Interactive Browsing and Navigation in Relational Databases

Minsuk Kahng, Shamkant B. Navathe, John T. Stasko, and Duen Horng (Polo) Chau
Georgia Institute of Technology
Atlanta, GA, USA
kahng@gatech.edu, sham@cc.gatech.edu, stasko@cc.gatech.edu, polo@gatech.edu

ABSTRACT

Although researchers have devoted considerable attention to helping database users formulate queries, many users still find it challenging to specify queries that involve joining tables. To help users construct join queries for exploring relational databases, we propose *ETable*, a novel presentation data model that provides users with a presentation-level interactive view. This view compactly presents one-to-many and many-to-many relationships within a single *enriched table* by allowing a cell to contain a set of *entity references*. Users can directly interact with this enriched table to incrementally construct complex queries and navigate databases on a conceptual entity-relationship level. In a user study, participants performed a range of database querying tasks faster with *ETable* than with a commercial graphical query builder. Subjective feedback about *ETable* was also positive. All participants found that *ETable* was easier to learn and helpful for exploring databases.

1. INTRODUCTION

A considerable challenge for non-technical users of relational databases is constructing *join* queries [29]. The join operation is required for even simple data lookup queries since relational databases store information in multiple separate normalized tables. Although database normalization provides many benefits for managing data (e.g., avoiding update anomalies), it significantly decreases the usability of database systems by forcing users to write many join queries to explore databases.

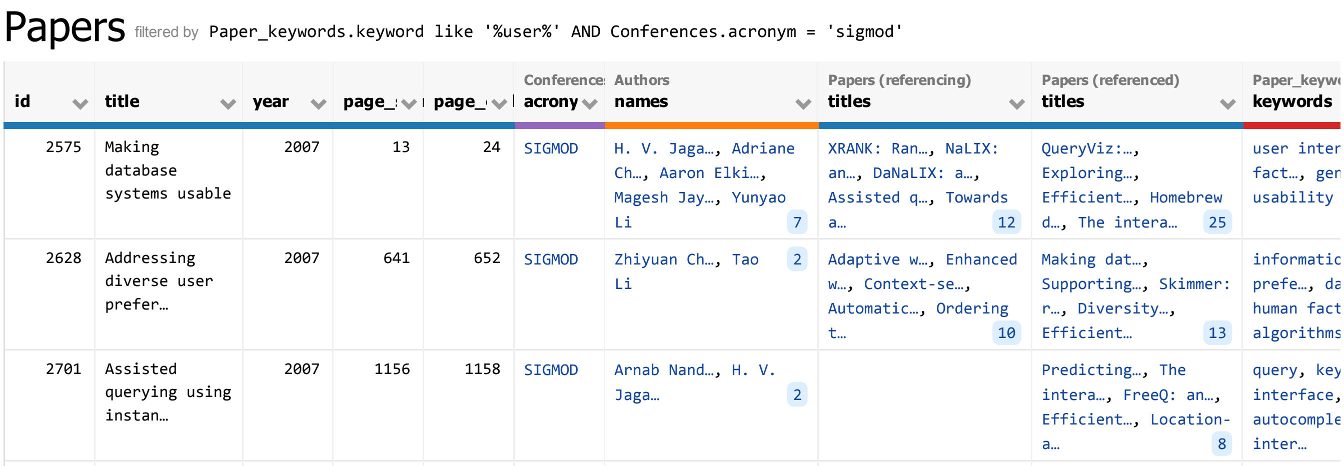
Constructing join queries is difficult for several reasons. The main reason is that users find it difficult to determine which relations to join among many relations. Understanding the role of each relation that represents a relationship of interest and finding the right join attributes are not trivial tasks, even when a schema diagram is given. To tackle this challenge, users often write complex queries by starting with a simpler query and iteratively adding operators [37]. Although this iterative strategy is helpful, it is still challenging because the format of join query results is hard to interpret. For example, consider a query that joins two relations in many-to-many relationships (e.g., `Papers` and `Authors` in Figure 3). A result of this query produces a large number of duplications (e.g., the title of each paper repeated as many times as the number of its authors). People represent the same information differently when they use a spreadsheet. For instance, they might create a cell containing multiple values separated by commas. Relational databases cannot represent data in this way because the relational model (as implemented in most relational DBMSs) requires that data be at least in the *first normal form*.

The usability challenge of writing complex queries has been studied by many researchers. Although *visual query builders* help people formulate SQL queries [13], they separate query construction and result presentation parts [29], introducing a usability gap between users' actions and their results [42, 37]. To overcome this limitation, researchers argue that database interfaces need to adopt the *direct manipulation* principle [42], a well-known concept in the *human-computer interaction (HCI)* area [29, 35]. It enables users to iteratively specify operators by directly interacting with result instances using simple interactions [35]. Researchers also argue that join query results should be represented in an easier-to-understand format that improves the interpretation of query results. Jagadish et al. [30] proposed the notion of the *presentation data model*, which they defined as a full-fledged layer above the logical and physical schema. This presentation layer allows users to better understand the query results without requiring full awareness of the schema. All this research strongly suggests the need for developing database interfaces that are usable, interactive, and interpretable.

We present *ETable*, a novel presentation data model with which users can interactively browse and navigate databases on an entity-relationship level without writing SQL. *ETable* presents a query result as an enriched table in which each cell can contain a set of *entity references*. By deliberately relaxing the *first normal form*, we compactly represent one-to-many and many-to-many relationships within a single table — a novel capability that enables users to more easily browse and interpret query results consisting of multiple relations. Figure 1 illustrates how *ETable* effectively presents a list of SIGMOD papers containing the keyword "user" from an academic paper database collected from DBLP and the ACM Digital Library (see Figure 3 for schema). Each row in *ETable* shows the base attributes and relevant entities of a paper, such as its authors and cited papers. If a relational database were used to obtain the same information, 9 tables would need to be joined, and the results produced would be hard to interpret (e.g., many duplicated cells).

To discover which relevant entities should be shown for each row, *ETable* uses a novel graph-based model called the *typed graph model (TGM)*, which frees users from concerning themselves with the complexity of the logical schema; users may instead focus on exploring and understanding the data set at the conceptual (or entity-relationship) level. The *typed graph model* stores relational data as graphs in which nodes represent entities (e.g., authors, papers) and

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.



Papers filtered by `Paper_keywords.keyword like '%user%' AND Conferences.acronym = 'sigmod'`

					Conference	Authors	Papers (referencing)	Papers (referenced)	Paper_keyw
id	title	year	page_	page_	acrony	names	titles	titles	keywords
2575	Making database systems usable	2007	13	24	SIGMOD	H. V. Jaga…, Adriane Ch…, Aaron Elki…, Magesh Jay…, Yunyao Li 7	XRANK: Ran…, NaLIX: an…, DaNaLIX: a…, Assisted q…, Towards a… 12	QueryViz:…, Exploring…, Efficient…, Homebrew d…, The intera… 25	user inter fact…, ger usability
2628	Addressing diverse user prefer…	2007	641	652	SIGMOD	Zhiyuan Ch…, Tao Li 2	Adaptive w…, Enhanced w…, Context-se…, Automatic…, Ordering t… 10	Making dat…, Supporting…, Skimmer: r…, Diversity…, Efficient… 13	informatic prefe…, da human fact algorithms
2701	Assisted querying using instan…	2007	1156	1158	SIGMOD	Arnab Nand…, H. V. Jaga… 2		Predicting…, The intera…, FreeQ: an…, Efficient…, Location-a… 8	query, key interface, autocomple inter…

Figure 1: *ETable* integrates multiple relations into a single enriched table that helps users browse databases and interactively specify operators for building complex queries. This example presents a list of SIGMOD papers containing the keyword "user" from an academic paper database collected from DBLP and the ACM Digital Library. Each column represents either a base attribute of a paper or a set of relevant entities obtained from other tables (e.g., `Conferences`, `Authors`). If a relational database were used to obtain the same information, 9 tables would need to be joined, and the results produced can be hard to interpret because of many duplicated cells.

edges represent relationships (e.g., those that relate authors to papers). This transformation enables *ETable* to retrieve other related entities through simple graph operations. For example, a given paper's authors, stored as direct neighbors, can be retrieved through a quick neighbor-lookup.

As the construction of complex queries and the exploration of data are inherently iterative processes, database exploration tools should provide easy-to-use operations to help users incrementally revise queries [16, 37, 35]. *ETable*'s direct manipulation interface enables users to directly work with and modify an existing enriched table to update its associated queries. For example, imagine a user, Jane, who would like to further explore the result in Figure 1. To see the detailed information about the authors of a particular paper, she clicks on its "author count" button (Figure 2-b). This simple interaction of tapping the button is translated into a series of *primitive operators* behind the scene, such as *Select*, as in selecting the row associated with a paper; and *Add*, as in adding and joining the `Authors` table with the `Papers` table. With a few rounds of similar interactions, Jane can incrementally build complex queries.

ETable's novel ideas work together to address an important, often overlooked problem in databases. The seminal vision paper by Jagadish et al. [29] introduced the notion of the presentation data model and argued the importance of direct manipulation interface. However, designing an easy-to-use system that meets these requirements is challenging. *ETable* is one of the first instantiations of this important idea, filling a critical research gap, by effectively integrating HCI principles to greatly improve database usability. To enable the creation of such a usable tool, *ETable* tightly integrates: (1) a novel hybrid data model representation, which advances over the relational and nested-relational models, to naturally represent entities and relationships; and (2) a novel set of interactions that closely work with the representation to enable users to specify expressive queries through direct manipulation. With *ETable*'s user interface, non-experts can easily and naturally explore databases without writing SQL, while *ETable* internally performs queries under the hood.

Through *ETable*, we contribute:

- A novel **presentation data model** that presents a query result as an enriched table for users to easily browse and explore relational databases (Section 3, 5);
- A **graph-based model**, called *typed graph model (TGM)* that provides an abstraction of relational databases, for users to explore data in *ETable* at a conceptual level (Section 4);
- A set of **user-level actions**, operations that users can directly apply to an enriched table to incrementally construct complex queries and navigate databases (Section 6.1);
- The **usable interface** of *ETable* that outperforms a commercial graphical query builder in a **user study**, in both speed and subjective ratings across a range of database querying tasks (Section 6, 7).

2. RELATED WORK

2.1 Database Usability & Query Specifications

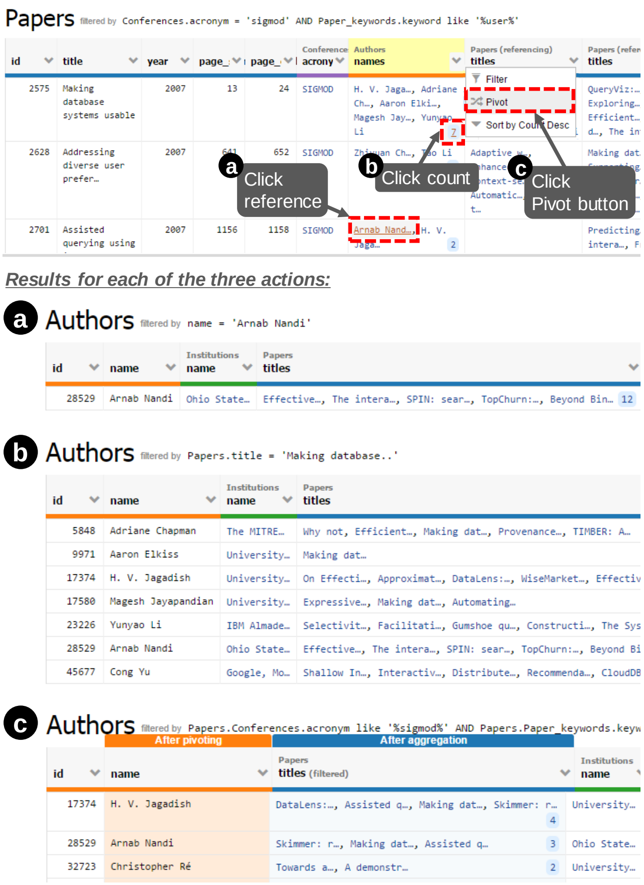
Since Query-by-Example (QBE) was developed in 1970s [48], database researchers have studied fairly extensively the usability aspect of database systems [29, 12, 2, 28]. Usability is important, especially because not all database users have expertise in writing complex queries; many non-technical users find it challenging to write even very simple join queries [29, 1]. Many existing approaches are aimed at assisting users with formulating queries. One representative method is the *visual query builder*, which enables users to visually manipulate schema elements on a graphical interface [13]. However, most visual querying systems require that users have precise knowledge of a schema, which makes it difficult for non-experts to use. This limitation can be relieved in *keyword search* systems, studied extensively in the last decade [27, 10, 4, 19], or natural language interfaces [33]. However, most of existing approaches [31, 23] separate queries and results so that users cannot directly refine query results, which decreases the usability of the systems. Nandi and Jagadish [37] argued that users' querying process is often iterative, so database systems should guide users toward interactively formulating and refining queries.

2.2 Direct Manipulation & Iterative Querying

Several database researchers argued that the usability of database querying systems can improve by adopting the *direct manipulation* paradigm [42], a well-established design principle in the HCI and information visualization areas. Acknowledging that users' needs are often ambiguous rather than precisely specifiable, researchers have developed many tools that enable users to interactively browse and explore databases [28, 11, 43]. Although they are not specifically designed for relational databases, a number of interactive visualization systems for entity-relationship data have been developed by information visualization researchers [32, 22, 21, 36]. For example, NetLens [32] visualizes relationships between two selected entity types in many-to-many relationships, and GraphTrail [22] visually summarizes each entity type and enables users to switch between entities. Although these visualization systems provide an overview of data sets, they are not suited for examining database instances along with attributes. In exploring and analyzing instance-level information, tabular interfaces, including spreadsheets, are better suited and often preferred by database users [24, 46, 35, 17, 25]. Tyszkiewicz [46] argued that spreadsheets can play a role as a database engine by using functions and macros. Liu and Jagadish [35] formally defined operators that interactively perform grouping operations within a spreadsheet. However, since the rigid tabular structure does not effectively present many-to-many relationships, the spreadsheet suffers from the same problems that relational databases have (i.e., a large number of duplications). To overcome this limitation, Jagadish et al. [30] proposed using a presentation view layer on top of underlying databases, which is the notion of the *presentation data model*, defined as a full-fledged layer on top of the logical and physical models. The challenge is to design presentation data models that help people easily understand join query results and interact with them.

2.3 Data Models for Effective Presentation

To develop an intuitive structure for presentation data models, we review a number of data models that conceptualize the mini-world represented in databases. One such example is the *nested relational model*, studied in the 1980s, which allows each cell to contain another table that presents one-to-many relationships in a single table [40, 39]. The nested model has been used in several studies for designing database interfaces. Bakke et al. [7] recently designed a direct manipulation interface for nested-relational databases, and DataPlay [3] also used the nested model for presenting query results. However, the model suffers from scalability issues because the sizes of the nested tables often become huge when an inner table contains a large number of associated rows or columns [8]. One way to tackle this problem is to replace the inner table with a set of pointers. For example, the *object-relational model* lets attributes be user-defined types that include pointers [44]. We adapt this idea by introducing an *entity reference* which compactly represents related entities. Another class of the data models that effectively conceptualize the real-world is the graph data model [6, 26, 14, 45]. It represents entities as nodes and relationships as edges based on the *entity-relationship model* [18, 9]. Catarci et al., [15] used a graph-style *translation layer* for their visual querying system. To provide users with an easy-to-understand view at an entity-relationship level, we also maintain a graph-style model, transformed from relational databases, under the presentation view.



Papers filtered by Conferences.acronym = 'sigmod' AND Paper_keywords.keyword like '%user%'

					Conference	Authors	Papers (referencing)	Papers (refer
id	title	year	page_	page_	acrony	names	titles	titles
2575	Making database systems usable	2007	13	24	SIGMOD	H. V. Jaga…, Adriane Ch…, Aaron Elki…, Magesh Jay…, Yunyao Li 7		QueryViz:…, Exploring…, Efficient…, d…, The in
2628	Addressing diverse user prefer…	2007	641	652	SIGMOD	Zhiyuan Ch…, Tao Li	Adaptive w…, …hance…, …ntext-se…, Automatic…, t…	Making dat…, Supporting…
2701	Assisted querying using …	2007	1156	1158	SIGMOD	Arnab Nand…, H. V. Jaga… 2		Predicting intera…, F

Results for each of the three actions:

a Authors filtered by name = 'Arnab Nandi'

		Institutions	Papers
id	name	name	titles
28529	Arnab Nandi	Ohio State…	Effective…, The intera…, SPIN: sear…, TopChurn:…, Beyond Bin… 12

b Authors filtered by Papers.title = 'Making database..'

		Institutions	Papers
id	name	name	titles
5848	Adriane Chapman	The MITRE…	Why not, Efficient…, Making dat…, Provenance…, TIMBER: A…
9971	Aaron Elkiss	University…	Making dat…
17374	H. V. Jagadish	University…	On Effecti…, Approximat…, DataLens:…, WiseMarket…, Effectiv
17580	Magesh Jayapandian	University…	Expressive…, Making dat…, Automating…
23226	Yunyao Li	IBM Almade…	Selectivit…, Facilitati…, Gumshoe qu…, Constructi…, The Sys
28529	Arnab Nandi	Ohio State…	Effective…, The intera…, SPIN: sear…, TopChurn:…, Beyond Bi
45677	Cong Yu	Google, Mo…	Shallow In…, Interactiv…, Distribute…, Recommenda…, CloudDB

c Authors filtered by Papers.Conferences.acronym like '%sigmod%' AND Papers.Paper_keywords.keyw

		Papers	Institutions
id	name	titles (filtered)	name
17374	H. V. Jagadish	DataLens:…, Assisted q…, Making dat…, Skimmer: r… 4	University…
28529	Arnab Nandi	Skimmer: r…, Making dat…, Assisted q… 3	Ohio State…
32723	Christopher Ré	Towards a…, A demonstr… 2	University…

Figure 2: Users can iteratively specify user-level actions by interacting with *ETable*. In this example, users can examine further information about paper authors in three ways: (a) clicking on an author's name; (b) clicking a paper's author count; (c) clicking on the pivot button.

3. INTRODUCING ETABLE

Before we describe the technical details of the proposed data models, we introduce *ETable* by describing what users see and how they can interact with it.

Representation. Figure 1 illustrates an enriched table that we call *Etable*. As mentioned earlier, it presents a list of SIGMOD papers containing the keyword "user" from our collected database (see Figure 3 for schema). Each row of *Etable* represents a single entity of the selected entity type (i.e., `Papers`); its column represents either a base attribute of the entity (e.g., year) or a set of relevant entities (e.g., authors, keywords). This representation is formed by pivoting a query result of a join of multiple tables (e.g., `Papers`, `Paper_keywords`, `Authors`) to a user-selected entity type (e.g., `Papers`). One advantage of this representation is that it can simultaneously present all relevant information about an entity in a single row (e.g., authors, keywords, citations). The relational model cannot represent all of this information in a single relation without duplications because every attribute value must be atomic. For instance, when the `Papers` table is joined with the `Authors` table, the paper information is repeated as many times as the number of authors, which prevents users from quickly interpreting the results. We integrate information spread across multiple tables into a single table by allowing each cell to contain a set of references to other entities.

Interactions. Users can interact with *Etable* to explore further information. For instance, to examine further information about the authors of the papers in Figure 1, users can create a new *Etable* that lists authors in several ways, as depicted in Figure 2: (1) If users are interested in one of the authors (e.g., Arnab Nandi), they can click on his name to create a new *Etable* consisting of one row that presents its attributes; (2) if users want to list the complete set of authors (e.g., all seven authors of the paper titled "Making database systems usable"), they can click on the author count in

Form	Source	Determining factor for mapping from a relational table
Node types	Entity tables	Relation with a single-attribute primary key
	Multi-valued attributes	Relation with two attributes; one of them is a foreign key of an entity relation
	Single-valued categorical attributes	Attribute of low cardinality
Edge types	One-to-many relationships	Foreign key between two entity relations
	Many-to-many relationships	Relation with a composite primary key; both are foreign keys of entity relations
	Multi-valued attributes	From an entity table to a multi-valued attribute
	Single-valued categorical attributes	From an entity table to a categorical attribute

Table 1: Categories of node and edge types based on how they are translated from relational schema

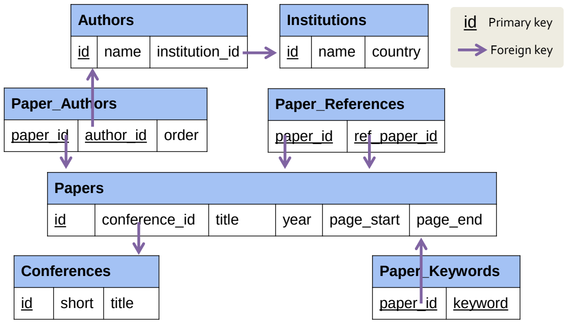


Figure 3: The relational schema of the academic data set used in this work, 7 relations in total.

the right corner of the cell (i.e., 7); and (3) if users want to list and sort the entities across the entire rows in a column (e.g., Who wrote the most papers about "user" in SIGMOD?), they can click on the *pivot* button on the column menu, which groups and sorts the authors based on the number of papers they have written. By gradually applying these operations, users can incrementally make sense of data and build complex queries.

4. TYPED GRAPH MODEL

In this section, we define a *typed graph model (TGM)* which enables users to explore relational databases on a conceptual entity-relationship level without having to know a logical schema. A relational schema and instances are translated into a *database schema graph* and *database instance graph* as a preprocessing step, and all operations specified by users on the *ETable* interface are executed over these graphs, not relational databases.

We represent entities and relationships as a graph with types and attributes. Each entity forms a node, and relationships among the entities become edges. A *typed graph database (TGDB)* consists of a *TGDB schema graph*, $\mathcal{G}_S$, and a *TGDB instance graph*, $\mathcal{G}_I$.

Definition 1. **Schema Graph.** A TGDB schema graph $\mathcal{G}_S$ is a tuple $(\mathcal{T}, \mathcal{P})$, where $\mathcal{T}$ represents a set of node types (or entity types[1]), and $\mathcal{P} \subseteq \mathcal{T} \times \mathcal{T}$ represents a set of edge types (or relationship types). Each node type $\tau_i \in \mathcal{T}$ is a tuple $(\alpha_i, \mathcal{A}_i, \beta_i)$, where α_i denotes the name of a node type, $\mathcal{A}_i$ is a set of single-valued attributes, and β_i is a label attribute chosen from one of the attributes and used to represent node instances of this type. Each edge type $\rho \in \mathcal{P}$ also has a name and a set of attributes. We denote the source and target node types of ρ as $source(\rho)$ and $target(\rho)$, respectively. All the edge types, except self loops, are bidirectional.

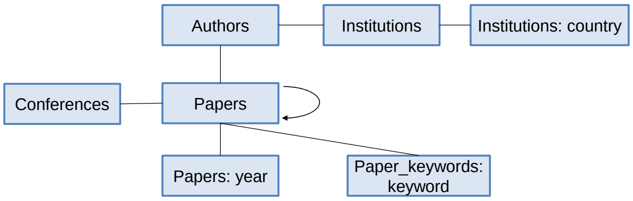


Figure 4: TGDB schema graph constructed from the relational schema in Figure 3. Each rectangle represents a node type, and each edge is an edge type.

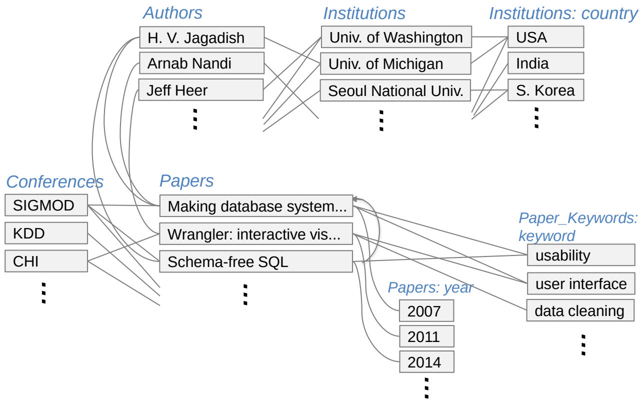


Figure 5: A part of the TGDB instance graph constructed from the academic data set used in this paper, following the schema in Figure 4. Node types shown in blue italic font.

Definition 2. **Instance Graph.** A TGDB instance graph $\mathcal{G}_I$, is a tuple (V, E), where V represents a set of nodes (or entities) and E represents a set of edges (or relationships) between two nodes. Every instance graph $\mathcal{G}_I$ has a corresponding schema graph $\mathcal{G}_S$, and the instance graph has a node type mapping function $type_\tau : V \rightarrow \mathcal{T}$ and an edge type mapping function $type_\rho : E \rightarrow \mathcal{P}$ that partition nodes V into $V_1, ..., V_{n_\mathcal{T}}$ and edges E into $E_1, ..., E_{n_\mathcal{P}}$. Each node $v \in V$ consists of a set of attribute values $v[A_{ij}]$ for the attributes of the corresponding node type and has a label defined as $label(v) = v[\beta_i]$. Each edge $e \in E$ consists of a set of attribute values $e[A_{ij}]$ for its type. We denote the source and target nodes of e as $source(e)$ and $target(e)$, respectively.

The typed graph model, similar to many graph data models [6, 26, 45], is much more effective for conveying a conceptual understanding of the mini-world represented in databases than the relational model. As it abstracts relational databases, users can ignore the logical and physical representation of data. Users can also easily understand the structure of data, since nodes always represent

[1] We use the words "node" and "entity" interchangeably. A node is used more formally; an entity is used more for presentation to users.

entities and edges represent relationships, Unlike TGM, the relational model is a mixture of entities, relationships, and multivalued attributes. Although some existing graph models are more expressive for representing a variety of relationships (e.g., hierarchical parent-child relationships among entities), we simply use nodes and edges to focus on making the semantics of the underlying relations more explicit by mapping to entities and relationships that they represent in the real world.

Relational databases can be translated into the TGDB schema and instance graphs in a near-automatic process. We adapt the reverse engineering literature pertaining to translating relational databases into several graph-style models [9, 20, 41]. A detailed procedure presented in Appendix A includes an analysis of a relational schema based on primary keys, foreign keys, and cardinalities for classifying tables into several categories, and a series of actions that create the schema graph. Table 1 summarizes the categories of node and edge types based on how they are determined from relational schema. Figures 4 and 5 illustrate a schema graph and a part of the instance graph constructed from an academic publication database whose schema is shown in Figure 3.

5. ETABLE PRESENTATION DATA MODEL

We present our *ETable* presentation data model for usable exploration of entities and relationships in databases.

5.1 Enriched Table

A query result in the *ETable* model is presented as an enriched table, which we also call *ETable*. An *ETable* R has a set of columns $\mathcal{A}$ and consists of a set of rows $r \in R$. The columns are categorized into two types: *single-attribute columns* and *entity-reference columns*. The value of the single-attribute column $r[A]$ is atomic as it is in the relational model. The value of the entity-reference column $r[A]$ contains a single or a set of *entity references*. The entity reference refers to another node in the database instance graph. Unlike a foreign key in the relational model, each entity reference is shown as a clickable label, similar to a *hyperlink* on a webpage. Just like how a hyperlink's hypertext describes the webpage that the link points to (instead of its URL), for example, *ETable* represents an author's entity reference by the author name (instead of the author ID).

The entity-reference columns present rich information spread across multiple relations within a single enriched table. While a foreign key attribute in the relational model contains only a single reference for a many-to-one relationship because of the first normal form, an entity-reference column can represent one-to-many relationships, many-to-many relationships, or multivalued attributes in a single column. Furthermore, the entity-reference column has advantages over the nested relational model which requires much screen space as it squeezes another table into cells, leading to inefficient browsing. Unlike the nested model, *ETable* presents clickable labels that compactly show information and allow users to further explore relevant information.

5.2 ETable Specification

An *ETable* can be specified by selecting specific elements of the TGDB database schema and instance graphs introduced in the previous section.

Definition 3. **ETable Query Specification.** An *ETable* R is specified by a *query pattern* Q, which is a tuple $(\tau_a, T, P, \mathcal{C})$.

1. **Primary node type** τ_a: It is one of the node types in the schema graph. Each row of *ETable* will represent a single node instance of the primary node type.

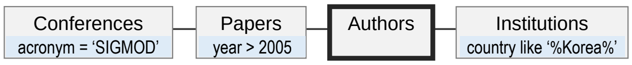


Figure 6: An example query pattern in a diagrammatic notation. It represents a query that finds a list of researchers who have published papers at SIGMOD after 2005 and are currently working at institutions in Korea.

2. **Participating node types** T: It is a set of node types chosen from the node types in the schema graph (i.e., $T = \{t_1, ..., t_{n_T}\}, \forall t_i \in \mathcal{T}$). It must contain the primary node type τ_a (i.e., $\tau_a \in T$). It determines the scope of data instances and is similar to a set of relations in SQL FROM clauses. A node type in the schema graph can exist multiple times in the participating node types, like a relational algebra expression can contain the same relation multiple times.

3. **Participating edge types** P: It is a set of edge types selected from the schema graph (i.e., $P = \{p_1, ..., p_{n_P}\}, \forall p_i \in \mathcal{P}$). It connects the participating nodes types, thus all the source and target nodes of these edges should exist in the participating node types (i.e., $source(p_i) \in T \wedge target(p_i) \in T, \forall p_i \in \mathcal{P}$).

4. **Selection conditions for node types** $\mathcal{C}$: It is a set of selection conditions $\mathcal{C} = (C_1, ..., C_{n_T})$ applied to each of the participating node types, i.e., C_i applies to $t_i \in T$.

A query pattern can be represented as an acyclic graph where one of the nodes is marked as a primary node type and any node can have selection conditions. For example, the query pattern in Figure 6 represents a query that produces a list of researchers who have published papers at SIGMOD after 2005 and are currently working at institutions in Korea.

5.3 Incremental Query Building with Primitive Operators

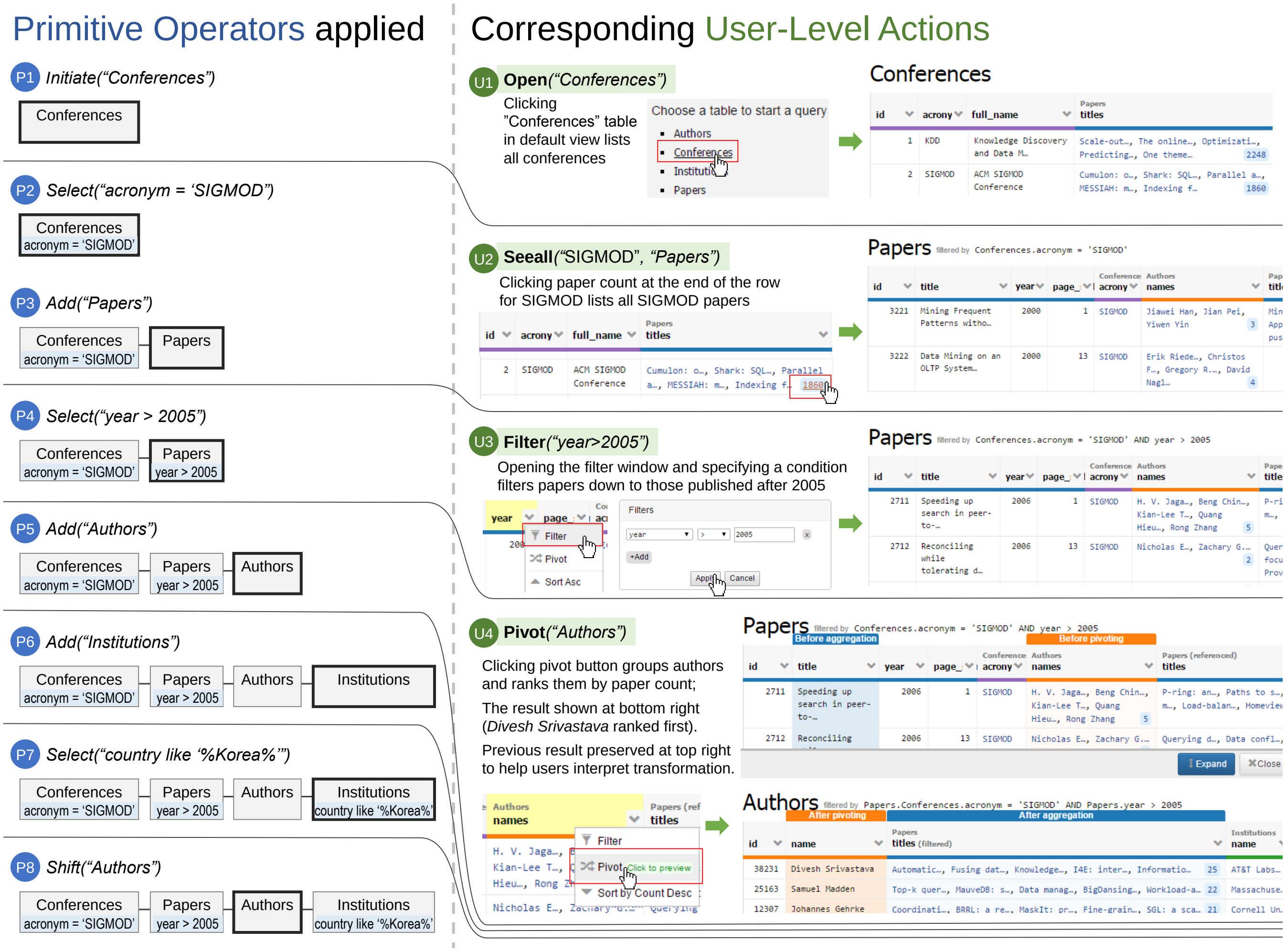
In *ETable*, a query pattern can be constructed and updated by *primitive operators*. Each operator builds on an existing *ETable* query to generate a new, updated *ETable* query. In this subsection, we describe these operators in detail. In Section 6.1, we will describe how users' actions performed on the *ETable* user interface will invoke these operators. Formally, given an *ETable* specification $Q(\tau_a, T, P, \mathcal{C})$, each of the following operator creates a new specification $Q'(\tau'_a, T', P', \mathcal{C}')$, except the *Initiate* operator which creates a new *ETable* from scratch.

1. **Initiation.** A new *ETable* can be created by selecting one of the node types τ_k in the schema graph. Its result lists the corresponding nodes.

   $$Initiate(\tau_k) = Q'$$
   where $\tau'_a = \tau_k,\ T' = \{\tau_k\},\ P' = \{\}$, and $\mathcal{C} = \{\}$.

2. **Selection.** *ETable* rows can be filtered based on their columns, similar to the selection operator in the relational model. Applying a selection condition C_k to the primary node type τ_a filters the rows of the current *ETable*.

   $$Select(C_k, Q) = Q'$$
   where $\tau'_a = \tau_a,\ T' = T,\ P' = P$, and $C'_a = C_k$.


Primitive Operators applied
P1 Initiate("Conferences")
Conferences
P2 Select("acronym = 'SIGMOD'")
Conferences acronym = 'SIGMOD'
P3 Add("Papers")
Papers
P4 Select("year > 2005")
Papers year > 2005
P5 Add("Authors")
Authors
P6 Add("Institutions")
Institutions
P7 Select("country like '%Korea%'")
Institutions country like '%Korea%'
P8 Shift("Authors")
Corresponding User-Level Actions
U1 Open("Conferences")
U2 Seeall("SIGMOD", "Papers")
U3 Filter("year>2005")
U4 Pivot("Authors")
Choose a table to start a query
Authors
Conferences
Institutions
Papers
Conferences
Papers filtered by Conferences.acronym = 'SIGMOD'
Papers filtered by Conferences.acronym = 'SIGMOD' AND year > 2005
Filters
+Add
Apply
Cancel
Filter
Pivot
Sort Asc
Before aggregation
Before pivoting
Expand
Close
Authors filtered by Papers.Conferences.acronym = 'SIGMOD' AND Papers.year > 2005
After pivoting
After aggregation
Click to preview
Sort by Count Desc

Clicking "Conferences" table in default view lists all conferences

Clicking paper count at the end of the row for SIGMOD lists all SIGMOD papers

Opening the filter window and specifying a condition filters papers down to those published after 2005

Clicking pivot button groups authors and ranks them by paper count;

The result shown at bottom right (*Divesh Srivastava* ranked first).

Previous result preserved at top right to help users interpret transformation.

Figure 7: An example of incrementally building a complex query: *find a list of researchers who have published papers at SIGMOD after 2005 and are currently working at institutions in Korea*. **Left**: constructing the query through a series of *ETable* primitive operators. **Right**: corresponding user actions in the interface that invoke the operators (Section 6.1 describes the user-level actions in detail). User actions for the operators P6-P8, similar to the others shown in the figure, are omitted for brevity.

3. **Adding a node type.** Another node type can be added to a query pattern to examine how it is related to the current primary node type. It corresponds to adding a join operator in the relational model. Selecting one of the node types that are linked to the primary node type τ_a by an edge type ρ_k (i.e., $source(\rho_k) = \tau_a$), adds it to the participating node types in the current query Q.

$$Add(\rho_k, Q) = Q'$$
$$\text{where } \tau'_a = target(\rho_k),\ T' = T \cup \{target(\rho_k)\},$$
$$P' = P \cup \{\rho_k\},\ \text{and } \mathcal{C}' = \mathcal{C} \cup \{\}.$$

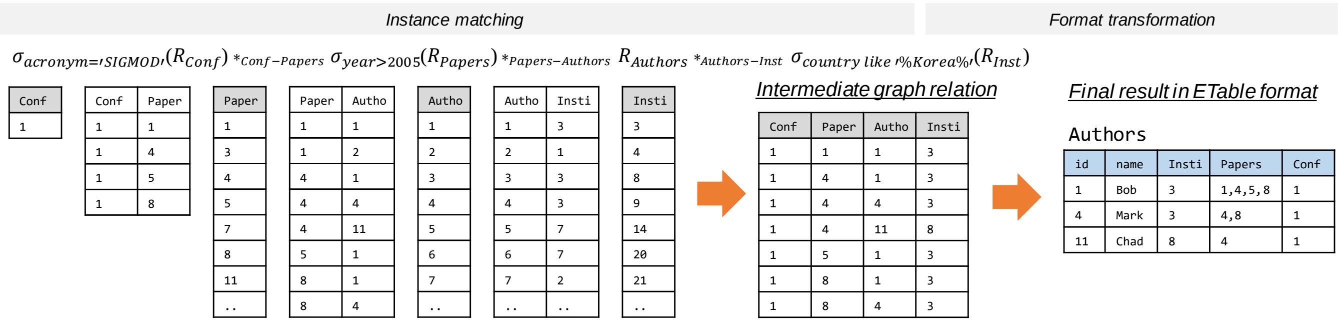
4. **Shifting focus to another participating node type.** The primary node type τ_a can be changed to one of the other participating node types τ_k. It can be thought of as representing the current join result from a different angle.

$$Shift(\tau_k, Q) = Q'$$
$$\text{where } \tau'_a = \tau_k,\ T' = T,\ P' = P,\ \text{and } \mathcal{C}' = \mathcal{C}.$$

The above primitive operators enable us to build any complex queries by incrementally specifying the operators one-by-one. Figure 7 (left) illustrates the query construction process consisting of 8 operators. A new query pattern can be created with *Initiate*; Selection conditions can be added with *Select*, just like writing expressions in WHERE clauses in SQL; and node types can be added with *Add*, just like adding relations to FROM clauses and setting one of them as a GROUP BY attribute. Also, the primary node type can be changed with *Shift*, similar to changing the GROUP BY attribute. A sequence of these operators specified constitutes a query pattern in the *ETable* model. These operators can be invoked by users on the user interface with *user-level actions*, which we will describe details in Section 6.1. The right side of Figure 7 shows how users can specify the same query through the user interface.

5.4 Query Execution

A query pattern is executed to produce a result in the *ETable* format. The execution process is divided into two steps: *instance matching* and *format transformation*. The first step extracts matched node instances from the TGDB instance graph, and the second step transforms a result from the first step into the *ETable* format.


Instance matching
Format transformation
$\sigma_{acronym='SIGMOD'}(R_{Conf}) *_{Conf-Papers} \sigma_{year>2005}(R_{Papers}) *_{Papers-Authors} R_{Authors} *_{Authors-Inst} \sigma_{country\ like\ '\%Korea\%'}(R_{Inst})$
Intermediate graph relation
Final result in ETable format
Authors

Conf
1

Conf	Paper
1	1
1	4
1	5
1	8

Paper
1
3
4
5
7
8
11
..

Paper	Autho
1	1
1	2
4	1
4	4
4	11
5	1
8	1
8	4

Autho
1
2
3
4
5
6
7
..

Autho	Insti
1	3
2	1
3	3
4	3
5	7
6	7
7	2
..	..

Insti
3
4
8
9
14
20
21
..

Conf	Paper	Autho	Insti
1	1	1	3
1	4	1	3
1	4	4	3
1	4	11	8
1	5	1	3
1	8	1	3
1	8	4	3

id	name	Insti	Papers	Conf
1	Bob	3	1,4,5,8	1
4	Mark	3	4,8	1
11	Chad	8	4	1

Figure 8: *ETable* query execution process consists of two steps: (1) the instance matching step which extracts matched instances from the instance graph and (2) the format transformation step which transforms the instances into the *ETable* format.

5.4.1 Instance Matching

The instance matching process finds a set of matched instances for a given query pattern. Formally, it returns a *graph relation* R^G, which consists of a set of tuples, each of which contains a list of node instances in the database instance graph. The graph relation is generated with an *instance matching function* $m(Q)$, which consists of a series of operations. The operations constitute primitives which make up a *graph relation algebra*.

A *graph relation* R^G, similar to a relation in the relational model, consists of a set of tuples with a set of attributes. The schema of the graph relation is defined as a set of node types $\mathcal{A} = (A_1, ..., A_n)$ where $A_i \in \mathcal{T}$. In other words, each attribute A_i corresponds to a node type. The node type τ_j determines the domain of the attribute (i.e., $domain_i = \{v|v \in V_j\}$). A *base graph relation* is defined as a graph relation with a single attribute. In other words, each node type $\tau_1, ..., \tau_n$ produces a base graph relation $R_1^G, ..., R_n^G$. A *non-base graph relation* can be created by applying the following graph relation operators to the base graph relations.

1. **Selection.** It filters tuples of a graph relation R using a selection condition C_i applicable to one of the node types A_i.

$$\sigma_{C_i}(R^G) = \{r|r \in R^G \wedge r[A_i] \text{ satisfies } C_i\}.$$

2. **Join.** It joins two graph relations R_1 and R_2 using edge types ρ_k. The attributes of the created graph relation is a concatenation of the attributes of the two graph relations.

$$R_1^G *_{\rho_k} R_2^G = \{(r_1, r_2)|r_1 \in R_1^G \wedge r_2 \in R_2^G \\ \wedge source(\rho_k) \in \mathcal{A}_1 \wedge target(\rho_k) \in \mathcal{A}_2\}.$$

We use a symbol, $*$, to differentiate it from the relational correspondence, $\bowtie$, and not to be confused with natural join.

3. **Projection.** It removes all attributes of the graph relations except the given attribute. Duplicated rows are eliminated.

$$\Pi_{A_i}(R^G) = \{r[A_i]|r \in R^G\}.$$

These operators enable us to define an instance matching function $m(Q)$. In fact, this function only requires the *Selection* and *Join* operators: the *Projection* operator will be used later in the format transformation step.

Definition 4. **Instance Matching.** Given a *ETable* query pattern $Q(\tau_a, T, P, \mathcal{C})$, a matching function m returns a graph relation R^G containing node instances in the instance graph $\mathcal{G}_I$.

$$m(Q) = \sigma_{C_1}(R_1^G) *_{p_1} \sigma_{C_2}(R_2^G) *_{p_2} ... *_{p_{n-1}} \sigma_{C_n}(R_n^G),$$

where R_i^G is a base graph relation obtained from a node type $t_i \in T$, i.e., $R_i^G = \{v|v \in V \wedge type(v) = t_i\}$, $C_i \in \mathcal{C}$ is a selection condition for R_i, and $p_i \in P$ is one of the edge types that joins graph relations on both sides, i.e., $p_i = \{p|p \in P \wedge source(p) \in \{t_1, ...t_i\} \wedge target(p) \in \{t_{i+1}, ...t_n\}\}$.

Figure 8 (left) illustrates the instance matching process. It returns a graph relation, which is an intermediate format to be transformed into the *ETable* format.

5.4.2 Format Transformation

A graph relation obtained from the *instance matching* function is transformed into the *ETable* format. We describe how rows and columns of *ETable* are determined from it.

The rows of *ETable* consist of nodes of the primary node type, filtered by all selection conditions in the query pattern. They are extracted from the instance matching result:

$$R = \{v|v \in \Pi_{\tau_a}(m(Q(\tau_a, T, P, \mathcal{C})))\}.$$

Given the result of the instance matching function, all attributes except the attribute representing the primary node type are discarded, and then, each of distinct node in that column becomes a row.

ETable has three types of columns to present rich information for each row. In addition to the attributes of the primary node types, which we call *base attributes* $\mathcal{A}_b$, we introduce two other types of columns for presenting a set of entity references: *participating node columns*, $\mathcal{A}_t$, and *neighbor node columns*, $\mathcal{A}_h$.

1. **List of base attributes** $\mathcal{A}_b$: It is a full set of the attributes A of the primary node type τ_a. The value of the column $A_j \in \mathcal{A}_b$ would be a single value:

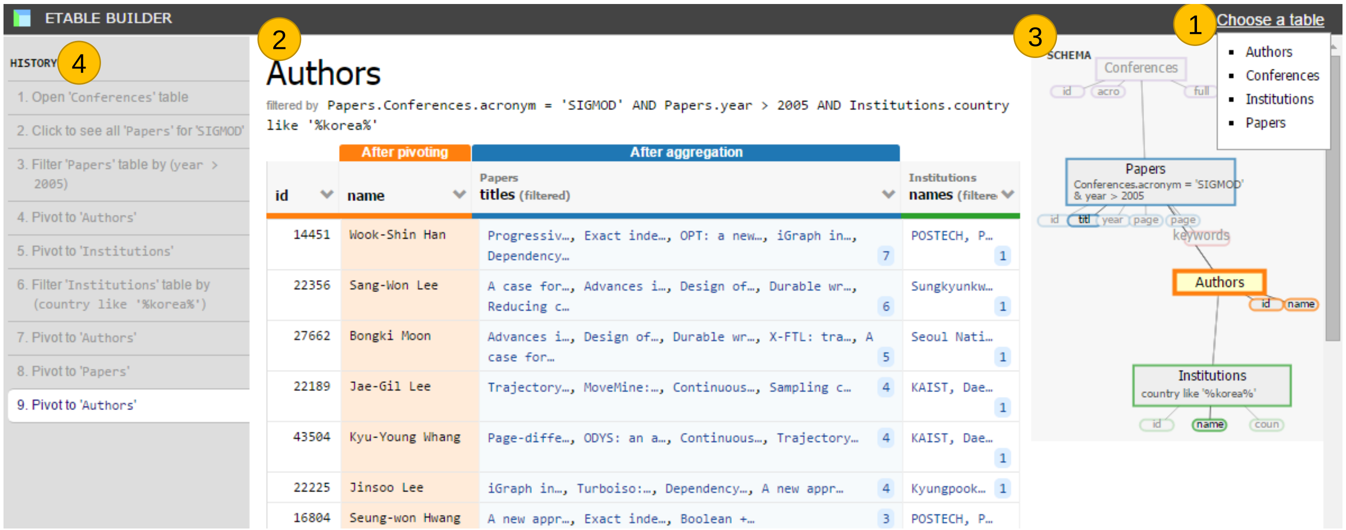
$$r[A_j] = v[A_j].$$

2. **List of participating node types** $\mathcal{A}_t$: It is a set of all the node types T in the query pattern, except the primary node type τ_a, i.e., $\mathcal{A}_t = \{\tau|\tau \in T \wedge \tau \neq \tau_a\}$. The value of the column $A_j \in \mathcal{A}_t$ would be a set of entity references:

$$r[A_j] = \{u|u \in V \wedge A_j = type(u) \\ \wedge \Pi_{type(u)}\sigma_{\tau_a=r}(m(Q))\}.$$

3. **List of neighbor node types** $\mathcal{A}_h$: It is a set of all the neighboring node types of the primary node type τ_a in the schema graph regardless of the query pattern, i.e., $\mathcal{A}_h = \{(\rho, \tau)|\tau \in \mathcal{T} \wedge \rho \in \mathcal{P} \wedge source(\rho) = \tau_a \wedge target(\rho) = \tau\}$. The value of the column $A_j \in \mathcal{A}_h$ would be a set of nodes references:

$$r[A_j] = \{u|u \in V \wedge e \in E \wedge A_j = (type(e), type(u)) \\ \wedge u = target(e) \wedge r = source(e)\}.$$



ETABLE BUILDER

HISTORY

1. Open 'Conferences' table
2. Click to see all 'Papers' for 'SIGMOD'
3. Filter 'Papers' table by (year > 2005)
4. Pivot to 'Authors'
5. Pivot to 'Institutions'
6. Filter 'Institutions' table by (country like '%korea%')
7. Pivot to 'Authors'
8. Pivot to 'Papers'
9. Pivot to 'Authors'

Authors

filtered by Papers.Conferences.acronym = 'SIGMOD' AND Papers.year > 2005 AND Institutions.country like '%korea%'

	After pivoting	After aggregation	
id	name	Papers titles (filtered)	Institutions names (filtere
14451	Wook-Shin Han	Progressiv…, Exact inde…, OPT: a new…, iGraph in…, Dependency… 7	POSTECH, P… 1
22356	Sang-Won Lee	A case for…, Advances i…, Design of…, Durable wr…, Reducing c… 6	Sungkyunkw… 1
27662	Bongki Moon	Advances i…, Design of…, Durable wr…, X-FTL: tra…, A case for… 5	Seoul Nati… 1
22189	Jae-Gil Lee	Trajectory…, MoveMine:…, Continuous…, Sampling c… 4	KAIST, Dae… 1
43504	Kyu-Young Whang	Page-diffe…, ODYS: an a…, Continuous…, Trajectory… 4	KAIST, Dae… 1
22225	Jinsoo Lee	iGraph in…, Turboiso:…, Dependency…, A new appr… 4	Kyungpook… 1
16804	Seung-won Hwang	A new appr…, Exact inde…, Boolean +… 3	POSTECH, P…

Choose a table

- Authors
- Conferences
- Institutions
- Papers

Figure 9: The *ETable* interface consists of (1) the default table list for initiating a query, (2) the main view presenting query results, (3) the schema view showing a query pattern, and (4) the history view listing operators specified by users. Users can build queries and explore databases by directly interacting with the interface.

Figure 8 (right) illustrates the results produced from the format transformation process. The first two columns are base attributes, and the rest of the columns are participating node columns. We omit neighbor node columns as some of these columns are the same as the participating node columns.

By transforming the graph relation into the *ETable* format, we compactly present join query results without duplications. Each row of *ETable* is uniquely determined by a node of a primary node type. The participating node columns show all the other entity types in the query pattern with respect to the primary node type. This transformation process is similar to setting one of the relations as a GROUP BY attribute in SQL, but while GROUP BY aggregates the corresponding instances into a single value (i.e., COUNT, AVG), *ETable* presents a list of the corresponding instances as entity references. The neighbor node columns are also useful for describing the rows of the *ETable*, although information in these columns is not obtained from the graph relation. These columns enable users to browse one-to-many or many-to-many relationships. Moreover, they provide users with a preview of possible new join operations as it presents all the join candidates. For instance, a *ETable* in Figure 1 consists of many neighbor node columns (e.g., Authors) that helps users browse rich information about each paper.

6. INTERFACE & SYSTEM DESIGN

ETable's interface (Figure 9) consists of four components: (1) the default table list, (2) the main view, (3) the schema view, and (4) the history view. The *default table list* presents a list of entity types in the schema graph. Users can pick one from the list to initiate a query. The *main view* presents an *ETable* executed based on a query pattern which is graphically shown over the *schema view*. Users can directly interact with the main view to update the current query. The list of actions specified by users is presented on the *history view*, which allows users to revert to a previous state.

6.1 User-Level Actions

Users can update the current query pattern by directly interacting with *ETable* via **user-level actions**. As shown in Figure 7, these actions in turn invoke their corresponding primitive operators (discussed in Section 5.3).

1. **Open a new table.** Users can open a new table by clicking a node type τ_k on the default table list. The action invokes the $Initiate(\tau_k)$ operator (Fig 7: action U1).

$$\textbf{Open}(\tau_k) = Initiate(\tau_k).$$

2. **Filter.** Users can filter the rows of the current *ETable* by inducing selection conditions via a popup window at the column header (Fig 7: action U3). Besides the base attributes, users can also filter rows by the labels of the neighbor nodes columns (e.g., authors' names), which is translated into subqueries. We currently provide only a conjunction of predicates, but it is straightforward to provide disjunctions and more operations. The action invokes the *Select* operator.

$$\textbf{Filter}(C, R) = Select(C, R).$$

3. **Pivot.** Users can change the primary node type by clicking the pivot button on the context menu for neighbor or participating node columns. It calls the *Add* operator if the column is the neighbor node type (Fig 7: action U4); it performs the *Shift* operator if it is the participating node type.

$$\textbf{Pivot}(\rho_l, R) = Add(\rho_l, R),$$
$$\text{or } \textbf{Pivot}(\tau_k, R) = Shift(\tau_k, R).$$

4. **See a particular node.** When users are interested in one of the entity references, they can click it to create a new *ETable* consisting of a single row presenting the clicked entity. Unlike the above actions, it invokes two primitive operators: it initiates a new *ETable*, and then perform the *Select* operator to show the single node. For the clicked node v_k:

$$\textbf{Single}(v_k, R) = Select(C, type(v_k), Initiate(type(v_k))),$$
$$\text{where } C = \{u | u = v_k\}.$$

5. **See all related nodes.** When users are interested in a full list of entity references, they can click a number (i.e., entity reference count) in the right corner of a cell (Fig 7: action U2). It also encapsulates two primitive operators. The operators invoked are different depending on whether the selected

Task	Category	#Relations
1. Find the year that the paper titled 'Making database systems usable' was published in.	Attribute	1
2. Find all the keywords of the paper titled 'Collaborative filtering with temporal dynamics'.	Attribute	2
3. Find all the papers that were written by 'Samuel Madden' and published in 2013 or after.	Filter	3
4. Find all the papers written by researchers at 'Carnegie Mellon University' and published at the KDD conference.	Filter	5
5. Which institution in South Korea has the largest number of researchers?	Aggregate	2
6. Find the top 3 researchers who have published the most papers in the SIGMOD conference.	Aggregate	4

Table 2: List of tasks. Task 1 & 2 retrieve attribute values, task 3 & 4 filter entities, task 5 & 6 perform aggregations.

column is *neighbor* or *participating* node column. For the *neighboring node column* ρ_l of v_k:

$$\textbf{Seeall}_h(v_k, \rho_l, R) = \mathit{Add}(\rho_l, \mathit{Select}(C, \mathit{type}(v_k), R)), \text{ where } C = \{u | u = v_k\},$$

and for the *participating node column* t_l:

$$\textbf{Seeall}_t(v_k, t_l, R) = \mathit{Shift}(t_l, \mathit{Select}(C, \mathit{type}(v_k), R)), R)), \text{ where } C = \{u | u = v_k\}\}.$$

ETable supports additional actions that help with database exploration, such as: (1) **Sort rows** based on the values in a column; (2) **Hide/show columns** to reduce visual complexity in the interface; and (3) **Revert to previous queries** via the left history panel.

6.2 Architecture

ETable system uses a three-tier architecture, consisting of (1) an interactive user interface front-end that can run in any modern web browsers, written in HTML, JavaScript, and D3.js[2]; (2) a Python-based application server; and (3) a PostgreSQL database backend. The PostgreSQL database stores TGDB schema and instance graphs in four relational tables: `nodes`, `edges`, `node_types`, and `edge_types`. A query pattern for *ETable* is translated into SQL queries that operate on the PostgreSQL database. To efficiently perform queries, we partition a long SQL query into multiple queries consisting of a fewer number of relations to be joined (i.e., each for a single entity-reference column) and merge them.

7. EVALUATION: USER STUDY

To evaluate the usability of *ETable*, we conducted a user study that tests whether users can construct queries quickly and accurately. We compared *ETable* with Navicat Query Builder.[3] Navicat is one of the most popular commercial database administration tools with a graphical query building feature. Graphical builders such as Navicat Query Builder have been commonly used as baseline systems in database usability research [35, 38, 7].

7.1 Experimental Design

Participants. We recruited 12 participants from our university through advertisements posted to mailing lists at our institution. All were graduate students who had taken at least one database course or had industry experience using database systems. The participants rated their experience in SQL, averaging at a score of 4.67 using a 7-point Likert scale (ranged from 3 to 6) with 1 being "having no knowledge" and 7 being "expert", which means most participants considered themselves non-expert database users. None of them had used the graphical query builder before. Each participant was compensated with a $15 gift card.

Data set. We used an academic publication data set used throughout this paper, which we collected from DBLP[4] and ACM Digital Library.[5] It contains about 38,000 papers from 19 top conferences in the areas of databases (e.g., SIGMOD), data mining (e.g,. KDD), and human-computer interaction (e.g., CHI), since 2000. A relational schema was designed using standard design principles, resulting in 7 relations with 7 foreign keys as depicted in Figure 3. As the main focus of this evaluation is on *ETable*'s usability, this data set creates a sufficiently large and complex database for such purpose.

Procedure. Our study followed a *within-subjects design* with two conditions: the *ETable* and Navicat conditions. Every participant first completed six tasks in one condition and then completed another six tasks in the remaining condition. The orders of the conditions were counterbalanced, resulting in 6 participants in each ordering. We generated two matched sets of tasks (6 tasks in each set) differing only in their specific values used for parameters such as the title of the paper. Before the participants were given the tasks to carry out for each condition, they went through a 10-minute tutorial for the tool they would use. For each task, the participants could ask clarifying questions before starting, and they had a maximum of 5 minutes to complete each task. After the study, they completed a questionnaire for subjective ratings and qualitative feedback. Each study lasted for about 70 minutes. Participants completed the study using Chrome browser, running on a Windows desktop machine, with a 24-inch monitor at a 1920x1200 resolution.

Tasks. We carefully generated two matched sets of 6 tasks that cover many database exploration and querying tasks. Table 2 shows one set (the other set is similar). The tasks fall into three categories: finding attribute values (Tasks 1 & 2); filtering (Tasks 3 & 4); aggregation (Tasks 5 & 6). The tasks were designed based on prior research studies and their categorization of tasks. Specifically, our categories are based on those used in database and HCI research [5, 34], and our tasks vary in difficulty as in [33].

Measurements. We measured participants' task completion times. If a participant failed to complete a task within 5 minutes, the experimenter stopped the participant and recorded 300 seconds as the task completion time. After completing tasks for both conditions, the participants filled out a post-questionnaire that asked for their subjective ratings about *ETable* (10 questions) and their subjective preference between two conditions (7 questions).

7.2 Results

Task completion times. The average task times for *ETable* were faster than those for Navicat for all six tasks. Figure 10 summarizes the task time results. We performed two-tailed paired t-tests. The differences were statistically significant for Tasks 1, 3, 5, and 6 ($p < 0.005$) and marginally significant for Tasks 2 and 4

[2] `https://d3js.org/`

[3] `http://www.navicat.com/`

[4] `http://dblp.uni-trier.de/`

[5] `http://dl.acm.org/`

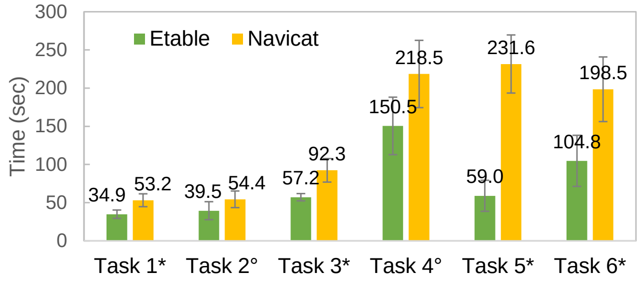


Figure 10: Average task completion time for each task. The error bars represent 95% confidence intervals for the mean. Participants performed the tasks faster with *ETable* than with Navicat. The * and $^\circ$ symbols indicate 99% and 90% statistical significance in the two-tailed paired t-tests, respectively.

($p = 0.052$, $p = 0.053$, respectively). The results of Task 2 may be explained by an outlier participant who did not understand the requirement that each row of the final results must represent a different keyword. Although Task 4 involves the highest number of operations that require participants to spend significant time in interpreting intermediate results before applying the next operators, *ETable* helped participants complete this task over 30% faster than Navicat.

The task completion times for *ETable* generally have low variance. The larger variance in Navicat is mainly due to syntax errors that the participants faced. Many participants, who are non-database experts, could not recall some SQL syntax and had trouble debugging errors. In particular, they had trouble specifying GROUP BY queries in Navicat. For example, many participants did not specify a GROUP BY attribute in their SELECT clauses in their first attempts. We also observed that many Navicat participants were overwhelmed by the complexity of the syntax of join queries [29] and preferred to specify new SQL queries from scratch instead of debugging existing ones when their original queries failed. Unlike graphical query builders such as Navicat, *ETable* helps non-experts gradually build complex queries without having to know the exact query syntax.

Subjective ratings. We asked participants to rate various aspects of *ETable* using 7-point Likert scales (7 being "strongly agreed"). Their subjective ratings were generally very positive (see Table 3). In particular, all participants found *ETable* easy to learn (i.e., rated 6 or 7), and almost all participants (11/12) found *ETable* easy to use and helpful for browsing data in databases. They also enjoyed using *ETable* (10/12) and would like to use software like *ETable* in the future (11/12). In response to the "*helpful to interpret and understand results*" question, one participant commented that "*there are too many attributes ..., which is not easy to interpret.*" To address this, as future work, we plan to develop techniques to rank and select the most important columns to show whenever a table has a large number of columns [47].

We also asked participants to compare *ETable* and Navicat in 7 aspects. All participants indicated that *ETable* was easier to learn and was more helpful in browsing and exploring data. A majority of participants liked *ETable* more (11/12) and found it easier to use (10/12). They would choose to use *ETable* in the future (10/12) and felt more confident using it (8/12). Half of the participants answered that *ETable* is more helpful in finding specific data than Navicat. This result was expected because *ETable*'s innovation focuses more on supporting data exploration.

Qualitative feedback. We asked participants about the features they liked about *ETable*. Many participants (9/12) explicitly mentioned the "pivot" feature. They said that the pivot feature enabled them to easily specify complex join queries. One participant said "*I also loved the pivot feature ... having multiple pivots throughout the course of forming a query. I messed up a query, but could still find the right answer by doing an appropriate pivot.*" In addition, many participants said that *ETable* provides an intuitive view to users. One said "*It is easy to see data from the perspective of what the users want to see/retrieve ...*" Another said "*Visually, I was able to see ... the effects of the SQL operations, which made it easier to use and verify intermediate results.*"

Question	Avg.
1. Easy to learn	6.42
2. Easy to use	6.33
3. Helpful to locate and find specific data	6.25
4. Helpful to browse data stored in databases	6.67
5. Helpful to interpret and understand results	5.58
6. Helpful to know what type of information exists	6.00
7. Helpful to perform complex tasks	6.00
8. Felt confident when using *ETable*	5.92
9. Enjoyed using *ETable*	6.42
10. Would like to use software like *ETable* in the future	6.50

Table 3: Subjective ratings about *ETable* using 7-point Likert scales (7: *Strongly Agreed*. 1: *Strongly Disagreed*).

8. EXPRESSIVENESS

This section discusses the expressiveness of the *ETable* model. We will first express the overall functionality of the *ETable* queries as a general SQL query pattern. By doing so, we will show how typical join queries can be translated into *ETable* queries, through multiple steps (similar to [35, 15]), demonstrating *ETable*'s expressiveness. Any join queries involving only FK-PK relationships on a relational database schema that meets *ETable*'s assumptions (detailed in Appendix) can be translated into an *ETable* query that operates on TGDB.

The overall functionality of *ETable* queries can be expressed as the following general SQL query pattern:

$$\begin{array}{l}
\texttt{SELECT } \tau_a\texttt{.*, ent-list(}t_1\texttt{), ent-list(}t_2\texttt{), } \ldots \\
\texttt{FROM } t_1, t_2, \ldots \\
\texttt{WHERE } \mathit{source}(p_1) = \mathit{target}(p_1) \texttt{ AND } \mathit{source}(p_2) = \\
\qquad \mathit{target}(p_2) \texttt{ AND } \ldots \texttt{ AND } C_1 \texttt{ AND } C_2 \texttt{ AND } \ldots \\
\texttt{GROUP BY } \tau_a\texttt{;}
\end{array}$$

where `ent-list` presents a list of corresponding entity references, similar to the `json_agg` operator in PostgreSQL.[6] Each of the four components in an *ETable* query (i.e., primary node type τ_a, node types T, edge types P, and selection conditions $\mathcal{C}$) maps to a clause in SQL: primary node type to GROUP BY clause; node_types to FROM clause; edge_types to join conditions; selection conditions to WHERE clause.

Following the above mappings, we now follow the approach similar to that in [35, 15] to show that *ETable* can expressively handle typical join SQL queries, through a step-by-step translation. That is, for any SQL join query following the above pattern, there exists an equivalent *ETable* query.

1. Transforms a relational algebra join expression ($R \bowtie R \bowtie \ldots$) to a graph relation correspondence $R^G * R^G * \ldots$ (described in Section 5.4) by analyzing the list of relations in

[6] `http://www.postgresql.org/docs/9.4/static/functions-aggregate.html`

the FROM clause, and the join conditions in the WHERE clause. (Each R^G is a node type; each $*$ an edge type.)

2. Applies the original selection conditions to the TGDB's node types;
3. If there is a *group by* attribute, transform it to the graph's primary node type; otherwise, if no *group by* attribute exists, arbitrarily set a primary node type.

ETable can express typical join queries consisting of the core relational algebra (i.e., relational algebra expression that does not contain set operations), which accounts for a large number of the database workloads. *ETable* additionally lets users choose a *primary node type* from the list of selected relations, and introduces the *entity-reference columns* (i.e., represented as `ent-list` in the above SQL pattern) to effectively present join queries. This paper focuses on the critical usability challenge that arises when joining several tables. In our future work, we plan to further increase *ETable*'s expressiveness of the presentation model to the full set of operators in the relational algebra, through introducing additional operators to support more complex queries (e.g., set operations, complex aggregations, etc.).

9. CONCLUSIONS

We proposed *ETable*, a new presentation data model for interactively exploring relational databases. The enriched table representation of *ETable* generates a holistic, interactive view of databases that helps users browse relevant information at an entity-relationship level. By directly interacting with the interface, users can iteratively specify operators, enabling them to incrementally build complex queries and navigate databases. *ETable* outperformed a commercial graphical query builder in a user study, in both speed and subjective ratings across a range of database querying tasks.

This work takes a first step towards developing a practically usable, interactive interface for relational databases, and opens up many interesting opportunities. Future research directions include: (1) incorporating more operations to further improve expressive power (e.g., set operations); (2) accelerating the execution speed of updated queries (e.g., by reusing intermediate results); (3) leveraging machine learning techniques to rank and select important columns to display. The above ideas could usher a new generation of interactive database exploration tools that will benefit all database users.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-1148903 and the NSF under Grant No. IIS-1563816.

11. REFERENCES

[1] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, et al. The beckman report on database research. *ACM SIGMOD Record*, 43(3):61–70, 2014.

[2] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, et al. The lowell database research self-assessment. *CACM*, 48(5):111–118, 2005.

[3] A. Abouzied, J. Hellerstein, and A. Silberschatz. Dataplay: interactive tweaking and example-driven correction of graphical database queries. In *UIST*, pages 207–218. ACM, 2012.

[4] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16. IEEE, 2002.

[5] R. Amar, J. Eagan, and J. Stasko. Low-level components of analytic activity in information visualization. In *IEEE INFOVIS*, pages 111–117. IEEE, 2005.

[6] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1, 2008.

[7] E. Bakke and D. Karger. Expressive query construction through direct manipulation of nested relational results. In *SIGMOD*, pages 1377–1392. ACM, 2016.

[8] E. Bakke, D. R. Karger, and R. C. Miller. Automatic layout of structured hierarchical reports. *IEEE TVCG*, 19(12):2586–2595, 2013.

[9] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin Cummings, 1992.

[10] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440. IEEE, 2002.

[11] M. Buoncristiano, G. Mecca, E. Quintarelli, M. Roveri, D. Santoro, and L. Tanca. Database challenges for exploratory computing. *ACM SIGMOD Record*, 44(2):17–22, 2015.

[12] T. Catarci. What happened when database researchers met usability. *Information Systems*, 25(3):177–212, 2000.

[13] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages & Computing*, 8(2):215–260, 1997.

[14] T. Catarci, G. Santucci, and M. Angelaccio. Fundamental graphical primitives for visual query languages. *Information Systems*, 18(2):75–98, 1993.

[15] T. Catarci, G. Santucci, and J. Cardiff. Graphical interaction with heterogeneous databases. *The VLDB journal*, 6(2):97–120, 1997.

[16] U. Cetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.

[17] K. S.-P. Chang and B. A. Myers. Using and exploring hierarchical data in spreadsheets. In *CHI*, pages 2497–2507. ACM, 2016.

[18] P. P.-S. Chen. The entity-relationship model: toward a unified view of data. *ACM TODS*, 1(1):9–36, 1976.

[19] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD*, pages 1005–1010. ACM, 2009.

[20] R. H. Chiang, T. M. Barron, and V. C. Storey. Reverse engineering of relational databases: Extraction of an eer model from a relational database. *Data & Knowledge Engineering*, 12(2):107–142, 1994.

[21] M. Dork, N. H. Riche, G. Ramos, and S. Dumais. Pivotpaths: Strolling through faceted information spaces. *IEEE TVCG*, 18(12):2709–2718, 2012.

[22] C. Dunne, N. Henry Riche, B. Lee, R. Metoyer, and G. Robertson. Graphtrail: Analyzing large multivariate, heterogeneous networks while supporting exploration history. In *CHI*, pages 1663–1672. ACM, 2012.

[23] J. Fan, G. Li, and L. Zhou. Interactive sql query suggestion: Making databases user-friendly. In *ICDE*, pages 351–362. IEEE, 2011.

[24] S. Few. *Show me the numbers: Designing tables and graphs to enlighten*. Analytics Press Oakland, CA, 2004.

[25] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *SIGMOD*, pages 1061–1066. ACM, 2010.

[26] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. V. Gucht. A graph-oriented object database model. *IEEE TKDE*, 6(4):572–586, 1994.

[27] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[28] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, pages 277–281. ACM, 2015.

[29] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, pages 13–24. ACM, 2007.

[30] H. V. Jagadish, A. Nandi, and L. Qian. Organic databases. In *Databases in Networked Information Systems*, pages 49–63. Springer, 2011.

[31] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1):695–709, 2008.

[32] H. Kang, C. Plaisant, B. Lee, and B. B. Bederson. Netlens: iterative exploration of content-actor network data. *Information Visualization*,

6(1):18–31, 2007.

[33] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[34] F. Li, T. Pan, and H. V. Jagadish. Schema-free sql. In *SIGMOD*, pages 1051–1062. ACM, 2014.

[35] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, pages 417–428. IEEE, 2009.

[36] Z. Liu, S. B. Navathe, and J. T. Stasko. Network-based visual analysis of tabular data. In *IEEE Conf. Visual Analytics Science & Technology*, pages 41–50. IEEE, 2011.

[37] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[38] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. *PVLDB*, 7(4):289–300, 2013.

[39] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM TODS*, 13(4):389–417, 1988.

[40] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information systems*, 11(2):137–147, 1986.

[41] J. F. Sequeda, M. Arenas, and D. P. Miranker. On directly mapping relational databases to rdf and owl. In *WWW*, pages 649–658. ACM, 2012.

[42] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16:57–69, 1983.

[43] M. Singh, M. J. Cafarella, and H. V. Jagadish. Dbexplorer: Exploratory search in databases. In *EDBT*, pages 89–100, 2016.

[44] M. Stonebraker and D. Moore. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., 1995.

[45] Y. Sun and J. Han. Mining heterogeneous information networks: principles and methodologies. *Synthesis Lectures on Data Mining & Knowledge Discovery*, 3(2):1–159, 2012.

[46] J. Tyszkiewicz. Spreadsheet as a relational database engine. In *SIGMOD*, pages 195–206. ACM, 2010.

[47] X. Yang, C. M. Procopiuc, and D. Srivastava. Summarizing relational databases. *PVLDB*, 2(1):634–645, 2009.

[48] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

APPENDIX

A. DATABASE TRANSLATION

This section describes a procedure for translating relational databases into database schema and instance graphs in the typed graph model. Our approach is based on the reverse engineering literature [9, 20, 15, 41]. We note that the following process cannot be applied to any relational schema, as relational schema do not contain all the semantics, but is a guideline for translations. We make several assumptions as in the literature [15, 9]. First, all the relations are in BCNF or 3NF. Second, there are no ternary relationships: all the relationships are binary. Third, for relationship relations, we assume that all attributes are foreign keys of the relations that participate in the relationship. Any attributes of the relationship itself are ignored. Finally, a relation representing a multivalued attribute always consists of two columns.

Identifying entity relations. This step identifies entity relations from a set of relations. Informally, entity relations refer to relations constructed from entity types in the entity-relationship model. We define an *entity relation* as a relation whose primary key does not contain a foreign key or a key inclusion dependent on any other attribute in any other relation [15, 9]. For each of the identified entity relations, the following process is performed.

1. A relation becomes a node type in the schema graph.
2. The relation name becomes the name of the node type.
3. All the attributes of the relation become the attributes of the node type.
4. One attribute selected by users becomes a *label attribute* for the node type.

We currently determine the *label attribute* based on a combination of heuristics, such as data type (e.g., text generally more interpretable than numbers) and cardinality. However, this label selection task is hard to fully automate. Thus, we also allow users to manually pick a desired label attribute. In our future work, we plan to investigate mixed-initiative approaches that allow human and computer to work together, so that we would provide an initial guess and recommend possible alternatives based on the heuristics, and allow the users to select attributes that are most meaningful to them.

Identifying 1:1 and 1:n relationships. Foreign keys, which are used to represent one-to-one and one-to-many relationships between entity relations in the relational model, are used to identify relationships between entity relations found above. For each foreign key, the following process is performed.

1. Each foreign key becomes an edge type in the schema graph. The source node would be a node type representing a relation containing the foreign key. The target node would be a node type representing a relation which the foreign key refers to.
2. Unless the source and target node types are the same, the edge types are duplicated with a reverse direction.
3. The label is defined as the name of the target node type. If the label is used by another edge type, a slightly different label will be created.

Identifying many-to-many relationships. Many-to-many relationships are represented as a separate table in the relational model. We identify these tables whose primary key is a concatenation of primary keys of two other entity relations. For each of the identified relationship relations, the following process is performed.

1. Each relationship relation becomes an edge type in the schema graph. The two other associated entity relations become source and target nodes.
2. The remaining steps are the same as above (i.e., Steps 2 & 3)

Identifying multivalued attributes. The relational model stores multivalued attributes in separate relations. We identify such relations. We assume these relations consist of only two attributes where both attributes make up the primary key and the first attribute is a foreign key to an entity relation. For each of this case, the following process is performed.

1. The second attribute becomes a node type in the schema graph.
2. The node type has one attribute which refers to itself. The label column is this only attribute.
3. An edge type is also created from the node type representing the entity relation to the newly created node type. It will be duplicated in a reverse direction.

Identifying categorical attributes. This step of identifying categorical attributes is optional, but we find it useful. People often perform GROUP BY operations over categorical attributes, and this step helps them perform such analysis. Any of the attributes of the entity relations could be selected by users. Often, attributes with low cardinality (e.g., less than 30) can be candidates for categorical attributes. For each of the selected attributes, the following process is performed.

1. Each attribute becomes a node type in the schema graph.
2. It has one attribute which refers to itself. The label column is this only attribute.
3. An edge type is also created from the node type representing the relation to the newly created node type. It will be duplicated in a reverse direction.

This creates a TGDB schema graph. Under the assumptions we made, the schema graph contains all the information in the original relational schema. Once the schema is translated, it is straightforward to create the corresponding TGDB instance graph.