIGMOD 1993*. New York, NY, USA: ACM Press, pp. 157–166.

[26] A. Olivé, "Integrity constraints checking in deductive databases," in *VLDB 1991*, pp. 513–523.

[27] J. Grant and J. Minker, "Deductive database theories," *The Knowledge Engineering Review*, vol. 4, pp. 267–304, 11 1989.

[28] ——, "The impact of logic programming on databases," *Communications of the ACM*, vol. 35, no. 3, pp. 66–81, 1992.

[29] I. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic is relevant," in *SIGMOD 1990*. New York, NY, USA: ACM, pp. 247–258.

[30] I. S. Mumick and H. Pirahesh, "Implementation of magic-sets in a relational database system," *SIGMOD Rec.*, vol. 23, no. 2, pp. 103–114, 1994.