Rewrite Techniques for Performance Optimization of Schema Matching Processes

Eric Peukert
SAP Research
01187 Dresden, Germany
eric.peukert@sap.com

Henrike Berthold
SAP Research
01187 Dresden, Germany
henrike.berthold@sap.com

Erhard Rahm
University of Leipzig
Leipzig, Germany
rahm@informatik.unileipzig.de

ABSTRACT

A recurring manual task in data integration, ontology alignment or model management is finding mappings between complex meta data structures. In order to reduce the manual effort, many matching algorithms for semi-automatically computing mappings were introduced.

Unfortunately, current matching systems severely lack performance when matching large schemas. Recently, some systems tried to tackle the performance problem within individual matching approaches. However, none of them developed solutions on the level of matching processes.

In this paper we introduce a novel *rewrite-based optimization technique* that is generally applicable to different types of matching processes. We introduce filter-based rewrite rules similar to predicate push-down in query optimization. In addition we introduce a modeling tool and recommendation system for rewriting matching processes.

Our evaluation on matching large web service message types shows significant performance improvements without losing the quality of automatically computed results.

Categories and Subject Descriptors

D.2.12 [**Interoperability**]: Data mapping

General Terms

Algorithms, Experimentation, Performance

Keywords

Schema Matching, Schema Mapping, Matching Processes

1. INTRODUCTION

Finding mappings between complex meta data structures as required in data integration, ontology evolution or model management is a time-consuming and error-prone task. We call this task schema matching, but it can also be labeled ontology alignment [26] or model matching [15]. Schema matching is non-trivial for several reasons. Schemas are often very large and complex. They contain cryptic element names, their schema documentation is mostly scarce or missing and the original authors are unknown. Some estimate, that up to 40% of the work in enterprise IT departments is spent on data mapping [3].

For that reason many algorithms, so called *matchers* were developed that try to automate the schema matching task partly. They compute correspondences between elements of schemas based on syntactical, linguistic and structural schema- and instance information and provide the user with the most likely mapping candidates [31, 29]. Many systems combine the results of a number of these matchers to achieve better mapping results [7, 4]. The idea is to combine complementary strengths of different matchers for different sorts of schemas. This balances problems and weaknesses of individual matchers, so that better mapping results can be achieved.

In a recent product release, SAP introduced a new business process modeling tool integrating automatic schema matching for the task of mapping large service interfaces. Computed correspondences are used as a recommendation and starting point to a manual mapping of service interfaces. Therefore suggestions need to have a good quality in order to avoid extra work for correcting wrongly identified correspondences. At the same time, the computation of mapping suggestions must be fast so that the user is not interrupted in the modeling process. After having spent too much time on waiting, some users will not apply auto matching recommendation again. Unfortunately, current state of the art matching systems severely lack performance when matching large schemas. For that reason, only a small set of matchers is currently used which restricts the achievable result quality.

The reasons for these performance problems are obvious. Schema matching is a combinatorial problem with at least quadratic complexity w.r.t. schema sizes. Even naive algorithms can by highly inefficient on large-sized schemas. Matching two schemas of average size N using k match algorithms results in a runtime complexity of $O(kN^2)$. Thus schema matching complexity can easily explode if multiple matchers are applied on bigger sized schemas. Even the most effective schema matching tools in the recent OAEI Ontology Alignment Contest suffered from performance issues [5].

As we will discuss in the section on related work only few systems have addressed the performance problem for schema matching. Unfortunately, most of the proposed techniques are built for individual matchers or are hard wired within specific matching processes.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

We therefore propose a generic rewrite-based approach for significantly improving the performance of matching processes. We are aiming at optimizing a matching process in a similar way as for query optimization in databases. We model matching processes as graphs and apply rewrite techniques to improve the performance and to retain or improve the result quality of matching processes. The process rewrite approach is orthogonal to existing techniques for improving the performance of matching systems. Since our approach is similar to query optimization in databases we expect a number of further opportunities to improve mapping performance in future.

Our contributions are:
Flexible matching process: We model schema matching processes as graphs where nodes are operators and edges represent data flow. We introduce a set of operators and their functionality. Our process graph model contains a filter operator that is able to prune comparisons between elements early in the matching process. This pruning is supported by a new data structure, a so called *comparison matrix*. This matrix will be used by matchers to decide what element comparisons are necessary. We develop two types of filter operators: a static threshold-based filtering and a dynamic filtering. The dynamic filtering allows to prune comparisons without losing on precision and recall.
Rewrite-based optimization: Analogous to cost-based rewrites in database query optimization, we treat the performance improvement problem as a rewrite problem on matching processes. This allows expressing performance and quality improvement techniques by special rewrite rules. In particular, we propose novel filter-based rewrite rules utilizing the proposed filter operator. A simple cost model is used to decide which parts of a matching process to rewrite.
A tool for modeling matching processes and a recommender system for rewrites: We developed a tool that can be used for modeling schema matching processes. The modeled processes can later be executed by users of schema matching systems. Additionally a recommender system was implemented that applies our found rewrite rules onto the modeled processes.
Real world evaluation We evaluated the performance improvement of our approach on real SAP service interfaces. Our result shows that a significant performance improvement can be achieved without losing the quality of the applied matching system.

The remainder of this paper is organized as follows: Section 2 gives an overview to existing approaches to improve schema matching. Section 3 introduces our matching process graph model. After that, Section 4 describes the graph based rewrite technique, the new filter operator, and the filter-based rewrite rule. In Section 5 our tool for modeling matching processes and our recommendation system is presented. Finally we evaluate our approach in Section 6 and finish with a conclusion in Section 7.

2. RELATED WORK

A wealth of schema matching techniques can be found in literature as surveyed in [29, 31]. Some techniques primarily rely on available schema information whereas others rely on instance data and additional sources like thesauri or dictionaries [7, 23, 27]. The way how the input information is processed highly influences individual performance properties of a matching algorithm. Element level techniques only consider schema elements in isolation such as string-based edit distance, n-gram and soundex code [17]. These techniques are simpler than structure-based approaches and can thus be executed faster.
All currently promoted matching systems use a combination of different matching techniques for improving the quality matching results. The topology of the matching system has a major impact on the performance and the quality of a matching task. Similar to [14] we identify three different types of matching topologies: (1) Parallel combination, (2) Sequential combination and (3) Iterative computation. Figure 1 visualizes the different topologies.

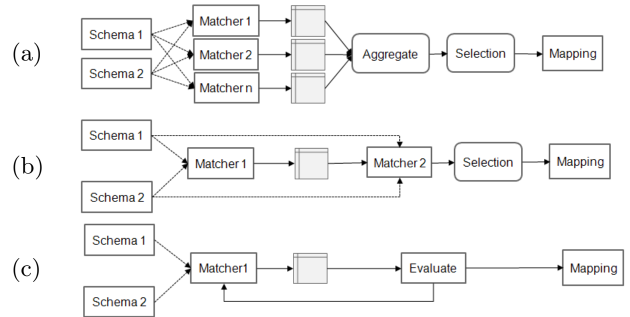

Figure 1: Matching system topoplogies: (a) parallel combination (b) sequential combination (c) iterative computation

Parallel combination implies that all matchers are executed independently, typically on the whole cross product of source- and target schema elements. Results of individual matchers, so called *similarity matrices*, are put into a *similarity cube* [7]. A *similarity aggregation* operation reduces the cube down to a single similarity matrix and a *selection operator* cuts off similarity values by using a threshold. Unfortunately, parallel combination approaches may result in performance problems if all matchers are executed for all comparisons.
Sequential combination systems rely on a sequence of matchers to narrow down a set of mapping candidates as done in CUPID [23] or within the ontology matching system Falcon AO [19]. This sequencing can improve performance since the search space is reduced by the first matchers. However, the performance improvement is achieved at the risk of losing possible mapping results. Unfortunately, building and extending such systems is cumbersome and the order of matchers within sequential systems is fixed. A step towards simplifying the construction of sequential matching strategies was the *refine operator* [6]. It allows to put a mapping result as input to a matcher that can then refine the found correspondences.
Iterative computation systems aim at an iterative improvement of the mapping result but they can severely loose on performance. For instance the ASMOV-System performed quite well on the OAEI2008 contest but took more than 3 hours to compute a result. In comparison to that, the RIMOM-System only took 24 minutes for the same mapping task. Another well-known representative of that group is the SimFlooding approach [24].
Some systems mix all three topologies in order to improve

their result mapping such as CUPID [23] that mixes parallel with sequential execution and ASMOV [34] that mixes all three topologies.

Recently some groups began to introduce so called meta matching systems [31, 21, 13]. These systems allow creating arbitrary matching processes with different topologies and selections of matchers. This can be used to create domain specific matching systems that achieve high result quality and also provide a good performance. However, meta matching systems introduce complex tasks to solve: which matchers to select, in what order to execute them and how to parameterize each matcher.

For the different types of systems several techniques for improving the performance were introduced:

Divide and conquer:

A number of systems apply a divide and conquer strategy when matching large schemas. They first try to manually or automatically identify relevant fragments [8], blocks [20, 18], partitions [1, 28] or clusters [32, 30]. The further matching is then performed on these identified schema parts, which reduces the search space. Unfortunately, this approach could worsen the overall result quality.

Filtering schema parts:

Some systems apply a schema reduction upfront by filtering out the relevant context [8] or by involving the user through a questionnaire [9]. Some systems automatically identify non-needed edges in the schema-graph structure [4] or apply heuristics to reduce the number of comparisons at the cost of quality [12]. Also the famous edit-distance algorithm can be improved by early pruning of comparisons [16]. Similar strategies for reducing the search space were proposed in the record-linkage area. These strategies are called blocking [2] and try to reduce the number of candidate record comparison pairs while still maintaining a reasonable linkage accuracy.

Avoiding repetitions:

A general performance technique is to avoid the repeated execution of the same subtask. For example, a pre-matching step such as tokenizing all labels avoids the repeated tokenization in later match comparisons [30].

Improved data structures:

A number of techniques use special data structures like indexes or hash tables to improve performance. Indexing helps to quickly identify the right elements to compare with. For instance, the B-Match-Approach [11] indexes tokens and its labels. That saves string comparisons based on the assumption that two similar labels share at least one common token. Others remove the nested looping effort since each element in the source needs to be compared to each element in the target by introducing a hash-join like method [4]. They also cache already computed results for later reuse.

Optimization of performance on the process level

Most performance improvement techniques mentioned so far are hard wired into fixed matching processes or act on the level of individual matchers. Some systems already try to tune so called meta matching systems at the level of processes as done in Apfel [13] and eTuner [21]. But they only focus on quality and not on performance.

Today, performance issues are only tackled indirectly. Given some quality and performance requirements, some systems support the automatic selection of appropriate matchers or through questionnaires [25]. For instance the RIMOM system [22] automatically selects or unselects label-based or structure-based matchers depending on the specifics of the input schemas. Other systems support the automatic selection of whole matching processes out of a set of given ones [33].

In this paper we focus on automatically finding the best order of matchers within a given matching process to improve runtime performance. The work that is closest to our work is the MatchPlanner-System [10]. Given a number of matchers the system learns a decision-tree that combines these matchers. As a side-effect, by restricting the number of matchers and the deepness of the tree a user can influence the performance of the matching system at the cost of result quality. Unfortunately, these decision trees need to be learned for every new domain. Also, the effort of executing the matchers of a given tree for every source/target element pair is still high.

In contrast to previous work, we aim at a flexible rule-based approach to optimize the performance of general match processes. With our approach we are able to model arbitrary parallel, sequential and iterative matching systems including the functionality that the refine operator offers. Also, we are able to transform parallel combinations of matchers into faster sequential combinations without losing on precision and recall.

3. MATCHING PROCESS MODEL

In this paper we follow a meta matching system approach. We treat matchers and matching systems as abstract components that can be organized in processes. Similar to the e-tuner system [21] we model matching systems as a complex graph-based matching process:

Definition 1. A *matching process* MP is represented by a matching process graph. The vertices in this directed graph represent operations from an operator library L. Each vertex can be annotated with a set of parameters. Edges within the graph determine the execution order of operations and the data flow (exchange of schemas and mappings between operations). In order to allow modeling performance aspects on the process level we also use so-called comparison matrices as part of the data flow.

Before describing our set of operations we first need to introduce some foundations of our matching process model such as schema, mapping, and comparison matrix:

A *schema* S consists of a set of schema elements. Each schema element s has a name, a data type, one or no parent schema element, and a set of children schema elements. The kind of schema is not restricted and can refer to any meta data structure that can be matched such as trees, ontologies, meta models, as well as database schemas.

A *mapping* M between a source schema S and target schema T is a quadruple (S, T, A, CM). The similarity matrix $A = (a_{ij})$ has $|S| \times |T|$ cells to represent a match result. Each cell a_{ij} contains a similarity value between 0 and 1 representing the similarity between the ith element of the source schema and the jth element of the target schema. The optional *comparison matrix* CM defines which elements of a source schema need to be compared with elements of the target in further match operations. This matrix is defined as $CM = (cm_{ij})$ with $|S| \times |T|$ cells. Each cell cm_{ij} contains a boolean value that defines whether the comparison should

be performed in the following match-operations. The role of the comparison matrix will be described in detail in the following sections.
Operations take the data of the input edges and produce data on an output edge. Each operation has a set of parameters that can be set. The operations are typed. This implies that the sequence of operations within a process is restricted by the input and output data of the individual operations, i.e. some operations in the graph need mappings as input whereas others need schemas.
We include the following operations in our operation library. Most of them were commonly proposed in literature [21, 3, 6]:

- The *SchemaInput* S_{in} and *MappingOutput* M_{out} are specific operations that represent the interface to a matching process. The schema-input operation takes a meta data structure as input and creates a *schema* as output, whereas the mapping-output takes a *mapping* as input and returns arbitrary mapping formats as output.
- The *SchemaFilter* operation SF filters incoming schemas to a given context, e.g. all non leaf elements could be removed from a schema. The output of the SchemaFilter is again a *schema* but with a possibly smaller set of schema elements. This operation is similar to context-filters in COMA++ that narrow the context that is required for a matching task.
- The *Match* operation mat either takes a source and a target schema S, T or a mapping M as input. If a mapping is given, the attached source and target schemas and the comparison matrix CM are used for matching. If no comparison matrix is given, it will be initialized by setting all its cells to true. A parameter defines the type of match algorithm to be used. The match operation returns a new mapping A: $A = mat(S, T, CM)$. It computes a similarity between two schema elements i and j if the value of cm_{ij} in CM is *true*.
- The *Selection* operation Sel takes a mapping A as input and produces a mapping B. It applies a condition *cond* on each cell. If the condition evaluates to false, the value of the cell is set to 0; otherwise $b_{ij} = a_{ij}$. Different selection conditions can be used such as: threshold, delta or topN [7].
- The *Aggregate* operation Agg takes n mappings $A_1, \ldots, A_n$ that refer to the same source and target schemas and aggregates them to a single mapping B using the aggregation function f. The entries of B are computed by $b_{ij} = f(a_{1ij}, \ldots, a_{nij})$. The behavior of the aggregation depends on the aggregation function f. We subsume common aggregation functions such as weighted sum, average or max under aggregate union Agg_{union} operations. However, f could also perform an intersection $Agg_{intsect}$ of given mappings: An entry in B contains a value greater than 0 only for those cells that have a value greater than 0 in all input mappings. The similarity value of each cell in B is calculated by applying f: $b_{ij} = f(a_{1ij}, \ldots, a_{kij})$ iff $\forall k : a_{kij} > 0$ otherwise $b_{ij} = 0$.

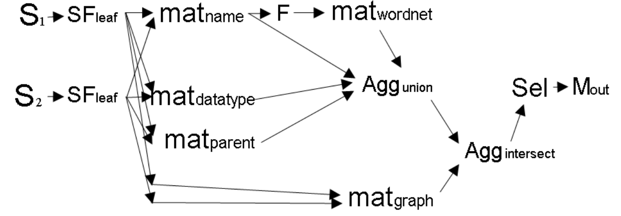

Figure 2: Example complex matching process

- The *Filter* operation F takes as input a mapping A, the comparison matrix CM that is referenced by A, and a filter condition *cond*. It applies the filter condition *cond* to all entries $a_{ij} \epsilon A$. The output of the filter operation is the mapping A together with its new comparison matrix CM'. If $cond(sim_{ij}) = true \wedge cm_{ij} = true$ then $cm'_{ij} = true$; otherwise $cm'_{ij} = false$; Later in this paper we propose two conditions for *cond* that is a static threshold-based condition and a dynamic condition. Throughout this paper, the filter operation is then either referenced by F_{th} for the threshold-based and F_{dyn} for the dynamic case.

The introduced set of operations is not complete (i.e a difference operation $Agg_{difference}$ could also be included), but it covers the most important operations that are needed to model a broad range of matching processes. In particular it contains operations and data structures that allow to improve the performance of matching processes. In Figure 2 an example of a complex matching process is given that can be constructed by applying our model. The visualized process contains parallel and sequential elements. Results of different types of matchers are treated differently. For instance, the non-matching element comparisons from the name matcher are matched with a wordnet matcher. An intersection of results from different match-approaches typically improves the overall precision.

4. GRAPH BASED REWRITE TECHNIQUE

After having defined the preliminaries of our graph model we are now introducing our novel graph based rewrite techniques for matching processes. What we describe can be seen in analogy to database query optimization. However, there are some major differences that will be discussed throughout this section. Particularly, our rewrite rules could lead to changes in the execution result of a matching process while database query optimization leaves the query results unchanged. Our understanding of rewrite-based process optimization can be now be defined as follows:

Definition 2. Given:

- A matching process MP as defined.
- A set of Rewrite Rules RW that transform a matching Process MP into a rewritten matching process MP'
- A utility function U running over a matching process MP. The function can be defined over precision, recall and/or performance of the matching process.

The goal of *rewrite-based matching process optimization* is to create a new matching process MP' by applying rule $rw \epsilon RW$ onto a matching process MP so that $U(MP') > U(MP)$.

In this initial paper we only introduce rewrite rules that improve the performance of a matching process. In the future, rewrite rules will also focus on improving the quality of matching processes. In this paper we describe a filter-based rewrite rule with two types of filter operators. One that improves speed without decreasing the quality, and another one where quality could decrease, but also increase, depending on the use case.

4.1 Filter Operators

In order to allow modeling performance aspects on the process level we use the introduced *Filter* operator and the *comparison matrix*.

Figure 3 gives an example of applying the comparison matrix in a sequential matching process. Two simple schemas are first matched using the name matcher mat_{name}. Element pairs with a similarity lower than 0.2 are pruned out by setting the cell in the comparison matrix to false (visualized as a cross in the bottom matrix). The following namepath matcher [7] $mat_{namepath}$ is only computing similarities for the comparisons that are still in the comparison matrix. In the example, more than half of the comparisons are pruned out. Note that possible matches might be pruned out early

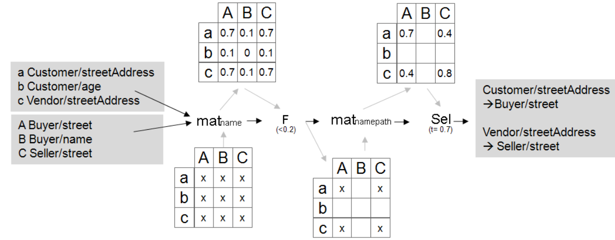

Figure 3: Example of filter process and its comparison matrix

even though they could be part of the overall mapping result. This behavior could drop recall but could also increase precision.

Our goal is to find the best sequential combination of matchers maximizing the number of pruned element pairs and at the same time minimizing the number of wrongly pruned comparisons that are part of the final mapping. The optimal solution would only drop comparisons that lost the chance to survive a later selection.

4.2 Incidence Graph

With the new filter operation we create sequential strategies that potentially perform much faster than their parallel combined equivalent. Figure 4(b) shows a matching process that executes a name matcher and a namepath matcher in sequence. Executing a label-based matcher before executing a structural matcher like a namepath matcher is quite common in sequential matching systems as discussed in Section 2. We took a small library of matchers and investigated the effect on performance and precision/recall (FMeasure)

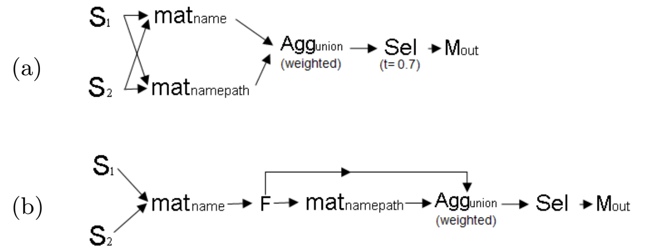

Figure 4: Combined and Sequential Process

of bringing all possible combinations of two matchers in a sequence. The library contains the following matchers: name (NM), namepath (NPM), children (CM), parent (PM), data type (DTM), leafs (LM). We selected a number of small sized schemas from the COMA evaluations [7], that contain representative properties for the domain of purchase order processing. For all pair wise combinations we created a parallel combined and a sequential combined strategy (see Figure 4(a) and (b)). In order to be comparable, we automatically tuned the aggregation weights, the selection thresholds and the filter thresholds for both strategies to achieve the best quality possible. For that purpose we applied a brute-force strategy of testing out the space of parameter settings in high detail.

From the best performing configuration we took the fastest ones and visualize their execution time in Figure 5. Since

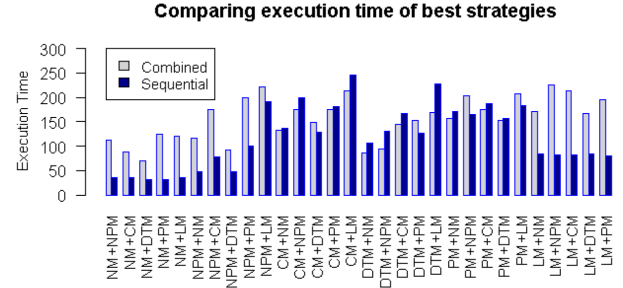

Figure 5: Comparing execution time of fastest, best performing sequential and combined strategies

we searched for the best quality configuration for sequential and combined strategies we found sequential strategies that did not filter at all. The achieved FMeasure of such sequential strategies is equal to the combined equivalent. These sequential strategies performed slower than the combined ones since they had to cope with the additional filter overhead. Yet the majority of combinations used a filter threshold that was bigger than 0. The execution times of those sequential strategies are significantly smaller than the combined ones. In some cases a significant part of comparisons was dropped out after the first matcher executed. Obviously there are some matchers that have better "filter-properties" whereas others should never serve as filter matcher. In order to make these observations reusable we represent the well-performing combinations in a special graph data structure called *Incidence Graph*.

Definition 3. The *Incidence Graph* is a directed graph that describes incidence relations between matchers mat from a given matcher library Mat in a given domain of schemas. The graph vertices represent match operations. If a significant speedup (> 20%) was achieved between matcher mat_a and matcher mat_b an edge (mat_a, mat_b) is added to the graph. Each match operation mat_x is annotated with the time R_x to execute the matcher on the simple mapping problem. Edges are annotated with the filter threshold that was found for the filter operator in the sequential execution example. They also store the percentage of the achieved speedup P_{ab} when executing mat_a before mat_b. P_{ab} can be computed as follows: $P_{ab} = 1 - (R_{seq}/R_{comb})$ with R_{seq} being the runtime of the sequential strategy on the given simple mapping problem and R_{comb} being the runtime for the combined strategy. The higher the value P_{ab} is the better the sequential speedup was.

Sometimes two edges (mat_1, mat_2) and (mat_2, mat_1) between two matchers mat_1 and mat_2 are put into the graph. This happens if two matchers behave similar and therefore serve as good filter matchers for one another.
Figure 6 shows the graph that we found for our given matcher library in the purchase order domain. For simplicity we omit the found filter thresholds. An edge from the name matcher to the leaf matcher states the following: The runtime of sequentially executing the name matcher before the leaf matcher was 71% faster than the parallel combination of these matchers (P_{ab}-value 0.71 on the edge). The average runtime R_x of the individual matchers on the given mapping problem is associated to the corresponding node. We ob-

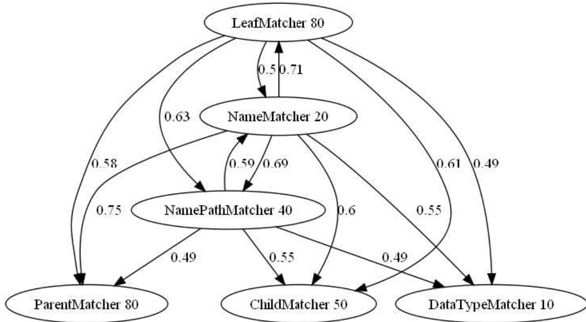

Figure 6: Incidence Graph Example containing only the individual runtimes and the achieved speed-up

served that the combinations that are encoded in the graph are quite stable for a given matcher library and different schemas. The filter thresholds were also nearly stable across different schemas. However, the graph must be recomputed for each new matcher library and schema domain since these properties are not generalizable. Later in the paper we will use the information about the achieved relative speedup and the individual runtime of matchers to decide, which sequential combination of two matchers is the best. In particular, the information in the graph will be an integral part of our simple cost model. Moreover, the graph will be used for the evaluation of our found rewrite rules.

4.3 Filter-based rewrite rule

After having introduced the matching process model, including the filter operator and the incidence graph, we can now define our matching process rewrite rules.

Definition 4. A *matching process rewrite rule* specifies a match pattern, a condition that has to hold, and a description of the changes to be applied for the found pattern instances.

We will use a simple notation for illustrating matching process rewrite rules that will be used within the following sections. The abstract notation containing pattern, condition and applied change is shown in Figure 7(a). If the condition evaluated over a found pattern instance is true, the changes below the horizontal bar are applied to the pattern instance.

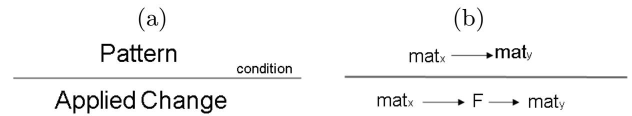

Figure 7: (a)Abstract matching process rewrite rule and (b) Simple example rule

Figure 7(b) shows a sample rule. An arrow between two operators specifies a direct path. The rule describes a pattern for two succeeding matchers mat_x and mat_y within a given process graph. The rule adds a filter operation F in between a found pair of these matchers to reduce the search space for mat_y.
In this paper we focus on filter-based rewrite rules. Yet, a number of other rules involving other schema and mapping operators are feasible. The use of filter rewrite rules is analogous to the use of predicate push-down rules for database query optimization which reduce the number of input tuples for joins and other expensive operators. The filter-based strategy tries to reduce the number of element pairs for match processing to also speed up the execution time.
The simple filter rule can be generalized to more complex patterns involving several matchers. An observation to utilize is that match processes are typically finished by selections to select correspondences exceeding a certain similarity threshold as likely match candidates. The idea is then to find rules for filtering out early those pairs that will be unable to meet the final selection condition.
We illustrate this technique with a rule for matching processes using several parallel matchers. The rule utilizes a relation $<_{profit}$ over a set of matchers Mat:
A set of matchers $Mat_p = \{mat_1 \dots mat_{n-1}\} \subset Mat$ profits from a matcher $mat_x \epsilon Mat$ with $mat_x \notin Mat_p$ written as $Mat_p <_{profit} mat_x$ if the following holds: There is a an edge in the incidence graph from mat_x to each $mat_i \epsilon Mat_p$.
Based on that relation we can now define a filter-based rewrite rule as shown in Figure 8. In our rewrite rules we introduce a special wildcard notation: For operators that have a schema as output we write $*_S$ and for operators that have a mapping as output we write $*_M$. The pattern on top of

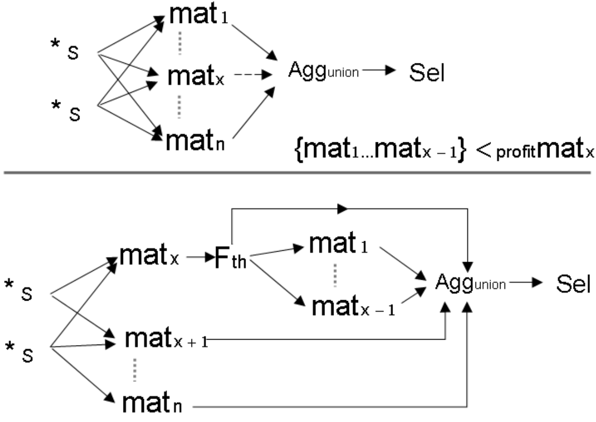

Figure 8: Filter-based rewrite rule ($Rule_S$) with schema input

the bar describes a part of a matching process that consists of two operators that output schemas $*_S$ followed by a set of matchers $\{mat_1 \ldots mat_n\}$ with $mat_x \epsilon \{mat_1 \ldots mat_n\}$ that are executed in parallel. Their result is aggregated in the Agg_{union} operation followed by a selection operation Sel. The condition evaluates to true, if a set of matchers profits from mat_x: $\{mat_1 \ldots mat_{x-1}\} <_{profit} mat_x$.
The applied rewrite below the bar adds a filter operation F after matcher mat_x. The behavior of the filter operation F will be described in the following paragraphs. The input of all matchers $\{mat_1 \ldots mat_{x-1}\}$ will be changed to F. The filtered result of mat_x will be added to the aggregation Agg_{union} and the original result of mat_x will be removed from the Agg_{union}. All matchers $\{mat_{x+1} \ldots mat_n\}$ that do not profit from mat_x remain unchanged.
The rule $Rule_S$ only covers patterns where the inputs are operators that provide schemas. Within complex matching processes that consist of sequential parts, the input to matchers could also be a mapping. In order to cover this, we added a second rewrite rule $Rule_M$ where the input is changed to $*_M$ (see Figure 9). After rewriting the process,

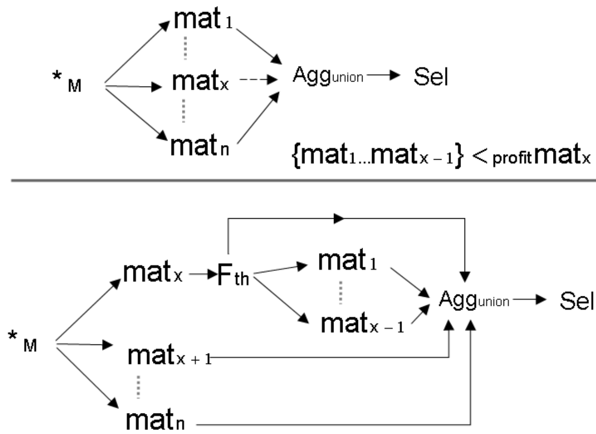

Figure 9: Filter-based rewrite rule ($Rule_M$) with mapping input

the condition of the filter operation F must be defined. We envision two types of filter conditions: a static and dynamic one.

4.3.1 Static filter condition

The static filter condition is based on a threshold. We refer to the threshold-based static filter with F_{th}. If a similarity value sim_{ij} in the input mapping is smaller than a given threshold, the comparison matrix entry is set to false. Each filter operation in a matching process could get different threshold-values that are adapted to the mapping they need to filter. In the rewrite we will reuse the annotated incidence graph that stores a filter threshold for each edge. If there are multiple outgoing edges from matcher mat_x we apply a defensive strategy: We set the filter-condition to the smallest threshold of all outgoing edges in the incidence graph from mat_x to matchers of $\{mat_1 \ldots mat_{x-1}\}$.

4.3.2 Dynamic filter condition

The dynamic filter condition adapts itself to the already processed comparisons and mapping results. Its overall idea is to filter out comparisons that already lost its chance to survive the final selection.
Given is a set of matchers $\{mat_1 \ldots mat_n\}$ that are contributing to a single aggregation operation Agg_{union}. Each matcher $mat_a \epsilon \{mat_1 \ldots mat_n\}$ has a weight w_a and computes an output mapping similarity value $sim_{a_{ij}}$. If the aggregation operation Agg_{union} applies a *weighted* aggregation function $aggsim_{ij}$ which is defined as follows:

$$aggsim_{ij} = \sum_{m=1}^{n} w_m * sim_{m_{ij}} \quad (1)$$

then the chance of not being pruned out can be computed after a matcher mat_x has been executed. Given the threshold $Sel_{threshold}$ of the final selection Sel the following condition can be checked:

$$(\sum_{\{mat_1 \ldots mat_n\}/mat_x} w_m * sim_{m_{ij}}) + w_x * sim_{x_{ij}} < Sel_{threshold} \quad (2)$$

If a matcher is not yet executed we consider it with the maximal possible similarity $sim_{m_{ij}} = 1$. If the computed aggregated similarity is smaller than the $Sel_{threshold}$ then the comparison can be pruned by setting the respective cell in the comparison matrix to false.
When more matchers are already executed, the actual similarities of matcher $sim_{m_{ij}}$ are known so that it will be much more probable that an element pair will be pruned. The dynamic filter condition ensures that the result of a filtered execution will not differ from a parallel execution. However, in most cases the dynamic filter does only begin pruning element pairs after some matchers have been executed.
Example:
Imagine three matchers with weights $w_1 = 0.3$, $w_2 = 0.4$ and $w3 = 0.3$ that contribute to an aggregation operation Agg_{union} and a following selection operation with a threshold of 0.7. If the first matcher computes a similarity for two elements $sim_{1_{ij}} = 0.2$ then the dynamic filter will not prune the comparison ((0.4*1 + 0.3*1) + 0.2 * 0.3 = 0.76 > 0.7). The more matchers are involved, the more unlikely it is that an element pair will be pruned early on. If the second matcher results in $sim_{2_{ij}} = 0.35$ then the element

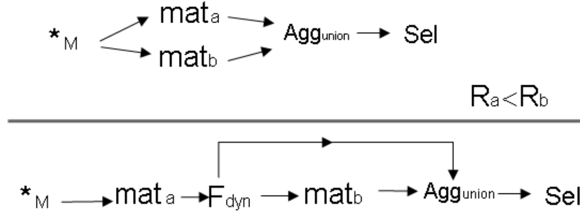

Figure 10: Filter-based rewrite rule ($Rule_{dyn}$) with mapping input and dynamic filter condition

pair can be pruned since it will never survive the selection. ((0.4*0.35 + 0.3*1) + 0.2 * 0.3 = 0.5 < 0.7).

This dynamic strategy can be softened by setting the worst case result similarities smaller than 1: $sim_{m_{ij}} < 1$ for matchers that have not yet been executed. However, similar to the static filter condition this could change the result of a matching process in comparison to the unchanged process. Since applying the dynamic filter condition can be done between arbitrary matchers without changing the final result we add a further rewrite rule $Rule_{dyn}$ (see Figure 10). Whenever two matcher mat_a and mat_b are executed in parallel, we apply these matchers in sequence and put a dynamic filter operator F_{dyn} in between them. The condition tries to ensure that the execution time of the first matcher R_a is smaller than the execution time of R_b. Typically this rewrite rule will be applied after the other filter rules have already been executed. Note that there also is an equivalent rule where the inputs are schemas instead of mappings.

4.4 Rewriting a matching process

To apply the matching process rewrite rule for filtering we need an algorithm to find suitable patterns for adaptation in order to improve performance. The algorithm *applyRule* takes an incidence graph IG, a matching process MP and a specific rewrite rule RW as input. First all matching pattern instances are identified in the given process (line 2). In line 5, for each pattern instance the cost C is computed as described in the following paragraph. The cost estimates are stored in a map (line 6). If the *costMap* is not empty, the best pattern instance is selected in line 8 and rewritten in line 9. The function *applyRule* will be called recursively in line 10 in order to iteratively rewrite all occurrences of the given pattern. The algorithm terminates when the *costMap* is empty and all possible pattern instances are rewritten (see line 7).

Algorithm *applyRule*
Input: Incidence Graph IG
Input: Matching Process MP
Input: Rewrite Rule RW
Output: Rewritten Process MP'
1. $MP' \leftarrow MP$
2. $patInstances \leftarrow findPatterns(RW, MP)$
3. $costMap \leftarrow \emptyset$
4. **for each** $pInst$ **in** $patInstances$
5. $cost \leftarrow computeCost(pInst, IG, RW)$
6. $costMap.put(pInst, cost)$
7. **if** $costMap.size() > 0$
8. $best \leftarrow costMap.getMinimum()$
9. $MP' \leftarrow rewrite(best, RW)$
10. $MP' \leftarrow applyRule(MP', IG, RW)$

We use a simple cost model based on the incidence graph to decide which pattern instance to rewrite for $Rule_S$ and $Rule_M$.

Definition 5. Given

- The incidence graph that contains individual runtimes of matchers R_m for all Mat matchers.
- The percentage of relative speedup P_{ab} between two matchers $mat_a, mat_b \in Mat$ as defined above. If there is no edge in the incidence graph from mat_a to mat_b then $P_{ab} = 0$

The cost $C_{x,\{1...n\}}$ of executing matcher mat_x before a set of matchers $\{mat_1 \dots mat_n\}$ can be computed by:

$$C_{x,\{1...n\}} = R_x + \sum_{a=1}^{n}(1 - P_{xa}) * R_a \qquad (3)$$

The rationale behind this cost-model is the following: The first matcher mat_x must be executed, hence its full runtime is considered. All matchers that have an incoming edge from mat_x add a fraction of their runtime cost to the overall cost that depends on the anticipated relative speedup P_{ab}. Computing the cost of a parallel execution of the given matchers is straightforward. Only the runtime-cost of all individual matchers need to be summed up.

Example:
Taking the values from the example incidence graph in Figure 6 the computed cost for first executing the name matcher and then executing all other matchers is: $20 + ((1-0.55)*10) + ((1-0.6)*50) + ((1-0.71)*80) + ((1-0.75)*80) + ((1-0.69)*40) = 100.1$. Whereas first executing the namepath matcher would generate higher cost: $40 + ((1-0.49)*10) + ((1-0.59)*20) + ((1-0.55)*50) + ((1-0.49)*80) + 80 = 196.6$.

5. MATCHING PROCESS EXECUTION SYSTEM

The overall architecture of our system is shown in Figure 11. It consists of three major components (1) a matching process modeling tool, (2) a matching process execution engine and (3) a matching process rewrite system.
We developed a schema matching system that is able to execute schema matching processes. Matching processes are defined at design time by applying our matching process graph model. The system is a meta matching system, i.e. we are agnostic towards the matcher library used. However, we implemented our own matcher library consisting of a number of operators and matchers. The system is able to

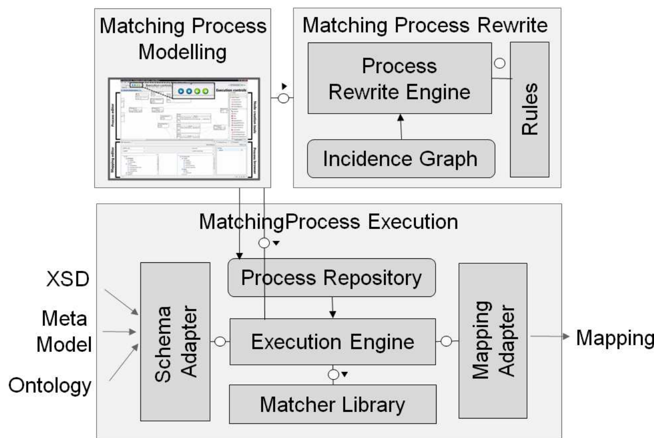

Figure 11: System architecture

match XML schemas, meta models or ontologies. In order to do this, we offer a number of adapters that transform these types of schemas into an internal model. Different selections of matchers can be used for defining a matching process for a specific domain. We also offer ways to export the found correspondences in domain specific mapping formats through mapping adapters.

5.1 Matching process modeling tool

In order to simplify the design of matching processes, we developed a graphical modeling tool for matching processes (see Figure 12 for a screenshot).

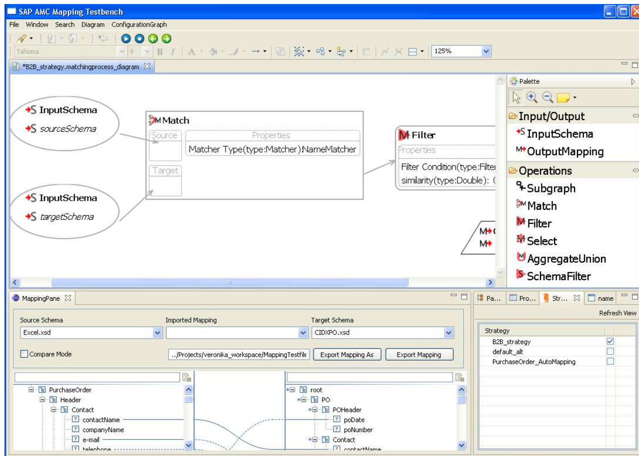

Figure 12: Process modeling Tool

Our matching process is visualized as a graph. This graph visualization makes relationships between operations and data explicit. Operations can be added to the graph by using drag and drop from the set of tools. A distinctive feature of our matching processes is the ability to contain another matching process as a subgraph. This hides complexity and improves reuse of processes. The user is able to easily drill down the hierarchy of subgraphs.

The problem of state of the art matching tools was that only highly skilled experts are able to exploit the auto matching potential. And even for them the process requires a high manual effort. Matching process designers model and tune matching processes to specific application areas. On request they are able to define new processes for given problem areas and store them in a central repository of "best practices" matching processes.

If a user within business applications like a business process modeling tool (see Introduction) wants to use auto matching, he simply gets a selection of predefined matching processes. The complexity of the underlying matching process is hidden to the user. He only has to select the most appropriate matching process for his matching problem which can be done automatically or supported through description.

With our matching process modeling tool we also introduce *debugging of matching processes*. Matching process debugging is primarily intended for matching process designers. We allow a graph designer to incrementally step through a matching process. On each step the input and output of an operation as well as its parameters can be visualized and changed using a graphical mapping view. Immediate feedback about the impact of parameter changes is given which helps to optimize individual parts of the process. The designer does not need to inspect concrete similarity values or matrices. Instead, the mapping visualization hides most of the complexity. Also the user is able to step back in a matching process, change parameters and operators and step forward with applied changes. This backward/forward stepping is a must in programming environments and helps to significantly improve the quality of a matching process. A user is able to exchange the order of operations. As discussed in Section 2 this could improve runtime performance.

5.2 Rewrite recommendation system

A distinctive feature of our matching process modeling tool is its connection to the developed matching process rewrite system introduced in this paper. The rewrite system supports the matching process designer in the development and tuning of a matching process. First the designer creates a graph with a number of matchers and operations. While developing the process, the designer asks the rewrite recommendation system to tune the performance of his strategy. It applies automatic rewrites on the graph to increase performance and quality. These rewrites can be confirmed or rejected by the process designer. The rewritten graph can be further extended or changed by the designer before finally storing it in the process repository. In this paper we introduced first rewrite rules, but we anticipate a number of further rules that improve speed as well as the quality of matching processes.

6. EVALUATION

In our evaluation we want to study whether the rewriting of matching processes can achieve a significant reduction of the overall execution time. For that purpose we compare the performance impact of applying the dynamic filter operations on a process with parallel matchers in comparison to the use of the static threshold-based filter operation. Then we investigate the influence of the static filter condition threshold on the final FMeasure and the execution time. We quantify our results by measuring the execution time (in milliseconds or seconds depending on the size of the mapping problem). For quantifying the quality we reuse the common FMeasure.

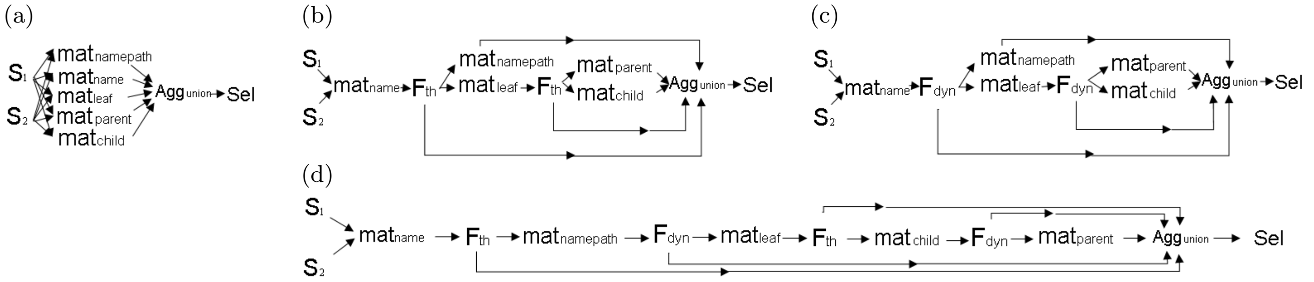

Figure 13: Rewritten matching processes after applying the rewrite rules

6.1 Setup

For our evaluation we took three different sets of schemas from the purchase order domain:

- A small set of simple purchase-order schemas that was taken by Do et al. in the COMA evaluation [7]
- A set of mid-sized SAP customer and vendor schemas of new SAP solutions
- Big-sized SAP XI and SAP PI Business Partner schemas and an SAP SRM Purchase Order message type

Table 1: Test Schemas

Schema Name	#Nodes	id
Apertum	143	S0
Excel	54	S1
CIDXPO	40	S2
Noris	56	S3
OpenTrans - Order	535	S4
SAP Business One - Business Partner	198	S5
SAP New - Vendor	173	S6
SAP New - Customer	222	S7
SAP PI - Business Partner Message	3953	S8
SAP XI - Business Partner	904	S9
SAP SRM - Purchase Order	5286	S10

Table 1 lists the different schemas used, the number of nodes of the schemas and a reference id for use in the description of the experiments. The chosen schemas exhibit most characteristics of complex matching scenarios such as verbose naming, flat structures and deeply nested structures. Also the size of the schemas is quite heterogeneous ranging from 40 elements to more than 5000 elements. For the evaluation we run several matching processes for computing mappings between combinations of the given schemas.
We computed the incidence graph for that matching library with 5 matchers: Name, Namepath, Leaf, Child, Parent and applied our different rewrite rules:

- (a) *Parallel*: First, no rewrite is applied and a parallel matching process is constructed.
- (b) *SequentialThreshold*: We applied the rewrite rules $Rule_S$ and $Rule_M$ to the parallel process. For the filter-operator we took the static filter F_{th}.
- (c) *SequentialDynamic*: We applied the rewrite rules $Rule_S$ and $Rule_M$ to the parallel process, but instead of F_{th} we took the dynamic filter F_{dyn}.
- (d) *SequentialBoth*: We applied $Rule_{dyn}$ onto the rewritten process from (b) to bring the remaining parallel operations in a sequence with a dynamic filter.

Figure 13 visualizes all generated matching processes used in the evaluation.

6.2 Results

Figure 14 shows the execution time results of the four evaluated matching processes on different combinations of schemas. As can be seen in all cases the execution times of the rewritten processes are significantly smaller. The improvement is stable over different sizes of schemas. It can be seen that ap-

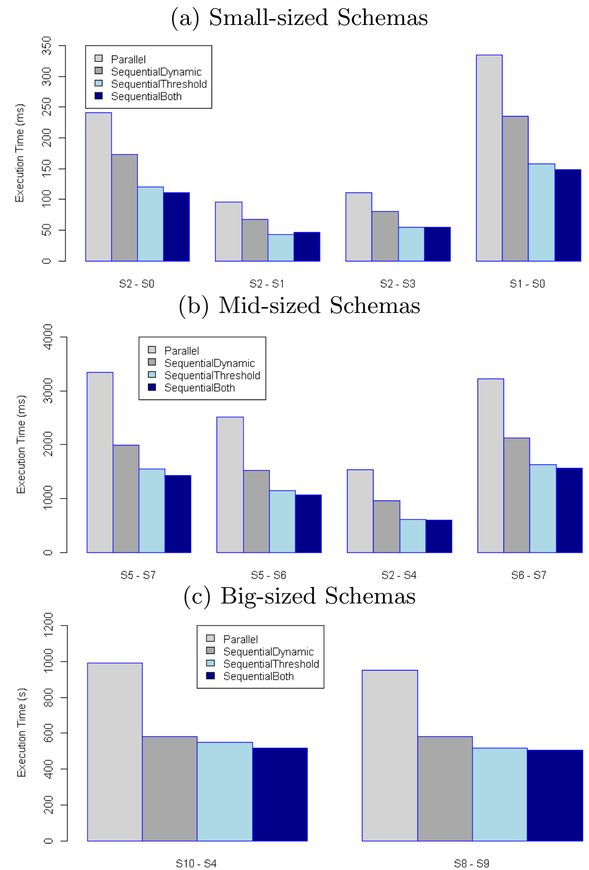

Figure 14: Comparing execution time between the combined and the rewritten processes.

plying the rewrite rules $Rule_M$ and $Rule_S$ and using the dynamic filter improves the performance by about 30-40% (SequentialDynamic). Here it is important to note that the result quality will never be different from its parallel combined equivalent (Parallel). By using the threshold-based process (SequentialThreshold) execution time is reduced by about 50%. For those schemas where we have a gold-standard mapping we could see that the FMeasure did also not deteriorate. This was achieved by a conservative filter condition to not prune element pairs relevant for the the final result. By applying the $Rule_{dyn}$ on a process that was already rewritten using a threshold-based filter an additional improvement can be achieved (SequentialBoth). Note that the first matcher still has to compute the full cross-product of source and target schema elements since the first filter is applied after that matcher. Therefore a significant part of the remaining 50% execution time is lost for that first matcher. By combining our rewrite rules with clustering or blocking strategies that we discussed in Section 2 we could further reduce the overall execution time significantly. Moreover, the rewrite rules allow to execute a matching process with more matchers within the same time-frame which could improve the matching quality.

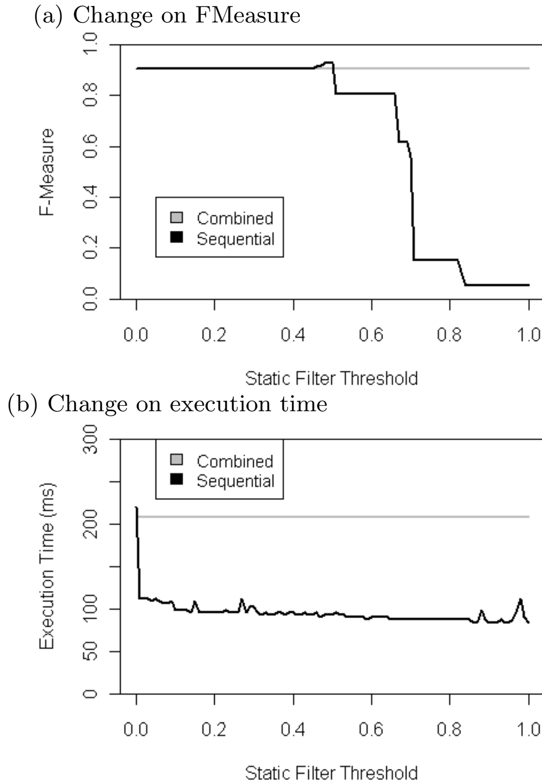

Figure 15: Influence on Execution Time and F-Measure of changing the filter threshold

In order to understand the influence of the filter threshold in the static filter-operator we changed the threshold from 0 (nothing is filtered) to 1 (everything is filtered). In Figure 15 the effect on FMeasure and performance is illustrated for matching schemas $S0$ and $S1$. It also explains the interplay of quality and execution time for a given matching problem. The graph (a) shows the FMeasure of the combined graph versus the FMeasure of the rewritten sequential strategy with a changing filter-threshold. Obviously for thresholds below 0.4 the FMeasure is not reduced. For higher threshold values above 0.4 the FMeasure begins to deteriorate drastically. But as can be seen in Figure 15(b) by increasing the filter threshold the execution time decreases rapidly already for very low thresholds while higher thresholds have only a modest impact. This behavior is explained as follows. The first matcher produces many element pairs with very low similarity values. If a filter threshold is set at a very low level e.g. 0.05 a very high number of pairs can be pruned. This significantly improves performance without changing the FMeasure since the pruned matches are very unlikely to contribute to the final result. Based on this observation we chose a very pessimistic threshold in our experiments since the performance improvements for higher thresholds are negligible.
Note that the slight increase of FMeasure at 0.45 is a positive side-effect of the filtering approach. Some matchers produce noisy results, which negatively influence the final weighted aggregation. The filter operator is able to prune out that noise. Hence a small increase of the FMeasure is possible.

7. CONCLUSION

In this paper we have introduced a generic approach to optimize the performance of matching processes. Our approach is based on rewrite rules and is similar to rewrite-based optimization of database queries. We initially focused on filter-based rewrite rules comparable to predicate push-down in database query plans. We assume that additional rewrite rules will allow further performance- or match quality improvements.
We implemented two versions of the filter-based strategy, a dynamic and static one. With our approach we are able to speedup matching processes executing several matchers in parallel. We rewrite the parallel processes into sequential processes and improve the performance significantly by early pruning many irrelevant element combinations. Our evaluation proves the effectivity of our approach on a number of SAP service interfaces that need to be matched when modeling Business Processes.
Also we further simplified the modeling of matching processes by introducing a dedicated graphical modeling tool. This tool significantly reduces the time for a matching process designer to create and tune a matching process to his needs. Here a special recommender system was developed that makes use of the presented rewrite rules.
A number of open issues will be addressed in our future work:
Identification of further rewrite rules We will develop further rewrite rules, that will not only focus on performance improvements but also on quality. Given the number of mapping and schema operators we see a big potential for improvement. Intersections of mappings typically help to improve precision. Thus applying two configurations of matchers and then intersecting the result instead of aggregating all into one AggregateUnion-operations will certainly be helpful.
Extend cost model Our cost-model seems to be sufficient for the filter-operation and its performance improvement goal. But when it comes to selecting between different types of rules a more advanced cost function is needed. That function should also cover quality aspects. It should be cus-

tomizable in order to let the user decide whether to focus on precision, recall or performance. Also, we experience major issues with memory usage for larger schemas. A cost function that also incorporates memory consumption would be highly beneficial.

Evaluate on other matcher frameworks We evaluated our approach on our matching library. But we are also planning to evaluate our finding on a much broader set of matcher libraries such as COMA++.

8. ACKNOWLEDGEMENT

This work was done in context of a doctoral work at the Database Group Leipzig lead by Prof. Dr. Erhard Rahm. The project was funded by means of the German Federal Ministry of Economy and Technology under the promotional reference “01MQ07012”.

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. *Proc. VLDB*, 2007.

[2] R. Baxter, P. Christen, and T. Churches. A Comparison of Fast Blocking Methods for Record Linkage. *ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[3] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. *Proc. SIGMOD*, 2007.

[4] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Rec.*, 33(4), 2004.

[5] C. Caracciolo et al. Results of the Ontology Alignment Evaluation Initiative 2008. *Third Int. Workshop on Ontology Matching*, 2008.

[6] H.-H. Do. *Schema Matching and Mapping Based Data Integration*. PhD thesis, University of Leipzig, 2005.

[7] H. H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approach. In *Proc. VLDB*, 2002.

[8] H. H. Do and E. Rahm. Matching large schemas: Approaches and evaluation. *Inf. Syst.*, 32(6), 2007.

[9] C. Drumm, M. Schmitt, H.-H. Do, and E. Rahm. Quickmig: automatic schema matching for data migration projects. *Proc. CIKM*, 2007.

[10] F. Duchateau, Z. Bellahsene, and R. Coletta. A Flexible Approach for Planning Schema Matching Algorithms. *Proc. OTM: On the Move to Meaningful Internet Systems*, 2008.

[11] F. Duchateau, Z. Bellahsène, M. Roantree, and M. Roche. An Indexing Structure for Automatic Schema Matching. *SMDB-ICDE: International Workshop on Self-Managing Database Systems*, 2007.

[12] M. Ehrig and S. Staab. QOM - Quick Ontology Mapping. *ISWC*, 2004.

[13] M. Ehrig, S. Staab, and Y. Sure. Bootstrapping ontology alignment methods with APFEL. *WWW*, 2005.

[14] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, 2007.

[15] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. *Proc. MoDELS*, 2008.

[16] L. Gravano et al. Approximate String Joins in a Database (Almost) for Free. *Proc. VLDB*, 2001.

[17] P. A. V. Hall and G. R. Dowling. Approximate String Matching. *ACM Comput. Surv.*, 12, 1980.

[18] W. Hu and Y. Qu. Block Matching for Ontologies. *ISWC*, 2006.

[19] W. Hu and Y. Qu. Falcon-AO: A practical ontology matching system. *Web Semant.*, 6(3), 2008.

[20] W. Hu, Y. Qu, and G. Cheng. Matching large ontologies: A divide-and-conquer approach. *Data Knowl. Eng.*, 67(1), 2008.

[21] Y. Lee, M. Sayyadian, A. Doan, and A. S. Rosenthal. etuner: tuning schema matching software using synthetic scenarios. *The VLDB Journal*, 16(1), 2007.

[22] J. Li, J. Tang, Y. Li, and Q. Luo. RiMOM: A Dynamic Multistrategy Ontology Alignment Framework. *IEEE Transactions on Knowledge and Data Engineering*, 21(8), 2009.

[23] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. *Proc. VLDB*, 2001.

[24] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. *Proc. ICDE*, 2002.

[25] M. Mochol, A. Jentzsch, and J. Euzenat. Applying an Analytic Method for Matching Approach Selection. *The First International Workshop on Ontology Matching*, 2006.

[26] N. F. Noy and M. A. Musen. The PROMPT suite: interactive tools for ontology merging and mapping. *Int. J. Hum.-Comput. Stud.*, 59, 2003.

[27] L. Palopoli, G. Terracina, and D. Ursino. DIKE: a system supporting the semi-automatic construction of cooperative information systems from heterogeneous databases. *Softw. Pract. Exper.*, 33, 2003.

[28] H. Paulheim. On Applying Matching Tools to Large-scale Ontologies. *The Third International Workshop on Ontology Matching*, 2008.

[29] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10, 2001.

[30] K. Saleem, Z. Bellahsene, and E. Hunt. PORSCHE: Performance ORiented SCHEma mediation. *Inf. Syst.*, 33, 2008.

[31] P. Shvaiko and J. Euzenat. A Survey of Schema-Based Matching Approaches. *Journal on Data Semantics IV*, 2005.

[32] M. Smiljanic, M. van Keulen, and W. Jonker. Using Element Clustering to Increase the Efficiency of XML Schema Matching. *Proc. ICDE Workshops*, 2006.

[33] H. Tan and P. Lambrix. A Method for Recommending Ontology Alignment Strategies. *ISWC/ASWC*, 2007.

[34] Yves R. Jean-Mary and M. R. Kabuka. Ontology matching with semantic verification. *Web Semantics: Science, Services and Agents on the World Wide Web*.