ChaseFUN: a Data Exchange Engine for Functional Dependencies at Scale

Angela Bonifati
University of Lyon 1
angela.bonifati@univ-lyon1.fr

Ioana Ileana
Paris Descartes University
ioana.ileana@parisdescartes.fr

Michele Linardi
Paris Descartes University
michele.linardi@parisdescartes.fr

ABSTRACT

Despite their wide use and importance, target functional dependencies (fds) 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 fds 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, and satisfying a set of logic-based constraints. These constraints typically include: source-to-target (s-t) tuple-generating dependencies (tgds) and target constraints such as target tgds and target equality-generating dependencies (egds). Target egds in turn include primary keys, and more general functional dependencies (fds) on the target schema. The produced solution of a DE process, called target solution, is generally obtained using the chase algorithm. Such algorithm iteratively applies s-t and target constraints 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 tgds and other complex constraints, target fds 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 fds 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 fds 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 fd 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 fds, 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 previosuly 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.

(i) Source Instance I

Active_Actors

name	*surname*	*age*
Leonardo	Di Caprio	40
John	Redmayne	33

Actor_Collaboration

name$_1$	*surname*$_1$	*name*$_2$	*surname*$_2$
Leonardo	Di Caprio	Matthew	David
Fredric	March	Miriam	Hopkins

Awarded_Actor

name	*surname*	*oscarName*	*year*
John	Redmayne	Best Actor	2014
Wallace	Beery	Best Actor	1932
Fredric	March	Best Actor	1932
Marlon	Brando Jr	Best Actor	1954
Marlon	Brando Jr	Best Actor	1972

(ii) Dependencies (uppercase for existential variables)

$\mathbf{m_1}$:
$Active\_Actors(n, s, a) \rightarrow Actor(n, s, Y_1, Y_2)$

$\mathbf{m_2}$:
$Awarded\_Actor(n', s', p', w') \rightarrow$
$Actor(n', s', T, T_1) \wedge Oscar\_Prize(p', w', T)$

$\mathbf{m_3}$:
$Actor\_Collaboration(n'', s'', n''', s''') \rightarrow$
$Actor(n'', s'', E_1, E_2) \wedge Actor(n''', s''', E_3, E_2)$

$\mathbf{e_1}$:
$Actor(n, s, p, w) \wedge Actor(n, s, p', w') \rightarrow$
$(p = p') \wedge (w = w')$

$\mathbf{e_2}$:
$Oscar\_Prize(p, w, z) \wedge Oscar\_Prize(p, w, z') \rightarrow (z = z')$

(iii) Target Instance (Solution) J
(values N_x are labelled nulls)

Actor

*name**	*surname**	*idRewarding*	*idClub*
John	Redmayne	N_5	N_6
Wallace	Beery	N_7	N_8
Marlon	Brando Jr	N_{13}	N_{14}
Leonardo	Di Caprio	N_{15}	N_{16}
Matthew	David	N_{17}	N_{16}
Fredric	March	N_7	N_{19}
Miriam	Hopkins	N_{20}	N_{19}

Oscar_Prize

*oscarName**	*year**	*idActor*
Best Actor	2014	N_5
Best Actor	1932	N_7
Best Actor	1954	N_{13}
Best Actor	1972	N_{13}

Figure 1: Running example: DE scenario involving actors, prizes and collaborations.

$\mathbf{m_1}$
$a_1m_1 = \{n : Leonardo,\ s : Di\ Caprio,\ a : 40,\ Y_1 : N_1,\ Y_2 : N_2\}$
$a_2m_1 = \{n : John,\ s : Redmayne,\ a : 33,\ Y_1 : N_3,\ Y_2 : N_4\}$

$\mathbf{m_2}$
$a_1m_2 = \{n' : John,\ s : Redmayne,\ p' : BestActor,\ w' : 2014,\ T : N_5,\ T_1 : N_6\}$
$a_2m_2 = \{n' : Wallace,\ s : Beery,\ p' : BestActor,\ w' : 1932,\ T : N_7,\ T_1 : N_8\}$
$a_3m_2 = \{n' : Fredric,\ s : March,\ p' : BestActor,\ w' : 1932,\ T : N_9,\ T_1 : N_{10}\}$
$a_4m_2 = \{n' : Marlon,\ s : Brando\ Jr,\ p' : BestActor,\ w' : 1954,\ T : N_{11},\ T_1 : N_{12}\}$
$a_5m_2 = \{n' : Marlon,\ s : Brando\ Jr,\ p' : BestActor,\ w' : 1972,\ T : N_{13},\ T_1 : N_{14}\}$

$\mathbf{m_3}$
$a_1m_3 = \{n'' : Leonardo,\ s'' : Di\ Caprio,\ n''' : Matthew,\ s''' : David,\ E_1 : N_{15},\ E_2 : N_{16},\ E_3 : N_{17}\}$
$a_2m_3 = \{n'' : Fredric,\ s'' : March,\ n''' : Miriam,\ s''' : Hopkins,\ E_1 : N_{18},\ E_2 : N_{19},\ E_3 : N_{20}\}$

(i) Set of assignments in their initial form (values N_x are labelled nulls)

$S_1 = \{a_1m_1,\ a_1m_3\}$
$S_2 = \{a_2m_1,\ a_1m_2\}$
$S_3 = \{a_2m_2,\ a_3m_2,\ a_2m_3\}$
$S_4 = \{a_4m_2,\ a_5m_2\}$

(ii) Saturation Sets

$a_1m_1 = \{n : Leonardo,\ s : Di\ Caprio,\ a : 40,\ Y_1 : N_{15},\ Y_2 : N_{16}\}$
$a_1m_3 = \{n'' : Leonardo,\ s'' : Di\ Caprio,\ n''' : Matthew,\ s''' : David,\ E_1 : N_{15},\ E_2 : N_{16},\ E_3 : N_{17}\}$

(iii) S_1 after chase

Actor:

Leonardo	Di Caprio	N_{15}	N_{16}
Matthew	David	N_{17}	N_{16}

(iv) Materialization of S_1 after chase

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[1] 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:

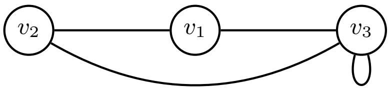
• Chase and assignments. Our chase flavor relies on the construction, selection and modification of a set of full s-t tgd assignments corresponding to the DE scenario. Each assignment is initially a mapping of universal variables in the s-t tgd body to source constants, further enriched with a mapping 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, the target set comprises the final (i.e. potentially modified by egd application) form of all assignments. We obtain the tuples in the target instance J by *materializing* this final form, i.e. by replacing variables in the tgds heads with their assigned values.

• Saturation Sets and chase order. One of the main reasons behind ChaseFUN's performance is its ability to tame the size of the intermediate target set during the chase, thus *systematically reducing the egd application scope*. To achieve such reduced size, we *group* assignments that are estimated to be at some point interacting via fds. We call such groups of assignments *Saturation Sets*. The chase of a Saturation Set will typically alternate between tgd steps and series of egd steps, applied to termination (i.e., until no egd remains applicable). Each Saturation Set thus acts as an *independent chase unit* that provides a part of the target solution. Figure 2(ii) shows a possible partition of the assignments in our running example into four Saturation Sets. We further show, in Figure 2(iii), the result of chasing S_1 (two s-t tgd steps corresponding to the assignments' selection, followed by an egd step with e_1 that modifies a_1m_1). Materializing this chase result yields the *Actor* tuples in Figure 2(iv). Note that these are indeed part of the target instance J in Figure 1(iii).

• The Conflict Graph and parallelization. To efficiently build Saturation Sets, our system uses a statically-built data structure called the *Conflict Graph*. Conflict Graph nodes correspond to s-t tgds, whereas edges witness the fact that the two s-t tgds, representing the connected nodes, have assignments that should potentially belong together in the

[1]A 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.



$$\mathbf{Areas(v_1)} = \{ca_1^1 = \langle (n, s), e_1\rangle\}.$$
$$\mathbf{Areas(v_2)} = \{ca_2^1 = \langle (n', s'), e_1\rangle, ca_2^2 = \langle (p', w'), e_2\rangle\}.$$
$$\mathbf{Areas(v_3)} = \{ca_3^1 = \langle (n'', s''), e_1\rangle, ca_3^2 = \langle (n''', s'''), e_1\rangle\}.$$

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 *PostgreSql9.4* 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 signficance 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[2].

Scenarios

SCENARIO	*s-t tgds*	*OF*	*OF+*	*OF++*	*# source tuples*
A	15	5 egds	10 egds	15 egds	500K
C	45	15 egds	30 egds	45 egds	1.5M
F	90	30 egds	60 egds	90 egds	3M

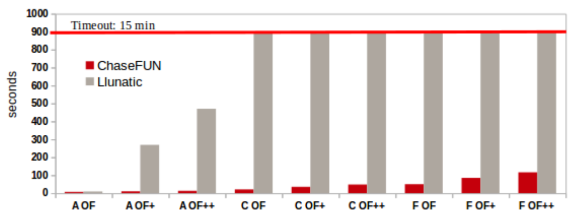


Figure 3: Evaluation and comparative assessment.

[2]We ran experiments on a 4-cores, i7-6600U 2.6 Ghz, 8GB RAM machine. We set a 15min timeout for all runs. We used the latest, most optimized version of Llunatic, as provided by its authors.

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.

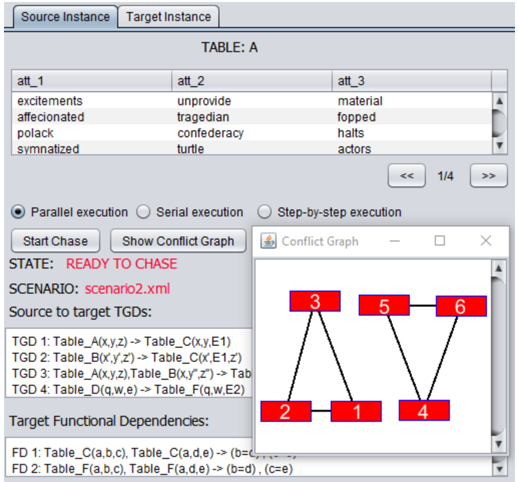


Figure 4: Ready to chase state for Scenario 2.

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.

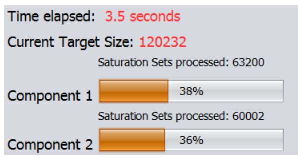


Figure 5: Progress information for Scenario 2.

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 ChaseFUN 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.

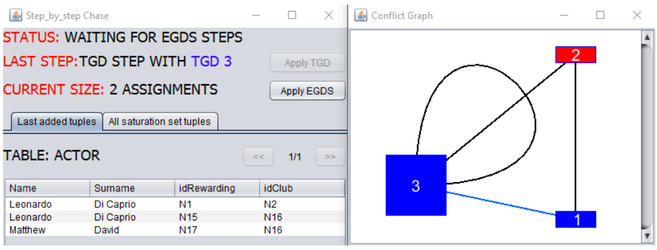


Figure 6: Step-by-step chase for Scenario 1

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 $a_1 m_3$ 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 $a_1 m_3$ has been added because of its estimated interaction with an assignment of m_1 (i.e. $a_1 m_1$). 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 ChaseFUN'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

[1] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[2] A. Bonifati, I. Ileana, and M. Linardi. Functional dependencies unleashed for scalable data exchange. In *Proceedings of SSDBM*, pages 2:1–2:12, 2016.

[3] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and cleaning. In *Proceedings of ICDE*, p. 232–243, 2014.

[4] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *PVLDB*, 4(12):1438–1441, 2011.