Table2Graph: A Scalable Graph Construction From Relational Tables using Map-Reduce

Sangkeun Lee, Byung H. Park, Seung-Hwan Lim, and Mallikarjun Shankar
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
Email: {lees4, parkbh, lims1, shankarm}@ornl.gov

***Abstract*—Identifying correlations and relationships between entities within and across different data sets (or databases) is of great importance in many domains. The data warehouse-based integration, which has been most widely practiced, is found to be inadequate to achieve such a goal. Instead we explored an alternate solution that turns multiple disparate data sources into a single heterogeneous graph model so that matching between entities across different source data would be expedited by examining their linkages in the graph. We found, however, while a graph-based model provides outstanding capabilities for this purposes, construction of one such model from relational source databases were time consuming and primarily left to ad hoc proprietary scripts. This led us to develop a reconfigurable and reusable graph construction tool that is designed to work at scale. In this paper, we introduce *Table2Graph*, the graph construction tool based on Map-Reduce framework over Hadoop. We also discuss results from applying *Table2Graph* to integrate disparate healthcare databases.**

I. Introduction

Data integration involves collocating data in different sources and structures and coalescing them into a unified model. Traditionally this requires a design of a common extract, transform, and load (ETL) process that, when applied separately to each disparate data source, produces data conforming to the unified database model in a data warehouse. The ETL process is repeatedly performed to incorporate the changes of the source data into the unified model.

The data warehouse-based solution has, among many others, an inherent drawback. Once the schemata of a unified model are established, it is realistically difficult to modify the design limiting analytics within the semantic structures of the model. Also a minor change in a source database schema may require a reimplementation of the ETL process due to its tightly coupled relation with the source schema. Furthermore, as the volume of each data source explodes, the ETL process needs to be executed more frequently, which is found to be not scalable.

Identifying correlations between the entities within or across disparate data sources (or databases) is found to be of great strategic importance in many areas. One example is the Section 6402(a) of the Patient Protection and Affordable Care Act (PPACA) of 2010. The statue seeks to protect the integrity of federal and State healthcare programs by identifying potential bad actors across multiple programs. For this goal, the Section 6402(a) mandates the Integrated Data Repository shall include, at a minimum, data from Medicare, Medicaid, State Childrens Health Insurance Programs (as known as CHIP), and Social Security for matching.

In collaboration with the Center for Medicare and Medicaid Services (CMS), Oak Ridge National Laboratory (ORNL) explored innovative methods and approaches to address data integration challenges of a Section 6402(a) implementation. Considering the aforementioned issues of the data warehouse-based solution and the primary requirement, i.e. *matching*, we decided to develop a model of flexible schema that also does not lose any source data information through an ETL process. In particular, we selected a heterogeneous graph-based model [1], which through semantic linking (not combining) integrates multiple data sources in terms of source features (represented as nodes and properties) and relationships (represented as edges) so that relationships between actors are represented as linkages over features in the graph. Unlike traditional relational models, a graph-based model directly embeds linkages in their natural forms, thus can handle relation-related queries very efficiently that would otherwise require multiple deep join operations in a normalized relational model.

Nevertheless an ETL process to construct a graph model from normalized relational models is still not a simple task even for constructing a graph-based consolidated model. First the consolidate graph model may include billions of nodes and edges. Data sources may also include billions of rows and hundreds of columns across multiple relational tables. Thus a scalable ETL solution should be developed so that rapid increase in source data would be incorporated into the consolidated model with negligible delay. Second the consolidate model may change its design or source databases update their schemata irregularly. Thus a reconfigurable ETL process should be developed for reuse without much modification. While exploring various options for these needs, we found that the most common practice was to write ad-hoc in-house scripts with no systematic tools available even for a single data source. For this reason, we designed a scalable and reconfigurable ETL tool named *Table2Graph*. With this tool, data practitioners can designate necessary mappings between data elements in source databases and a graph model, which will be then automatically converted into a MapReduce code for instantiating the graph. Such a user configuration is stored in an XML file and any future change can be applied directly to the file.

The rest of the paper is organized as follows. In section II, we give brief overviews on NoSQL databases and graph conversion processes in general. Section III describes *Table2Graph* in detail. Section IV reports an empirical study on performance of *Table2Graph*. Finally, section V concludes the paper with discussion on future directions.

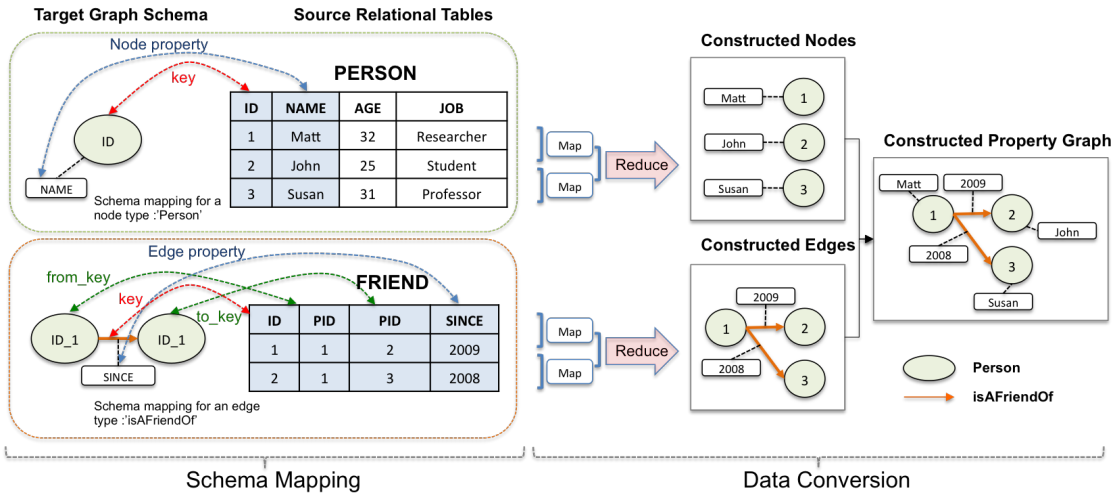


Fig. 1. An Overview of Two Main Steps, Schema Mapping and Data Conversion, in Table2Graph

II. Related Work

In this section, we describe our decision to adopt the graph database. In particular, we give a brief overview on emerging database technologies and compare their pros and cons with respect to our mission. Also related works to *Table2Graph* are discussed.

A. NoSQL Database Technologies

For our purpose of data integration - matching medical service providers, as illustrated briefly in the previous section, a relational data model was found to be inadequate mainly due to its ineptitude in performing relation-related queries, inflexible schemata, and the scalability issue. In search for an alternate data model, we investigated various emerging NoSQL databases [2] such as column-oriented stores [3], [4], [5], [6], document-oriented databases [7], [8], and graph databases [9], [10], [11].

A column-oriented database helps speed up calculations by taking advantage of the observation that repeated elements in a column can be summarized. Additionally, these databases offer the opportunity for advanced data compression. However, built on the relational model, it is not much flexible to adapt and change data schemata. Furthermore relationship searches still rely on the traditional multiple inter-table join operations to extract indirect connections between data sets. This constraint adds complexity to the column-store architecture's ability to store and explore indirect relationships that may be several tables removed.

As opposed to rigid *schema* of a relational database, a document-oriented database operates around a less restricted notion of *document* that can be viewed as flexible schema. Hence, this model is suitable for creating non-rigid data models. Emphasizing storage reliability and scale, some document-oriented databases are also built on MapReduce [12] on Hadoop [13]. We found a document-oriented database a good candidate to integrate disparate data sources, for it is flexible and scalable. However, matching entities are conceptually the same as in the relational models.

A graph database is designed to support graph-like queries such as computation of the shortest paths between nodes or detection of communities in the graph. It does not impose the rigid schema requirement, instead stores data and relationships in a schema-free manner. It is inherently faster in modeling and identifying association between data sets. Since it does not require expensive join operations and can be instantiated on distributed data framework such as Hadoop, it naturally scales. The schema-free property allows convenient insertion of new data types or entities into the model by simply introducing new nodes and edges. We found a graph database to be most suitable for data integration and subsequent analysis for our purposes.

B. Relational to Graph Data Conversion

There are several existing approaches to converting relational data into graph data. Fong et al. [14] and Fernandez et al. [15] introduced approaches to converting relational data into semi-structured XML documents. We can consider that these approaches deal with a specific type of graph, as XML documents are rooted directed connected graphs [16]. In the domain of Semantic Web research, with the emergence of Linked Open Data [17], there have been many works and tools which aim to make relational data available as RDF (Resource Description Framework) representation. RDF is a standard language for data interchange in World Wide Web, and it is based on idea of expressing resources as RDF *triples*, where each *triple* consists of a subject, a predicate, and an object. A RDF dataset is a *graph* in that each triple in the dataset is be represented by a node for the subject, an edge for the predicate, and a node for the object. R2O [18] is a language for mapping relational database schema and ontologies implemented in RDFS or OWL. D2RQ [19] aims to directly make non-RDF relational databases available as virtual RDF graphs. Virtuoso RDF Views [20] is another declarative language for defining the mapping of SQL data to pre-existing RDF vocabularies and aim to transform the result set of a SQL SELECT query into a set of triples. Triplify [21] translates resulting relations from HTTP-URI requests RDF statements. In many of those existing researches, each tuple in the relational table is converted into a node and each foreign key-primary key link is transformed

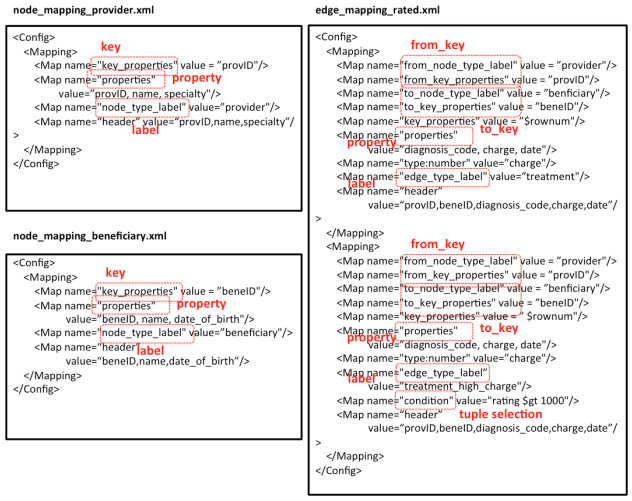
into a directed edge between the corresponding nodes. Due to the simplicity of the approach, constructed graph by the approach cannot capture all semantics implied in the relational database; for example, this approach will not create any edges with properties. Note that there can be various relationships with properties in real world applications. Moreover, although RDF is acknowledged as a generic data model, too much of its simplicity often leads to complex graph structure with large number of nodes and edges.

Recently, Virgilio et al. [22] presented an approach to converting relational database to graph database that uses the *property graph*. Property graph is notable because it is expressive to cover many real world applications and exploited in the state-of-the-art general purpose graph databases such as *Neo4J*[9], *Titan*[10] or *DEX*[11]. The authors focus on speeding up processing queries over the constructed graph by building graph structure based on joinable tuple aggregations. We notice that the approach has two main limitations. First, the authors only considered simplified version of a property graph where only nodes have properties, while edges have just labels that represent relationships between data in nodes. Second, it always deterministically converts one source relational database to one specific structure of graph database; however, in the real world, depending on different analysis or needed queries, graph structure that we want to create can vary even with the same source relational database.

In this paper, we also adopt the *property graph* model like [22], but we allow both nodes and edges to have a set of key-value pairs, called properties. [23] showed an effort to take advantage of Map-Reduce Framework and Hadoop for scalable graph construction. However, in this approach, users are required to write application-specific parsers and routines in Map function, which is a burden to database administrators. With the Table2Graph system, instead of users' programming efforts, it only requires user's efforts to write mappings between source and target databases, which can be done without having much knowledge about complex programming or query languages.

III. Table2Graph: System Design

This section describes the design of *Table2Graph*. Among many other variations of graph models, we chose the property graph model, for it can retain a rich set of information about a node or edge in terms of properties. In fact, this model is adopted by many state-of-the-art general purpose graph databases such as *Neo4J* [9], *DEX* [11] and *Titan* [10]. A property graph is a directed multigraph $G = (\mathcal{V}, \mathcal{E}, T_{\mathcal{V}}, T_{\mathcal{E}})$, where $\mathcal{V}$ is a finite set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a finite multi-set of edges, $T_{\mathcal{V}}$ is a finite set of node types, and $T_{\mathcal{E}}$ is a finite set of edge types. Each node is mapped to a node type by a mapping function $\phi_{\mathcal{V}} : \mathcal{V} \rightarrow T_{\mathcal{V}}$, and each edge is mapped to an edge type by another mapping function $\phi_{\mathcal{E}} : \mathcal{E} \rightarrow T_{\mathcal{E}}$. A node $v_i \in \mathcal{V}$ or an edge $e_k(v_i,v_j) \in \mathcal{E}$ has a set of <*attribute, value*> pairs which constitute the properties. Then, *schema*, a node and edge type-level template, of a property graph G is defined as a directed graph G_S=(T_V, T_E), where T_V is a finite set of node types, and $T_E \subseteq T_V \times T_V$ is a finite set of edge types. In analogy to a schema of the relational model, a schema of the graph model defines the types and properties following which a graph is instantiated.



(a) An Example of Schema Mapping Documents

nodes.json

```
{'_type':'provider', '_id':'provider://1', 'provID':'1', 'name':'Matt', 'specialty':'Radiology'}
{'_type':'provider', '_id':' provider://2', 'provID':'2', 'name':'John', 'specialty':'Cardiology'}
{'_type':'beneficiary', '_id':'beneficiary://1', 'beneID':'1', 'name':'Jack',
'date_of_birth':'1983-10-31'}
{'_type':' beneficiary', '_id':' beneficiary://2', 'beneID':'2', 'name':'Henry',
'date_of_birth':'1981-02-28'}
```

edges.json

```
{'_type':'treatment', '_from_id':'provider://1', '_to_id':' beneficiary://1', 'diagnosis_code':'580.1', ,
'charge':250, 'date':'2012-09-16'}
{'_type':'treatment', '_from_id':'provider://1', '_to_id':' beneficiary://2', 'diagnosis_code':'759.9',
charge:1200, 'date':'2012-10-31'}
{'_type':'treatment', '_from_id':'provider://2', '_to_id':' beneficiary://1', 'diagnosis_code':'139.8',
charge:120, 'date':'2013-10-31'}
{'_type':'treatment', '_from_id':'provider://1', '_to_id':'beneficiary://2', 'diagnosis_code':'759.9',
charge:1200, 'date':'2012-10-31'}
```

(b) An Example of Constructed Property Graph in JSON Format

Fig. 2. (a) shows simplified examples of node and edge mapping *XML* documents used in the *Table2Graph* system, and (b) shows an example of property graph constructed using the mappings, represented in *JSON* format.

As depicted in Fig. 1, *Table2Graph* operates in two steps that are *Schema Mapping* and *Data Conversion*. In the former, users create mapping from source relational table schema to the target property graph schema, and the latter transforms tuples of the source tables into nodes and edges of a graph as specified in the mappings.

A. Schema Mapping

The *Schema Mapping* step is designed to allow flexible configurations and edits. With this capability, a data practitioner may create a mapping for a graph model and reuse the same mapping for another model with minimal manual work. Mappings from a data source to the target model is stored in an *XML* document (refer to Fig. 2(a)). A mapping is done separately for nodes and edges. For a node, this involves identifying an attribute (or a set of attributes) of the source schema to be converted as a target node, other attributes that are included as properties in the target node, and a label that annotates the node. We particularly denote those attributes and label as **key**, **property**, and **label**. Similarly, an edge schema mapping includes **key**, **property**, and **label**. In addition, a pair of **from_key** and **to_key** is also included to designate the edge direction, that is **key** of the connected nodes (start and destination). Also both node and edge mappings can be

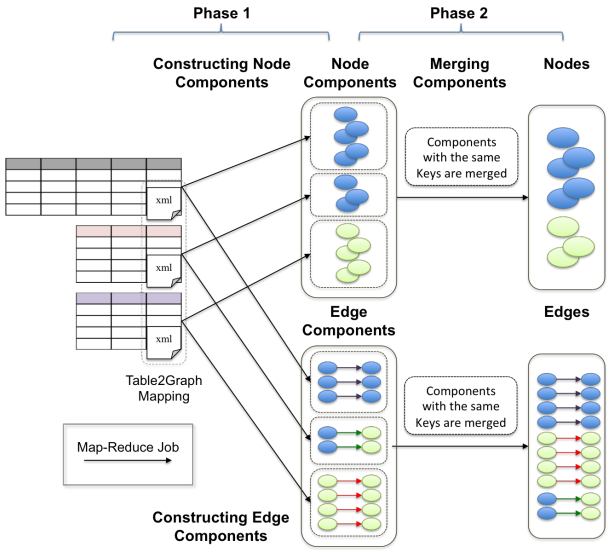


Fig. 3. The Map-Reduce workflow for data conversion in Table2Graph

constrained to limit the inflow of source data into the target model.

B. Data Conversion

The *Data Conversion* step executes a number of Map-Reduce jobs over two phases. During the first phase, multitudes of Map-Reduce jobs collectively create temporary nodes and edges, which we will call node and edge components. A uniform resource identifier (URI) is used to uniquely identify a node or an edge during this phase. Subsequently, during the second phase, these node or edge components of the same *URI* are merged to produce unique nodes and edges. Fig. 4 shows a detailed Map-Reduce workflow of *Data Conversion*.

A Map-Reduce job in *Data Conversion* is classified to one of the following four job types:

- Node component constructor, $\mathbb{N}$
- Edge component constructor, $\mathbb{E}$
- Node set merger, $\mathbb{M}_{\mathbb{N}}$
- Edge set merger, $\mathbb{M}_{\mathbb{E}}$

Before describing these job types (or operators) in detail, let us define several utility functions that are commonly used in all operators. First *getURI*$(value_1, \ldots)$ is an hash function that returns a unique URI given an arbitrary number of inputs. *mergeNodeComponent*$(v_1, v_2, \ldots)$ is a function which takes an arbitrary number of node components as inputs and returns a unique node as a result of merging the input components. Here all input node components $v_1, v_2, \ldots$ are of the same *URI*. One important aspect of *mergeNodeComponent*$(\cdot)$ is to merge the properties of the all input components. If different values exit for the same attributes of input nodes, either the values are merged into one value (e.g. array) or one value is selected depending on the user option. Similarly, *mergeEdgeComponent*$(e_1, e_2, \ldots)$ returns a unique edge as the result of merging edge components that have the same *URI*.

Algorithm 1 shows Map-Reduce functions for node component constructor $\mathbb{N}$. It takes five input arguments **R**, **U**, **P**, φ, and l. **R** is a source relation, **U** is a set of *URI* attributes, and **P** is a set of node property attributes. φ is a propositional formula composed of conditional elements and the logical operators $\vee$ (and), $\wedge$ (or), and $\neg$ (negation), and l is the node type label. Then, it returns a set of l node components by converting relation R. The *URI* of generated nodes will be assigned based on the values of attributes in **U**. Each node has properties, in other words, attributes that are in the **P**. Notice that a *Mapping* element in the mapping configuration file can be interpreted as inputs to the Algorithm 1. **U** is achieved from the 'key_properties' *Map*, **P** is achieved from 'properties' *Map*, l is achieved from 'node_type_label' *Map*, and φ is achieved from 'condition' *Map*.

Algorithm 1 Node Component Constructor $\mathbb{N}$(**R**, **U**, **P**, φ, l)

<Map Function>
INPUT
  R$(A_1, \ldots, A_n)$: A source relation
  U=$\{A_{u_1}, \ldots, A_{u_{|U|}}\}$: A set of URI attributes
  P=$\{A_{p_1}, \ldots, A_{p_{|P|}}\}$: A set of property attributes
  (**U**, **P** $\subset \{A_1, \ldots, A_n\}$)
  φ: A propositional formula
  l: a Node type label
BEGIN
  if φ!=$empty$ **then**
    R' $\leftarrow \sigma_{\varphi}$(**R**)
    /* σ is the selection operation in relational algebra */
  else
    R' $\leftarrow$ **R**
  end if
  for all tuple $r \in$ **R**' **do**
    if U!=ϕ **then**
      uri $\leftarrow$ *getURI*(l, $r[A_{u_1}]$, …, $r[A_{u_{|U|}}]$);
    else
      uri $\leftarrow$ get a new *URI* that has not been used by other nodes or edges;
    end if
    create a l type node component v whose *URI* is *uri*;
    for all A_{p_i} in **P** **do**
      if $r[A_{p_i}]$!=*null* **then**
        Add (A_{p_i}, $r[A_{p_i}]$) to the node v's property set;
      end if
    end for
    key $\leftarrow$ *uri*
    Output <*key*, v>;
  end for
END

<Reduce Function>
INPUT
  key, (v_1, v_2, …)
BEGIN
  Output <*key*, *mergeNodeComponent*(v_1, v_2, …)>
END

Similar to the node component constructor, Edge component constructor $\mathbb{E}$ takes following arguments as inputs. **R**

is a source relation, **U** is a set of *URI* attributes, **P** is a set of edge property attributes, φ is a propositional formula, and l is the edge type label. l_{from} is node type label for start nodes and l_{to} is node type label for destination nodes. $\mathbf{U}_{from}$ and $\mathbf{U}_{to}$ are *URI* attributes that are exploited to generate *URI*s of start and destination nodes. Algorithm 2 shows how the edge constructor $\mathbb{E}$ generates edge for given parameters. *Mapping* element in an edge mapping document can be interpreted as inputs of the Algorithm 2. **P** is achieved from 'properties' *Map*, l is achieved from 'edge_type_label' *Map*, and φ is achieved from 'condition' *Map*. l_{from}, l_{to}, $\mathbf{U}_{from}$, $\mathbf{U}_{to}$ are achieved from respective 'from_node_type_label' *Map*, 'to_node_type_label' *Map*, 'from_key_properties' *Map*, and 'to_key_properties' *Map*.

Algorithm 2 Edge Component Constructor $\mathbb{E}$(**R**, **P**, φ, l, $\mathbf{U}_{from}$, $\mathbf{U}_{to}$)

<**Map Function**>
INPUT
R($A_1, \ldots, A_n$): A source relation
P=$\{A_{p_1}, \ldots, A_{p_{|P|}}\}$: A set of property attributes
$\mathbf{U}_{from}$=$\{A_{s_1}, \ldots, A_{s_{|U_s|}}\}$: A set of URI attributes for start nodes
$\mathbf{U}_d$=$\{A_{d_1}, \ldots, A_{d_{|U_d|}}\}$: A set of URI attributes for destination nodes
($\mathbf{U}_{from}$, $\mathbf{U}_{to}$, **P** $\subset \{A_1, \ldots, A_n\}$)
φ: A propositional formula
l_{from}: A node type label for start nodes
l_{to}: A node type label for destination nodes
l: An edge type label
BEGIN
if φ!=$empty$ **then**
R' $\leftarrow \sigma_{\varphi}$(**R**)
else
R' $\leftarrow$ **R**
end if
for all tuple $r \in$ **R**' **do**
$uri_{from} \leftarrow$ *getURI*(l_{start}, $r[A_{s_1}], \ldots, r[A_{s_{|U_s|}}]$);
$uri_{to} \leftarrow$ *getURI*(l_{to}, $r[A_{d_1}], \ldots, r[A_{d_{|U_d|}}]$);
create a l type edge $e(v_{from}, v_{to})$, where the node v_{from}'s *URI* is uri_{from} and the node v_{to}'s *URI* is uri_{to};
for all A_{p_i} in **P** **do**
if $r[A_{p_i}]$!= *null* **then**
Add (A_{p_i}, $r[A_{p_i}]$) to the edge e's property set;
end if
end for
key $\leftarrow$ *getURI*(l, uri_{from}, uri_{to});
Output <*key*, e>;
end for
END

<**Reduce Function**>
INPUT
key, (e_1, e_2, ...)
BEGIN
Output <*key*, *mergeEdgeComponent*(e_1, e_2, ...)>
END

Component merging provides a way to construct nodes and edges from multiple source tables individually and merge them to construct complete nodes or edges. Let us consider an example in Fig. 4(a), where a persons' information is separately stored in PERSON and ADDRESS tables. A simple approach to create a type of nodes or edges from more than one table is through *join* operations within relational databases, followed by schema mapping step for the large joined table and the data conversion step (Fig. 4(a)). This approach has limitation in that *join* can be very expensive for large relational tables. Performing such *join* operations by taking advantage of Map-Reduce is useful to scale up the graph construction process (Fig. 4(b)).

Map-Reduce jobs for node constructor (Algorithm 2) and edge constructor (Algorithm 2) can be executed one or more times per each table as depicted in the Fig. 4(b). Each execution of Map-Reduce job will produce either a set of <key, v> pairs or a set of <key, e> pairs, where v is a node component and e is an edge component. So, there can be multiple temporary node sets or edge sets as intermediate results. *Node Set Merger* (Algorithm 3) $\mathbb{M}_{\mathbb{N}}(\mathcal{V})$ merges all node component sets resulted in the Phase 1 and returns the final set of nodes. *Edge Set Merger* (Algorithm 4) $\mathbb{M}_{\mathbb{E}}(\mathcal{E})$ does the same for edge component sets.

Algorithm 3 Node Component Set Merger $\mathbb{M}_{\mathbb{N}}(\mathcal{V})$

<**Map Function**>
INPUT
$\mathcal{V}$: A set of sets that are composed of <key, v> pairs resulted by the Node Component Constructor, where v is a node component
BEGIN
for all V_i in $\mathcal{V}$ **do**
for all <key, e> in V_i **do**
Output <key, v>
end for
end for
END

<**Reduce Function**>
INPUT
key, (v_1, v_2, ...)
BEGIN
Output *mergeNodeComponent*(v_1, v_2, ...)
END

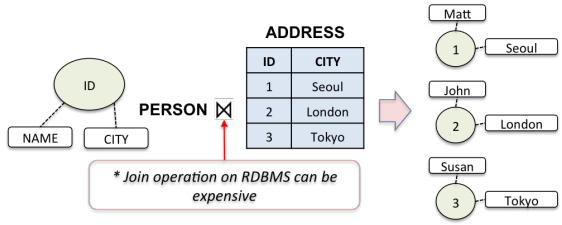
IV. Experiments

Table2Graph is designed to convert source relational data sets into a consolidated graph model at scale. In order to demonstrate its scalability, we measured the time required to construct the consolidated graph model when different sizes of computing resources are used. The intention was to show the expected performance growth of *Table2Graph* as the bigger resources are used. For this end, we performed experiments using Amazon Web Service (AWS)'s public cloud with the cluster sizes of 1, 2, 4, and 8. We used the *m2.2xlarge* instance type, which has 4 CPUs, 34.2GB of memory, and 850GB of locally attached file I/O storage. A summarized specification of the instance type can be found in Amazon's web site (http://aws.amazon.com/ec2/previous-generation/). Also we employed Hadoop 2.4.0 and scheduled

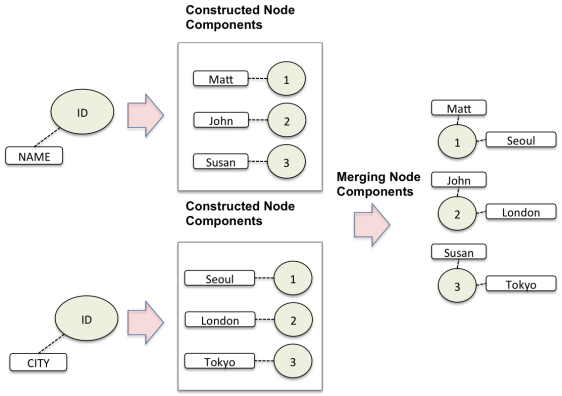
Algorithm 4 Edge Component Set Merger $\mathbb{M}_{\mathbb{E}}(\mathcal{E})$

<Map Function>
INPUT
$\mathcal{E}$: A set of sets that are composed of <key, e> pairs resulted by the Edge Component Constructor, where e is an edge component
BEGIN
for all E_i in $\mathcal{E}$ **do**
for all <key, e> in E_i **do**
Output <key, e>
end for
end for
END

<Reduce Function>
INPUT
key, (e_1, e_2, ...)
BEGIN
Output *mergeEdgeComponent*(e_1, e_2, ...)
END



(a) Joining two source tables and constructing a graph



(b) Constructing node components from individual source tables and merging them to construct complete nodes

Fig. 4. Two approaches to constructing nodes from multiple source tables

all independent MapReduce jobs to simultaneously run in order to maximize the parallelism of conversion process.

We applied *Table2Graph* to integration of three databases into a consolidated graph model, which are NPPES[1] (Medicare), **PECOS**[2] (Medicare), and **TMSIS**[3] (Medicaid data from State of Texas). Fig. 5 shows a simplified structure of the

[1]National Plan & Provider Enumeration System
[2]Provider Enrollment, Chain, and Ownership System
[3]Transformed Medicaid Statistical Information System

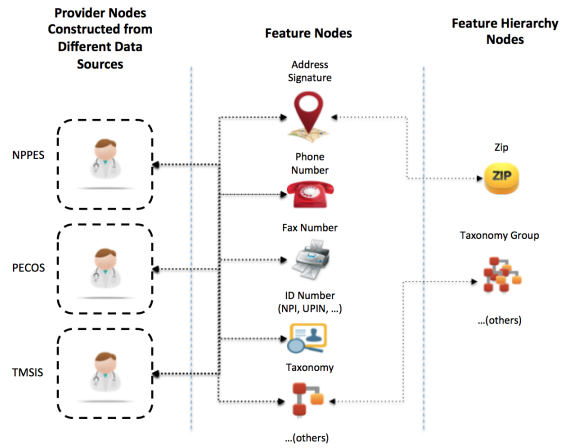


Fig. 5. Simplified structure of the constructed graph structure using three healthcare datasets

model constructed from the three data sources, where only a few nodes and edges are introduced for illustration purposes.

Of the three data sources that were used to construct the consolidate graph model, we tested the *Table2Graph* only with **NPPES** (National Plan & Provider Enumeration System) dataset[4] leaving the other two data sets out of the experiment. The reason is that these data sets, **PECOS** and **TMSIS** contain protected health information and cannot be placed into public computing facilities like Amazon Web Service. Moreover, with the project with CMS expired, we had to discard the data sets. Thus when we designed a systematic experimental setting for a scalability test, the two data sets were not available. However, although we do not present any detailed numbers, we observed the similar performance growth when all three data sets were actually used for the integration.

NPPES is a healthcare provider dataset that is disseminated monthly by CMS. It includes a healthcare provider's demographic features such as address, gender, taxonomy, other identifiers, and etc. The data set used for the experiment is the version of June, 2013. The size of the dataset is 5.07 GB, with 4,181,747 records. Fig. 6 shows the construction time for each cluster setting. The constructed graph has 16,543,479 nodes 48,415,932 edges. We could construct the graph 3.41 times faster with 8 worker nodes than the single worker node case. This experiment confirms that *Table2Graph* can be exploited to construct a very large-scale graph in a reasonable processing time, and the processing time can be significantly reduced as the number of worker nodes increases.

V. Conclusions

A graph-based model is used in a wide range of domains for its capability of fast associative data set processing. Examples include electricity grids, transportation routes, paths of disease outbreaks, and relationships among scientific studies. In this paper, we described another application domain of a graph-based model, linking disparate data sources of healthcare service providers for the purpose of matching. Although many efficient graph analytic algorithms for our mission are available, there still exists a wide gap between data sources

[4]http://nppes.viva-it.com/NPI_Files.html

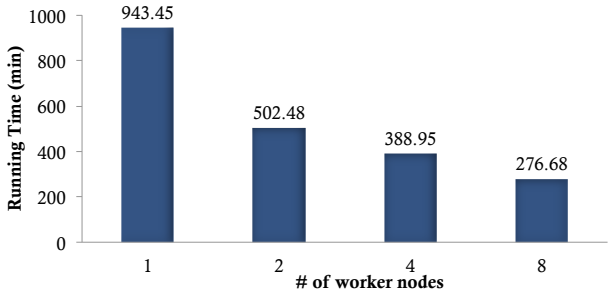


Fig. 6. Graph Construction Time for NPPES Dataset

and data analytic systems, that is, conversion of the source data into a consolidated graph model. To address this, we introduced *Table2Graph* that is devised to facilitate a ETL process of converting multiple relational data tables into a graph-based model. Based on MapReduce framework over Hadoop, *Table2Graph* is designed to process voluminous data using commodity computing resources. Although, preliminary, our empirical performance study demonstrates its scalability. *Table2Graph* is also highly configurable and adaptable so it can be applied to an arbitrary domain with ease. To further extend *Table2Graph* for better accessibility and usability, we are currently pursuing the followings.

Developing a semi-automatic schema mapping: The current version of *Table2Graph* requires users to write *XML* documents manually. *Table2Graph* may be further extended to provide guidance. For this end, we are exploring text mining and data distribution analysis to identify semantic similarities between data sets. Such a capability, once incorporated into the tool, will provide recommended mappings to user through an interactive Graphical User Interface (GUI). We believe this will greatly benefit domain experts with little or no database understandings.

Adopting in-memory large-scale data processing system: Nowadays, in-memory large-scale data processing systems, such as Apache Spark [24] are gaining much attention from both academia and industry. We found these technologies will reduce an ETL process time greatly. We are currently developing a new version *Table2Graph* to take advantages of such an in-memory distributed computing technology.

ACKNOWLEDGEMENT

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] S. Lee, S. Park, M. Kahng, and S.-g. Lee, "Pathrank: Ranking nodes on a heterogeneous graph for flexible hybrid recommender systems," *Expert Systems with Applications*, vol. 40, no. 2, pp. 684–697, 2013.

[2] C. Strauch, U.-L. S. Sites, and W. Kriha, "Nosql databases," *URL: http://www. christof-strauch. de/nosqldbs. pdf (07.11. 2012)*, 2011.

[3] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1664–1665, 2009.

[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 553–564.

[5] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems," *Dept. of Computer Science, Purdue University*, 2006.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[7] K. Chodorow, *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[8] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide*. " O'Reilly Media, Inc.", 2010.

[9] J. W. I. Robinson and E.Elfrem, "Graph databases," in *Graph Databases*. O'Reilly Media, 2013.

[10] M. Broecheler, D. LaRocque, M. A. Rodriguez, P. Yaskevich, S. Mallette, and K. Yim, "Titan, distributed graph database," https://github.com/thinkaurelius/titan/wiki.

[11] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey, "Dex: high-performance exploration on large graphs for information retrieval," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. ACM, 2007, pp. 573–582.

[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1–10.

[14] J. Fong, H. K. Wong, and Z. Cheng, "Converting relational database into xml documents with dom," *Information and Software Technology*, vol. 45, no. 6, pp. 335–355, 2003.

[15] M. F. Fernandez, A. Morishima, D. Suciu, and W.-C. Tan, "Method for converting relational data into xml," Aug. 31 2004, uS Patent 6,785,673.

[16] P. Buneman, "Semistructured data," in *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1997, pp. 117–121.

[17] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 3, pp. 1–22, 2009.

[18] J. Barrasa Rodríguez, Ó. Corcho, and A. Gómez-Pérez, "R2o, an extensible and semantically based database-to-ontology mapping language," 2004.

[19] C. Bizer and A. Seaborne, "D2rq-treating non-rdf databases as virtual rdf graphs," in *Proceedings of the 3rd international semantic web conference (ISWC2004)*, vol. 2004, 2004.

[20] O. Erling and I. Mikhailov, "Rdf support in the virtuoso dbms," in *Networked Knowledge-Networked Media*. Springer, 2009, pp. 7–24.

[21] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, "Triplify: light-weight linked data publication from relational databases," in *Proceedings of the 18th international conference on World wide web*. ACM, 2009, pp. 621–630.

[22] R. De Virgilio, A. Maccioni, and R. Torlone, "Converting relational to graph databases," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 1.

[23] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: Scalable graph etl

framework," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 4.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.