Reverse Engineering Models from Databases to Bootstrap Application Development

Ankit Malpani[#1i], Philip A. Bernstein[*2], Sergey Melnik[&3i], James F. Terwilliger[*4]

[#]*Indian Institute of Technology Madras, India*

[1]ankit@cse.iitm.ac.in

[*]*Microsoft Research, USA*

[2]Phil.Bernstein@microsoft.com

[4]James.Terwilliger@microsoft.com

[&]*Google Inc., USA*

[3]melnik@google.com

Abstract— **Object-relational mapping systems have become often-used tools to provide application access to relational databases. In a database-first development scenario, the onus is on the developer to construct a meaningful object layer for the application because shipping tools, as ORM tools only ship database reverse-engineering tools that generate objects with a trivial one-to-one mapping. We built a tool, EdmGen++, that combines pattern-finding rules from conceptual modelling literature with configurable conditions that increase the likelihood that found patterns are semantically relevant. EdmGen++ produces a conceptual model with inheritance in Microsoft's Entity Data Model, which Microsoft's Entity Framework uses to support an executable object-to-relational mapping. The execution time of EdmGen++ on customer databases is reasonable for design-time.**

I. Introduction

Object-relational mapping (ORM) tools like Hibernate [6], Oracle TopLink [8], Ruby on Rails [10], and Microsoft Entity Framework [1] have become popular tools for application development. Part of the appeal of these tools is that a developer can construct query and update operations in the native object-oriented model of the application.

When constructing an application from scratch, there are two common scenarios: model-first and database-first. There is mature and robust tool support for the model-first scenario, as generating a database schema from an object or conceptual model is generally an algorithmic process [6,8]. However, it is more difficult to create a useful object model that captures inheritance and other semantics that is present in a database but hidden in its schema and data. Hence, it is often performed manually. Each ORM tool listed above ships with a tool that generates an object model from a database to bootstrap development, but the generated model is isomorphic to the database schema — a class per table and a relationship per foreign key, with no inheritance or recognition of other relationships.

Research projects spanning several decades have attempted to develop algorithms for generating models with inheritance from database instances in an automated fashion [4,7,9]. These approaches use patterns discovered in schema or data to infer data semantics and constraints, and exploit those constraints to infer the presence of constructs in the client data model. Neither dedicated modelling tools nor ORM reverse engineering tools employ pattern-finding techniques.

Identifying potential inheritance hierarchies in a database is beneficial because such hierarchies leverage the expressive power of the conceptual or object model, and are more expressive than the storage relations. However, strictly applying pattern-matching rules to a database instance may generate a conceptual model with unmanageably many constructs and inflated inheritance hierarchies. Generated sub-types can be statistically insignificant or meaningless because a small amount of data happens to match a pattern by accident. In addition, rules involve queries that run on schema and data, and thus have an associated cost in disk I/O that increases with the size of the database. Simply running rules indiscriminately any time a rule can possibly be applied is not efficient and may not yield useful results.

The Entity Framework (EF), part of the recent release of ADO.Net from Microsoft, gives developers object-oriented access to relational data by deriving classes from an extended entity-relationship conceptual representation of data called the Entity Data Model (EDM). We have developed a tool called EdmGen++ that generates a valid EDM conceptual model and a correct mapping between the model and the database to provide developer access. Pre- and post-conditions for each rule eliminate inheritance relationships that are unlikely to be semantically meaningful. The pre-conditions also improve performance by reducing the number of patterns that match a rule, applying rules judiciously over a large search space.

The contributions of this work are:

- A set of pattern-matching rules inspired by prior work in model generation that identify EDM constructs in a database instance, and an explicit rule ordering that improves performance.
- Pre- and post-conditions for rule execution to determine if a rule is likely to produce useful results.
- Details of a prototype implementation[ii], and a performance analysis of EdmGen++ on customer databases.

In Section 2, we provide an overview of our demonstration of the EdmGen++ tool. Section 3 provides some insight into

[i] Work performed while at Microsoft Research

[ii] A version of our prototype is available for public download at: http://code.msdn.microsoft.com/EdmGen2

the implementation of EdmGen++ and a performance analysis of the tool running on real-world data sets. Section 4 concludes by comparing EdmGen++ to related work.

II. What is Demonstrated

We demonstrate a database-first application engineering scenario, starting with nothing but a database instance. After running EdmGen++, the developer is given a semantically meaningful model for that database, classes corresponding to the model, and a valid mapping to those classes that supports queries and updates to the database. We show application code that issues queries and updates against models generated from example databases, returning live results.

We use the terms entities, associations, and properties to refer to constructs in the client-side EDM model. We use tables, foreign keys, and columns to refer to constructs in the server-side relational database instance.

Consider the database instance in Figure 1(a). The schema contains three tables connected by two foreign keys, both of which have one-to-one cardinality since their source columns are also primary keys. Starting with the schema and data of this database instance, we can construct the conceptual model found in Figure 1(b). The relationship between Figure 1(a) and Figure 1(b) can be explained as follows:

- Table Person is broken into entities Person, Student, and Alumnus based on the values stored in the column type. This relationship is an example of a table-per-hierarchy pattern for storing inheritance trees as tables.
- Table Details is vertically merged into Person entity.
- Table Academics is vertically merged into Student entity.

Each of the above relationships arises from a pattern in the data in Figure 1(a). For instance, the table-per-hierarchy arrangement arises from identifying a relationship between values in the "type" column and the presence or absence of values in other columns.

Figure 1(c) shows another conceptual model for the same database instance. The conceptual model is valid in the sense that it can map to the tables, columns, foreign keys, and distributions of null values in the database, but this conceptual model contains entities that Figure 1(b) does not. It is debatable whether these two additional entities are significant; for instance, the MinorStudent entity has many fewer rows than its parent, and HonorStudent has only a single property. In our demonstration, we show how a user can specify parameters to EdmGen++ to quantify what it means to be statistically significant.

III. Implementation

The core of our implementation is a set of rules that mine the database for patterns in data and schema that provide evidence of inheritance or partitioning relationships. The general algorithm is:

1. Generate a default model and one-to-one mapping from the database, where tables map to entities, columns map to properties, and foreign keys map to associations. If any table consists only of foreign keys to two other tables, translate it into a many-to-many association instead.
2. For each of the rules, loop over the constructs (entities, associations, and properties) in the current conceptual model. If the construct matches the pre-conditions for the rule and the pattern, then evaluate the rule.
3. If the rule has post-conditions, evaluate them on the output of the rule. If the conditions fail, roll back the effect of the rule.
4. Repeat steps 2 and 3 for each of the rules.

A. Rules and Order

Table 1 describes the six rules that our implementation currently uses, in the order that they are applied.

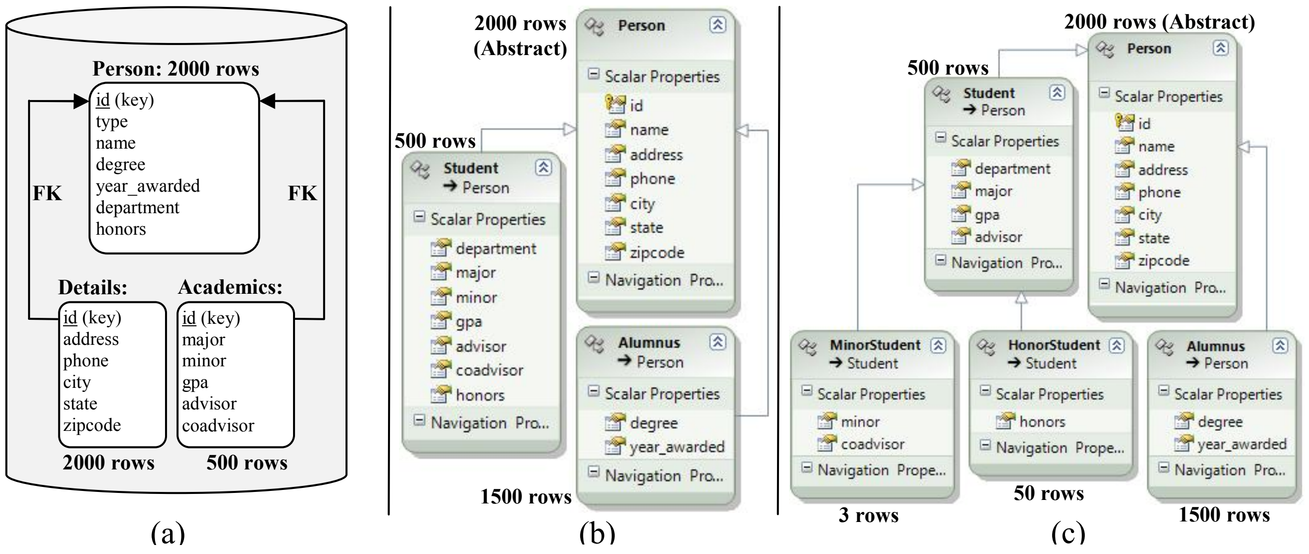

Figure 1. A database schema with row counts (a) and two candidate conceptual models for that database (b, c)

TABLE I
THE ORDERED LIST OF RULES THAT EDMGEN++ EXECUTES. NOTATION $A \le B$ REPRESENTS RELATIONSHIP "A IS A PARENT OF B".

Name	Description	Pattern	Action Overview
TPH1	Table-per-hierarchy pattern: search for discriminator columns	A is an entity, P is a property of A, Count(Distinct A.P) is below some threshold	• Create a new abstract entity E with properties Keys(A). • Create a new entity V, $E \le V$, for each distinct value in A.P. For each non-key property C in A, add property C to V if $\pi_C\sigma_{P=V}A$ contains at least one non-null value. This step effectively partitions E into as many classes as there are elements in A.P's active domain. • If property C belongs to all children of E, move C from children to E. • For each sibling pair C, D of descendants of E, if Properties(C) $\subseteq$ Properties(D), remove D from its parent and introduce $C \le D$.
TPH2	Table-per-hierarchy pattern: search for mutual exclusion relationships between columns	A is an entity, A has at least one property that allows null values	• Use an FP-tree to determine mutual existence (A.P is null iff A.Q is null) and mutual exclusion (A.P is null iff A.Q is not null) • Use the existence and exclusion dependencies to find a set of property sets $P_1,\ldots,P_n$ such that each property set has mutual existence and the set of property sets is mutually exclusive (e.g., if P_2 holds non-null values for row R, then R is null on all other sets). • Partition entity A into entities $B_1,\ldots,B_n$, one for each property set P_i. • Run recursively over the newly-generated entities.
TPC	Horizontal partitioning and Table-per-concrete type pattern: search for tables with overlapping columns and non-overlapping key extents	A, B are entities, Key(A) = Key(B), $\pi_{Key(A)}A \cap \pi_{Key(B)}B = \emptyset$, A and B columns in common above some threshold	• If Properties(A) = Properties(B) and A and B participate in associations with the same entities, construct a new entity M with properties Properties(A) $\cup$ {P}, where P is a provenance column. • If Properties(A) $\subset$ Properties(B), introduce relationship $A \le B$. • Otherwise, create new abstract entity C with properties (Properties(A) $\cap$ Properties(B)), and relationships $C \le A$ and $C \le B$.
PDA	Push down associations: move associations to subtypes	A, B, C are entities, $A \rightarrow B$ $B \le C$, Count($A \bowtie C$) = Count($A \bowtie B$)	• Move association $A \rightarrow B$ to $A \rightarrow C$.
TPT	Table-per-type pattern: use associations to look for inheritance and vertical partitioning	A, B are entities, $A \rightarrow B$ in a one-to-one association	• If Count(A) = Count(B), remove $A \rightarrow B$ and merge entities A and B into a single entity A by taking the union of their columns. • Otherwise, remove $A \rightarrow B$, introduce inheritance relationship $A \le B$.
ABS	Abstraction check: determine if an entity should be marked as abstract	A, $E_1,\ldots,E_n$ are entities, $\forall_i\, A \le E_i$, $\nexists B$ ($B \notin \{E_1,\ldots,E_n\}$ and $A \le B$), $E_1 \cup \ldots \cup E_n = A$	• Set table A to be abstract.

Some details are omitted for brevity; for instance, rule TPH1 also uses associations to look outside a table for discriminator columns. The order of the rules is important for performance reasons, as earlier rules reduce the number of candidate matches for later rules, in general.

Four rules in Table 1 search for patterns that correspond to the three primary methods for mapping inheritance relationships to tables: table-per-hierarchy (TPH), table-per-concrete type (TPC), and table-per-type (TPT). These rules also search for evidence of vertical or horizontal partitioning of data across tables as special cases of the TPT and TPC patterns. Finally, two rules in the table are "clean-up" rules that adjust associations (moving them as far down inheritance hierarchies as is consistent with the data) and identify abstract entities.

B. Parameterized Pre-Conditions

Several of the pattern-matching rules contain pre-conditions that must be true about an entity, collection of entities, or association before performing any data mining that requires running queries. Pre-conditions avoid executing rules that are unlikely to produce meaningful inheritance relationships. They limit the range of possible inheritance in the output and the I/O cost. Each pre-condition (and post-condition, introduced momentarily) has a default value that we determined empirically from running EdmGen++ on live databases and visually inspecting the results.

Number of distinct values in a column. Rule TPH1 depends on finding a column with few active domain values.

Number of columns in common between tables. Rule TPC requires two tables to have columns in common to consider the tables as candidates for merging into a table-per-concrete-type hierarchy. The column count includes both key and non-key columns. This pre-condition is necessary because tables may incidentally have columns with identical names and domains (common cases include "Name" and "id").

TABLE II
RESULTS OF RUNNING EDMGEN++ ON A COLLECTION OF REAL-WORLD DATA SETS USING DEFAULT PARAMETERS

DB	Tables	Size	Time	# Found
1	70	11K rows/table	13 sec	7
2	45	6K rows/table	21 sec	14
3	93	44K r/t (1.15 GB)	8.5 min	44
4	176	36K r/t (2 GB)	7.5 min	26
5	85	500 rows/table	50 sec	51
6	187	5K r/t (1.8 GB)	6.5 min	97
7	125	25K r/t (1.3 GB)	1 min	10

C. *Parameterized Post-Conditions*

Some rules have post-conditions that run after rule application to limit inheritance in the final model to instances that are likely to be semantically relevant. Post-conditions do not limit I/O cost because they must run after mining conceptual dependencies and applying (but not committing) the rule.

Child extent size relative to parent. For any new inheritance relationship discovered by rule TPH2, we test to see if the new child entity has a significant number of instances. "Significant" in this case is defined as the ratio of row counts in the child to the parent (our default value is 5%). If this ratio is below the threshold, the new inheritance is rejected.

Multiplicity of child properties. For any new inheritance relationship discovered by rules TPH1 and TPH2, we check whether the candidate child entity too few unique properties to be meaningful as its own entity (for instance, only one or two).

D. *Frequent Pattern Trees*

We use Frequent Pattern Trees (FP-trees) [5] in two of the rules in Table I to analyze the frequencies of co-occurrences of features in an item collection. An FP-tree can be built in two passes over the collection of items: the first to determine the frequency of each feature, and the second to build the tree.

Rule TPH2 uses an FP-tree to correlate the presence of null values in properties. The "feature" is each non-key property, and the item collection is the entity's set of rows. A non-null value for a property means the property "occurs" for that row.

Rule TPC uses an FP-tree to correlate the presence of schema elements across entities. The "feature" is the name and data type of a non-key property. The item collection is the set of all entities with a common primary key that have no parent. A feature "occurs" if an entity has a property with the feature's name and data type. The FP-tree thus organizes pairs of entities that are candidates for the rule.

E. *Performance Results and Discussion*

We ran EdmGen++ against demonstration and real-world data sets using the default values for pre- and post-conditions. Table II shows, for a sampling of those runs, the number of tables and size of each data set, execution time, and the number of inheritance relationships discovered. These results demonstrate that, on databases with gigabytes of data and tens to hundreds of tables, EdmGen++ generates models in a suitable amount of time for a design-time activity. Our performance analysis is a work in progress; for the demonstration, we will present timings over much larger database instances and will profile the relative cost of each rule.

We chose default condition values after testing EdmGen++ with varying parameters. For instance, for the relative child size post-condition, increasing the parameter from 0% to 5% (the default value and the case shown in Table 2) on DB 6 reduced inheritance relationships from 319 to 97, and reduced the number of entities in the generated model by over half. Manual inspection revealed that most of the cut entities were similar to MinorStudent in Figure 1(c), with few enough rows and columns that the semantic relevance of the entity is questionable. Increasing the parameter from 5% to 10% reduced inheritance count from 97 to 81, and began to cut entities with significant column counts.

IV. RELATED WORK

Many research projects have shown how to generate rich conceptual or object models from instances. Lammari et al. summarize such methods, both for generation of Extended ER diagrams and of object models [7]. They provide their own method (MeRCI) that has features that EdmGen++ currently lacks, such as examination of DML in logs and deduction of default values. They developed a prototype, but do not analyze its performance or the relevance of generated models.

Hainaut et al. describe a pattern-based approach to generating an EER conceptual model from schema and data [4]. They describe case studies and a prototype implementation, DB-MAIN, which is the only publicly-available modeling tool we are aware of that can reverse engineer a non-trivial model from a database [3]. In DB-MAIN, reverse engineering is done manually, where the user chooses a rule to apply to the current conceptual model, and the rule is applied to all matching patterns. No rules apply to instance data, and no analysis is performed on the quality of the resulting models.

EdmGen++ conditions are inspired by work on schema summarization by Yu and Jagadish [11] and object-oriented quality design metrics [2]. In future work, we may consider other metrics referenced in their work as guidelines for generating more relevant models.

REFERENCES

[1] A. Adya, J. A. Blakeley, S. Melnik, S. Muralidhar, and the ADO.NET Team. Anatomy of the ADO.NET Entity Framework. SIGMOD, 2007.

[2] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 1994, 20(6):476–493.

[3] DB Main. http://www.db-main.eu/.

[4] J.-L. Hainaut et al. Database reverse engineering: from requirements to CARE tools. Journal of Automated Software Engineering, 1996, 3(1):9–45.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. SIGMOD, 2000, 1–12.

[6] Hibernate. http://www.hibernate.org/.

[7] N. Lammari, I. Comyn-Wattiau, and J. Akoka. Extracting generalization hierarchies from relational databases: A reverse engineering approach. Journal of Data and Knowledge Engineering, 2007, 63(2):568–589.

[8] Oracle TopLink. http://www.oracle.com/technology /products/ias/toplink/index.html.

[9] S. Ramanathan and J. Hodges. Extraction of object-oriented structures from existing relational databases. SIGMOD Record, 26(1):59–64.

[10] Ruby on Rails. http://rubyonrails.org/.

[11] C. Yu and H. V. Jagadish. Schema Summarization. VLDB 2006, 319–330.