The iBench Integration Metadata Generator

Patricia C. Arocena [*]
University of Toronto
prg@cs.toronto.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Radu Ciucanu [†]
University of Oxford
radu.ciucanu@cs.ox.ac.uk

Renée J. Miller [*]
University of Toronto
miller@cs.toronto.edu

ABSTRACT

Given the maturity of the data integration field it is surprising that rigorous empirical evaluations of research ideas are so scarce. We identify a major roadblock for empirical work - the lack of comprehensive metadata generators that can be used to create benchmarks for different integration tasks. This makes it difficult to compare integration solutions, understand their generality, and understand their performance. We present iBench, the first metadata generator that can be used to evaluate a wide-range of integration tasks (data exchange, mapping creation, mapping composition, schema evolution, among many others). iBench permits control over the size and characteristics of the metadata it generates (schemas, constraints, and mappings). Our evaluation demonstrates that iBench can efficiently generate very large, complex, yet realistic scenarios with different characteristics. We also present an evaluation of three mapping creation systems using iBench and show that the intricate control that iBench provides over metadata scenarios can reveal new and important empirical insights. iBench is an open-source, extensible tool that we are providing to the community. We believe it will raise the bar for empirical evaluation and comparison of data integration systems.

1. INTRODUCTION

Despite the large body of work in data integration, the typical evaluation of an integration system consists of experiments over a few real-world scenarios (e.g., the Amalgam Integration Test Suite [20], the Illinois Semantic Integration Archive, or Thalia [15]) shared by the community, or *ad hoc* synthetic schemas and data sets that are created for a specific evaluation. Usually, the focus of such an evaluation is to exercise and showcase the novel features of an approach. It is often hard to reuse these evaluations.

[*]Supported in part by NSERC BIN.

[†]Supported in part by EPSRC platform grant DBOnto.

Proceedings of the VLDB Endowment, Vol. 9, No. 3

Patterson [21] states that "when a field has good benchmarks, we settle debates and the field makes rapid progress." Our canonical database benchmarks, the TPC benchmarks, make use of a carefully designed schema (or in the case of TPC-DI, a data integration benchmark, a fixed set of source and destination schemas) and rely on powerful data generators that can create data with different sizes, distributions and characteristics to test the performance of a DBMS. For data integration however, data generators must be accompanied by metadata generators to create the diverse metadata (schemas, constraints, and mappings) that fuel integration tasks. To reveal the power and limitations of integration solutions, this metadata must be created systematically controlling its detailed characteristics, complexity and size. Currently, there are no openly-available tools for systematically generating metadata for a variety of integration tasks.

1.1 Integration Scenarios

An integration scenario is a pair of a source and target schema (each optionally containing a set of constraints) and a mapping between the schemas. Integration scenarios are the main abstraction used to evaluate integration systems. Typically in evaluations, the relationship between the source and the target schema is controlled to illustrate different features of a system. We present the two primary ways such scenarios have been generated and illustrate how they have been used in evaluating different integration tasks.

Example 1 (Primitives). *Libraries of* ***primitives*** *can be used to create scenarios of different shapes and sizes. A set of synthetic* schema evolution *scenarios modeling micro and macro evolution behavior was proposed for evaluating mapping adaptation [25]. A set of finer-grained* schema evolution primitives *was proposed for testing mapping composition [9]. Each primitive is a simple integration scenario. A more general set of* mapping primitives *was proposed in STBenchmark, to permit the testing and comparison of mapping creation systems [2]. The primitives used in each approach depended on the integration task. In STBenchmark, the task was mapping creation. For composition [9], some additional primitives that were trivial for mapping creation (like an* `add-attribute` *primitive) were used because they can make the composition non-trivial (i.e., the composition may be a second-order (SO) mapping [14]).*

Using mapping primitives one can either test a single solution or compare multiple solutions on integration scenarios with different properties (by selecting different subsets of primitives). In addition, scalability can be tested by combining primitives and applying them repeatedly to generate

larger scenarios (larger schemas and mappings). As an alternative to using mapping primitives to create integration scenarios, one can also use *ad hoc* integration scenarios tailored to test specific features of an integration method. This approach was the norm before the primitive approach was introduced, and remains common.

EXAMPLE 2 (AD HOC SCENARIOS). *MapMerge [1] is a system for correlating mappings using overlapping sets of target relations, or using relations that are associated via target constraints. To evaluate their approach, the authors used three real biological schemas (Gene Ontology, UniProt and BioWarehouse) and mapped the first two schemas to the third (BioWarehouse). These real scenarios reveal applicability of their technique in practice, but do not allow control over the metadata characteristics (number of mappings, their complexity, etc.). Unfortunately, the primitives of previous approaches did not meet their needs as the mappings created by sets of primitives rarely shared target relations and the scenario generators did not permit detailed control over how target constraints were generated.*

Hence, to evaluate MapMerge, the authors created their own ad hoc *integration scenarios. These were designed to let the researchers control the degree of sharing (how many mappings used the same target relation) and the set of target schema constraints, the two crucial parameters for the MapMerge algorithm. This set-up permitted control over the mapping scenario complexity which was defined as the number of hierarchies in the target schema. However, this definition of complexity is not appropriate for other tasks (e.g., mapping composition) that do not depend so directly on the characteristics of a schema hierarchy. Furthermore, this scenario generator is limited to this one specific schema shape. These characteristics make it unlikely that others can benefit from this test harness for evaluating their work.*

The widespread use of *ad hoc* scenarios is necessitated by the lack of a robust metadata generator.

1.2 Contributions

We present iBench, a flexible integration metadata generator and make the following contributions.

- We define the metadata-generation problem and detail important design decisions for a flexible, easy-to-use solution to this problem (Section 2). Our problem definition includes the generation of (arbitrary) independent schemas with an (arbitrary) set of mappings between them, where the independent variables controlled by a user include the schema size, the number of mappings (mapping size), the mapping complexity, and the mapping language. The problem definition also permits the generation of schemas where the relationship between the schemas is controlled in a precise way. Hence, this relationship can be viewed as an independent variable, controlled by a user of the generator through the use of schema primitives.
- We show that the metadata-generation problem is NP-hard and that even determining if there is a solution to a specific generation problem may be NP-hard.
- We present MDGen, a best-effort, randomized algorithm, (Section 3) that solves the metadata-generation problem. The main innovation behind MDGen is in permitting flexible control over independent variables describing the metadata in a fast, scalable way.
- We present an evaluation of the performance of iBench (Section 4) and show that iBench can efficiently generate both large integration scenarios and large numbers of scenarios. We show that iBench can easily generate scenarios with over 1 Million attributes, sharing among mappings, and complex schema constraints in seconds. iBench can be easily *extended* with new (user-defined) primitives and new integration scenarios to adapt it to new applications and integration tasks. We present a performance evaluation of this feature where we take several real-world scenarios and scale them up by a factor of more than 10^3 and combine these user-defined primitives with native iBench primitives. We show that this scaling is in most cases linear.
- We demonstrate the power of iBench by presenting a novel evaluation of three mapping discovery algorithms, Clio [12], MapMerge [1], and ++Spicy [17] (Section 5). Our evaluation systematically varies the degree of source and target sharing in the generated scenarios. This reveals new insights into the power (and need for) mapping correlation (used in MapMerge) and core mappings (used in ++Spicy) on complex scenarios. As the first generator that varies the amount of generated constraints, we show for the first time how this important parameter influences mapping correlation. We also quantify how increasing target keys improves the quality of ++Spicy's mappings.

2. METADATA-GENERATION PROBLEM

We now motivate our design decisions and formalize the metadata-generation problem. We use a series of examples which showcase the need for certain features. This culminates in a formal problem statement and a complexity analysis which demonstrates that the problem is NP-hard.

2.1 Generating Integration Scenarios

EXAMPLE 3. *Alberto has designed a new mapping inversion algorithm called* mapping restoration. *Like the Fagin inverse [11], not every mapping has a restoration, but Alberto believes that most mappings do. To demonstrate this empirically, he would like to compare the success rates of implementations for restoration and the Fagin inverse. To do this, he must be able to generate large sets of mappings.*

Alberto's problem would be solved by a mapping generator. We define an *integration scenario* as a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where $\mathbf{S}$ and $\mathbf{T}$ are schemas and Σ is a set of logical formulas over $(\mathbf{S}, \mathbf{T})$. Relation (and attribute) names in the schemas may overlap, but we assume we can identify each uniquely (for example, using $\mathbf{S}.R.A$ and $\mathbf{T}.R.A$). We generalize this definition later (Definition 2) to permit constraints (e.g., keys, foreign keys), instances of the schemas, and transformations implementing the mappings.

To fulfill Alberto's requirements, a metadata generator is needed that can efficiently create integration scenarios. One approach would be to generate schemas and constraints and apply an existing mapping discovery algorithm to generate mappings for these schemas. However, this can be quite expensive and the output would be biased based on the idiosyncrasies of the algorithm. We take a different approach motivated by data generation and permit a user to control the basic metadata characteristics. Ideally, Alberto should be able to specify the values (or admissible intervals of values) for the characteristics he would like to control as an input to the generator. The metadata generator then has to create an integration scenario (or many scenarios) that fulfill these restrictions. This is only possible if the restricted

Parameter	Description
$\pi_{SourceRelSize}$	Number of attributes per source relation
$\pi_{TargetRelSize}$	Number of attributes per target relation
$\pi_{SourceSchemaSize}$	Number of relations in source schema
$\pi_{TargetSchemaSize}$	Number of relations in target schema
$\pi_{NumMaps}$	Number of mappings
$\pi_{SourceMapSize}$	Number of source atoms per mapping
$\pi_{TargetMapSize}$	Number of target atoms per mapping
$\pi_{MapJoinSize}$	No. of attributes shared by two atoms
$\pi_{NumSkolem}$	% of target variables not used in source
$\pi_{SkolemType}$	Type of value invention (`all`, `random`, `key`)
$\pi_{SkolemSharing}$	% of value inv. fcts shared amg mappings
$\pi_{SourceSharing}$	% mappings that share source relations
$\pi_{TargetSharing}$	% mappings that share target relations

Table 1: Scenario Parameters

Parameter	Min	Max
$\pi_{SourceRelSize}$ and $\pi_{TargetRelSize}$	2	5
$\pi_{SourceSchemaSize}$ and $\pi_{TargetSchemaSize}$	2	10
$\pi_{NumMaps}$	1	5
$\pi_{SourceMapSize}$	1	1
$\pi_{TargetMapSize}$	1	10
$\pi_{NumSkolem}$	0%	50%

Table 2: MD task: indep. schemas & LAV mappings

characteristics can be measured efficiently, e.g., the number of source relations. To define this more precisely, we will assume relational schemas and mappings specified as source-to-target tuple-generating dependencies, arguably the most commonly used mapping language in integration [24].

A *source-to-target tuple-generating dependency* [13] (s-t tgd) is a formula of the form $\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow \exists \mathbf{y}\ \psi(\mathbf{x}, \mathbf{y}))$, where $\mathbf{x}$ and $\mathbf{y}$ are disjoint vectors of variables; $\phi(\mathbf{x})$ and $\psi(\mathbf{x}, \mathbf{y})$ are conjunctions of atomic formulas over the source schema $\mathbf{S}$ and target schema $\mathbf{T}$, respectively. All variables of $\mathbf{x}$ are used in ϕ and all variables of $\mathbf{y}$ are used in ψ. S-t tgds are sometimes called global-and-local-as-view (GLAV) mappings [16]. Local-as-view (LAV) mappings have a single atom in ϕ whereas global-as-view (GAV) mappings have a single atom in ψ and no existential variables ($\mathbf{y}$ is empty).

A major choice in the design of a metadata generator is what scenario characteristics should be controllable by the user. On the one hand, we would like to give the user full control over the generation, on the other hand, we do not want to overwhelm her with a plethora of rarely used options. Clearly, the number of mappings generated and size of these mappings (determined as the number of conjuncts) should be controllable. A user may also want to control the incompleteness in a mapping, i.e., the number of existential variables used in the target expression ($\|\mathbf{y}\|$). Using this intuition, one can construct a generator that creates two independent schemas of a specified size (number of relations and attributes per relation) and mappings (of a specified size) between the schemas. To model this, we use scenario parameters in our problem definition (Table 1) and allow a user to specify a minimum and maximum value for each. We will explain the last few later in this section.

Definition 1 (MD Task). *Let Π be a set of scenario parameters (shown in Table 1). A metadata-generation task Γ is a set of constraints of the form $min_\pi < \pi < max_\pi$ where $\pi \in \Pi$. For a given integration scenario $\mathcal{M}$ we say that $\mathcal{M}$ is a solution for Γ if $\mathcal{M} \models \Gamma$.*

Note that we do not require that every scenario characteristic is constrained by an MD Task. This enables a user to only vary the characteristics she wants to control.

Example 4. *To evaluate his hypothesis that restoration is more successful for LAV than for GLAV mappings, Alberto creates the MD Task in Table 2 to create LAV mappings (and later GLAV mappings by increasing parameter $\pi_{SourceMapSize}$) and runs his algorithm over the generated mappings. A scenario that is a solution for the task in Table 2 is shown in Figure 1. The red lines represent correspondences (mapping variables shared between the source and target), ignore the black lines for now. Mappings for this scenario are m_1 and m_2.*

$m_1 : \forall a, b\ Cust[a, b] \rightarrow \exists c\ Customer[c, a, b]$
$m_2 : \forall a, b\ Emp[a, b] \rightarrow \exists c\ Person[c, b]$

To see how the compliance of the scenario with the constraints of the task is evaluated consider $\pi_{SourceRelSize}$. The task requires that $2 \leq \pi_{SourceRelSize} \leq 5$. Since both relation Cust and Emp have 2 attributes, the scenario fulfills this constraint. As another example, the task specifies that up to 50% of the target variables may be existential (Skolems). Both m_1 and m_2 fulfill this constraint (1/3 and 1/2).

2.2 Controlling Incompleteness

Example 5. *Alberto has extended his system to support second-order (SO) tgds [14], a broader class of mappings with finer control over incompleteness. He needs a generator that can create SO tgd mappings and control the incompleteness. To create SO tgds that are not equivalent to any s-t tgd, he would like to control how arguments of Skolem functions representing unknown values are chosen and how Skolem functions are reused among mappings.*

While s-t tgds permit incompleteness via existential variables, this incompleteness can also be represented via Skolem functions. Indeed, because of this, we called the scenario parameter controlling this $\pi_{NumSkolem}$. The mapping m_1 from Example 4 is equivalent to the following SO tgd.

$m_1 : \exists f\ \forall a, b\ Cust[a, b] \rightarrow Customer[f(a, b), a, b]$

Using this representation, some limitations of s-t tgds become clear. First, all functions depend on all universally quantified variables in the mapping. Second, two mappings cannot share the same value-invention (Skolem) function (i.e., function f cannot be used in both m_1 and m_2). We define the metadata-generation problem to include scenarios that use plain SO tgds [4], a strict superset of s-t tgds with more flexible value-invention semantics. Arenas et al. [4] argue that plain SO tgds are the right choice for important integration problems related to composition and inversion.

A plain SO tgd [4] is an existential second-order formula of the form: $\exists \mathbf{f}(\ (\forall \mathbf{x_1}(\phi_1 \rightarrow \psi_1)) \wedge \cdots \wedge (\forall \mathbf{x_n}(\phi_n \rightarrow \psi_n))\ )$ where (a) each ϕ_i is a conjunction of relational source atoms over variables from $\mathbf{x_i}$ such that every variable from $\mathbf{x_i}$ appears in at least one atom and (b) each ψ_i is a conjunction of atomic formulas that only uses terms of the form $T(y_1, ..., y_k)$ where $T \in \mathbf{T}$ and each y_i is either a variable from $\mathbf{x_i}$ or a term $f(x_1, ..., x_k)$ where f is a function symbol in $\mathbf{f}$ and each $x_j \in \mathbf{x_i}$. We refer to each clause of a plain SO tgd, $\forall \mathbf{x_i}(\phi_i \rightarrow \psi_i)$, as a mapping to keep the analogy with s-t tgds.

To control the complexity of SO tgd generation, we use additional parameters. The $\pi_{SkolemSharing}$ parameter controls the percentage of Skolems (from $\mathbf{f}$) shared between mappings. The $\pi_{SkolemType}$ controls how the functions are parameterized: for value *All* each function depends on all universal variables in the mapping, for *Random* each function depends on a random subset of universal variables, and for *Keys* each function depends on the primary keys of the mapping's source relations. The settings of $\pi_{SkolemSharing} = 0\%$ and $\pi_{SkolemType} = All$, limits the mappings to s-t tgds.

EXAMPLE 6. *By modifying the parameter* $\pi_{SkolemType}$ *in Table 2 to* Random *and* $\pi_{SkolemSharing}$ *to a max of 100%, the following two mappings may be created as a solution.*

$$\exists f\ (\forall a, b\ Cust[a, b],\ Emp[a, b] \rightarrow Insider[f(a), a, b]$$
$$\forall a, b\ Emp[a, b] \rightarrow Person[f(a), b])$$

In addition to supporting scenario parameters that control schema and mapping size, we let a user limit the mapping language to the important classes of s-t tgds, LAV (by restricting the number of source atoms per mapping to 1 using parameter $\pi_{SourceMapSize}$), and GAV (by restricting the number of target atoms per mapping $\pi_{TargetMapSize} = 1$ and $\pi_{NumSkolem} = 0\%$) that have been identified as important in a survey of mapping languages [24]. Many integration systems do not support all languages, and even when multiple mapping languages are supported, a system may provide different performance depending on the language.

2.3 Schema Primitives

EXAMPLE 7. *Alice has built a system that creates executable data exchange transformations from s-t tgds. To thoroughly test her approach, she needs a large number of diverse scenarios (schemas and mappings). She decides to follow Alberto's methodology to generate large sets of mappings. For each she creates a transformation script and runs it over source databases (created by a data generator like ToXgene [8]). While she was able to test her system's performance over a wide variety of metadata with little effort, she feels that the performance of her system on independent schemas with random mappings is not necessarily predictive of its performance for real world settings. Similar to how database benchmarks such as TPC-H provide realistic queries and data she would like to create scenarios that give her a better understanding of how her system will perform on real mappings that are likely to arise in practice.*

As Alice in the example, current evaluations do not test integration systems on arbitrary scenarios, rather they control quite precisely the relationship between the source and target schema. This has often been modeled using mapping primitives. A mapping primitive is a parameterized integration scenario that represents a common pattern. For instance, vertical and horizontal partitioning are common patterns that a metadata generator should support. Primitives act as templates that are instantiated based on user input. A metadata generator should allow a user to determine what primitives to use when generating an integration scenario. By doing so, the user can control the relationship between the schemas. For example, she can use only primitives that implement common transformations used in ontologies (e.g., that increase or decrease the level of generalization in an `isa` hierarchy) or she can choose to use only those primitives that implement common relational schema evolution transformations. The two existing mapping generators follow this approach. The first using schema evolution primitives proposed by Yu and Popa [25] to test a mapping adaptation system and extended by Bernstein et al. [9] to test a mapping composition system. The second, STBenchmark, used a set of transformation primitives for testing mapping creation and data exchange systems [2].

EXAMPLE 8. *Consider the integration scenario in Figure 1 that could be created using two primitives. The first,* ***copy-and-add-attribute*** *(ADD), creates the source relation* `Cust(Name, Addr)` *and copies it to a target relation that contains another attribute,* `Customer(Name, Addr, Loyalty)`*. The new attribute* `Loyalty` *does not correspond to any source attribute. The second primitive,* ***vertical-partitioning-hasA*** *(VH) creates a source relation,* `Emp(Name, Company)`*, and vertically partitions it into:* `Person(Id, Name)` *and* `WorksAt (EmpRec,Firm,Id)`*. The VH primitive creates a* ***has-a*** *relationship between the two target relations (modeled by a foreign key (FK)). Specifically, the primitive VH creates new keys for the two target relations (*`Id` *for* `Person` *and* `EmpRec` *for* `WorksAt`*) and declares* `WorksAt.Id` *to be a FK for* `Person`*.*

A generator should support a comprehensive set of primitives described in the literature and should be extensible to incorporate new, user-defined primitives (see Section 2.5). Furthermore, a generator should avoid proliferation of primitives. To achieve this, following the approach of STBenchmark, primitives should be parameterized. Primitives should work seamlessly with scenario parameters. The generator should create scenarios that include mappings (relations and constraints) created by primitives and independent mappings (relations and constraints) whose generation is controlled by the scenario parameters.

Comprehensive Set of Primitives. A partial list of primitives is given in Figure 4 (the complete list of 17 primitives is in our technical report [6].) We define the metadata-generation problem to use primitives including the mapping primitives of STBenchmark [2] (e.g., *VP*) and the schema evolution primitives [9, 25] (e.g., *ADD*). We also include as primitives, *ad hoc* scenarios that we found in evaluations in the literature. For example, *VA* (see Example 2) is the schema hierarchy scenario used to evaluate MapMerge [1]. In the MD Task, a user specifies the number of times each primitive should be instantiated via a parameter θ_P (with default value of 0) where P is the name of the primitive.

Parameterized Primitives. As done in STBenchmark, primitives can be parameterized to change features like the number of relations created. This permits, as a simple example, a single parameter $\lambda_{NumPartitions}$ (we will use λ for primitive parameters and θ for setting the number of instances of a primitive) to be used to vertically partition a relation into two, three, or more relations.

EXAMPLE 9. *Alice has created the task in Table 3 (left) to create LAV mappings. Notice she has not set the source (or target) schema size parameter or the number of mappings parameter. Hence, mappings will only be created by invoking primitives. This configuration will create n source relations and n mappings for* $40 \leq n \leq 100$ *(depending on how many primitives are instantiated). The number of target relations depends on the number of partitions chosen for each primitive instantiation (these primitives use the* $\lambda_{NumPartition}$ *parameter so each primitive invocation will create between 2 and 10 target relations). Alice finds that her transformations perform well on the hundreds of scenarios she creates using this configuration file. She now wants to test how quickly performance deteriorates as the scenarios become less well-behaved with more portions of the schemas and mappings being independent. She now uses the configuration file in Table 3 on the right and creates 200 scenarios. These scenarios will all have exactly 100 primitive invocations, but in addition will have between 0 and 100 arbitrary mappings (and relations). She can then plot her performance vs. the percentage of independent mappings (0 to 50% independent mappings).*

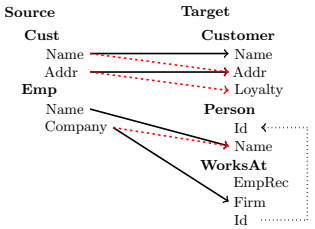


Figure 1: Random Mapping (-), ADD and VH Primitives (-)

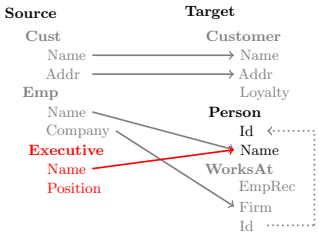


Figure 2: ADL with Target Sharing

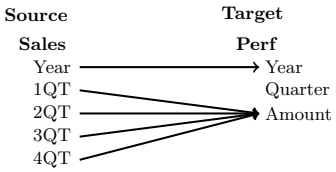


Figure 3: A Pivot UDP

Name	Description	Example	FKs
ADD	Copy a relation and add new attributes	$R(a,b) \rightarrow S(a,b,f(a,b))$	No
ADL	Copy a relation, add and delete attributes in tandem	$R(a,b) \rightarrow S(a,f(a))$	No
CP	Copy a relation	$R(a,b) \rightarrow S(a,b)$	No
HP	Horizontally partition a relation into multiple relations	$R(a,b) \wedge a = c_1 \rightarrow S_1(b)$ $R(a,b) \wedge a = c_2 \rightarrow S_2(b)$	No
SU	Copy a relation and create a surrogate key	$R(a,b) \rightarrow S(\underline{f(a,b)},b,g(b))$	No
VH	Vertical partitioning into a HAS-A relationship	$R(a,b) \rightarrow S(\underline{f(a)},a) \wedge T(\underline{g(a,b)},b,f(a))$	Yes
VI	Vertical partitioning into an IS-A relationship	$R(\underline{a},b,c) \rightarrow S(\underline{a},b) \wedge T(\underline{a},c)$	Yes
VNM	Vertical partitioning into an N-to-M relationship	$R(a,b) \rightarrow S_1(\underline{f(a)},a) \wedge M(\underline{f(a)},\underline{g(b)}) \wedge S_2(\underline{g(b)},b)$	Yes
VP	Vertical partitioning	$R(a,b) \rightarrow S_1(\underline{f(a,b)},a) \wedge S_2(\underline{f(a,b)},b)$	Yes

Figure 4: Exemplary Native Primitives

Parameter	Min	Max
θ_{VH}	10	25
θ_{VI}	10	25
θ_{VNM}	10	25
θ_{VP}	10	25
$\lambda_{NumPartitions}$	2	10
$\pi_{SourceRelSize}$	5	10
$\pi_{TargetRelSize}$	7	12

Parameter	Min	Max
θ_{VH}	25	25
θ_{VI}	25	25
θ_{VNM}	25	25
θ_{VP}	25	25
$\lambda_{NumPartitions}$	3	3
$\pi_{SourceRelSize}$	5	10
$\pi_{TargetRelSize}$	7	12
$\pi_{NumMappings}$	100	200
$\pi_{SourceSchemaSize}$	100	200
$\pi_{TargetSchemaSize}$	300	400

Table 3: Example MD Tasks

Primitives and Scenario Parameters. To satisfy Alice's needs, unlike previous metadata generators (e.g., STBenchmark) where the scenario is generated as a union of a number of primitive instantiations, we define the metadata generation problem in a more flexible fashion where the user can request a number of instances for each primitive type and/or use scenario parameters to generate independent metadata.

2.4 Sharing Among Mappings

EXAMPLE 10. *Alice notices an anomaly in her results. Even though her transformations remove subsumed (redundant) tuples created by older transformation systems like Clio [22], varying the amount of redundancy in the source instances does not affect performance for schemas created only using primitives. She realizes that no target relation is being populated by more than one mapping, i.e., there is no redundancy created in the target. She has forgotten to set an important iBench parameter controlling the degree to which mappings can share source or target relations.*

By combining multiple instances of the primitives, a user can generate diverse integration scenarios with a great deal of control over the scenario characteristics. However, while real-world schemas contain instances of these primitives, they often contain metadata that correspond to a combination of primitives. To create such realistic scenarios, it is important to permit sharing of metadata among mappings and control the level of sharing. For example, in a real-world scenario typically some relations may be used by multiple mappings (e.g., employees and managers from a source schema are both mapped to a person relation in the target schema), but it is uncommon for all mappings to share a single source relation. This type of sharing of metadata has been recognized as an important feature by Alexe et al. [2], but they tackle this issue by providing some primitives that combine the behavior of other simpler primitives.

EXAMPLE 11. *Suppose we apply a third primitive,* **copy-add-delete-attribute** *(ADL). Without sharing, this primitive would create a new source and target relation where the latter is a copy of the former excluding (deleting) some source attribute(s) and creating (adding) some target attribute(s). With sharing, primitives can reuse existing relations. If we enable target sharing, then an application of ADL may create a new source relation* `Executive` *and copy it into an existing target relation. In our example, see Figure 2, the existing target* `Person` *is chosen (this relation is part of a VH primitive instance). ADL deletes the source attribute* `Position` *and adds the target attribute* `Id`*. By addition, we mean that no attribute of the source relation* `Executive` *is used to populate this target attribute. Notice that the resulting scenario is a very natural one. Parts of the source relations* `Emp` *and* `Executive` *are both mapped to the target* `Person` *relation while other parts (other attributes) of these source relations are partitioned into a separate relation (in the case of* `Emp`*) or removed (in the case of* `Executive`*).*

To create realistic metadata, a generator could create new primitives that represent combinations of the existing primitives (as done in STBenchmark). However, implementing all possible combinations is infeasible in terms of implementation effort. In addition, it is likely overwhelming for a user to choose from long lists of primitives, making such a generator hard to use. We define the metadata-generation problem using an alternative route under which a user is able to control the amount of source or target **sharing** among invocations of the primitives and among independent mappings.

2.5 User-Defined Primitives

EXAMPLE 12. *Alice wishes to test her solution on mappings that include a pivot. A simple example of this is depicted in Figure 3. There are four separate source attributes*

representing quarters, each containing the sales amount for that quarter. In the target, information about quarters is represented as data (the `Quarter` *attribute). The desired mapping is m_p (omitting the universal quantification).*

m_p : *Sales(Y,1QT,2QT,3QT,4QT)* → *Perf(Y, '1', 1QT), Perf(Y, '2', 2QT), Perf(Y, '3', 3QT), Perf(Y, '4', 4QT)*

Sharing provides great flexibility in creating realistic and complex scenarios. However, a user may sometimes require additional control. For example, a proposed integration technique may be designed to exploit scenarios with a very specific shape. Despite our attempt to be comprehensive in creating primitives, there may be other shapes of interest. In addition, it is often helpful to use real scenarios (for example, the real schemas currently being used to evaluate integration systems) as building blocks in creating integration scenarios. We do not see metadata generation as a replacement for the use of real schemas, real mappings and real data, rather we would like generation to make such real scenarios even more valuable. Hence, we include **user-defined** primitives (UDP) in the definition of the metadata generation problem.

The advantage of supporting UDPs is that we can systematically scale these scenarios (creating new scenarios with many copies of the new primitive) and can combine them with other native primitives or other UDPs. All native primitives and UDPs should be able to share metadata elements.

Example 13. *By creating a new primitive **Pivot**, Alice can now combine this primitive with other primitives to create large scenarios with many pivots and can combine these with vertical or horizontal partitioning. If desired, the UDPs and these native primitives may share metadata (relations). This permits her to understand the impact of pivoting (accuracy and performance) on her integration task.*

In Section 4.2, we will show how some already commonly used real scenarios can be turned into UDPs.

2.6 Generation of Other Metadata

Example 14. *Patricia has designed a new algorithm to rewrite SO tgds into s-t tgds. Her algorithm uses functional dependencies (FDs) to rewrite SO tgds that could not be rewritten without these additional constraints. The algorithm is sound, but not complete. To test her algorithm, she needs to not only be able to generate SO tgds, but also vary the amount of source and target constraints. We have defined the metadata-generation problem to include not only mappings, but also constraints and as a result, she was able to use an early prototype of iBench to generate 12.5 million SO tgds and evaluate how the success rate of her algorithm is affected by the number of constraints [7].*

Primitives permit the controlled generation of specific constraints. In addition a metadata generator should be able to generate arbitrary constraints such as FDs (including keys) and INDs (including foreign-keys). The user must be able to control the type and complexity of constraints (e.g., the number of attributes in key constraints or the number of non-key functional dependencies). Hence, the full list of scenario parameters (28 included in our technical report) also includes parameters for this purpose [6]. In addition, we have scenario parameters that control other parts of the metadata generation process like renaming. By default, primitives (and independent mappings) reuse source names in the target and invent random names (using the name generator of STBenchmark) for target attributes and relations that are not mapped from the source. This can be changed to invoke renaming on some or all mappings. Renaming is implemented as a plug-in, we plan to include more flexible name generators in the future.

In addition, a user may request the generation of data (just source, or both source and target), and transformation code that implements the generated mapping. We now give the complete definition of an integration scenario.

Definition 2. *An* integration scenario *as a tuple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\mathbf{S}}, \Sigma_{\mathbf{T}}, \Sigma, I, J, \mathcal{T})$, where $\mathbf{S}$ and $\mathbf{T}$ are schemas, $\Sigma_{\mathbf{S}}$ and $\Sigma_{\mathbf{T}}$ are source and target constraints, Σ is a mapping between $\mathbf{S}$ and $\mathbf{T}$, I is an instance of $\mathbf{S}$ satisfying $\Sigma_{\mathbf{S}}$, J is an instance of $\mathbf{T}$ satisfying $\Sigma_{\mathbf{T}}$, and $\mathcal{T}$ is a program that implements the mapping Σ.*

A user may request any subset of scenario metadata types. In addition, she may specify whether J should be a solution for I (meaning $(I, J) \models \Sigma$ and $\mathcal{T}(I) = J$) or whether J should be an independent instance of $\mathbf{T}$.

2.7 Complexity and Unsatisfiability

We now formally state the metadata-generation problem and investigate its complexity, showing that it is NP-hard in general. Furthermore, we demonstrate that there exist tasks for which there is no solution. Proofs to the theorems can be found in our technical report [6]. These results have informed our MDgen algorithm presented in Section 3.

Definition 3 (Metadata-Generation Problem). *Given an MD Task Γ, the metadata-generation problem is to produce a solution $\mathcal{M}$ for Γ.*

Theorem 1 (Complexity). *The metadata-generation problem is NP-hard.*

Furthermore, the problem of checking whether a scenario $\mathcal{M}$ is a solution for a task is also NP-hard.

Theorem 2. *Let Γ be a task and $\mathcal{M}$ a scenario. The problem of checking whether $\mathcal{M} \models \Gamma$ is NP-hard.*

It is possible to define tasks for which no solution exists. For instance, consider a task with no target sharing, a target schema of size one, and two CP primitive instances. There is no integration scenario that is a solution for this task, because for a scenario to contain two instances of the CP primitive with no sharing of target relations, the scenario has to contain at least two target relations.

3. THE MDGEN ALGORITHM

We now present our MDGen algorithm that solves the metadata-generation problem for an input task Γ. Since the problem is NP-hard in general, a polynomial-time solution must either be an approximation or incomplete. As discussed in Section 2.7, the user may accidentally specify a task that has no solution. We could deal with this by aborting the generation and outputting the conflicting requirements (to help a user see what needs to be relaxed to avoid conflicts). Or alternatively, we can relax the constraints (e.g., by increasing the relation size) and produce a solution for the relaxed constraints. We have chosen the second approach since we found it more useful in using the generator ourselves, as a relaxation that increases the size of one or more scenario parameters is usually sufficient. We present a greedy algorithm where we never reverse a decision once it has been made. Since our algorithm is best-effort (and checking if a solutions exists is hard), we might relax the constraints even when an exact solution exists.

Algorithm 1 `MDGen` (Γ)

1: Initialize empty Integration Scenario $\mathcal{M}$
{Instantiate Primitives}
2: **for each** $\theta \in \Theta$ **do**
3: numInst = random(min_θ, max_θ)
4: **for** $i \in \{1, \ldots,$ numInst$\}$ **do**
5: InstantiatePrimitive($\theta, \mathcal{M}, \Gamma$)
{Add Independent Mappings/Relations}
6: RandomMappings($\mathcal{M}, \Gamma_\pi$)
{Value Invention}
7: ComplexValueInvention($\mathcal{M}, \Gamma$)
{Generate Additional Schema Constraints}
8: GenRandomConstraints($\mathcal{M}, \Gamma$)
{Rename Schema Elements}
9: ApplyRenamingStrategy($\mathcal{M}, \Gamma$)
{Generate Data}
10: **for each** $S \in \{\mathbf{S}, \mathbf{T}\}$ **do**
11: **for each** $R \in S$ **do**
12: genData($\mathcal{M}, R, \Gamma$)

Algorithm 2 `InstantiatePrimitive` ($\theta, \mathcal{M}, \Gamma$)

1: {Determine Sharing}
2: sourceShare = (random$(0, 100) < \pi_{sourceShare}$)
3: targetShare = (random$(0, 100) < \pi_{targetShare}$)
{Determine Restrictions on Scenario Elements}
4: Req = determinePrimReqs(θ, Γ)
{Generate Source Relations}
5: **for** $i \in \{1, \ldots, Req(\|S\|)\}$ **do**
6: **if** sourceShare = true **then**
7: tries = 0
8: **while** $R \not\models Req \wedge tries++ < MAXTRIES$ **do**
9: R = pickSourceRelation($\mathcal{M}, \Gamma, i$)
10: **end while**
11: **if** $R \not\models Req$ **then**
12: R = genSourceRelation($\mathcal{M}, Req, i$)
13: **else**
14: R = genSourceRelation($\mathcal{M}, Req, i$)
{Generate Target Relations}
15: Repeat Lines 5 to 14 using Target Schema
{Generate Mappings}
16: **for** $i \in \{1, \ldots, Req(\|\Sigma_{ST}\|)\}$ **do**
17: σ = genMapping($\mathcal{M}, Req, i$)
{Generate Transformations and Constraints}
18: genFKs($\mathcal{M}, Req$)
19: genTransformations($\mathcal{M}, Req$)

3.1 Overview

Algorithm 1 (MDGen) takes as input an MD Task Γ. We first instantiate primitives (native and UDPs). For each $\theta \in \Theta$, we uniformly at random create a number of instances (described in more detail in Section 3.2) that lies between min_θ and max_θ (Lines 1 to 2 of Algorithm 1). During this process we keep track of the characteristics of the scenario being generated. We always obey the restrictions on the number of primitive instantiations and primitive parameters, but allow violations of scenario parameters. Thus, we guarantee that a returned solution obeys Γ_Θ and Γ_λ (which is always possible), but not that this solution fulfills Γ_Π. To create independent parts of the schemas and mappings, we use a primitive called random mappings (Line 6). This primitive takes as input the number of source relations, target relations, and mappings to create. These values are set based on Γ_Π minus the number of relations (mappings) that have created been during primitive instantiation. If we have already created larger schemas or more mappings than requested by Γ_Π, then additional metadata is not created (and the solution may violate the respective scenario parameters). We found this solution most satisfying for users. In general, if the relationship between the source/target schemas does not matter for an evaluation, then the user will just use scenario primitives (recall Example 4 where Alberto just needed a large set of random mappings over independent schemas) and the generator will always find a correct solution. However, if the relationship matters and primitives are used, we always satisfy the primitive constraints, and relax, if necessary, scenario restrictions.

Additional scenario parameters are handled within separate post-processing steps which are applied to the scenario (with mappings) that we have created. These include value invention, the generation of additional constraints, and the renaming of target relations and attributes (if requested). For reasons of space, we only explain one exemplary of these postprocessing steps, value invention, in Section 3.3.

We also generate data for the source schema (by calling ToXgene, a general purpose data generator [8]). The user can control the number of tuples generated per relation (via the scenario parameter $\pi_{RelInstSize}$). In the future, we plan to give the user control over which value generators are used to create attribute values (types of data generators and the probability of using them). However, this is not implemented in the current release.

3.2 Instantiate Primitives

As mentioned before, our algorithm is greedy in the sense that once we have created the integration scenario elements for a primitive instance we never remove these elements. In this fashion, we iteratively accumulate the elements of a scenario by instantiating primitives. Algorithm 2 is used to instantiate one primitive of a particular type θ. When $\pi_{SourceShare}$ or $\pi_{TargetShare}$ are not set (their default is 0), each primitive instantiation creates new source and/or target relations. Otherwise, we achieve reuse among mappings by using relations created during a previous primitive instantiation instead of generating new relations. The choice of whether to reuse existing relations when instantiating a primitive is made probabilistically (Lines 2 and 3) where $\pi_{SourceShare}$ (respectively, $\pi_{TargetShare}$) determines the probability of reuse. Once we have determined whether we would like to reuse previously generated schema elements or not, the next step is to determine the requirements Req on scenario elements based on the primitive type θ we want to instantiate, the scenario parameters Γ_Π, and primitive parameters Γ_λ (Line 4). Recall that our algorithm makes a best-effort attempt to fulfill the input task restrictions. To avoid backtracking and to resolve conflicts between primitive requirements and the scenario restrictions our algorithm violates scenario parameters if necessary. For instance, assume the user requests that source relations should have 2 attributes ($\pi_{SourceRelSize}$) and that VP primitives should split a source relation into three fragments ($\pi_{NumPartitions}$). It is impossible to instantiate a VP primitive that fulfills these conditions, because to create three fragments, the source relation has to contain at least three attributes. We define a precedence order of parameters and relax restrictions according to this order until a solution is found. In this example, we would choose to obey the restriction on $\pi_{NumPartitions}$ and violate the restriction on $\pi_{SourceRelSize}$. In Lines 5 to

Algorithm 3 `ComplexValueInvention` $(\mathcal{M}, \Gamma)$

```
1: VI ← attribute → Skolem {Map from attributes to Skolems}
   {Associate Victim Attributes With Skolems}
2: for each i < (getNumAttrs(S) · Π_SkolemSharing) do
3:     S.A = pickRandomAttr(S) {Pick random attribute}
4:     A⃗ = pickRandomAttr(S, π_SkolemType)
5:     f = pickFreshSkolemName()
6:     Add(VI, S.A, f(A⃗))
   {Adapt Mappings}
7: for each σ ∈ Σ_st do
8:     for each (S.A → f(A⃗)) ∈ VI do
9:         if S ∈ σ then
10:            x_A ← vars(σ, S.A)
11:            args ← vars(σ, A⃗)
12:            term ← f(A⃗)[A⃗ ← args]
13:            for each R(x⃗) ∈ RHS(σ) do
14:                R(x⃗) ← R(x⃗)[x_A ← term]
```

14, we generate source relations from scratch or reuse existing source relations. Since not every source relation can be reused (it may not fit the requirements of the primitive) we select relations and check whether they meet the requirements for the current primitive. These source relations are not picked completely at random, but we only choose among relations that fulfill basic requirements of the primitive (e.g., a minimum number of attributes) to increase the chance that sharing is successful. After `MAXTRIES` tries we fall back to generating a source relation from scratch. When generating a relation we also generate primary key constraints if required by the primitive or requested by the user. The generation of target relations (Lines 15) is analogous to the source relation generation. Then, we generate mappings between the generated source and target relations. The structure of the generated mappings is given by the primitive type (see Table 4) and further influenced by certain primitive and scenario parameters. For instance, $\theta_{NumPartitions}$ determines the number of fragments for a VP primitive and $\pi_{SourceRelSize}$ determines the number of attributes per relation. The generation of transformations is similar.

3.3 Value Invention

Part of the value invention process has already been completed while instantiating primitives. Based on the parameter $\pi_{SkolemType}$, we have parameterized value invention in mappings as described in Section 2.2. In addition, we use the scenario property $\pi_{SkolemSharing}$ to control how much additional sharing of Skolem functions we inject into mappings. Algorithm 3 outlines the process. We randomly select attributes from the generated source schema (called victims), and associate them with fresh Skolem terms that depend on other attributes from this source relation (Lines 2-6). For each selected attribute $S.A$, the attributes which are used as the arguments of the Skolem function are chosen based on parameter $\pi_{SkolemType}$, e.g., if $\pi_{SkolemType} = Keys$ then we use the key attributes of relation S. In Lines 7-14, we rewrite any mappings that use a victim to use the generated Skolem term instead. Here $\psi[x \leftarrow y]$ denotes the formula derived from ψ by replacing all occurrences of x with y. For instance, assume the Skolem term $f(a)$ was assigned to attribute b of relation $S(a, b)$. We would transform the mapping $S(x_1, x_2) \rightarrow T(x_1, x_2)$ into $S(x_1, x_2) \rightarrow T(x_1, f(x_1))$. Hence, if S is used in three mappings that exchange b, then the Skolem function f will be shared by these mappings.

3.4 Complexity and Correctness

Aside from solving the MD generation problem, the main goal in the design of MDGen was scalability. MDGen runs in PTime in the size of the generated integration scenario.

Theorem 3 (Complexity). *The MDGen algorithm runs in time polynomial in n, where n is the maximum of the number of created relations and created mappings.*

Since by design the algorithm may violate input scenario parameters, in order to produce a solution without backtracking, a main question is what kind of correctness guarantees can be given for the solution returned by the algorithm under such relaxations of the inputs. If the algorithm does not have to relax any restriction, then the scenario produced by the algorithm is a solution to the input task.

Theorem 4 (Correctness). *Let Γ be an MD Task. MDgen fulfills the following correctness criteria.*

- *If MDgen returns a result without relaxing any restrictions given by the task, then the result is a solution for Γ.*
- *A solution returned by the algorithm always conforms to the primitive restrictions given in the input task.*
- *Let Γ_{relax} be the MD Task generated by replacing the restrictions in the task Γ with their relaxed version produced by the algorithm when running over Γ. The scenario returned by the algorithm is a solution for Γ_{relax}.*

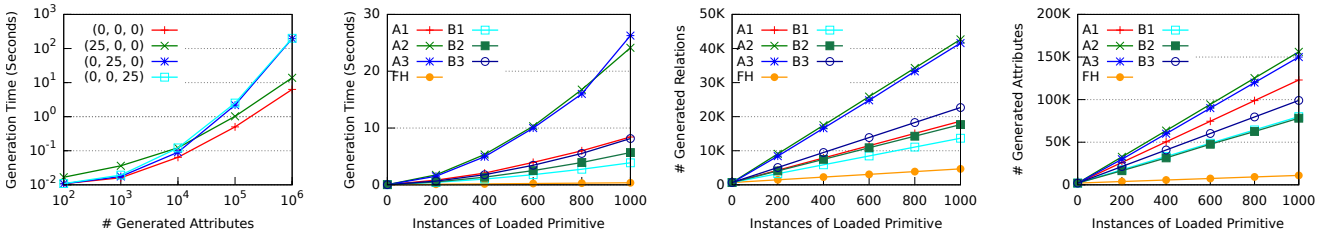
4. iBENCH EVALUATION

We now evaluate the scalability of iBench for various tasks, using native primitives and UDPs. We ran our experiments on an Intel Xeon X3470 with 8×2.93 GHz CPU and 24GB RAM, reporting averages over 100 runs.

4.1 Native Metadata Scenarios

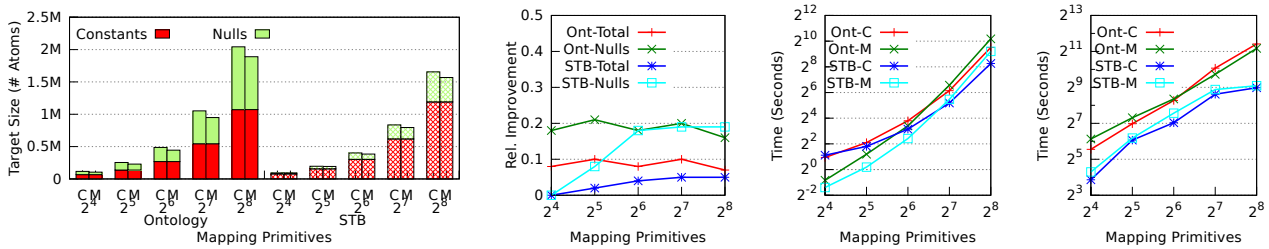
We conducted four experiments to investigate the influence of the schema size on the time needed to generate large metadata scenarios. All four experiments share a baseline configuration, that uses the same parameter settings for the number of attributes per relation (5-15), the same range for the size of keys (1-3), same size for mappings (2-4 source relations and 2-4 target relations). We generated scenarios of various sizes (100 up to 1M attributes where we use the sum of source and target attributes as a measure of schema size) by using the baseline configuration and varying the number of primitives. In these experiments, iBench always produced solutions satisfying all configuration constraints.

Figure 5(a) shows the metadata generation time in seconds for generating four types of scenarios (on logscale). The first configuration denoted as (0,0,0) has no constraints, no source sharing, and no target sharing. The other three configurations are created by introducing 25% FD constraints (25,0,0), 25% source sharing (0,25,0), and 25% target sharing (0,0,25), respectively. iBench can generate scenarios with 1M attributes in 6.3 sec for the (0,0,0) configuration and shows a linear trend as expected. For the 25% constraint configuration, we also observe a linear trend: 13.89 sec for a 1M attribute scenario. For the source and target sharing configurations, we observed a non-linear trend. While iBench generates scenarios with 100K attributes in 2.1 and 2.5 sec, respectively, for 1M attributes, iBench requires several minutes. Here, we noticed high variance in elapsed times: 215.91 sec with a standard deviation of 11.67 sec for source sharing, and 213.89 sec with a standard deviation of 14.14 sec for target sharing. This variance is due



(a) Gen. Time (s) - Native (b) Gen. Time (s) - UDPs (c) Gen. Relations: UDPs (d) Gen. Attributes: UDPs

Figure 5: iBench Scalability Results for Native and UDP Primitives



(a) Ontology+STB: #Mappings (b) Rel. Improv.: 6(a) (c) Gen. Time: 6(a) (d) Exec. Time: 6(a)

Figure 6: Scalability of Clio (C) and MapMerge (M)

to the greedy, best-effort nature of the sharing algorithm. Despite this, we are still able to generate in reasonable time scenarios that are 10-100 times larger, and with much more realistic sharing among mappings, than used in any previous evaluation of which we are aware.

4.2 UDP-based Metadata Scenarios

To analyze the performance of iBench's UDP feature, we used seven real-life metadata scenarios from the literature and we scale them by factors of up to 1,000 times. The first three UDPs are based on the Amalgam Schema and Data Integration Test Suite [20], which describes metadata about bibliographical resources. We denote them by A1, A2, and A3. The next three UDPs are based on a biological metadata scenario called Bio [3], which employs fragments of the Genomics Unified Schema *GUS* (`www.gusdb.org`) and the generic Biological Schema *BioSQL* (`www.biosql.org`). We denote them by B1, B2, and B3. The last UDP (denoted as FH) is based on a relational encoding of a graph data exchange setting [10], comprising information about flights (with intermediate stops) and hotels. For each UDP, we vary the number of instances from 0 to 1,000. In all cases, we also requested 15 instances of each native primitive. We present the generation time in Figure 5(b), and the numbers of generated relations and attributes in Figure 5(c) and 5(d), respectively. As we scale the number of loaded UDPs, the metadata generation time grows linearly in most cases. We observe the worst behavior for A2 and A3, which contain the largest number of relations and up to 40 FKs (Figure 5(c)). While profiling our code, we realized that this non-linear behavior is due to a limitation in the Java-to-XML binding library we used for manipulating relations in the UDP and will be optimized for future iBench releases.

5. INTEGRATION SYSTEM EVALUATION

To showcase how iBench can be used to empirically evaluate schema mapping creation, we present a novel evaluation comparing MapMerge [1] against Clio [12] and ++Spicy [17].

Systems. Clio creates mappings and transformations that produce a universal solution, MapMerge correlates Clio mappings to remove data redundancy, while ++Spicy creates mappings and transformations that attempt to produce core solutions [19]. We obtained Clio and MapMerge from the authors and downloaded ++Spicy from the web (`www.db.unibas.it/projects/spicy`). Since these three systems create mappings, we used iBench to generate schemas, schema constraints, mappings, and instance of the source schemas. We omitted the iBench mappings from the input, so each system created its own mappings and own transformation code (which we ran over the iBench data to create target data that we could compare). We used an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM. Notice that this is a different machine than the rack server used in Section 4 since all three systems we compare required input given through a user-interface. As this approach is more human labor-intensive, here we report averages over 5 runs.

Original MapMerge Evaluation. The original evaluation compared transformations that implement Clio's mappings (input to MapMerge) with transformations that implement correlated mappings produced by MapMerge. The evaluation used two real-life scenarios of up to 14 mappings, and a synthetic scenario (Example 2) with one source relation, up to 272 binary target relations and 256 mappings. It was concluded that MapMerge improved the quality of mappings by both reducing the size of the generated target instance, and increasing the similarity between the source and target instances. We implemented the synthetic scenario used in the original evaluation as an iBench primitive (VA) both to further our goal of having a comprehensive set of primitives and to perform a sanity (reproducibility) check [6]. We obtained the same generated instance size and comparable times to the original evaluation (these measures are described below). Our goal in the new experiments was to test the original experimental observations over more diverse and complex metadata scenarios. In particular, we use iBench to generate scenarios with a variable number of con-

straints and variable degree of sharing to explore how these important factors influence mapping correlation.

Original ++Spicy Evaluation. ++Spicy evaluated the quality of its solutions using four fixed scenarios from the literature with 10 (or fewer) tgds and 12 (or fewer) egds. They compared the size of core solutions that were computed w.r.t. tgds only with the size of core solutions that used both tgds and egds. They also used synthetic scenarios (created using STBenchmark and similar to our STB scenarios described below), to evaluate the scalability of their algorithms. While their scalability experiments control the number of constraints, their quality evaluation did not. Our goal was to quantify the quality gains of ++Spicy as the number of constraints is varied and as the degree of sharing is varied (hence on more complex scenarios). Also we compare ++Spicy directly to MapMerge on the same scenarios, a comparison that has not been done before.

Metadata Scenarios. We designed two scenarios using iBench. The *Ontology* scenarios consist of primitives that can be used to map a relational schema to an ontology. Three of these primitives are different types of vertical partitioning, into a HAS-A, IS-A, or N-to-M relationship (VH, VI, VNM). The fourth primitive is ADD (copies and adds new attributes). The *STB* scenarios consist of primitives supported by STBenchmark [2]: CP (copy a relation), VP (vertical partitioning), HP (horizontal partitioning), and SU (copy a relation and create a surrogate key). The *Ontology* scenarios produce at least twice as many value inventions (referred to hereafter as *nulls*) as the *STB* scenarios (e.g., one instance of each ontology primitive - ADD, VH, VI, and VNM - yields 8 nulls, while one instance of each *STB* primitive - CP, VP, HP, and SU - only yields 4 nulls). The *Ontology* scenarios also include more keys compared to *STB*.

Measures. Intuitively, mappings that produce smaller target instances are more desirable because they produce less incompleteness. To assess the quality of the correlated mappings, we measure the *size* of the generated target instance and the *relative improvement* (of MapMerge or ++Spicy w.r.t. Clio), where the size of an instance is the number of atomic values that it contains [1]. Assuming that the Clio mappings produce an instance of size x and the MapMerge mappings produce an instance of size y, the relative improvement is $(x - y)/x$. The original MapMerge evaluation [1] used this measure and also measured the similarity between source and target instances using the notion of full disjunction [23]. We were not able to use full disjunction in our experiments because we use arbitrary constraints and sharing, and these features often break the gamma-acyclicity precondition required for the implementation of this measure. Instead, we measure the number of nulls in the generated target instance. ++Spicy used a similar measure, however, they measure size in tuples rather than atomic values. We also report the performance in terms of time for generating mappings and time for executing them.

5.1 Scalability of Clio and MapMerge

For the two aforementioned scenarios, we generate an equal number of instances for each primitive for varying powers of two. On the x-axis of Figures 6(a) to 6(b) we report the total number of primitives, e.g., the value 128 for *Ontology* scenarios corresponds to primitives ADD, VH, VI, VNM each occurring 128/4=32 times. We generated additional INDs for 20% of the relations. We used 25% source and target sharing for *Ontology* scenarios. For *STB* scenarios, we also used 25% target sharing, but 50% source sharing to compensate for sharing that naturally occurs (e.g., primitives such as HP generate multiple mappings sharing relations, hence sharing occurs even if 0% sharing is requested). We set source relation size parameter (2-6) and create a source instance with 1,000 tuples per relation.

The target instance sizes are shown in Figure 6(a) - as expected linear in the number of primitives. Clio (C) produces the same number of constants and more nulls than MapMerge (M). The *Ontology* scenarios produce more nulls and larger instances than the *STB* scenarios. Figure 6(b) shows that in general, the relative improvement of MapMerge over Clio is higher for *Ontology* compared to *STB*. An important outcome of our experiments is that the benefits of MapMerge w.r.t. Clio mappings are more visible in scenarios involving more incompleteness. We notice that the relative improvement remains more or less constant (the exception is on small *STB* scenarios, where Clio and MapMerge are almost indistinguishable), because the characteristics of the scenarios are the same for different sizes.

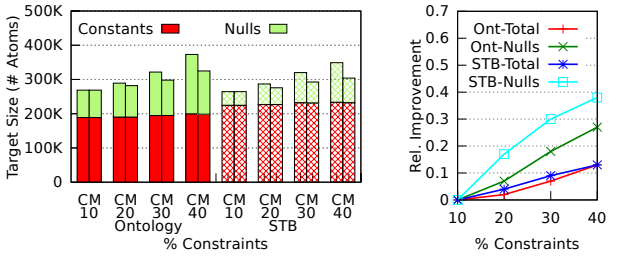
We present the mapping generation and execution time (logscale) in Figure 6(c) and 6(d), respectively. Generating and executing mappings for the *Ontology* scenarios takes longer than for the *STB* scenarios due to the amount of nulls. Although MapMerge requires more time than Clio to generate mappings, it requires less time to execute them. This behavior is more visible for larger number of mappings because in our experiments this implies larger target instances (up to 2M atoms, in contrast to the original MapMerge evaluation that only considered up to 120K atoms).

Our findings extend the original MapMerge evaluation in two ways: (i) they do not report mapping execution time (they have only the generation time), and (ii) MapMerge has greater benefit on the *Ontology* scenarios, which are defined using iBench primitives that are not covered by previous scenario generators nor by the existing MapMerge evaluation. Indeed, the very flexible value invention provided by iBench reveals another strong point about MapMerge, not considered before. MapMerge not only generates smaller instances compared to Clio, but is also more time efficient.

5.2 Impact of Constraints and Sharing

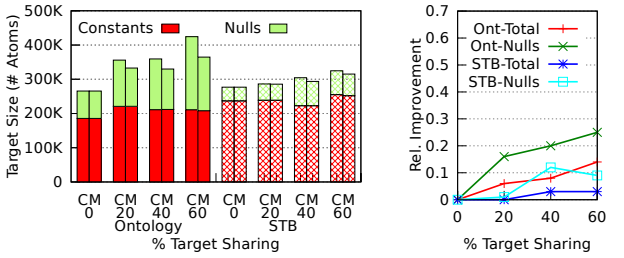
We now study the impact of increasing the number of constraints and amount of sharing for relations with (3-7) attributes and 1,000 tuples. We ran three experiments, and for each scenario, we use 10 instantiations of each of its primitives. First, we vary the number of source and target INDs from 10 to 40%, with no source or target sharing. We present the size of the generated target instance for both scenarios in Figure 7(a) and the relative improvement in Figure 7(b). Here the relative improvement is not constant, but improves as the number of constraints increases. Second, we vary target sharing from 0-60% and no random constraints. Source sharing is fixed to 20% for *Ontology* and to 50% for *STB*. We present the size of the generated target instance for both scenarios in Figure 7(c) and the relative improvement in Figure 7(d). Third, we vary the amount of source sharing from 0-60%. Target sharing is fixed to 20% and there are no random constraints. Target instance sizes are shown in Figure 7(e) and the relative improvement in Figure 7(f).

Our experiments reveal that both sharing and constraints increase the relative improvement of MapMerge over Clio.



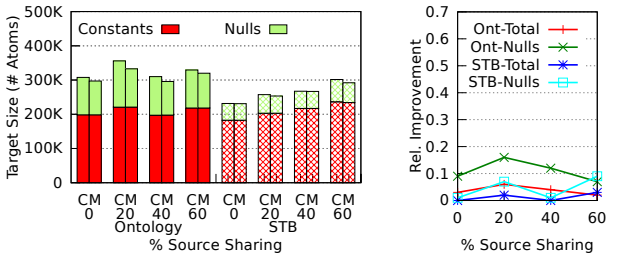
(a) % Constraints

(b) Rel. Improv.: 7(a)



(c) Target Sharing

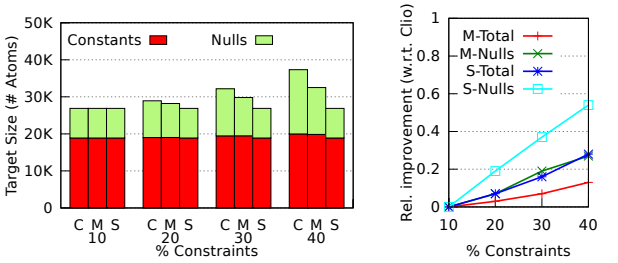
(d) Rel. Improv.: 7(c)



(e) Source Sharing

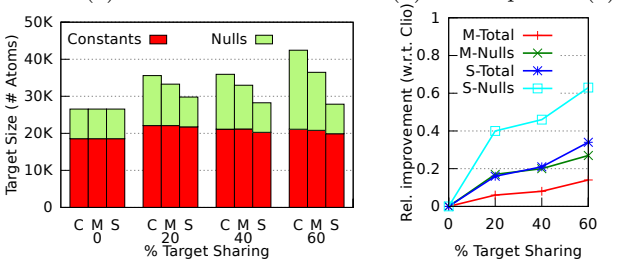
(f) Rel. Improv.: 7(e)

Figure 7: Evaluation of Clio (C) and MapMerge (M)



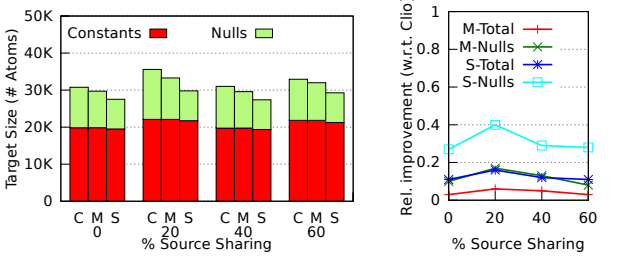
(a) % Constraints

(b) Rel. Improv.: 8(a)



(c) Target Sharing

(d) Rel. Improv.: 8(c)



(e) Source Sharing

(f) Rel. Improv.: 8(e)

Figure 8: Evaluation of ++Spicy (S)

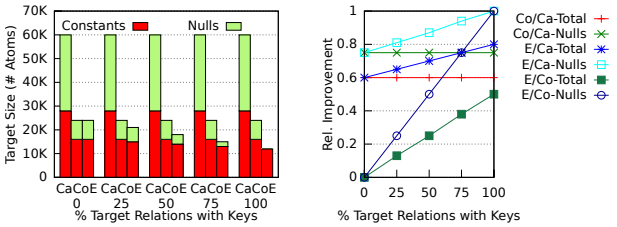
We observe that for *STB*, the biggest gain comes from using constraints, while for the *Ontology* scenarios the biggest gain comes from target sharing. This suggests that the benefits shown in Figure 6(b) (for scenarios combining constraints and sharing) are due to both factors. MapMerge is most useful for mappings that share relations or contain INDs. Our results highlight the impact of MapMerge in novel scenarios, going beyond the original evaluation.

5.3 Comparison with ++Spicy

Despite producing target instances smaller than Clio, MapMerge may not produce *core* solutions. Here, we include a comparison with ++Spicy [18]. More precisely, we present two experiments: (i) we first compare the core solutions produced by ++Spicy with the MapMerge and Clio solutions [19], and (ii) we independently study the impact of using egds in the mapping, an important ++Spicy feature [17]. For (i), we took 10 instances of each *Ontology* primitive, and source relations with (3-7) attributes and 100 tuples. We focus on the *Ontology* scenarios that have both more keys and more nulls, making the comparison with MapMerge clearer. We used ++Spicy in its default configuration i.e., with core rewriting but without egd rewriting. In Figure 8, we report the generated target instance size and the relative improvement w.r.t. Clio for both MapMerge and ++Spicy, for three different settings: we vary the amount of source and target INDs (8(a) and 8(b)), of target sharing (8(c) and 8(d)), and source sharing (8(e) and 8(f)), respectively (variations over the same ranges while keeping the other parameters fixed to the same values as in Section 5.2). The size of the instance generated by ++Spicy remains constant regardless of the number of additional constraints (cf. 8(a)). This is in contrast to both Clio and MapMerge where the size increases. This indicates that the core solutions of ++Spicy are effectively removing redundancy created by chasing over these additional INDs. The relative improvement is always greater for ++Spicy compared to MapMerge (cf. 8(b), 8(d), and 8(f)). These benefits come naturally with a time cost: while generating Clio or MapMerge mappings took up to 8 sec, generating the ++Spicy mappings took up to 105 sec.

5.4 Evaluation of ++Spicy Using UDPs

In our second ++Spicy experiment, we quantified the benefits of using egds in mapping creation. Neither Clio nor MapMerge uses egds, hence we compared the solution given by ++Spicy in the presence of egds with its canonical and core solutions. The advantages of using egds can be seen in many integration scenarios generated by iBench, but to best highlight the gains, we report on one experiment in which we created a specially designed UDP. The UDP contains source address information shred across three relations that needs to be combined into a single target relation. The UDP is easy to create (the UDP code took only a few minutes to write and is 79 lines of XML that we include in our TR). The source has three relations `Person`(`name`, `address`), `Address`(`occupant`, `city`) and `Place`(`occupant`, `zip`), and two



(a) Target Instance Size (b) Rel. Improv.: 9(a)

Figure 9: ++Spicy: Canonical Solution (Ca), Core Solution (Co), Solution in the Presence of EGDs (E)

FKs (from the two `occupant` attributes to `Person.name`). In the target, we have one relation `LivesAt`(`name`, `city`, `zip`). There is a correspondence towards each target attribute from the source attribute having the same name. In the experiment, we invoked the UDP 40 times with source relations containing 100 tuples, and we varied the number of target relations having a primary key from 0 to 100%. We report the generated target size and the relative improvement in Figure 9(a) and 9(b), respectively. The canonical and core solutions have constant sizes. As the number of keys increases, ++Spicy uses them on more and more relations to remove incomplete redundant tuples as expected. Notice that iBench makes the generation of such an experiment very easy. Once the UDP is created, iBench can apply it on data instances of different sizes and characteristics (for ++Spicy this would include reuse of values) and can invoke it an arbitrary number of times with other primitives or with additional (arbitrary) mappings. Despite the fact that the complexity of the mapping generation problem is hard in the presence of UDPs, iBench makes it easy to create and apply them flexibly to quickly develop robust evaluations.

6. CONCLUSION

We presented the first version of iBench, an open-source metadata generator [5]. We used the system to conduct a new comparative evaluation of MapMerge and ++Spicy, systems that had been evaluated individually but not compared directly. We also presented a new evaluation of ++Spicy to quantify the influence of target key constraints on the quality of their solution, an evaluation not done by the authors for lack of appropriate metadata generation tools. Moreover, iBench was an essential tool in a large scale empirical evaluation we have conducted in previous work [7]. Our hope is that iBench will be a catalyst for encouraging more empirical work in data integration and a tool for researchers to use in developing, testing and sharing new quality measures. In the future, we will extend the prototype with new functionality (with community help).

Acknowledgements. We thank the creators of STBenchmark (Alexe, Tan, Velegrakis) for sharing their code with us; the creators of MapMerge (Alexe, Hernández, Popa and Tan) for sharing MapMerge; Mecca and Papotti for helping us to use ++Spicy; and all for verifying our experimental conclusions.

7. REFERENCES

[1] B. Alexe, M. A. Hernández, L. Popa, and W.-C. Tan. MapMerge: Correlating Independent Schema Mappings. *VLDB J.*, 21(2):191–211, 2012.

[2] B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *PVLDB*, 1(1):230–244, 2008.

[3] B. Alexe, B. ten Cate, P. G. Kolaitis, and W.-C. Tan. Designing and Refining Schema Mappings via Data Examples. In *SIGMOD*, pages 133–144, 2011.

[4] M. Arenas, J. Pérez, J. L. Reutter, and C. Riveros. The Language of Plain SO-tgds: Composition, Inversion and Structural Properties. *J. Comput. Syst. Sci.*, 79(6):763–784, 2013.

[5] P. C. Arocena, R. Ciucanu, B. Glavic, and R. J. Miller. Gain Control over your Integration Evaluations. *PVLDB*, 8(12):1960–1963, 2015.

[6] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench Integration Metadata Generator: Extended Version. Technical report, U. of Toronto, 2015. `https://bitbucket.org/ibencher/ibench`.

[7] P. C. Arocena, B. Glavic, and R. J. Miller. Value Invention in Data Exchange. In *SIGMOD*, pages 157–168, 2013.

[8] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: An Extensible Template-based Data Generator for XML. In *WebDB*, pages 49–54, 2002.

[9] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing Mapping Composition. *VLDB J.*, 17(2):333–353, 2008.

[10] I. Boneva, A. Bonifati, and R. Ciucanu. Graph Data Exchange with Target Constraints. In *EDBT/ICDT Workshops–GraphQ*, pages 171–176, 2015.

[11] R. Fagin. Inverting Schema Mappings. *TODS*, 32(4):24, 2007.

[12] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236, 2009.

[13] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[14] R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *TODS*, 30(4):994–1055, 2005.

[15] J. Hammer, M. Stonebraker, and O. Topsakal. THALIA: Test Harness for the Assessment of Legacy Information Integration Approaches. In *ICDE*, pages 485–486, 2005.

[16] M. Lenzerini. Data Integration: a Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[17] B. Marnette, G. Mecca, and P. Papotti. Scalable Data Exchange with Functional Dependencies. *PVLDB*, 3(1):105–116, 2010.

[18] 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.

[19] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings. In *SIGMOD*, pages 655–668, 2009.

[20] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam Schema and Data Integration Test Suite. `www.cs.toronto.edu/~miller/amalgam`, 2001.

[21] D. Patterson. Technical Perspective: For Better or Worse, Benchmarks Shape a Field. *CACM*, 5(7), 2012.

[22] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

[23] A. Rajaraman and J. D. Ullman. Integrating Information by Outerjoins and Full Disjunctions. In *PODS*, pages 238–248, 1996.

[24] B. ten Cate and P. G. Kolaitis. Structural Characterizations of Schema-Mapping Languages. *Commun. ACM*, 53(1):101–110, 2010.

[25] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. *VLDB*, pages 1006–1017, 2005.