Extending ER Models to Capture Database Transformations to Build Data Sets for Data Mining *

Carlos Ordonez, Sofian Maabout[2], David Sergio Matusevich, Wellington Cabrera
University of Houston, USA
[2]LaBRI, France

Abstract

In a data mining project developed on a relational database, a significant effort is required to build a data set for analysis. The main reason is that, in general, the database has a collection of normalized tables that must be joined, aggregated and transformed in order to build the required data set. Such scenario results in many complex SQL queries that are written independently from each other, in a disorganized manner. Therefore, the database grows with many tables and views that are not present as entities in the ER model and similar SQL queries are written multiple times, creating problems in database evolution and software maintenance. In this paper, we classify potential database transformations, we extend an ER diagram with entities capturing database transformations and we introduce an algorithm which automates the creation of such extended ER model. We present a case study with a public database illustrating database transformations to build a data set to compute a typical data mining model.

1 Introduction

The entity-relationship (ER) model [1, 2] provides diagram notation and methods to design a database, by defining its structure before storing information. On the other hand, the relational model provides a precise mathematical definition to store information in the form of tables (relations) interrelated by foreign keys, whose basic structure is determined by the ER model [1]. Relational algebra [2] provides a formal computation mechanism to query tables in a database combining select, project, join and aggregation (SPJA) operations. In this work, we are concerned with modeling all potential database transformations via relational queries to build a data set that is used as input by a typical data mining algorithm [3]. In general, such data set has a tabular form, where every row corresponds to an observation, instance or point (possibly varying over time) and every column is associated to an attribute or feature. The main issue is that relational queries in a data mining project create many temporary tables (static) or views (dynamic), which are not represented as entities in an existing ER model. Such collection of transformation tables and disconnected queries complicate database management, software development and software maintenance. The problem is even more acute when multiple data sets are derived. Our approach is related to an extract-load-transform (ELT) process, in which tables are cleaned and transformed *after* they are loaded into the database. On the other hand, traditional Extract-Transform-Load (ETL) tools can compute data transformations, but mainly to build data warehouses. In ETL tools most data transformation happens outside the DBMS, before loading data. In general, tools that perform data reverse engineering [4, 5, 6] do not give an abstract, well-defined representation of database transformations computed with relational queries, nor do they have a data set with variables (in a statistical sense) as the target of such transformations. We propose to promote derived data sets and associated temporary tables as an optional extended view of an ER model. By doing so, users can become aware of the content of the database and hence help them reuse what has already been computed. Indeed, understanding transformation tables in abstract form can be useful not only for optimizing the generation of new data sets, but also for tracing their provenance (lineage) [7]. Our approach goes further by considering several stages of data transformation, mathematical functions and the powerful SQL `CASE` statement, which does not have a 1-1 correspondence with relational algebra.

Based on the motivation introduced above, we extend an ER diagram with entities that represent database transformations used to build data sets. Our proposed extended ER model can help analysts understand previous database transformations and then motivate reuse of existing temporary tables, views and SQL scripts, thereby saving time and space. We introduce an algorithm to automate the process of extending an existing ER model based on a sequence of

*© Elsevier, 2014. This is the author's **unofficial** version of the work. The official version of this article was published in Data & Knowledge Engineering (DKE Journal), 2014. DOI: http://dx.doi.org/10.1016/j.datak.2013.11.002

SQL transformation queries given in a script. The paper is organized as follows: Section 2 provides background information on the ER model and relational databases. Section 3 formalizes the problem with relational algebra, classifies database transformations, explains how to extend an existing ER model and introduces an algorithm to automate the ER model extension. In Section 4 we discuss a case study with a real database. We discuss closely related work in Section 5. Section 6 concludes the paper.

2 Preliminaries

We first present definitions and assumptions. Then we introduce a small database that will be used throughout the paper to illustrate our contributions.

2.1 ER model and its Mapping to a Relational Model

A relational database is denoted by $D(\mathcal{T}, I)$, where $\mathcal{T} = \{S_1, \ldots, S_n\}$, is a set of n source tables and I is a set of integrity constraints. Columns in tables are denoted by $A_1, A_2, \ldots$, corresponding to attributes in an ER model. They are assumed to have the same name across all entities. The most important constraints are entity and referential integrity, which ensure validity of primary keys and foreign keys. Entity integrity requires that every relation should have a primary key. On the other hand, referential integrity is an assertion of the form $S_i(K) \rightarrow S_j(K)$, where K is the primary key of S_j and K is a foreign key in S_i. Referential integrity requires that the inclusion dependency $\pi_K(S_i) \subseteq \pi_K(S_j)$ holds.

As it has become standard in modern ER tools, each entity corresponds to a table and each relationship is represented by foreign keys. We assume 1:1 relationships can be merged into one entity because they share the same primary key. Moreover, we assume M:N (many to many) relationships eventually get mapped to a "linking" entity, which connects both entities taking their respective foreign keys as its primary key. Therefore, it is reasonable to assume only 1:N (1 to many) and N:1 (many to 1) relationships exist between entities in a normalized database. Because relationships can be read in both directions it suffices to base our discussion on 1:N relationships. In short, we will work with the final ER model, ready to be deployed as physical relational tables, with a one to one correspondence between entities and tables. We use modern ER notation, which has been influenced by UML (Unified Modeling Language). Entities are represented by rectangles, attributes are grouped into key and non-key attributes and relationships are represented by lines. Entities are linked by relationships denoted with solid lines, where a crow feet indicates the "many" side of the relationship.

The ER to relational model mapping is the standard mechanism to convert an ER conceptual model into a physical model that can be deployed with data definition languages (DDL) in SQL. Database system textbooks (e.g. [2]) present a standard procedure for mapping (or converting) an ER model to a relational model. This mapping is not lossless, in the sense that not all semantics captured in ER are distinguishable in the relational model. In practice, an ER model is deployed in the form of a relational database schema, where entities and relationships are represented by tables and referential integrity constraints.

2.2 Relational Algebra and SQL

We assume database transformations are computed with the SQL language. However, in order to have precise and short mathematical notation we study database transformation with relational queries, which have set semantics and obey first order logic. We use an extended version of relational algebra, where π, σ, $\bowtie$, $\cup$ and $\cap$ are standard relational operators and π is used as an extended operator to compute `GROUP BY` aggregation queries (e.g. grouping rows to get sum(), count()). Relational queries combine SQL aggregate functions, scalar functions, mathematical operators, string operators and the SQL `CASE` statement. Given its programming power and the fact that it is incompatible with relational algebra, we study the `CASE` statement as a special operator.

Outer joins are essential for the construction of data sets. However, full outer joins are not useful in data mining transformations because, generally speaking, the referencing table is the table whose rows must be preserved, whereas right outer joins can be rewritten as equivalent left outer joins. Consequently, an inner (and natural) join is a particular case of a left outer join returning less rows. In short, we consider left outer join a prominent operator needed to merge tables to build the desired data set.

Finally, it is necessary to define the class of queries that are valid to transform tables. We define a *well formed* query as a query that complies with the following requirements:

- A well formed query always produces a table with a primary key and a potentially empty set of non-key attributes.
- In a well formed query having join operators, each join operator is computed based on a foreign key and primary key from the referencing table and the referenced table, respectively.

Intuitively, every table produced during the transformation process can be joined with other tables in the database, based on appropriate foreign keys and primary keys. We also consider well formed queries as closed operators, in the sense that queries can be combined with other queries, giving rise to more complex well formed queries. In this manner we eliminate from consideration SQL queries that produce tables incompatible with the transformation process.

2.3 Motivating Example

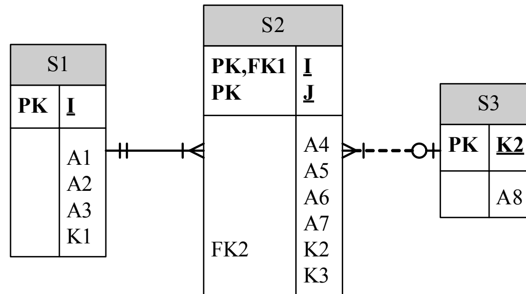

Figure 1: ER model for example database.

Figure 1 shows a small database following our notation. Tables S_1, S_2, S_3 represent three source tables from a normalized database. Intuitively, S_1 represents the object to analyze (e.g. customer). S_2 corresponds to a large transaction table constantly receiving new records corresponding to specific objects (e.g. products purchased by customer). Finally, S_3 gives detailed information about what each object does (e.g. detailed product information). This small database will be used to illustrate our proposed ER model extension and algorithm. From a data mining perspective, we will build a data set used as input for a regression model, where the data set will have several explanatory variables and one target variable. Such variables *do not exist* in the database: they will be derived with transformation queries.

3 Extending ER Model with Database Transformations

This section presents our main contributions. We start by formally defining a sequence of transformation queries. We then define the data set as well as its properties in the context of the ER model. We then provide a comprehensive taxonomy of database transformations using relational algebra as a theoretical foundation. We compare potential solutions to represent database transformations, emphasizing the gap between a logical and a physical database model. Based on the classification of database transformations, we introduce an algorithm to extend an ER model, which takes as input the database schema behind the ER model and a sequence of SQL queries transforming such database.

3.1 Database Transformation Queries

The main goal of this kind of sets of transformation queries is to build a single table, that will be used as input for a data mining algorithm. Such table will be the result of computing a sequence of queries, defined below.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be the set of m tables representing the existing *source* entities in the original ER model. Let $\mathcal{Q} = [q_0, q_1, q_2, \ldots, q_n]$ a sequence of $n+1$ relational queries producing a chain of database *transformations*, where q_0 determines the universe of data mining records to be analyzed. The desired data set X is the result of q_n. Let $\mathcal{T} = [T_0, T_1, T_2, \ldots, T_n]$ be the sequence of *transformation* tables, where $X = T_n$. Table T_0 will be used to build the final data set X with left outer joins. Notice that the order of tables in the sequence is important, since we assume table T_j might depend on a subset of transformation tables, where T_i is one of those tables and $i < j$. Each relational query is an SPJA expression (extended with `CASE` statements) combining tables from $\mathcal{S}$ and $\mathcal{T}$.

Example (typical SQL script): Based on the database example introduced in Section 2.3 here we present a sequence of transformation SQL queries to build a data set. Notice source tables are S1, S2 and S3, whereas transformation

tables are T_1, T_2, and so on. In several cases T_i depends on T_{i-1}. In other cases a query combines source and transformation tables. The last query merges all partial transformation tables into a single data set. Notice a left outer join using T_0 as the "universe" table is computed on the last query.

```
/* q0: T0, universe */
SELECT I,  /* I is the record id, or point id mathematically */
       CASE
          WHEN A1='married' or A2='employed' THEN 1
          ELSE 0
       END AS Y,/* binary target variable */
       A3 AS X1   /* 1st variable */
INTO T0
FROM S1;

/* q1: denormalize and filter valid records */
SELECT S2.I,S2.J,A4,A5,A6,A7,K2,K3
INTO T1
FROM S1 JOIN S2 ON S1.I=S2.I
WHERE A6>10;

/* q2: aggregate */
SELECT I, sum(A4) AS X2,sum(A5) AS X3,max(1)  AS k /* k is FK */
INTO T2
FROM T1
GROUP BY I;

/* q3: get min, max */
SELECT  1 AS k, min(X3) AS minX3, max(X3) as maxX3
INTO T3
FROM T2;

/*q4: math transform */
SELECT I,
       log(X2) AS X2  /* 2nd variable */
       (X3-minX3)/(maxX3-minX3) AS X3 /* 3rd variable range [0,1]*/
INTO T4
FROM T2 JOIN T3 ON T2.K=T3.K;   /* get the min/max */

/* q5: denormalize, gather attribute from referenced table S3 */
SELECT I,J,A7,A8
INTO T5
FROM T1 JOIN S3 ON T1.K2=S3.K2;

/* q6: aggregate with CASE */
SELECT I, sum(CASE WHEN A7='Y' THEN A8 ELSE 0 END) AS X4
INTO T6
FROM T5
GROUP BY I;

/* q7: data set, star join this data set can be used for:
 logistic regression, decision tree, SVM */
SELECT T0.I,X1,X2,X3,X4,Y
INTO X
FROM T0 JOIN T4 ON T0.I=T4.I
        JOIN T6 ON T0.I=T6.I;
```

3.2 Data Set Schema

We define a data set as follows: The data set is an entity $X(K, A)$ with two attribute sets, where K is the primary key (possibly composite) and A is a set of non-key attributes. From a formal modeling standpoint the data set corresponds to a weak entity [2] that depends on some source entity [2]. Nevertheless, we have decided not to show the data set as a weak entity in the ER model with a special diagram notation. In data mining terms, X will be the input for a statistical model like linear regression, PCA, Naive Bayes or clustering (among others). In particular, if all non-key attributes are real numbers then X represents a matrix.

We now give an overview of how each attribute set is built, starting with K. The first consideration is whether K is a simple or composite key. The most common case is that K is simple with a 1-1 correspondence to some existing

source entity (e.g. customer id). In such case, we will sometimes use i instead of K to make reference to the ith point or i^{th} observation explicit. On the other hand, if K is composite we can consider two possibilities:

- it takes primary keys from several entities (e.g. customer id, product id), similar to a cube entity with a composite key
- or it is the result of adding a new key attribute that was not originally part of any entity (e.g. customer id, month id).

In the latter case, it is likely that there exists a *dimension* table where the new key attribute is already available (e.g. a cube dimension table). Evidently, given m source tables (entities) there should be $m-1$ foreign keys linking them in order to compute $m-1$ joins. In other words, the primary key of the data set comes from combining the primary keys of the m source tables.

We now discuss A, the non-key attributes set. Assume $T = T_1 \bowtie T_2 \ldots \bowtie T_k$. The goal is to further transform T into X. We should note that T is not in 3NF and if T has a composite key it is likely that it is not in 2NF either. We argue that each attribute in A can only come directly from: a table, an aggregation or an expression (arithmetic, logical, string) combining one or more attributes together. Basically, attributes coming directly from another table are not transformed, but they transform the target table. Therefore, a transformed attribute must either be the result of an aggregation summarizing several rows into one, or computing some derived expression, resulting in denormalization. In the next subsection we will explore data transformations in more depth.

Example (data set schema): As explained in Section 2.3, the data mining goal is to build a data set used as input for a regression model. The data set X schema is $X(I, X_1, X_2, X_3, X_4, Y)$, where $K = \{I\}$ and $A = \{X_1, X_2, X_3, X_4, Y\}$. In this case, X_1, X_2, X_3, X_4 represent explanatory (input) variables and Y is a target variable we wish to predict.

3.3 Classification of Database Transformations

We discuss database transformations at two levels: entity level and attribute level. In relational database terms they correspond to tables and columns referenced and created by SQL queries.

Entities

As mentioned above, our goal is to extend the existing ER model with new entities providing a representation for data transformations. We will call existing entities source entities, based on the fact that they represent source tables. The new entities will be called transformation entities, computed with SPJA queries. We should point out that a single SQL query may create many temporary tables during its evaluation. In order to have a complete representation each nested SQL query will correspond to one temporary table, which in turn will be represented by one relational algebra query and one transformation entity. Generalizing this idea, the data transformation process will create a set of transformation tables that will be successively joined and aggregated. The final result is one table: the desired data set X. Such database transformation process with nested queries (i.e., based on views or temporary tables) will result in a tree (similar to a parse tree from a query), where internal nodes are joins, leaves are tables and the root is one final query returning the data set. Notice that we assume π (projection) is associated with aggregations (i.e., `GROUP BY` queries in SQL). In the ER diagram, the new entities will be linked with existing source entities and previous transformation entities.

The π operator eliminates duplicates, if any. Even though SQL has bag semantics, allowing duplicate rows, such semantics do not make sense from a data mining perspective. The reason is simple. Let us assume that $T = T_1 \bowtie T_2 \ldots \bowtie T_k$ on the corresponding foreign keys. If we project columns from T they must either include the primary key of T (being the primary key of some table T_i) or they include a `GROUP BY` key to compute aggregations. In relational algebra terms, when computing data transformations the π operator does not make sense if it does not include the primary key of T and it does not have aggregations to derive non-key attributes. We formalize such property as follows:

Proposition: Let $T = T_1 \bowtie T_2 \ldots \bowtie T_k$ on appropriate foreign keys. Every query used to transform T either:

1. Includes the primary key of T which comes from some T_i or
2. it does not include the primary key of T, but it includes a subset of k primary keys of k tables T_i to later compute group-by aggregations.

Proof sketch: All aggregation queries are assumed to have grouping columns in order to identify the object of study. That is, they represent `GROUP BY` queries in SQL. Therefore every $\bowtie$ must include the primary key of either joined table in order produce a data set. An aggregation must use keys to group rows (otherwise, records cannot be identified and further processed) and the only available keys are foreign keys.

Attributes

We now turn our attention to data transformations at the column (attribute) level. We distinguish two mutually exclusive database transformations:

1. Denormalization, which brings attributes from other entities into the transformation entity or simply combines existing attributes.

2. Aggregation, which creates a new attribute grouping rows and computing a summarization.

We assume queries do not compute redundant aggregations (like calling sum() several times).

We will begin by discussing denormalization transformations. When a denormalization brings attributes from other entities the attribute values themselves are not transformed. In other words, the table structure might change, but the column values remain the same. When attributes come directly from other entities, without transforming their values, they can have three roles: they can be part of the primary key, part of a foreign key or they can be a non-key attribute. Attributes that are part of the primary key or the foreign key can later be used to group rows to compute aggregations, which are the second class of database transformations.

In general, non-key attributes will be combined together using all SQL operators, programming constructs and functions. Arithmetic expressions and string expressions fall into this category. We should note that *date* expressions can be considered a mix of string and arithmetic expressions. The only non-key denormalization transformation that deserves special attention is the SQL `CASE` statement, discussed below.

The `CASE` statement is a powerful construct that simulates a variable assignment in a programming language. In its simplest form a `CASE` statement assigns a value to a `SELECT` term when some predicate is true, and a default value otherwise.

Example (CASE statement for binary coding): Consider a CASE statement with our toy database:

```
SELECT
  ..
  CASE
    WHEN A1='married' or A2='employed' THEN 1
    ELSE 0
  END AS binaryVariable
  ..
FROM ..
```

This CASE statement derives a binary attribute nor present before in the database. This example illustrates one of two important issues introduced by the `CASE` statement:

1. it may create new values, not present on any previous column.

2. it may introduce nulls which were not present before.

In other words, from a theory perspective, it can create a new domain and it can introduce nulls, even if referential integrity is enforced (when a `CASE` statement dynamically creates a new foreign key). These two issues can be summarized as follows:

Proposition: An SQL `SELECT` statement with `CASE` statements cannot be evaluated with an equivalent SPJA query.

We now turn our attention to aggregations. Aggregations will use the π operator having a grouping attribute(s) and a list of aggregations (e.g. sum, count). Then a `GROUP BY` query will partition output attributes into key attributes and aggregated attributes. In general, aggregations will return numbers, but we must notice that "max()" aggregations are allowed with strings or dates given as arguments.

Example (aggregation by groups): $\pi_{K,sum(A)}(T)$ is the equivalent of `SELECT K,sum(A) FROM T GROUP BY K;`. Notice T itself may be the result of another query. The output attribute names in the transformed entity can be extracted from the SQL statements, since we assume they populate temporary tables.

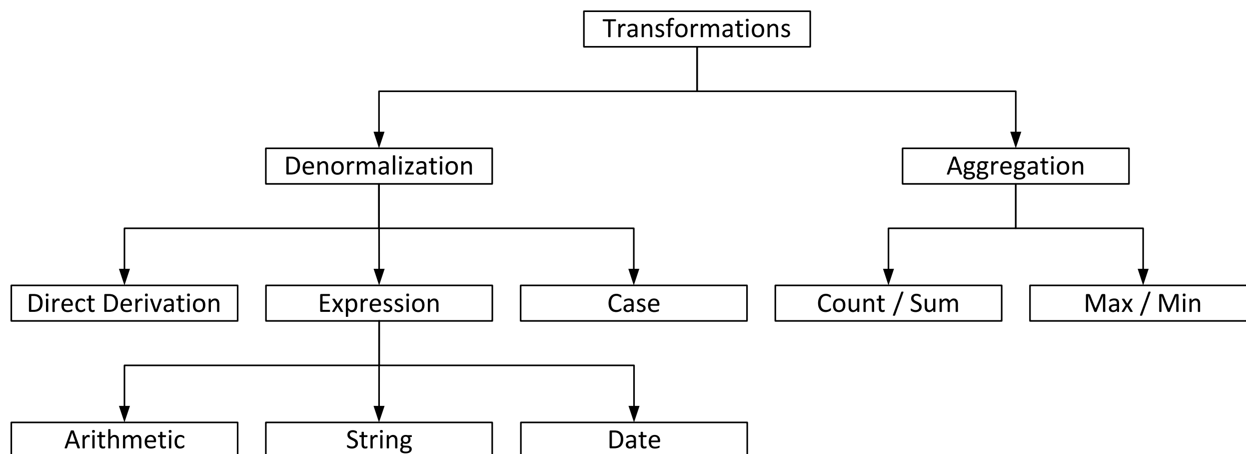

Figure 2: Classification of database transformations.

In short, all potential database transformations are either denormalization or aggregation. Figure 2 shows a diagram summarizing the classification of database transformations.

Example (classifying database transformations): All queries in the input script from Section 3.1 can be classified into either aggregation or normalization transformation classes introduced above. Clearly, queries that derive attributes or have joins represent denormalization (e.g. q1, q4) whereas queries that call aggregation functions or have a GROUP BY clause correspond to aggregation (e.g. q2,q3). Query q6 deserves special attention because it combines a CASE statement with an aggregation. Notice that a query combining joins and an aggregation is split into two queries, doing denormalization and aggregation, separately.

3.4 ER Diagram Representation of Database Transformations

In this section we explain how to show actual SQL queries in an ER diagram, without altering ER diagram notation. We must emphasize that the ER model works at a conceptual-logical level and SQL works at a physical level. Therefore, our proposed notation brings both closer, enriching the ER model with the ability to represent changes to database elements, but mixing database modeling with database processing. We consider three major aspects:

1. Choosing a mathematical notation to represent complex database transformations and reason about them,
2. Extending or maintaining ER diagram notation and,
3. Showing database transformations at the attribute level in a succinct and intuitive manner, including column provenance (original source table).

The first consideration is to choose which abstract notation is best to represent database transformations. We believe the best way to represent transformations is relational algebra, given its precise mathematical definition and the fact that SQL statements tend to be verbose and SQL has bag semantics, which hinder reasoning about sets.

The second and perhaps more thorny aspect is ER diagram notation. After exploring several alternatives, keeping in mind the comprehensive set of diagram symbols available in the ER model, we believe adding new diagram notation to the ER model is not a good idea. On the other hand, given the large number of temporary tables and columns created by SQL queries, we believe it is useful to automatically extend an existing ER diagram, adding entities which correspond to transformation queries. This automated extension will be done by a new algorithm, which we introduce in Section 3.5.

The ER diagram notation has many variants: clearly the classical notation with small boxes for entities and ellipses for attributes is inadequate as it does not scale to a large number of entities and much less to a large number of attributes. That is why we defend the idea of using UML notation as most modern ER tools do. A transformation entity represents a weak entity in a strict modeling sense since each object existence depends on the source entity.

As previously discussed entities will be broadly classified as source or transformation entities, where transformation entities are either aggregations or denormalization. Therefore, each database transformation entity will be labeled as “denormalization: <name>” or “aggregation: <name>”.

We now discuss attribute manipulation. As mentioned above, we must track provenance in order to help the user (data miner, statistician) make sense of the transformations. We propose the following notation: a single dot to indicate the source table if such table participates in a join in the corresponding query (e.g. $T.A$). Otherwise, we propose to use a double dot to indicate the column originates from a previous temporary table (e.g. $T..A$). Based on the fact that SPJA queries cannot express `CASE` computations it is necessary to define a special notation to represent `CASE` statements. Basically, there are two elements to such a statement: a predicate combining comparisons with `and/or/not` and the value(s) to be assigned. Therefore, we can think of a `CASE` statement as a functional form which returns a value based on the predicate being true or false. Given its similarity to the "?:" operators in the C++ language this is our chosen notation.

Example (CASE statement abstract representation): In our example the CASE expression is represented as "A1='married' or A2='employed' ? 1:0". This is a shorter expression than the `CASE` syntax and it is intuitive to any C++ programmer.

There are further SQL query elements that we will not show in the ER diagram query: the `WHERE` clause and the `HAVING` clause. When there is a filter predicate in the transformation table it can be represented with the σ operator omitting the filter predicate since it does not transform the input table. We treat the `HAVING` clause in an analogous manner.

The actual SQL statements, including each temporary table and its corresponding `SELECT` are selectively displayed on a per-entity basis. In other words, we avoid showing SQL statements in the ER diagram. We call such top-down approach a "zoom in" view of the transformation entity. We emphasize that this detailed entity view does not alter ER notation and it can be easily implemented on a GUI.

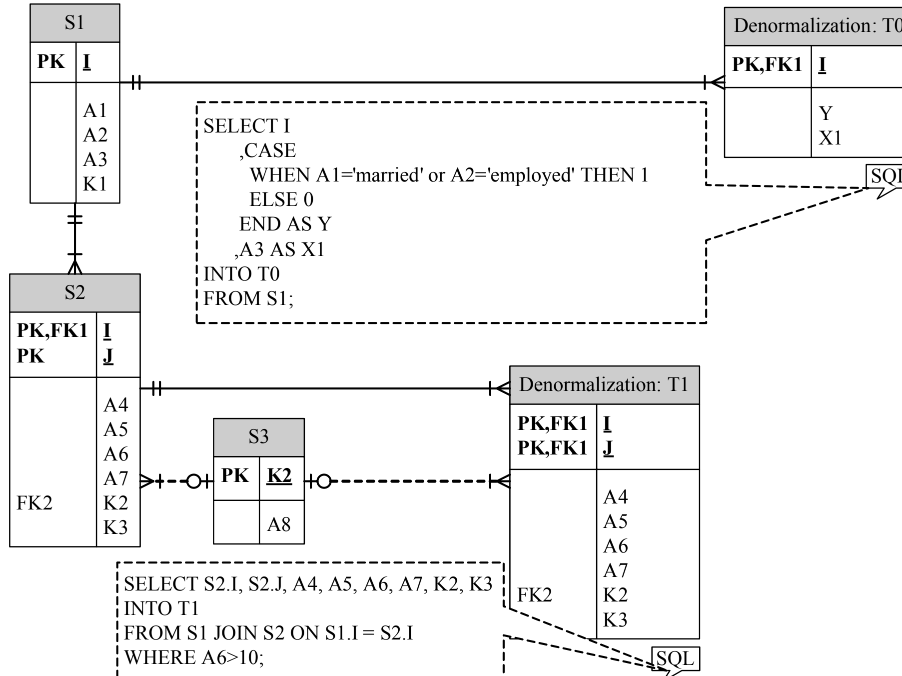

Figure 3: Denormalization transformations for example database.

Examples (denormalization): Based on our small example database Figure 3 shows representative denormalization transformations. T0 is the table where every object of analysis is selected (e.g. customer). Notice there is no WHERE clause. T0 also derives the target binary variable Y, based on marital status and employment. Finally, it renames A3 to become the first variable. T1 gathers columns from S2 to later aggregate them. Figure 4 presents enlightening aggregations. T2 aggregates A4 and A5 from the previously constructed denormalized table T1 to obtain variables X1 and X2, and introduces a constant foreign key to later join with a global summarization (used to rescale X3). T6 corresponds to an aggregation filtering rows based a comparison; such a filter generally helps by removing outliers or unwanted information from the variable. In extended relational algebra such query would require a computation in two steps: denormalization and aggregation, but in SQL the CASE expression is pipelined into the aggregation.

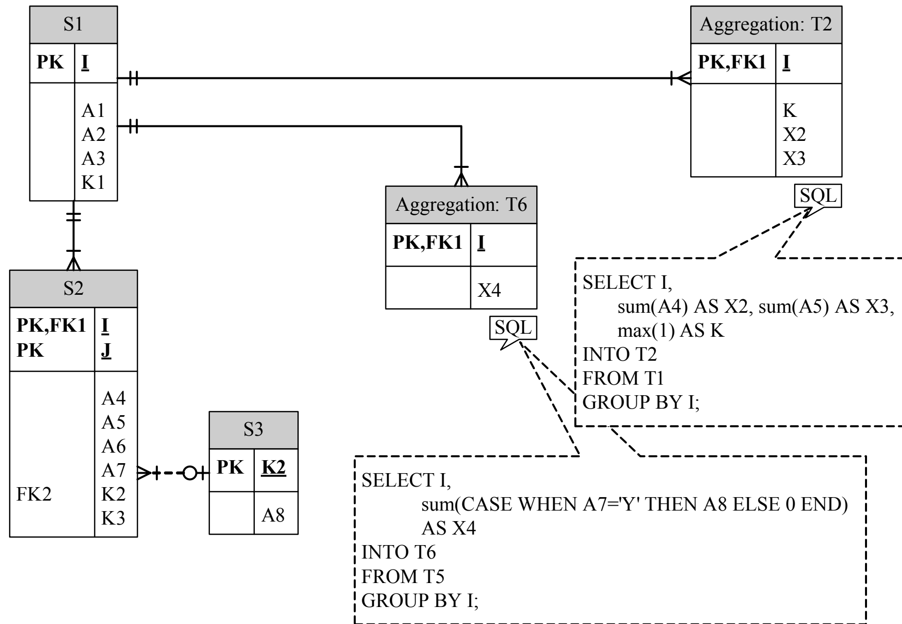

Figure 4: Aggregation transformations for example database.

3.5 Algorithm to Extend an Existing ER Model with Database Transformations

Given an ER model and a sequence of transformation SQL queries Q (defined above) that create data set X, we present the following algorithm to build the extended ER model. The algorithm can be applied on different SQL scripts producing different data sets, reusing existing database transformations when possible. The input is a standard ER model given as a collection of table schemas and the output is the extended ER model with the database transformation entities. The transformation entities schemas, together with their primary keys and foreign keys, can be imported into an ER diagram building tool. Our algorithm was programmed in C#. The program output helped us automatically extend an existing ER model from a real database, which we will explain in detail in Section 4.

```
Denormalization: T0(I,Y,X1                ,PK(I)   ,FK(S1.I));
Denormalization: T1(I,J,A4,A5,A6,A7,K2,K3 ,PK(I,J) ,FK(S2.I,S2.J),FK(S3.K2));
Aggregation:     T2(I,X2,X3,K             ,PK(I)   ,FK(S1.I));
Aggregation:     T3(K,minX3,maxX3         ,PK(K));
Aggregation:     T4(I,X2,X3               ,PK(I)   ,FK(S1.I));
Denormalization: T5(I,J,A7,A8             ,PK(I,J) ,FK(S2.I,S2.J));
Aggregation:     T6(I,X4                  ,PK(I)   ,FK(S1.I));
Denormalization: X(I,X1,X2,X3,X4,Y        ,PK(I)   ,FK(S1.I));
```

Figure 5: Output of our program to extend the ER model of toy database.

Algorithm to extend an ER model with transformation entities

Input: Schemas of $S_1, S_2, \ldots$ and query sequence $q_0, q_1, q_2, \ldots$

Output: $T_0, T_1, T_2, \ldots$

1. Initialize extended ER model with original ER model; labeling each entity as "source" entity: $S_1, S_2, \ldots$.
2. Create a transformation entity j for every intermediate table, numbering it according to query q_j in the sequence. If there is a `GROUP BY` aggregation label the entity "aggregation", otherwise label it "denormalization". Treat nested queries and views as additional temporary tables, as an option depending on the level of detail desired. Label each transformation table as T$< int >$, where $< int >$ is an increasing integer.
3. For each non-key attribute link it to either a denormalization expression or an aggregation function. Track provenance (lineage) of attributes coming from the denormalization process. Link foreign key attributes as foreign keys of source entities (instead of foreign keys to other transformation entities).
4. Create an entity for the final data set. This target data set entity will be highlighted in the ER model and labeled "X" or "data set".

Example (program output): Given as input the table schemas and an SQL script our program classifies each query as aggregation or denormalization and it produces entity labels to extend the existing ER model with transformation entities. The output of our program with the example SQL script presented in Section 2.3), is shown in Figure 5. Notice transformation tables include primary keys and foreign keys in order to automate the creation of the extended ER model. Since entities have labels with a T* prefix (i.e., transformation) they can be selectively displayed or hidden by the ER modeling GUI. To make presentation concise, Figure 6 shows only the three source entities and the data set for our small example.

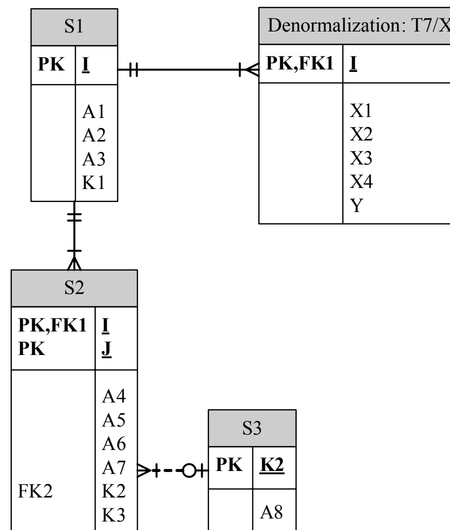

Figure 6: Extended ER model showing only source tables and data set for example database.

4 Case Study

We present a case study with a publically available database [8]. Our discussion closely follows the order in which we presented our contributions in Section 3. We start by describing the database and the data mining problem. We then explain the data set used for data mining analysis, explaining each attribute (primary key, attributes for analysis). Having the data set schema in mind, we explain how the existing ER model is extended by our algorithm. We end the test case discussion highlighting important data transformations in the extended ER diagram. All our ER diagrams were developed with Microsoft Visio 2010, which uses modern ER notation (similar to UML class diagrams, but without methods). Our algorithm was programmed in the C# language. The DBMS was Microsoft SQL Server.

4.1 Database Description

Adventure Works Cycles, a fictitious company name, represents a large, multinational manufacturing company. This company manufactures and sells metal and composite bicycles to North American, European and Asian commercial markets. The AdventureWorks2012 OLTP is a test database of Adventure Works Cycles provided by Microsoft for the SQL Server DBMS. The database has 70 tables with historic sales information of bicycles spanning 4 years (2005-2008). Figure 7 shows the main entities (tables) used in our analysis.

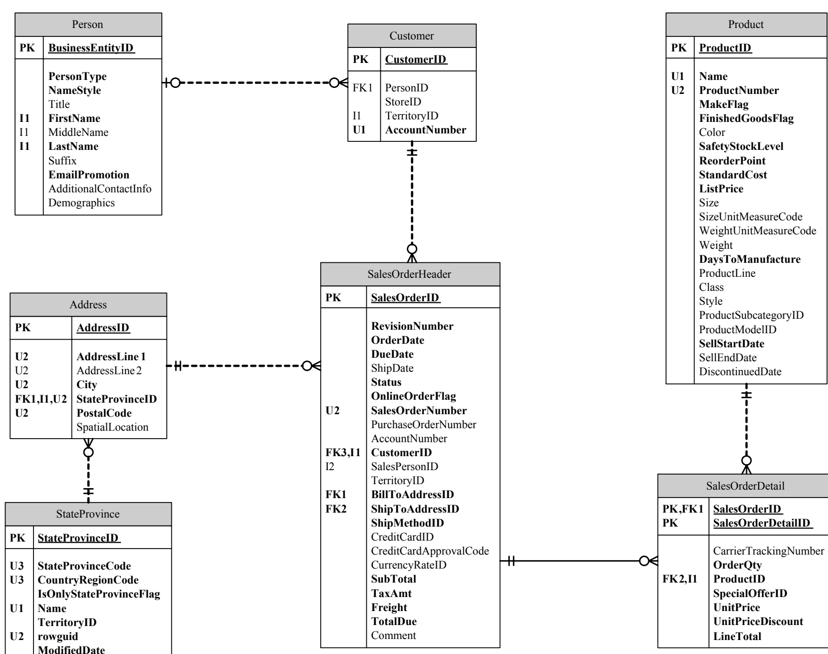

Figure 7: Original ER model for case study database.

4.2 Data Mining Model

Our goal is to predict whether a customer will buy or not within a 6 month time window, which is a representative data mining model in retail forecast analysis. The 6 month window represents a reasonably long time to analyze purchasing behavior. In most data mining projects in a store database there is a large transaction line item table that provides detailed sales information about what each customer bought at a specific date and time. Such table needs to be aggregated, joined and transformed in order to create variables for statistical analysis. The most common type of analyses in a store are customer behavioral segmentation (i.e., categorizing customers based on their buying preferences), churn prediction (that is, predicting if a customer will buy again or not). Our case study aligns with predicting customer churn.

We now discuss the data mining model. As mentioned above, we computed a classification model to predict whether customers will make a purchase or not in the next 6 months based on their purchases during the past 6 months. The data set was partitioned into 2/3 for training and 1/3 for testing, respectively. The chosen classifier was the well-known Naive Bayes, available in the R statistical package. After doing some iterations adding variables,

```
Denormalization:T1( PK(CustomerID),                       ,FK(Customer.CustomerID)
                                                          ,FK(Person.BusinessEntityID));
Denormalization:T2( PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(Person.BusinessEntityID)
                                                          ,FK(SalesOrderHeader.SalesOrderID));
Denormalization:T3( PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID)
                                                          ,FK(Address.StateProvinceID));
Denormalization:T4( PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID));
Denormalization:T5( PK(CustomerID,SalesOrderID,ProductID) ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID)
                                                          ,FK(Product.ProductID));
Denormalization:T6( PK(CustomerID,SalesOrderID,ProductID) ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID)
                                                          ,FK(Product.ProductID));
Denormalization:T7( PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
Denormalization:T8( PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID));
Aggregation:    T9( PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID));
Aggregation:    T10(PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID));
Denormalization:T11(PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID))
Denormalization:T12(PK(CustomerID,SalesOrderID)           ,FK(Customer.CustomerID)
                                                          ,FK(SalesOrderHeader.SalesOrderID));
Aggregation:    T13(PK(CustomerID)                        ,FK(Customer.CustomerID));
```

Figure 8: Classification of transformation entities in case study database (T13 corresponds to data set X).

removing outliers and rebuilding data sets, we were able to obtain accurate classification models. In our case study we explain the final data set and corresponding database transformations, where the classifier accuracy reached 75% with 10-fold cross validation.

4.3 Input: Existing ER model and SQL script

In this section we explain the original ER model and the sequence of transformation queries $q_0, q_1, \ldots$ to build the data set X. There were a total of 14 queries in the script joining, aggregating and filtering seven source tables (i.e., $S_1, S_2, \ldots, S_7$). We should mention there was one query which only filtered rows without doing any database transformation. The analytic record was a customer, whose primary key which was present in almost every transformation table. Total purchasing amounts per transaction were already summarized in a sales header table, which had a relationship to a sales detail information, per product purchased. The sales detail table was the largest in our analysis and it contained the most valuable information to understand why customers buy or not. Product purchases were summarized by a sequence of queries. Such queries created several numeric attributes, including money amounts, tax, item quantity, item size, product class and so on. Denormalization queries gathered information scattered in normalized tables. On the other hand, aggregations computed sum() and max() on columns to add attributes (variables) to the data set X.

The final data set, described in more detail below, ended up having $p = 16$ attributes and n =11,911 records to compute the classification model. From a statistical point of view, our customer data set is multivariate, where each variable is numeric. Variables had mixed types, including categorical and numerical values, but categorical variables were transformed into binary variables. We created a binary class variable that indicated if a customer was active or not during the first six months (January to June) of the most recent sales year.

4.4 Output: New Entities Classified by Transformation and Extended ER Model

We start by showing the output of our algorithm to produce entity names and attributes, as well as classifying them by type of transformation. In order to show output concisely at a high level, Figure 8 shows only the entity name, its type, and primary/foreign key information.

Since there were 13 database transformation entities, each having more than 10 attributes, we show only selected transformation entities to illustrate the output of our tool and the extended ER diagram. These database transformations show key steps to gather information coming from normalized tables and the illustrate the main variables derived with aggregations. Figure 9 shows representative denormalization transformations. T6 is the last step getting all detailed information on products purchased by the customer. T6 takes advantage of T5 (not shown), which combines customer

and sales detail information. Several numeric attributes in T6 will be summarized to create variables that will be present on the final data set. T12 creates the target variable “class”, based on the 6-month window. At this point T12 carries a lot of important statistical variables, which help understanding customer purchasing preferences and behavior. T12 puts together information about the customer and all the detail of sales transactions (i.e. each item purchased). T12 uses T11, which in turn joins sales header and sales detail information. Figure 10 provides a zoom in view of entities T6 and T12. We can see the specific SQL expressions to compute each column as well as their provenance, which appears in comments.

Figure 11 presents important aggregation transformations, which assume all required columns are present in denormalized form. T9 summarize products purchased by the customer based on detailed sales information. Notice the data mining model variables intend to explain why a customer comes back to the store. T13 produces the final data set X where there is a unique record per customer. A GROUP BY aggregation creates numeric and categorical variables that give a detailed profile of each customer. Notice this customer summary eliminates the need to analyze each product separately, but it is feasible to create more accurate classification models adding variables for each product or product category. In an analogous manner, Figure 12 provides a zoom in view of entities T9 and T13. We can see the GROUP BY columns above and the specific SQL functions to compute each column below, as well as provenance for all columns.

To conclude our case study, Figure 13 presents the source entities and the data set X represented by T13. Notice the ER diagram shows a 1-1 relationship between the source table and the data set since the data set effectively extends the original Customer table with statistical variables (i.e., it represents a weak entity [2]).

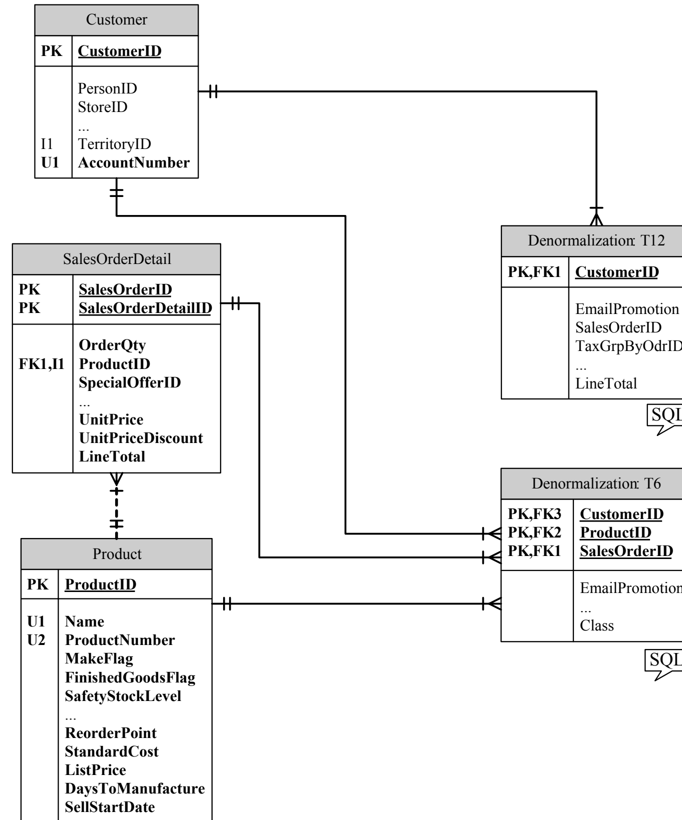

Figure 9: Denormalization transformations in case study database: extended ER model.

5 Related Work

The ER model dates back to the seminal paper [1], in which the basic concepts of entity and relationships are proposed to design databases. Research work on ER modeling can be broadly classified as models for transactional databases and models for data warehouses [9]. Models for transactional databases ensure the database can be maintained in

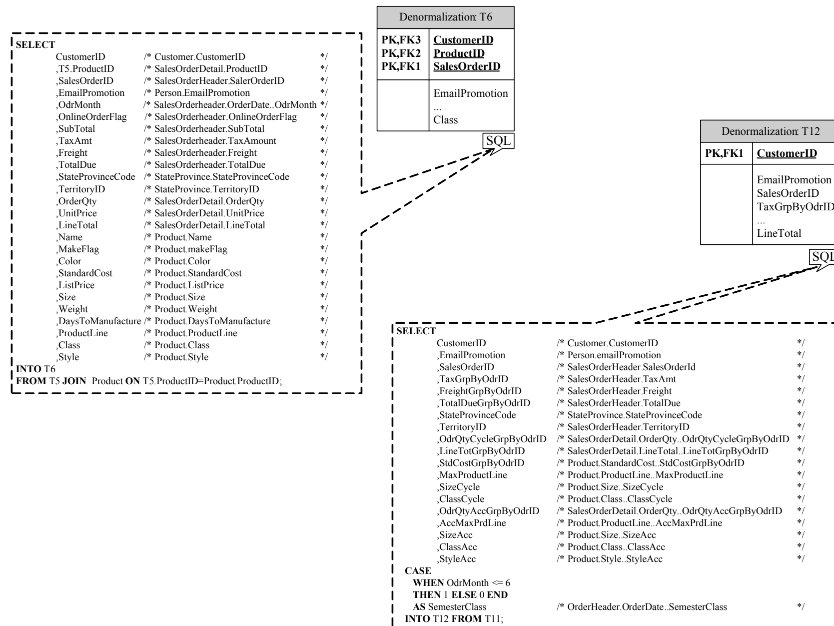

Figure 10: Denormalization transformations in case study database: zoom in view.

a valid state (complete and consistent), whereas models for data warehousing enable multidimensional analysis on cubes [10, 11, 12]. Since we are concerned with analyzing the database rather than updating it our extended ER model is more closely related to models for data warehouses. In both models aggregations and joins (denormalization) are the fundamental database operators. However, there are fundamental differences between both kinds of models. Cube dimensions and measures are identified early on the database design to denormalize tables, whereas attributes in the data set are built later, during the iterative data mining process. Generally speaking, data transformations for statistical analysis are more complex since they also involve mathematical functions and complex logic in the `CASE` statement. On the other hand, provenance [7] identifies the source tables in a data warehouse where columns originate from. There is recent interest on building abstract models to understand how ER models are transformed, combined with other models or extended with external database sources: by defining an EER (extended ER) metamodel [13] or considering big data [14]. Provenance comes in the form of enriched metadata. We have adapted provenance to trace the source tables an attribute in the data set comes from. A closely related work that studied the problem of transforming tables from a modeling perspective is [15], where the authors study the transformation of a database schema into another equivalent one in order to facilitate information exchange with another database; [15] compares alternatives schemas, but does not consider queries combining aggregations, denormalization and math functions on attributes to transform tables like we do. The only work we found that studies how to represent a data set for data mining is [16], where the authors exploit UML to extend an existing data warehouse model to represent attributes used in classification, a prominent statistical learning problem; [16] focuses on the data set, rather than representing data transformations to build it.

The closest approach in the literature is data reverse engineering [17, 18]. There are ETL tools which perform reverse data engineering, such as Pentaho Kettle, but they are unable to capture the whole transformation process. For instance, Pentaho can display joins to illustrate the denormalization process. However, Pentaho Kettle has a heavier focus on ETL and cube processing than on data mining. Specifically, it does not consider the representation of `CASE` statements, which carry a significant weight in data transformation; it also does not make a clear distinction between denormalization and aggregation. In contrast, we assume SQL queries consume tables already stored on the database. We consider more complex data transformations beyond denormalization with joins. Our goal is to enable the user to extend and reuse existing tables and views created with SQL queries.

Our work shares some similarities with efforts to represent transformations in ETL (Extract-Transform-Load) processes [19, 20]. However, there is a fundamental difference: we study the transformation of existing tables. Therefore, our work resembles an ELT process where tables are transformed after being loaded into a data warehouse. Another important difference is that we do not represent transformations to integrate databases or solve data quality problems like ETL does.

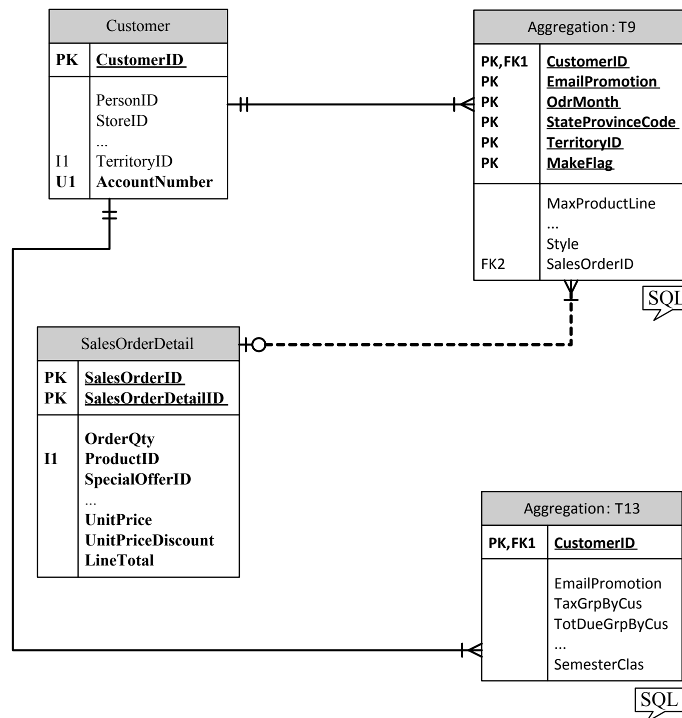

Figure 11: Aggregation transformations in case study database: extended ER model.

From the query processing side, SQL has been extended with operators to transform tables for cube exploration and data mining. Horizontal aggregations represent a combination of pivoting and aggregation in a single query [21], which are useful to create data sets. It is essential to represent such data transformation in an abstract form, as we do in our extended ER model. On the other hand, the importance of understanding denormalized tables has been highlighted before [22]. In [22] there is a proposal to analyze inclusion and functional dependencies in denormalized tables built with relational queries. However, the goal is to transform tables into a 3NF database model that can be represented in ER form, rather than providing an abstract representation of existing database transformations.

6 Conclusions

In a data mining project executed on a relational database the largest amount of time is spent on preparing and cleaning data with SQL queries, which creates many temporary tables, views and many lines of SQL code in a disorganized manner. With that motivation in mind, we proposed extending an ER model with new entities to represent database transformations and we introduced an algorithm to automate the process. Such extended ER model can help analysts reuse existing tables or views and can help understanding a set of complex SQL queries at a high level. We focused on database transformations to build a data set for statistical or machine learning analysis. Our work bridges the gap between a logical database model represented by a standard ER model and a physical database model represented by SQL queries. Our extended ER model has two kinds of entities: source entities and transformation entities, which correspond to normalized tables and temporary tables created with SQL queries, respectively. We proposed to classify database transformations into two main categories: denormalization and aggregations. Database transformations computed by SQL queries can be selectively displayed on a per-entity basis with a zoom in view on the extended ER diagram. A case study on a real database illustrates the usefulness of our proposal. This validation demonstrates our extended ER model allows the user to navigate and understand a large set of transformation tables, providing easier, more organized and higher level understanding than reading SQL queries.

Our reverse data engineering approach offers many opportunities for future research. We intend to automate reusability of tables and queries: We want to develop algorithms that match a new query to the closest database transformation query and corresponding entity, helping the user modify an existing query instead of adding a new one. It is necessary to study complex data transformations in depth, including variable coding, binning, pivoting and cubes. We want to determine when it is feasible to design transformation entities before writing SQL queries to close

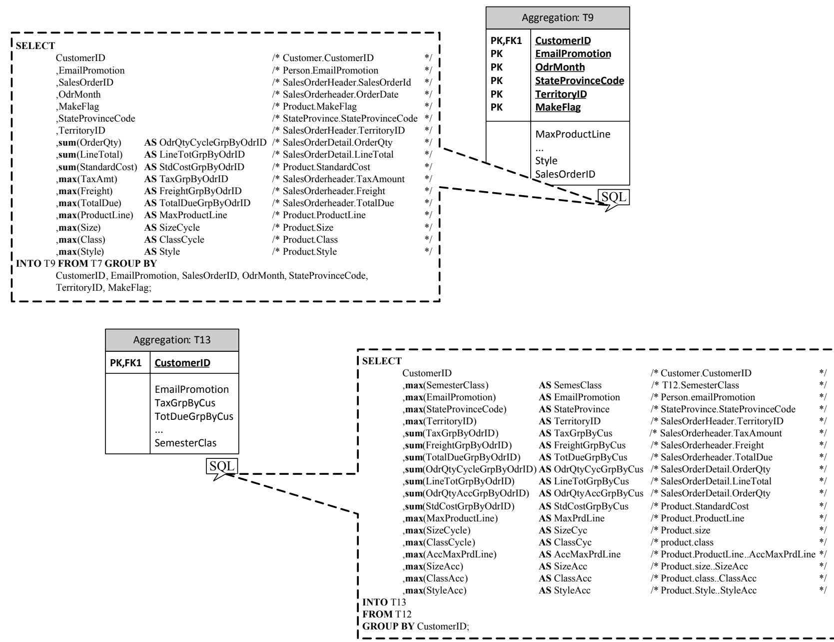

Figure 12: Aggregation transformations in case study database: zoom in view.

the loop. From a normalization perspective, it is necessary to study aggregations choosing a best table schema to maintain consistency and to minimize storage of redundant information. The SQL `CASE` statement is powerful, but as noted above, it introduces many issues from a theoretical and modeling perspective. Specifically, we want to extend relational algebra with new data types and logical operators to handle all potential forms of the `CASE` statement. We intend to develop metrics to quantify savings in software development effort (such as lines of source code, number of functions, time spent by developers or analysts) and database maintenance (number of tables that can be reused, number of attributes that can be used by multiple models, free disk space).

References

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

[2] R. Elmasri, S. Navathe, Fundamentals of Database Systems, 4th Edition, Addison-Wesley, 2003.

[3] C. Ordonez, Data set preprocessing and transformation in a database system, Intelligent Data Analysis (IDA) 15 (4) (2011) 613–631.

[4] C.Y.C., M. Gertz, N. Sundaresan, Reverse engineering for web data: From visual to semantic structure, in: Proc. ICDE Conference, 2002, pp. 53–63.

[5] A. Cleve, J. Hainaut, Dynamic analysis of SQL statements for data-intensive applications reverse engineering, in: Proc. WCRE Conference, 2008, pp. 192–196.

[6] J. Hainaut, J. Henrard, V. Englebert, The nature of data reverse engineering, in: Proc. of Data Reverse Engineering Workshop (DRE), 2000, pp. 1–10.

[7] B. Tomazela, C. Hara, R. do Rodrigues Ciferri, C. D. de Aguiar Ciferri, Empowering integration processes with data provenance, Data & Knowledge Engineering (DKE). 86 (2013) 102–123.

[8] Adventure Works sample data warehouse, `http://technet.microsoft.com/en-us/library/ms124623(v=sql.105).aspx` (2008).

[9] E. Malinowski, E. Zimányi, A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models, Data & Knowledge Engineering (DKE) 64 (1) (2008) 101–133.

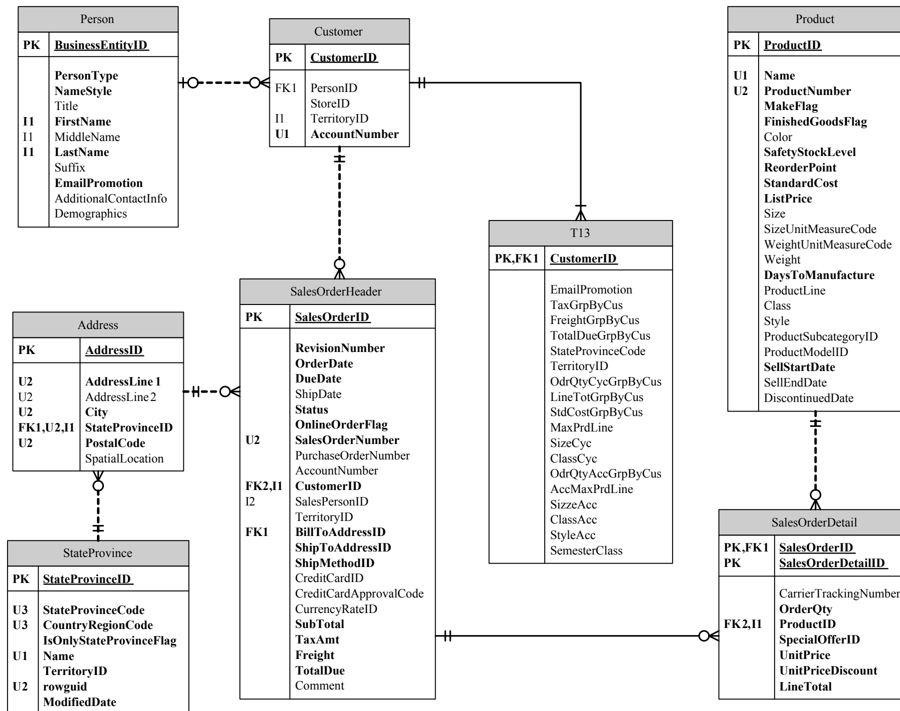

Figure 13: Extended ER model of case study database including the data set.

[10] E. Malinowski, E. Zimányi, Hierarchies in a multidimensional model: From conceptual modeling to logical representation, Data & Knowledge Engineering (DKE) 59 (2) (2006) 348–377.

[11] J.-N. Mazón, J. Lechtenbörger, J. Trujillo, Solving summarizability problems in fact-dimension relationships for multidimensional models, in: Proc. ACM DOLAP, ACM, 2008, pp. 57–64.

[12] N. Prat, J. Akoka, I. Comyn-Wattiau, A UML-based data warehouse design method, Decision Support Systems 42 (3) (2006) 1449 – 1473.

[13] R. Fidalgo, E. D. Souza, S. España, J. Castro, O. Pastor, Eermm: A metamodel for the enhanced entity-relationship model, in: Proc. ER Conference, 2012, pp. 515–524.

[14] S. Ceri, E. Valle, D. Pedreschi, R. Trasarti, Mega-modeling for big data analytics, in: Proc. ER Conference, 2012, pp. 1–15.

[15] A. Poulovassilis, P. McBrien, A general formal framework for schema transformation, Data & Knowledge Engineering (DKE) 28 (1) (1998) 47–71.

[16] J. Zubcoff, J. Trujillo, Conceptual modeling for classification mining in data warehouses, in: Proc. DaWaK Conference, LNCS, Springer, 2006, pp. 566–575.

[17] K. Davis, Lessons learned in data reverse engineering, in: Proc. of Reverse Engineering, 2001, pp. 323–327.

[18] F. Arcelli, G. Viscusi, M. Zanoni, Unifying software and data reverse engineering: a pattern based approach, in: Proc. of International Conference on Software and Data Technologies (ICSOFT), 2010, pp. 208–231.

[19] P. Vassiliadis, A survey of Extract-Transform-Load technology, International Journal of Data Warehousing and Mining (2009) 1–27.

[20] P. Vassiliadis, A. Simitsis, E. Baikousi, A taxonomy of ETL activities, in: DOLAP, 2009, pp. 25–32.

[21] C. Ordonez, Z. Chen, Horizontal aggregations in SQL to prepare data sets for data mining analysis, IEEE Transactions on Knowledge and Data Engineering (TKDE) 24 (4) (2012) 678–691.

[22] J. Petit, F. Toumani, J. Boulicaut, J. Kouloumdjian, Towards the reverse engineering of denormalized relational databases, in: S. Y. W. Su (Ed.), Proc. ICDE Conference, 1996, pp. 218–227.