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 Lille
radu.ciucanu@inria.fr

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 one 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). We show that iBench can be used to create very large, complex, yet realistic scenarios. Our evaluation of iBench demonstrates that it can efficiently generate large scenarios with different characteristics. We also present an evaluation of two mapping creation systems using iBench and show that the intricate control that iBench provides over metadata scenarios can reveal new and important empirical insights into integration solutions. 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., Amalgam Integration Test Suite [20] the Illinois Semantic Integration Archive, or Thalia [16]) 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.

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. We present the two primary ways such scenarios have been generated and illustrate how they have been used in evaluating different integration tasks.

EXAMPLE 1.1 (PRIMITIVES). *Libraries of **primitives** can be used to create integration scenarios of different shapes and sizes. A set of synthetic* schema evolution *scenarios modeling micro and macro evolution behavior were proposed for evaluating mapping adaptation [24]. A set of finer-grained* schema evolution primitives *were proposed for testing a mapping composition system [7]. Each primitive is a simple integration scenario. Similarly, a set of* mapping primitives *were 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 or discovery. For composition [7], some additional primitives that were trivial for mapping creation (like an* `add-attribute` *primitive) were used because they can make the composition nontrivial (that is, the composition may be a second-order (SO) tuple-generating-dependency (TGD) [12]).*

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 1.2 (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 no allow control over the metadata characteristics (number of mappings, their complexity, etc.) Such control is needed to evaluate the performance of the technique. 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 use of ad hoc scenario generation is widespread in integration research and is necessitated by the lack of a common, shared integration scenario generator.

1.2 Metadata Generator Requirements

To meet the needs of a large variety of integration tasks, we enumerate some basic requirements for a generator.

Generation of Different Types of Metadata. Different integration tasks require different types of inputs and produce different outputs. To test mapping creation systems, the schemas, the schema constraints (and sometimes instances of the schemas) are used as input. The mapping itself can be used as the *gold standard* to evaluate the accuracy of a system. To test mapping composition, sets of pairs of schemas and the mapping between them are used as input (where the target schema of one scenario is the source schema of another). A data exchange system takes a pair of schemas and a mapping as input and produces transformation code creating a target instance from a source instance. A metadata generator should be able to produce integration scenarios containing any of these types of metadata: schemas, correspondences, mappings, transformations, and instance data.

Concise Specification of Scenario Characteristics. Extensive empirical evaluation of the quality and performance of integration tasks requires a metadata generator that can produce integration scenarios with varying parameters. Ideally, the user can provide the generator with a concise specification of characteristics to efficiently produce a set of integration scenarios for evaluating the behavior of her system when varying one (or more) of the input parameters. For example, to test how well a system scales in the instance size, the researcher would use the generator to generate a set of integration scenarios with the same schema and mapping characteristics, but different instance sizes. A system's execution time can be measured over these scenarios. Similarly, to measure how sensitive transformations produced by a data exchange system are to correlations among mappings within a scenario (that is, to mappings that share the same source or target relations), a researcher could use the generator to produce a set of scenarios with different amounts of correlation. To efficiently support this type of usage, the user has to be able to concisely specify the characteristics of the scenarios that should be produced.

Realistic and Diverse Scenarios. Synthetic scenarios produced by a generator should be similar to real-world scenarios. In particular, mappings and schemas should follow patterns that are common in real-world scenarios.

Scale Real-world Scenarios. Real-world integration scenarios are great tools for evaluation. However, real-world scenarios usually are of small size which limits their applicability for empirical studies. A metadata generator should be able to scale real-world scenarios while preserving their characteristics. In addition, the generator should be able to combine scaled real-world scenarios with synthetic scenarios to create scenarios with fine-tuned characteristics.

Support Complex Metadata. The metadata generator should support rich languages for constraints and expressive mapping languages [23], including languages like SO TGDs which are important for integration tasks like composition. The user should be able to easily create scenarios with constraints and mappings in the language of her choice.

1.3 Contributions

- We define the metadata-generation problem and detail important requirements for a flexible, easy-to-use solution to this problem (Section 2).
- We present iBench, an open-source metadata generator that enables the fast, scalable development of integration scenarios (Section 3). The system provides the user with intricate control over the metadata-generation process (through a configuration) and satisfies all the requirements we have enumerated. A user may choose which types of primitives to use, the size of the schemas, the existence and type of schema constraints, the mapping language: ST TGDs (aka GLAV mappings), nested TGDs [13], or SO TGDs [12]. Importantly, the user may also control how interconnected the mappings are (that is, the degree to which mappings share source relations or target relations) and the value invention semantics used within mappings. This control is the main innovation of the generator and distinguishes it from previous mapping generators that use only a few primitives designed for a single integration task and permit very limited or fixed sharing among mappings.
- We present the MDGen algorithm that solves the metadata generation problem for a given configuration (Section 4). The configuration allows a user to specify ranges for parameter values (e.g., the size of relations or degree of sharing among many others). MDGen is a randomized algorithm that invokes primitives satisfying a specification.
- We present an evaluation of the performance of iBench (Section 6) 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 MapMerge [1], comparing it to two other systems Clio [10] and ++Spicy [18] (Section 7). Our evaluation systematically varies the degree of source and target sharing among mappings in the generated scenarios. This reveals new insights into the power (and need for) mapping correlation on complex mappings. As the first generator that varies the amount of generated constraints, we show for the first time how this important parameter influences mapping correlation.

The flexibility of iBench and the focus on providing comprehensive control over properties of the metadata (including schemas, constraints, and mappings) mean that iBench can be used for a wide range of integration and metadata management tasks. iBench permits innovative evaluations of existing methods that reveal new insights into their performance over scenarios with differing degrees of complexity.

2. METADATA GENERATION PROBLEM

We now define the metadata generation problem and illustrate its challenges with examples. Furthermore, we define a set of user requirements on the form, size and characteristics of the metadata to be generated. and define what it means for an integration scenario to satisfy these requirements. The metadata generation problem is the problem of generating a solution (that is, an integration scenario) that complies with the user specification.

2.1 Integration Scenario

An integration scenario is the output of a metadata-generator , i.e., the schemas, constraints (including the mappings), transformations (if requested), and data (if requested).

DEFINITION 2.1 (INTEGRATION SCENARIO). *An integration scenario is a tuple* $\mathbb{S} = (\mathbf{S}, \mathbf{T}, \Sigma, \mathcal{I}, \mathcal{J}, \mathcal{T})$ *where* $\mathbf{S}$ *and* $\mathbf{T}$ *are a source and a target schema,* Σ *are constraints over these schemas (both mappings and integrity constraints on the schemas),* $\mathcal{T}$ *is a transformation that implements the mappings in* Σ*, and* $\mathcal{I}$ *(respectively,* $\mathcal{J}$*) is a source (respectively, target) instance. When non-empty, we require that the transformation is a valid implementation of the mappings* $(\mathcal{I}, \mathcal{T}(\mathcal{I})) \models \Sigma$.

2.2 Primitives

We use mapping primitives as the basic building-block for metadata generation. A mapping primitive is a parameterized integration scenario that represents a common pattern. For instance, vertical partitioning and horizontal partitioning are common patterns that a metadata generator should support. Primitives act as templates that are instantiated based on user input. The generator should allow a user to both determine what primitives to use when generating an integration scenario and also other characteristics of the generated scenario such as the number of attributes per relation, the dictionary used for schema element names, and the constraints on the relations.

Primitives have the advantage that they can be used to create specific targeted micro-benchmarks for testing very specific functionality of an integration solution. They can also be combined together to create larger benchmarks (i.e., to standardize sets of integration scenarios). A user can select which primitives to use to create a benchmark. 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.

EXAMPLE 2.2. *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 two target relations:* `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`*.*

The primitives determine the shape of the schemas, i.e., the relations and the required schema constraints. They also determine attribute correspondences (which source attributes are mapped to which target attributes). The primitives however are parameterized. Example parameters are the number of attributes per relation or per constraint (e.g., for a key or FK), and the number of relations (so VH can decompose into many target relations).

2.3 Sharing Among Primitives

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 primitives.

EXAMPLE 2.3. *Suppose we apply a third primitive,* **copy-add-delete-attribute** *(ADL). Without sharing, this primitive would create a new source relation and new target relation where the target relation is a copy of the source but with one or more source attribute(s) deleted and one or more target attribute created (added). However, with sharing, we can apply primitives to existing source (or target) relations. As an example, if we enable target sharing, then an application of ADL could 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 (a relation that in this example was created by an application of the VH primitive). In our example, 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*

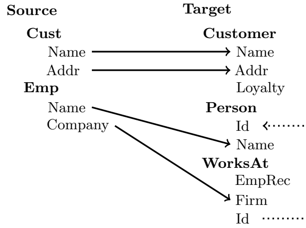


Figure 1: ADD and VH Primitives

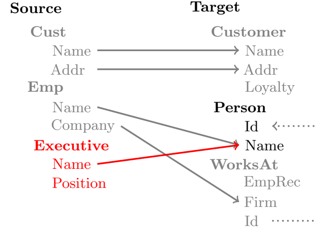


Figure 2: ADL with Target Sharing

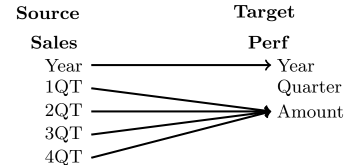


Figure 3: A Pivot UDP

source relations are partitioned into a separate relation (in the case of `Emp`*) or removed (in the case of* `Executive`*).*

To be able to create such realistic metadata, a generator could create new primitives that represent combinations of the existing primitives. 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 (hundreds or thousands), 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.

2.4 User-Defined Primitives

Sharing provides great flexibility in creating realistic and complex scenarios. However, a user may sometimes require additional control. One example of this is when a proposed integration technique is designed to exploit scenarios with a very specific shape (for example, the *ad hoc* scenarios we discussed in the introduction). In addition, it is often helpful to use real scenarios as building blocks in creating integration scenarios. Hence, we include in the metadata problem definition, the use of **user-defined** primitives (UDP). 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 2.4. *Suppose an integration developer wishes to test her solution on mappings that include a pivot. A simple example of this is depicted in Figure 3. In the source, there are four separate attributes representing quarters, each containing the sales amount for that quarter. In the target, information about quarters is represented as data (the value of the* `Quarter` *attribute). The desired mapping is m_p.*

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

*By creating a new primitive **Pivot**, a user can now combine this primitive with other primitives to create large scenarios with many pivots and can combine these with vertical or horizontal partitioning or other native primitives. This permits her to understand the impact of pivoting (the accuracy and performance) on her integration task (for example, on her composition or mapping inversion algorithm).*

2.5 Value Invention

Many integration scenarios are incomplete, i.e., there exist elements in one schema for which there is no corresponding element in another. For tasks like data exchange or mapping composition, we need to model how values are created for these elements. This is a task known as object identification (OID) or *value invention* [17]. Following the literature, we use Skolem functions to formally model how these incomplete or missing values are correlated to known values.

Example 2.5. *Continuing Example 2.2 (Figure 1), this scenario has four attributes for which value invention is required (*`Customer.Loyalty`*,* `Person.Id`*,* `WorksAt.EmpRec`*, and* `WorksAt.Id`*). We can model each with a Skolem function. Primitives may define a default semantics for value invention. For example, the ADD primitive creates a Skolem function whose arguments are all attributes in the source relation. If this is the case, then we can represent the mapping that the ADD primitive creates as:*

$$m_{a_2}: \exists f\ \forall N, A\ Cust(N, A) \rightarrow Customer(N, A, f(N, A))$$

In the metadata generator, we would like each primitive to determine a mapping. To do this, it should provide a semantics for value invention.

Example 2.6. *Consider an invocation of the ADL primitive where the user has requested two attributes deleted and two added. The result could be a source* `S(Id,City,Mgr)` *and target* `T(Name,Region,Type)`*, where* `Id` *is mapped to* `Name`*. There are many possible semantics for value invention. Three examples are depicted below.*

$$m_1 : \exists f, g\ S(Id, City, Mgr) \rightarrow T(Id, f(Id, City, Mgr), g(Id, City, Mgr))$$
$$m_2 : \exists f, g\ S(Id, City, Mgr) \rightarrow T(Id, f(Id), g(Id, City))$$
$$m_3 : \exists f, g\ S(Id, City, Mgr) \rightarrow T(Id, f(City), g(Mgr))$$

Each depicts a different semantics. For m_1, the two invented values depend on the entire source tuple. But for m_3, there is a constraint that if two source tuples have the same `City` *then they will necessarily create target tuple(s) with the same* `Region` *(and a similar constraint for* `Mgr` *and* `Type`*). Of course there are many other possible mappings. The choices are exponential in the number of attributes. A choice is important as it not only defines (possibly different) sets of solutions for data exchange, but it may influence whether the mapping has an inverse or how it can be composed. And each choice is a valid semantics and one a metadata generator should be able to create.*

Value invention poses a challenge for metadata generation. First, we would like to give users flexibility in choosing the value invention semantics. This is particularly important since this semantics can change the power of the language needed to express the generated mappings. For instance, mapping m_1 above could alternatively be expressed as an ST TGD (GLAV mapping) by replacing the Skolem with an existential. However, mapping m_3 is not an ST TGD [12]. Some integration solutions are designed to work only with ST TGDs, so it must be possible to restrict the generated mappings to ST TGDs. Others are designed to work with nested mappings or with SO TGDs, so it is equally important to be able to create mappings that are nested (or SO) and not equivalent to a mapping in a simpler language.

Parameter	Description
$\pi_{relSize}$	Number of attributes per relation.
$\pi_{sourceShare}$	% source relations used in more than one mapping.
$\pi_{joinSize}$	Length of join chains for some mapping primitives
$\pi_{relInstSize}$	Number of tuples per relation instance.
$\pi_{invType}$	Type of value invention (`key`, `all`, `random`).

Table 1: Scenario Parameters (Excerpt)

One option is to implement a different primitive for each of the value invention semantics, but this leads to an exponential blow up in the number of primitives required (exponential in the number of attributes). Hence, we define the metadata generation problem to require that the value invention semantics used by primitives and used within an integration scenario be parameterized. A user should be able to choose among possible value invention semantics and mapping languages.

2.6 Schema Constraints

To accurately model specific transformations, primitives may generate schemas with constraints. An example is VH (Example 2.2). Primitives may create source constraints, target constraints or both. In addition to primitive-created constraints, it is important for a metadata generator to permit a user to specify if there should be additional constraints on the schema and to specify the number and characteristics of these constraints.

EXAMPLE 2.7. *Our recent paper [5], showed that in the presence of source functional dependencies (FD), SO TGDs may be equivalent to first-order mappings. Continuing Example 2.6, if either of the FDs $S.City \rightarrow S.Mgr$ or $S.Mgr \rightarrow S.City$ holds in the source, then mapping m_3 is equivalent to a first-order nested mapping. To evaluate how often SO TGDs are equivalent to first-order mappings in realistic schemas, a metadata generator was needed that could generate large sets of scenarios with varying amounts and types of FDs.*

2.7 Transformations

For integration tasks such as data exchange and instance-based schema matchers, a metadata generator should be able to create executable transformations that implement the generated mappings.

2.8 Name and Data Generation

Integration tasks such as schema matching can be sensitive to both the schema elements names and their correlation. To support such tasks, a metadata generator should provide flexible ways of generating names for schema elements. In addition, many integration solutions (for problems including schema matching and mapping creation) may leverage data (schema instances). An important challenge when generating data is to generate schema instances that satisfy the constraints defined over the schema.

2.9 The Metadata Generation Problem

A user of the generator should have a high level of control over the output but should also be able to generate a set of scenarios with *approximately* the same shape and size. So to define the metadata generation problem, we use configurations where the user is permitted to give ranges for parameters. Note that we distinguish between two types of parameters: *scenario parameters* restrict the shape of the generated integration scenario; *primitive parameters* restrict the usage of primitives, i.e., the number of instances for a primitive type.

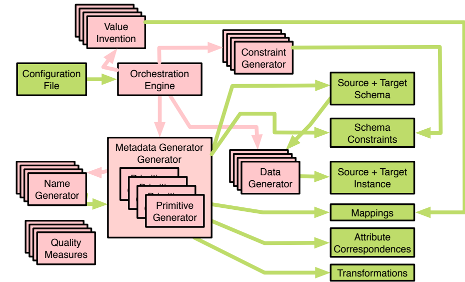


Figure 4: iBench Architecture

DEFINITION 2.8 (INPUT CONFIGURATION). *Let Π and Θ be sets of parameters and primitives, respectively. An input configuration $\Gamma = (\Gamma_\Pi \cup \Gamma_\Theta)$ is a set of scenario restrictions Γ_Π of the form $min_\pi < \pi < max_\pi$ where $\pi \in \Pi$ and Γ_Θ is a set of primitive restrictions of the form $min_\theta < \|\theta\| < max_\theta$ where $\theta \in \Theta$.*

Intuitively, the primitive parameters restrict the usage of primitives when creating an integration scenario $\mathbb{S}$ that conforms with an input configuration. The number of instantiations of a primitive θ used to create $\mathbb{S}$ must be between min_θ and max_θ. We call a scenario constructed in this fashion *compliant* with the primitive parameters Γ_Θ of a configuration. For such a scenario $\mathbb{S}$, we use Ω_Θ to denote the number of instances for each primitive types that were used to create $\mathbb{S}$.

DEFINITION 2.9. *Given a configuration Γ, a primitive instantiation is an assignment $\Omega_\Theta : \Theta \rightarrow \mathbb{N}$ conforming with Γ_Θ, i.e., for each $\theta \in \Theta$ we have $min_\theta \leq \Omega_\Theta(\theta) \leq \max_\theta$. A scenario $\mathbb{S}$ constructed by instantiating primitives according to Ω_Θ is called compliant with Γ_Θ.*

In addition to restricting what primitives can be used to create a scenario, the configuration also allows the user to place restrictions on the general shape of the output scenario using the scenario parameters Γ_Π. A few examples for these parameters are shown in Table 1. To be a solution for a configuration Γ, a scenario $\mathbb{S}$ should not only comply with Γ_Θ, but also fulfill the restrictions on scenario parameters in Γ_Π. Note that there exist configurations for which it is impossible to fulfill both conditions and, thus, no solution exists for such configurations (this is discussed further in Section 4).

DEFINITION 2.10 (METADATA GENERATION PROBLEM). *Let Π be a set of scenario parameters and Θ be a set of primitives. A solution for a configuration Γ is a scenario $\mathbb{S}$ that is complaint with Γ_Θ and that satisfies the scenario parameters: $\mathbb{S} \models \Gamma_\Pi$.*

3. THE iBENCH GENERATOR

In this section, we present iBench[1] as a solution to the metadata generation problem. The details of the scenario generation algorithm will be discussed in Section 4.

[1] http://dblab.cs.toronto.edu/project/iBench/.

Name	Description	Example	Skolems
ADD	Copy a relation and add new attributes	$R(a,b) \rightarrow S(a,b,f(a,b))$	Variable
ADL	Copy a relation, add and delete attributes in tandem	$R(a,b) \rightarrow S(a,f(a))$	Variable
CP	Copy a relation	$R(a,b) \rightarrow S(a,b)$	-
HP	Horizontally partition a relation into multiple relations	$R(a,b) \wedge a = c_1 \rightarrow S_1(b)$ $R(a,b) \wedge a = c_2 \rightarrow S_2(b)$	-
SU	Copy a relation and create a surrogate key	$R(a,b) \rightarrow S(\underline{f(a,b)}, b, g(b))$	Fixed/Variable
VH	Vertical partitioning into a HAS-A relationship	$R(a,b) \rightarrow S(\underline{f(a)}, a) \wedge T(\underline{g(a,b)}, b, f(a))$	Fixed
VI	Vertical partitioning into an IS-A relationship	$R(\underline{a}, b, c) \rightarrow S(\underline{a}, b) \wedge T(\underline{a}, c)$	-
VNM	Vertical partitioning into an N-to-M relationship	$R(a,b) \rightarrow S_1(\underline{f(a)}, a) \wedge M(\underline{f(a)}, \underline{g(b)}) \wedge S_2(\underline{g(b)}, b)$	Fixed
VP	Vertical partitioning	$R(a,b) \rightarrow S_1(\underline{f(a,b)}, a) \wedge S_2(\underline{f(a,b)}, b)$	Variable

Figure 5: Exemplary Native Primitives

3.1 iBench Architecture

The architecture of the system is depicted in Figure 4. The Name and Data Generators are connected as plug-ins to permit easy customization with new generators. Metadata generation is driven by a configuration file defining an input configuration Γ. Our open source repository includes configuration files for: our evaluation (Section 7), an evaluation of a mapping simplification algorithm [5], and for evaluating a provenance-based mapping debugger [15].

3.2 Primitives

In iBench, we use a set of primitives that includes the mapping primitives of STBenchmark [2] and the evolution primitives of Bernstein et al. [7]. We have added a set of additional primitives for tasks beyond evolution and mapping. A partial list of primitives is given in Figure 5 (for a complete list see our web page). Note that the example mappings in the figure represent the simplest version of each primitive. The generated mappings may be much more involved and diverse. For instance, a vertical partitioning may split a relation into more than two fragments (depending on the $\pi_{joinSize}$ configuration parameter) and the number of attributes in a relation may vary ($\pi_{relSize}$).

Examples of primitives are the *ADD*, *VH*, and *ADL* primitives from Section 2 (the first is from Bernstein et al. [7], the second two are new to iBench). iBench includes primitives that model different variants of vertical partitioning such as transforming subclass hierarchies (modeled relationally) or has-a relationships. STBenchmark's version of vertical partitioning corresponds to VP, where the partitioned relations both have an invented key and there is a foreign-key generated between the invented keys (meaning the target relations are in an ISA relationship). Bernstein et al. [7] use a similar definition but without value invention (VI in Figure 5) where the key of the source relation is used as the key of the target relations. Based on the configuration parameter $\pi_{mapLang}$ we either express mappings as ST-TGDs or SO-TGDs. Since not all primitives can be expressed fully as ST-TGDs we have to apply some reasonable simplifications. For instance, the Skolem functions f and g used to generate keys by VH for relations S and T depend on different subsets of the input attributes which cannot be expressed in an ST-TGD. We simply use different existential variables in that case. In the configuration file, a user specifies the number of times each primitive should be instantiated (Γ_Θ introduced in Section 2.9). Primitives instantiation will be discussed in Section 4.

3.3 Sharing among Mappings of Primitives

The degree of *sharing* between mappings directly influences the complexity of the mapping scenario. We allow a user to directly control whether (and to what degree) the mappings will reuse the same source relations (called *source sharing*) and also the degree to which they reuse the same target relations (*target sharing*). For example, to create a scenario with 20% source sharing, intuitively, we could invoke 90% of the requested primitives (without sharing so each creates new metadata). For the final 10% of the primitives, we find source relations with the right shape required by the primitive (for example, a source relation with a key if the primitive requires a key) and reuse these relations, creating new target relations. The result will be that 20% of the mappings share (at least one) source relation with another mapping (details are given in Section 4.1).

3.4 User-Defined Primitives

We support two mechanisms to define a UDP. In the first, a user implements a new primitive type in our (open-source) system. A much easier alternative is using the ability of iBench to load any existing integration scenario from a file. This scenario can then can be used in the same way as other native iBench primitive type. The first option allows for tighter integration with iBench, e.g., the primitive can be parameterized in the same way as native primitives (to create relations of different sizes, etc.). The second option allows for rapid development of new primitives. In particular, if the user has an existing real or synthetic integration scenario from a previous evaluation (for example, an *ad hoc* scenario from Example 1.2) the only effort required is to translate this scenario (its schemas, constraints and mappings) into the XML scenario format supported by iBench. Our current implementation already ships with XML files for the Amalgam Integration Benchmark Scenarios [20] and the biological schemas used to evaluate Eirene [3] .

3.5 Value Invention

Each primitive in iBench has a default value invention semantics. This includes UDPs where the value invention semantics is specified using Skolem functions (or existentials) in the mapping implementing the UDP (read from an XML file). Hence, a developer can create a primitive that encodes a specific value invention semantics required by her integration task. The default semantics for primitives is shown by example in Figure 5. Recall that we desire the ability to generate **any** value invention semantics for primitives so that a new primitive is not required to model each (of the large number of) different value invention semantics for a given scenario. To achieve this, we allow a primitive developer to indicate whether Skolemization is *fixed* or *variable*.

For primitives with variable Skolemization, a user can override the default semantics provided by the primitive using one of three $\pi_{invType}$ settings: *All, Key*, or *Random*. Using the *All* setting, all source attributes are used as arguments for Skolem terms in a primitive. Using the *Key* setting, the arguments are the keys to the source relation(s) used by the primitive. Using *Random*, the arguments are randomly chosen source attributes of the primitive subject to the requirement that any constraints of the primitive (e.g., a foreign key) are satisfied.

EXAMPLE 3.1. *For Example 2.3 (Figure 2), the ADL primitive has a default behavior of Skolemizing by the exchanged variables (*`Executive.Name` *in this example). This could be changed by using Skolem mode* All *(creating m_{adl}).*

$$m_{adl} : \exists k\, \forall n, p\; Executive(n, p) \rightarrow Person(k(p, n), n)$$

The Skolem mode Random *would let the generator randomly pick whether to use n or p (or both) as arguments.*

The *Random* setting is useful for creating second-order mappings and can do so even using a primitive that has by default a first-order value invention semantics.

We also permit a user to take an integration scenario and inject additional **Skolem Noise**. Composition of schema mappings, like schema evolution, can lead to complex interactions between Skolem terms (specifically the arguments of the Skolem functions). In addition to Skolem terms introduced in our primitives, we introduce additional Skolem terms by randomly choosing source attributes to be replaced (in the target) with Skolem functions (see Section 4.2).

3.6 Schema Constraints

iBench supports relational schemas with functional dependencies (including keys) and inclusion dependencies (including foreign-keys). We let the user control the type and complexity of constraints (e.g., the number of attributes in key constraints or the number of non-key functional dependencies). We support multiple configuration parameters that control generation of additional random constraints which are not based on the primitives.

3.7 Transformations

iBench can produce an executable transformation that implements the generated mappings. Currently we support transformations expressed in SQL. iBench comes equipped with tools for loading a mapping scenario into a relational database by creating the schemas, loading the generated source data, and running the transformations. This is useful for integration tasks that require a target schema instance in addition to a source schema instance, e.g., data exchange or instance based matching.

3.8 Name and Data Generation

We encapsulate the name generation in a plugin. Thus, several implementations of this plugin can be used to generate different name distributions. Currently we support a dictionary-based generator. The main challenge in generating a source instance is to fulfill the constraints defined over the schema and comply with input value distributions (set in our configuration file). In our first release of iBench, we use the *ToXGene* generator for XML data [6]. ToXGene supports both the specification of value distributions and data that conforms to referential and key constraints. When generating massive datasets, e.g., for Big Data integration scenarios, parallelization of data generation is crucial. For example, Ghazal et al. [14] discuss how to generate dependent data in parallel with low communication overhead. Our future work will consider how to apply these ideas to integration scenarios with complex mappings and both source and target constraints.

Algorithm 1 `MDGen` (Γ)

```
1: for each θ ∈ Θ do
2:     Ω_Θ(θ) = RANDOM(min_θ, max_θ)
   {Instantiate Primitives}
3: for each θ ∈ Θ do
4:     for i ∈ {1, ..., Ω_Θ(θ)} do
5:         INSTANTIATEPRIMITIVE(θ, 𝕊, Γ_Π)
   {Value Invention}
6: COMPLEXVALUEINVENTION(𝕊, Γ)
   {Generate Additional Schema Constraints}
7: GENRANDOMCONSTRAINTS(𝕊, Γ)
   {Generate Data}
8: for each S ∈ {S, T} do
9:     for each R ∈ S do
10:        GENDATA(𝕊, R, Γ_Π)
   {Serialize Outputs}
11: OUTPUT(𝕊, Ψ)
```

4. THE MDGEN ALGORITHM

We now present our *MDGen* algorithm that solves the metadata generation problem for a configuration Γ. Our algorithm is greedy in that we are never reversing a decision once it has been made. We use a best effort approach for dealing with configurations that have no solution and to compensate for the greediness. We always obey the primitive restrictions, but allow violations of scenario restrictions. Thus, we guarantee that a solution is returned that obeys Γ_Θ, but not that this solution will fullfil Γ_Π.

Algorithm 1 shows the metadata generation algorithm for iBench. The algorithm takes as input a configuration Γ and output specification Ψ. The output specification determines what integration scenario elements should be generated, e.g., should data to be generated or not. We support multiple formats for storing the generated metadata including the XML file format developed for iBench mentioned before as well as formats for particular elements, e.g., storing schemas as XML schema files. As a first step we create a primitive instantiation Ω_Θ. For each $\theta \in \Theta$ we uniformly at random select a value for $\Omega_\Theta(\theta)$ that lies between min_θ and max_θ (lines 1 to 2 of Algorithm 1). The next steps, instantiation of primitives and value invention, are explained in Section 4.1 and 4.2, respectively. We then generate random constraints in addition to the constraints that are generated based on the chosen primitives. If requested by the user in the output specification Ψ we generate data for the source schema. The user can control the number of tuples per generated per relation ($\pi_{relInstSize}$) and value generators used to create attribute values (types of generators and associated probability to use them). Finally, we serialize the generated scenario using the specified output format.

4.1 Instantiate Primitives

Note that 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 fash-

Algorithm 2 `InstantiatePrimitive` $(\theta, \mathbb{S} = (\mathcal{S}, \Sigma, \mathcal{I}), \Gamma_\Pi)$

1: sourceShare = false, targetShare = false
{Determine Sharing}
2: **if** RANDOM$(0, 100) < \pi_{sourceShare}$ **then**
3: sourceShare = true
4: **if** RANDOM$(0, 100) < \pi_{targetShare}$ **then**
5: targetShare = true
{Determine Restrictions on Scenario Elements}
6: Req = DETERMINEPRIMREQS(θ, Γ_Π)
{Generate Source Relations}
7: **if** sourceShare = true **then**
8: **for** $i \in \{1, \ldots, Req(\|T\|)\}$ **do**
9: **if** targetShare = true **then**
10: tries = 0
11: **while** $R \not\models Req \wedge tries + + < MAXTRIES$ **do**
12: R = PICKSOURCERELATION$(\mathbb{S}, \Gamma_\theta, i)$
13: **end while**
14: **if** $R \not\models Req$ **then**
15: R = GENSOURCERELATION$(\mathbb{S}, Req, i)$
16: **else**
17: R = GENSOURCERELATION$(\mathbb{S}, Req, i)$
{Generate Target Relations}
{Generate Mappings}
{Generate Transformations}
18: ...

ion, we iteratively accumulate the elements of a scenario by instantiating primitives. Algorithm 2 is used to instantiate one primitive of a particular type θ. We realize sharing among mapping primitives (as determined by the $\pi_{sourceShare}$ and $\pi_{targetShare}$ parameters) by reusing relations in the schema that we have created during previous calls to Algorithm 2. The choice of whether to reuse existing relations when instantiating a primitive is made probabilistically (lines 2 to 6) where $\pi_{sourceShare}$ respective $\pi_{targetShare}$ 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 and the scenario parameters Γ_Π (line 8). Recall that our algorithm makes a best effort attempt to fulfill the input scenario restrictions. To avoid backtracking and to resolve conflicts between primitive requirements and the scenario restrictions our algorithm violates scenario restrictions if necessary. For instance, assume the user requests that relations should have 2 attributes ($\pi_{relSize}$) and that VP primitives should split a source relation into three fragments ($\pi_{joinSize}$). 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 $\pi_{joinSize}$ and violate the restriction on $\pi_{relSize}$. In lines 8 to 17, we generate source relations from scratch or reuse existing source relations. Since not every source relation can be reused we randomly pick source relations and check whether they meet the requirements for the current primitive. After `MAXTRIES` tries we fall back to generating a source relation from scratch. For reasons of space we do not show the details for target relation generation (which is analogous to source relation generation), mapping generation, and transformation generation.

4.2 Value Invention

Algorithm 3 `ComplexValueInvention` $(\mathbb{S}, \Gamma)$

Input : Integration Scenario $\mathbb{S}$
Output : An Updated Integration Scenario
1: $VI \leftarrow attribute \rightarrow Skolem$ {Map from attributes to Skolems}
{Associate Attributes With Skolems}
2: **for each** $i <$ (GETNUMATTRS$(\mathbf{S}) \cdot \Pi_{complexVI}$) **do**
3: $S.A$ = PICKRANDOMATTR$(\mathbf{S})$ {Pick random attribute}
4: $\vec{A}$ = PICKRANDOMATTR$(S, \pi_{SkolemMode})$
5: f = PICKFRESHSKOLEMNAME()
6: ADD(VI,$S.A$, $f(\vec{A})$)
{Adapt Mappings}
7: **for each** $\sigma \in \Sigma_{st}$ **do**
8: **for each** $(S.A \rightarrow f(\vec{A})) \in VI$ **do**
9: **if** $S \in \sigma$ **then**
10: $x_A \leftarrow vars(\sigma, S.A)$
11: $args \leftarrow vars(\sigma, \vec{A})$
12: $term \leftarrow f(\vec{A})[\vec{A} \leftarrow args]$
13: **for each** $R(\vec{x}) \in RHS(\sigma)$ **do**
14: $R(\vec{x}) \leftarrow R(\vec{x})[x_A \leftarrow term]$

Part of the value invention process has already been completed while instantiating primitives. Based on the parameter $\pi_{invType}$, we have parameterized the value invention in mappings as described in Section 3.5. In addition, we use the scenario property $\pi_{complexVI}$ to control how much additional **Skolem Noise** we inject into the mapping. Algorithm 3 outlines the process of injecting Skolem noise. 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 $\pi_{invType}$, e.g., if $\pi_{invType} = Keys$ then we use the key attributes of relation S. In lines 7-14, we rewrite any mappings that use a victim to now use the generated Skolem term. 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 $S(x_1, x_2) \rightarrow T(x_1, f(x_1))$.

5. RELATED WORK

Our solution builds on foundational work including efforts for creating benchmarks for specific integration tasks, collection and description of real-world integration scenarios, and quality measures for comparing integration solutions.
Scenario Generators. Scenario generators have been used to evaluate a number of different integration tasks. A set of synthetic *schema evolution* scenarios were proposed for evaluating a mapping adaptation system [24], a set of *schema evolution primitives* were proposed to evaluate a mapping composition system [7], and a set of *mapping scenarios* were defined in STBenchmark [2] to evaluate and compare mapping creation systems. STBenchmark raised the issue of enabling schema reuse among primitives to generate more diverse and realistic scenarios but tackles this issue by providing some primitives that combine the behavior of other simpler primitives. No scenario generator has a general solution for sharing and none of these approaches provide for flexible value invention. (To be fair, all of these generators create ST TGDs which support very limited value invention.) We have developed a principled approach for handling sharing and value invention that avoids an explosive

growth of the number of primitives. In addition, these early scenario generators focused primarily on varying the characteristics of the (ST TGD) mappings they produced. iBench is the first scenario generator that supports the generation of plain SO TGDs [4], a much larger class of mappings useful in many integration tasks including composition and inversion. In addition, we let a user control not only the mappings, but also the mapping language, the schemas, and their constraints. The latter is especially important for integration systems that depend on (or benefit from) the presence of certain types of schema constraints (and we exploit this feature in our evaluation of MapMerge and ++ Spicy in Section 7).

Real-world Scenarios and "Fixed" Benchmarks. Several real-world scenarios and synthetic benchmarks have been used to evaluate data integration systems. The Thalia [16] benchmark models a University schema and consists of several challenges including value conversion, language translation, and resolving structural heterogeneity. Another example is the Amalgam [20] benchmark which consists of several schemas and datasets storing bibliographic information. Many other real-world scenarios, e.g., the Illinois Semantic Integration Archive (`http://pages.cs.wisc.edu/~anhai/wisc-si-archive/`) have been published and used in evaluating integration systems. Real-world scenarios provide for the highest level of realism when evaluating an integration system, but only allow very limited evaluations. For instance, it is not possible to vary parameters such as schema and instance size and types of mappings which is necessary for a rigorous experimental evaluation. With iBench we leverage the extensive effort spent on creating these scenarios through UDPs. Any existing mapping scenario can be loaded as a new primitive and, thus, can be scaled in multiple dimensions to support broad evaluations.

Quality Metrics. Different integration tasks have different outputs, thus, unsurprisingly, a number of quality metrics have been proposed to evaluate specific output types. Here we only showcase a few examples to demonstrate that these measures can be quite complex and diverse. For mapping creation or adaptation, we can measure the similarity between instances produced by data exchange. Alexe et al. [1] proposed a metric for measuring the preservation of data correlations in a target instance wrt. a source instance. Recently, Mecca et al. [19] introduced the so-called instance quality metric, which can be used to measure the similarity of a generated output instance with respect to an expected one. A long-term goal for iBench is to get the community to contribute implementations of their metrics to enable future evaluations exploiting these metrics.

6. IBENCH EVALUATION

We now evaluate the scalability of iBench for various input parameters, using native primitives and UDPs.

6.1 Native Metadata Scenarios

We conducted four experiments to investigate the influence of the schema size on the time needed to generate large metadata scenarios. We ran these experiments on an Intel Xeon X3470 with 8×2.93 GHz CPU and 24GB RAM, reporting averages over 100 runs. All four experiments share a baseline configuration, that uses the same number of attributes per relation (10 attributes $\pm$ 5), the same size for keys (2 $\pm$ 1) , and the same length of join paths in mappings (3 $\pm$ 1). We generated scenarios of various sizes (100 up to 1M attributes) by using the baseline configuration and varying the amount of primitives.

Figure 6(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 0% of 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. For the random constraints 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 $\pm$ 1.17 sec with a standard deviation of 11.67 sec for source sharing, and 213.89 $\pm$ 1.30 sec with a standard deviation of 13.09 sec for target sharing. This variance is due 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 that we are aware of.

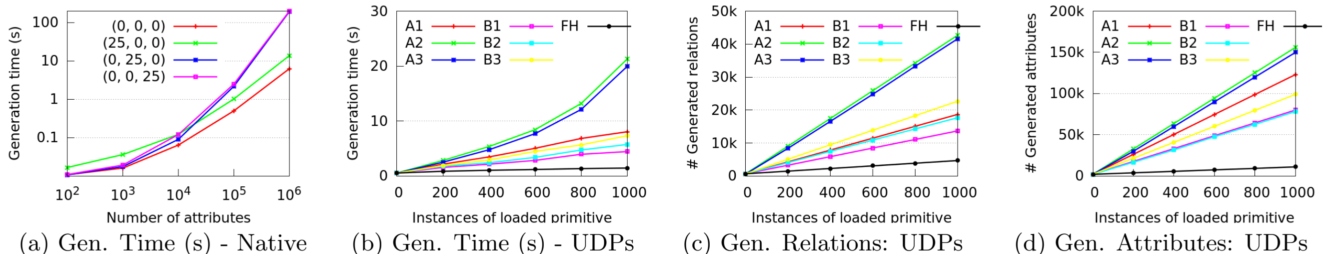
6.2 UDP-based Metadata Scenarios

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. We ran these experiments on an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM. 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 relational 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 [8], 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 6(b), and the numbers of generated relations and attributes in Figure 6(c) and 6(d), respectively. As we scale the number of loaded UDPs, the metadata generation time grows linearly in the majority of cases. We observe the worst behavior for A2 and A3, which contain the largest number of relations (Figure 6(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.

7. INTEGRATION SYSTEM EVALUATION

To showcase how iBench can be used to empirically evaluate an integration task, we present a novel evaluation comparing MapMerge [1] against Clio [10] and ++Spicy [18].

Systems. Clio creates mappings (given correspondences) and transformations that produce a universal solution, MapMerge invokes Clio then correlates mappings to remove data



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

Figure 6: iBench Scalability Results for Native and UDP Primitives.

redundancy, while ++Spicy creates mappings and transformations that attempt to produce core solutions. We obtained Clio including MapMerge from the authors [1] and downloaded ++Spicy from the web.

Original MapMerge Evaluation. The original MapMerge 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 biological scenarios of up to 14 mappings, and a synthetic scenario with one source relation, up to 272 binary target relations and 256 mappings. The synthetic scenario was based on an authority pattern, i.e., a special case of vertical partitioning, where data over a denormalized source schema is transformed into a target schema containing several hierarchies. Each hierarchy has at its root an authority relation and the other relations refer to the authority through FKs. 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. Our goal was to test these observations over more diverse and complex metadata scenarios. In particular, we use iBench to generate scenarios with random constraints and sharing to explore how these parameters influence mapping correlation.

Metadata Scenarios. We designed two scenarios using iBench. The *Ontology* scenarios consists 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 hereafter as *nulls*) as the *STB* scenarios (e.g., one instance of each ontology primitive - ADD, VH, VI, and VNM - yields 8 nulls or more, while one instance of each *STB* primitive - CP, VP, HP, and SU - only yields 4 nulls). An important outcome of our experiments is that the benefits of MapMerge w.r.t. Clio mappings are more visible in scenarios involving more nulls.

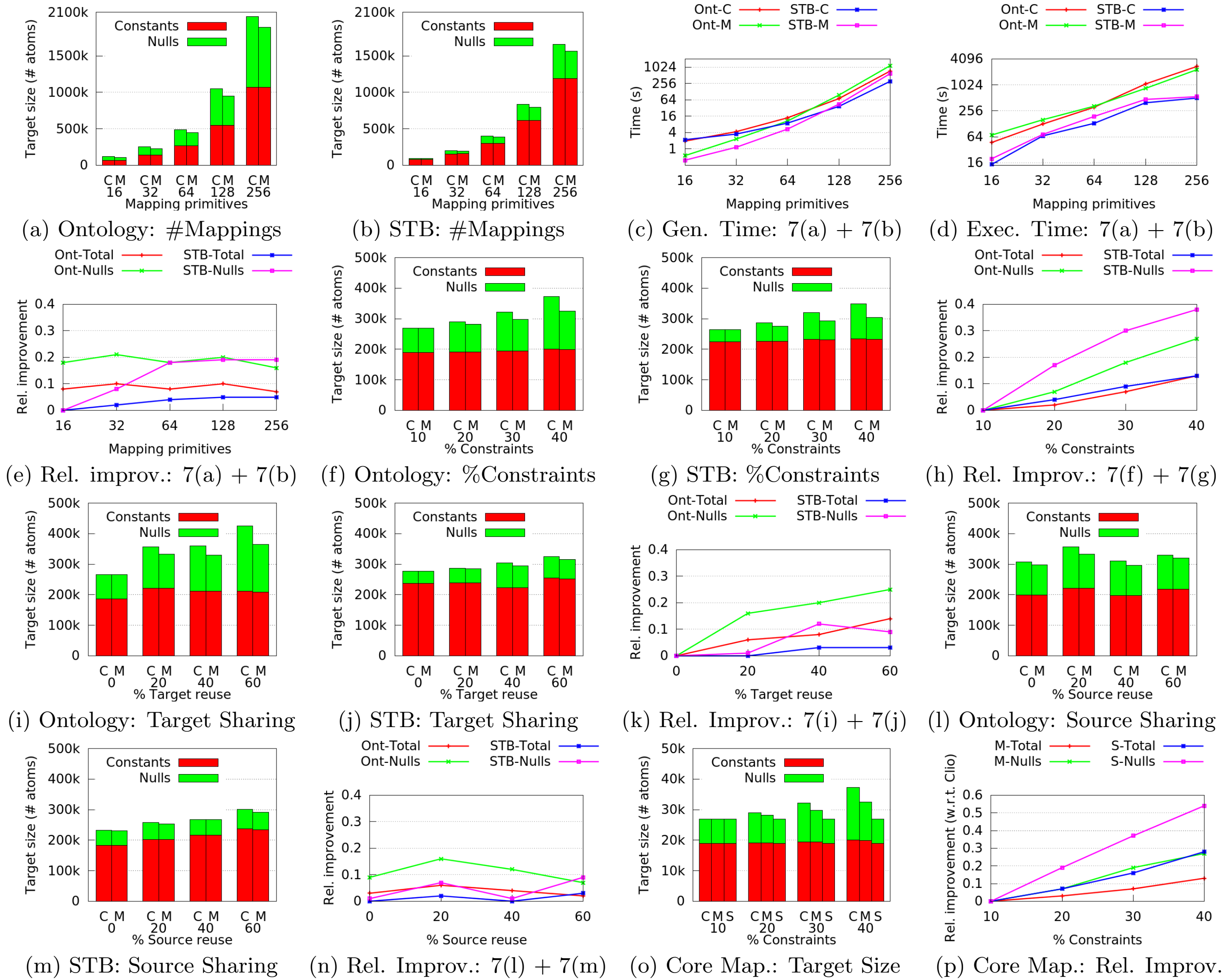
Measures. Intuitively, mappings that produce smaller target instances are more desirable because they produce less incompleteness. To assess the quality of the correlated mappings, we measure the *size* of the generated target instance and the *relative improvement* (of MapMerge w.r.t. Clio), where the size of an instance is the number of atomic values that it contains [1]. The original MapMerge evaluation [1] used this measure and also measured the similarity between source and target instances using the notion of full disjunction [22]. 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 [9] precondition required for the implementation of this measure. Instead, we measure the number of nulls in the generated target instance. We also report the performance of Clio and MapMerge in terms of time for generating and executing mappings using the same machine as in Section 6.2. We implemented the authority scenario used in the original MapMerge evaluation as an iBench primitive to perform a sanity check, and obtained the same generated instance size and comparable times to the original evaluation.

7.1 Scalability of Clio and MapMerge

For the two aforementioned scenarios, we generate an equal number of occurrences for each primitive and vary this number according to consecutive powers of two. On the x-axis of Figures 7(a) to 7(e) 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 random inclusion dependencies 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 horizontal partitioning generate multiple mappings sharing relations, hence sharing occurs even if 0% sharing is requested). We require each source relation to have 4 ± 2 attributes and 1,000 tuples.

The target instance size for the studied scenarios is shown in Figure 7(a) and 7(b) - 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 instances of bigger size than the *STB* scenarios. Figure 7(e) shows that in general, the relative improvement of MapMerge over Clio is higher for *Ontology* compared to *STB*. Additionally, 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 7(c) and 7(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



(a) Ontology: #Mappings (b) STB: #Mappings (c) Gen. Time: 7(a) + 7(b) (d) Exec. Time: 7(a) + 7(b)

(e) Rel. improv.: 7(a) + 7(b) (f) Ontology: %Constraints (g) STB: %Constraints (h) Rel. Improv.: 7(f) + 7(g)

(i) Ontology: Target Sharing (j) STB: Target Sharing (k) Rel. Improv.: 7(i) + 7(j) (l) Ontology: Source Sharing

(m) STB: Source Sharing (n) Rel. Improv.: 7(l) + 7(m) (o) Core Map.: Target Size (p) Core Map.: Rel. Improv.

Figure 7: Evaluation of Clio (C), MapMerge (M), and ++Spicy(S)

(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 for two reasons: (i) they do not report the mappings execution time (they have only the generation time), pointing to a benefit of having a metadata generator that includes not only mappings but transformations, and (ii) the behavior that we report is exposed in 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.

7.2 Impact of Random Constraints and Reuse

We now study the impact of random constraints and sharing for relations with 5 ± 2 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 amount of random inclusion dependencies 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(f) and 7(g), respectively, and the relative improvement in Figure 7(h). Here the relative improvement is not constant but improves as the number of constraints increases.

Second, we vary target sharing by varying the amount 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(i) and 7(j), respectively, and the relative improvement in Figure 7(k).

Third, we vary the amount of sharing source relations from 0-60%. Target sharing is fixed to 20% and there are no random constraints. Target instance sizes are shown in Figure 7(l) and 7(m), respectively, and the relative improvement in Figure 7(n).

Our experiments reveal that both sharing and target constraints increase the relative improvement of MapMerge over Clio. We observe that for *STB*, the biggest gain comes from using constraints, while for *Ontology* scenarios the biggest gain comes from sharing. This suggests that the benefits shown in Figure 7(e) (for scenarios combining constraints

and sharing) are coming from both factors. MapMerge is most useful for mappings that share relations or contain target inclusion dependencies, which aligns with the intuition of correlating mappings. Our results highlight the impact of MapMerge in novel scenarios, going beyond the original evaluation.

7.3 Impact of Core Mappings

Despite producing target instances smaller than Clio, MapMerge has no guarantee to produce *core* solutions [11]. Here, we include a brief comparison with ++Spicy [18] that exploits FDs and rewrites mappings to produce core solutions. We took 10 instances of each *Ontology* primitive, no sharing, and source relations with 5 ± 2 attributes and 100 tuples. We vary the amount of random constraints from 10 to 40%. We present the generated target instance size in Figure 7(o) and the relative improvement with respect to Clio for both MapMerge and ++Spicy in Figure 7(p). We observe that the size of the instance generated by ++Spicy remains constant regardless of the random constraints. Therefore, its relative improvement is greater than for MapMerge. These benefits come naturally with a time cost: while generating Clio or MapMerge mappings took up to 8 seconds, generating the ++Spicy mappings took up to 25 seconds.

8. CONCLUSION

Developing a system like iBench is an ambitious goal. We presented the first version of iBench. We use the system to conduct a new evaluation of MapMerge that reveals new insights into its performance and considered how MapMerge compares with other systems including Clio and ++Spicy. In addition, iBench was an essential tool in a large scale empirical evaluation we have conducted in previous work [5]. 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 and testing new quality measures. In the future, we will extend the prototype with new functionality (with help from the community). Although not part of our initial release of iBench, we would also like to be able to generate queries expressed over a generated source or target schema. Our goal is to support the evaluation of virtual data integration solutions that translate target queries into source queries, or alternatively to evaluate what is lost in data exchange when a source query is translated and evaluated on an exchanged target instance.

9. ACKNOWLEDGEMENTS

We thank Bogdan Alexe and Lucian Popa for sharing the MapMerge code and verifying our experimental conclusions.

10. 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. Kolaitis, and W. 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, B. Glavic, and R. J. Miller. Value Invention in Data Exchange. In *SIGMOD*, pages 157–168, 2013.

[6] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. Toxgene: An extensible template-based data generator for XML. In *WebDB*, pages 49–54, 2002.

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

[8] I. Boneva, A. Bonifati, and R. Ciucanu. Graph data exchange with target constraints. In *EDBT/ICDT Workshops–GraphQ*, pages 171–176, 2015.

[9] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[10] 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.

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

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

[13] A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.

[14] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD*, pages 1197–1208, 2013.

[15] B. Glavic, J. Du, R. J. Miller, G. Alonso, and L. M. Haas. Debugging Data Exchange with Vagabond. *PVLDB*, 4(12):1383–1386, 2011. System Demo.

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

[17] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*, pages 455–468, 1990.

[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, S. Raunich, and D. Santoro. What is the IQ of your Data Transformation System? In *CIKM*, pages 872–881. ACM, 2012.

[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] A. Rajaraman and J. D. Ullman. Integrating information by outerjoins and full disjunctions. In *PODS*, pages 238–248, 1996.

[23] B. ten Cate and P. G. Kolaitis. Structural characterizations of schema-mapping languages. *Commun. ACM*, 53(1):101–110, 2010.

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