ABSTRACT

Despite their wide use and importance, target functional dependencies (fdfs) are still a bottleneck for the state-of-the-art Data Exchange (DE) engines. The consequences range from incomplete support to support at the expense of an important overhead in performance. We demonstrate here ChaseFUN, a DE engine that succeeds in effectively mitigating and taming this overhead, thus making target fdfs affordable even for very large-sized, complex scenarios. ChaseFUN is a custom chase-based system that essentially relies on exploiting chase step ordering and constraint interaction, so as to piecemeal process, parallelize and dramatically speed-up the chase. Interestingly, the structures and concepts at the core of our system moreover allow it to seamlessly uncover a range of usually opaque details of the chase. As a result, ChaseFUN’s two main strengths are: (i) its significant scalability and performance and (ii) its ability to provide detailed, granular insight on the DE process. Across our demonstration scenarios, we will emphasize our system’s practical performance and ability to scale to very large source instances and sets of constraints. Furthermore, we will aim at providing the user with a novel, behind-the-scenes view on the internals of the ongoing chase process, as well as on the intrinsic structure of a DE scenario.

CCS Concepts

• Information systems → Data exchange;

1. INTRODUCTION

Over the last decade, a plethora of mapping systems, including commercial ones such as IBM Rational Data Architect and research prototypes [1], have been developed for data transformation and data integration tasks. Data Exchange (DE) is one of the core processes of data transformation, relying on first-order logic and as such mainly pursued in research implementations. It revolves around translating data adhering to a source schema into data compliant with a target schema. The produced solution of a DE process, called the target schema. The produced solution of a DE process, called the target solution. Until a fix point (i.e. termination) is reached; the chase result upon termination then yields the target solution.

Existing DE engines span from completely covering all the above classes of constraints to supporting only subsets thereof. Indeed, custom chase engines [3] have been conceived for computing DE solutions under a wide range of constraints. While such engines may show high efficiency when dealing with tgdgs and other complex constraints, target fdfs yet hinder their performance and scalability. Alternatively, to aim for performance, DE engines like [4] have focused on outputting a set of SQL queries whose execution yields the target solution. While fast indeed, this approach is, however, mostly limited to s-t constraints. Extensions to subsets of target fdfs were shown possible, but typically requiring the additional input of source constraints [4].

Contributions. We demonstrate ChaseFUN, a novel chase-based engine for Data Exchange in the presence of arbitrary target fdfs and in the absence of source constraints. Our demonstration’s first focus will be on emphasizing our system’s performance on such DE scenarios. Indeed, as we will show, ChaseFUN is able to dramatically speed-up fdfs evaluation by leveraging constraints’ interaction and chase step ordering, and exploiting the granular processing and parallelization opportunities yielded by such concepts. By showcasing our system’s performance and scalability on large and complex DE scenarios, we then aim at showing that efficient support is yet attainable for general target fdfs, despite the overhead brought in by these constraints.

Interestingly, the concepts that stand at the core of ChaseFUN’s performance endorse our system with an additional property: the ability of shedding light on the internals of DE scenarios and the corresponding chase sequences. To this end, ChaseFUN offers several features allowing the user to consult and examine scenario and chase-related data, including a particularly informative step-by-step execution of the chase procedure. Accordingly, our demonstration’s second focus will be on showcasing such features, thus providing the user with a novel, behind-the-scenes view on the underpinnings of DE. To the best of our knowledge, such view has never been previously proposed by a chase-based DE engine.

Paper layout. We present an overview of ChaseFUN in Section 2 and the demonstration details in Section 3.
A detailed description of these concepts is available in [2].

(2) Set of assignments in their initial form (values $N_2$ are labelled nulls)

Figure 2: Assignments and Saturation Sets for the DE scenario in Figure 1.

2. SYSTEM OVERVIEW

Main algorithmic concepts. To efficiently produce DE solutions, ChaseFUn relies on a series of algorithmic concepts which we synthetically illustrate hereafter by means of a DE example depicted in Figure 1: Figure 1(i) shows the source instance $I$; (ii) shows the s-t tgds $(m_1, m_2, m_3)$ and target fds $(e_1$ and $e_2$); finally, (iii) shows the target solution $J$. This example shows a recurring transformation task, that of taking overlapping data across source tables (e.g. the actors who are active, who collaborate with each other, and win prizes) and injecting them into one or two target tables by merging duplicates via target functional dependencies. Transformations of this kind, involving general target fds and no source constraints, are indeed crucial in DE. Using our example, we describe hereafter the key concepts and tools used by ChaseFUn:

- Chase and assignments. Our chase flavor relies on the construction, selection and modification of a set of full s-t tgd assignments corresponding to the DE scenario. Each assignment is initially a mapping of universal variables in the s-t tgd body to source constants, further enriched with a mapping of existential variables in the s-t tgd head to fresh labelled nulls. Initial assignments for our running example are illustrated in Figure 2(i). Chase steps with s-t tgds consist in the selection of a yet available assignment, which is marked as no longer available and added to a target set. Chase steps with egds (fds) in turn modify assignments within the current target set. Upon termination of the chase.

1A detailed description of these concepts is available in [2].
same Saturation Set. We call this kind of relation a conflict between two s-t tgds. The Conflict Graph further characterizes, via conflict areas adorning vertices, the interaction we would expect between assignments of the respective s-t tgds. The Conflict Graph for our running example is depicted below, with vertices $v_1$, $v_2$, $v_3$ corresponding to s-t tgds $m_1$, $m_2$, $m_3$.

Areas($v_1$) = \{ca_1 = \langle (n, s), e_1 \rangle\},
Areas($v_2$) = \{ca_2 = \langle (n', s'), e_1 \rangle, ca_2' = \langle (p', w'), e_2 \rangle\},
Areas($v_3$) = \{ca_3 = \langle (n'', s''), e_1 \rangle, ca_3' = \langle (n''', s'''), e_1 \rangle\}.

By $v_1$ and $v_2$’s adornments we infer that any assignments of $m_1$ and $m_2$ may trigger the fd $e_1$, if they agree on the values for $n$ and $n'$, respectively $s$ and $s'$. Thus, since they exhibit such agreement, $a_{2|m_1}$ and $a_{1|m_2}$ must belong together in the same Saturation Set, i.e. $S_2$ in Figure 2(ii).

Besides its important role in Saturation Set construction, the Conflict Graph also provides very interesting parallelization opportunities. Indeed, one can show that a Saturation Set can never span across several connected components of the graph. ChaseFUN thus proceeds to Saturation Set construction and chase in parallel for each of the Conflict Graph’s connected components. Coupled to the Saturation Set-chase paradigm, parallel processing in turn further boosts our system’s speed and scalability.

Implementation and assessment. We have implemented ChaseFUN in Java (JVM version 1.8) using a JDBC interface for communication with an underlying PostgreSQL DBMS system. To stress-test ChaseFUN we have used several scenarios generated by using iBench[1], a novel data integration benchmark for generating arbitrarily large and complex schemas and constraints. We have considered three types of scenarios, in increasing complexity order: (i) OF scenarios generated with the default iBench object fusion primitive; (ii) OF+ scenarios, generated by combining the iBench object fusion and vertical partitioning primitives; (ii) OF++ scenarios, obtained by further modifying OF+ to yield s-t tgds with up to three atoms in the head. To further provide scale and assess the significance of ChaseFUN’s performance, we comparatively ran, on the same scenarios, one of the best DE engines currently available, namely the Llunatic system[3]. Figure 3 shows several measures obtained during this comparative evaluation.

DE workflow. Our system runs the DE process as a transition among four states, detailed hereafter.

- **1. Initial state:** waiting to load scenario. Prior to any interaction, ChaseFUN bootstraps with loading a Data Exchange scenario, comprising source and target schemas, constraints (s-t tgds and target fds) and source instance tuples.
- **2. Ready to chase state.** Once a scenario has been loaded, the Conflict Graph and the initial assignments are further computed. The system then reaches the Ready to chase state, where the user can browse scenario-related data: source and target schemas, source instance, s-t tgds and their assignments, target fds, as well as the Conflict Graph.

ChaseFUN provides windows and subwindows where baseline information can be selectively displayed by clicking on the corresponding tabs. Details can be further obtained by clicking on displayed elements. Figure 4 shows some of the system’s visual feedback in the Ready to chase state for our demonstration Scenario 2.

- **3. Chase in progress state.** Pressing the Start Chase button triggers the start of the chase procedure, with a choice among three chase modes. The first two modes both imply a continuous run, corresponding to a serial (sequential) and respectively parallel processing of the connected components in the Conflict Graph. The third mode in turn is aimed at allowing the user to peak into the chase, via a step-by-step execution. We detail this mode at the end of this section.

Throughout the chase, our system displays a range of useful information regarding the current state and evolution of the chase. This comprises progress bars for each connected component, the time spent chasing so far, as well as the evolving size of the solution so far constructed, by progressive materialization of completed Saturation Sets. Figure 5 illustrates such progress-related information.

- **4. Final state:** chase completed. Upon chase completion, in addition to previously available information, the user has access to the contents of the solution, as well as to a wide range of time and size statistics. She may export these...
statistics and/or wraparound to the initial state to run Chase-FUN on a new DE scenario.

**Step-by-step chase.** An essential feature of ChaseFUN is that of providing a detail-oriented, debug-like, step-by-step chase mode, aimed towards learning and understanding how the chase goes and what ChaseFUN’s unit actions are. When the step-by-step option is selected, a new window pops up, allowing the user to incrementally run and inspect the results of each Saturation Set’s construction and chase, alternating between tgds and egds. To improve the understanding of this process, ChaseFUN will provide a range of additional status information and visual cues.

![Step-by-step chase for Scenario 1](image)

**Figure 6: Step-by-step chase for Scenario 1**

Figure 6 shows a snapshot of the step-by-step chase for our demonstrated Scenario 1. This scenario corresponds to our running example in Figure 1 and we refer the reader to the detailed description of this example above. The snapshot corresponds to the construction and chase of the Saturation Set $S_1$. In particular, it depicts the state reached after the addition of the assignment $a_1m_3$ to $S_1$. The user has thus previously launched two tgd steps, namely for $m_1$ and $m_3$, whose corresponding Conflict Graph nodes have accordingly changed colour. Furthermore, the last tgd to add an assignment being $m_3$, its corresponding node is emphasized (enlarged). The edge linking $m_1$ and $m_3$ is equally emphasized (shown in blue), since $a_1m_3$ has been added because of its estimated interaction with an assignment of $m_1$ (i.e. $a_1m_1$). A subwindow displays the tuples obtained by the materialization of the current Saturation Set. Since after each tgd step egds must be applied, this is signaled to the user via the status information and the available button. Expectedly, once the user launches the next egds step, the tuples shown in Figure 6 will evolve to become the tuples shown in Figure 2(iv).

The step-by-step chase is importantly made available by our system’s “by design” granular processing of the chase, keeping the user-intended information small enough to remain easily accessible and understandable. To account for large-sized scenarios, ChaseFUN additionally provides pause/continue-like interactions, by letting the user alternate between the continuous serial and the step-by-step mode over the course of a single chase sequence.

### 3. DEMONSTRATION OVERVIEW

**Scenarios.** We will demonstrate our system on scenarios of increasing complexity in terms of both the number of constraints and the source instance size, namely one synthetic and three iBench-based[1] DE scenarios detailed hereafter.

- **Scenario 1** is our simplest scenario, corresponding to our running example in Figure 1 and comprising 9 tuples in the source, 3 $s$-$t$ tgds, 2 egds and a single connected component in the Conflict Graph.
- **Scenario 2** is on the mid-low side of the complexity spectrum. It comprises 400K tuples in the source and is built using twice the iBench default object fusion primitive (see Section 2), yielding 6 $s$-$t$ tgds, 2 egds, and 2 connected components in the Conflict Graph.
- **Scenario 3** increases the source instance size to 1M tuples, and further raises complexity by (i) increasing the number of iBench object fusion primitives applied and (ii) further plugging-in the vertical partitioning iBench primitive (in terms of Section 2 notation, this is an OF⁺ scenario). It includes 30 $s$-$t$ tgds, 30 egds, and 10 connected components in the Conflict Graph.
- **Scenario 4** raises the bar to 3M tuples in the source, and a larger yet number of constraints: 90 $s$-$t$ tgds and 90 egds, yielding a Conflict Graph of 30 connected components. We obtain this scenario by plugging in both object fusion and vertical partitioning primitives and further increasing the number of atoms in the $s$-$t$ tgds heads. Scenario 4 is in fact our OF⁺ stress-test scenario $F$ in Figure 3.

**Showcased features and messages conveyed.** On the above scenarios, we will demonstrate our system’s features and interactions described in Section 2, emphasizing Chase-FUN’s two main strengths:

- **Performance.** We will showcase our system’s processing speed and ability to scale for large and complex Data Exchange scenarios with target fds. As also witnessed by our experimental assessment, we are indeed not aware of a previous DE engine able to equate or outperform ChaseFUN in such settings. Since parallelization is one of our key performance factors, we will moreover show its impact and benefits by providing comparative runs using the parallel and serial chase modes offered by ChaseFUN. To present performance results, we will in particular focus on Scenarios 3 and 4. We also offer the possibility of live running comparative assessments of our system, such as the one charted in Section 2.

  - **User-intended view on the DE internals.** We will showcase the available Conflict Graph metadata, enabling a global, synthetic view on the links and interplay of constraints in the demonstrated DE scenarios. We will further emphasize the usefulness of the chase progress information provided by our system, as a first and important solution against the opacity problem of the chase operated by DE engines. Finally, we will extensively present the step-by-step chase mode described in Section 2, aimed at offering a novel, behind-the-scenes, refined view of the “low-level” granular operations of the DE process. We will showcase these capabilities on all demonstrated scenarios, and use Scenario 1 for an end-to-end presentation of the step-by-step run. Our demonstration will particularly focus on these detail and introspection opportunities provided by ChaseFUN. Indeed, to the best of our knowledge, ours is the first DE engine to provide the users with such informative and instructive features.

### 4. REFERENCES