ENHANCING DATA MODELS WITH TUNING TRANSFORMATIONS

Jason E. Mattinson and Andrew J. McAllister

Faculty of Computer Science, University of New Brunswick

Fredericton, New Brunswick, Canada, E3B 5A3
tel: (506) 452-6328 fax: (506) 453-3566
email: AndrewM@unb.ca

Abstract

Maintaining consistency between an entity-relationship model and the relational database it represents can be problematic. As a database evolves, reverse engineering can be used periodically to capture an up-to-date model. There are difficulties, however, in obtaining accurate conceptual data requirements using automated reverse engineering tools. A new approach is proposed that ensures data models are up-to-date and removes the need for reverse engineering. Tuning transformations are introduced as a concise method for specifying database design modifications related to performance tuning. This method reduces the effort required to keep models up-to-date. Maintaining an entity-relationship model together with concise specifications of tuning transformations enables a tuned relational database definition to be generated automatically.

***Key words*:** conceptual data model, relational database, reverse engineering, physical database design, tuning transformations

1. Introduction

The purpose of round-trip engineering (RTE) for a relational database is to help maintain consistency between the database definition and the conceptual data model for that database. A typical RTE process is depicted in Figure 1. Database definition often begins with the creation of a conceptual model of data requirements using a notation such as entity-relationship (ER) modeling [3]. Tools that support conceptual modeling typically provide the ability to generate automatically SQL data definition language (DDL) code ("create table" statements, etc.) from the ER model. (Prominent examples of

Figure 1: A round-trip engineering process

such tools include Oracle Designer/2000 [1] and Platinum ERWin from Computer Associates [6].) The code can then be used by a relational database management system to generate the database. This type of automatic generation is referred to as forward engineering.

A database generated in this manner is normally based on some standard conversion method, resulting in a highly normalized database definition. Consider the example ER model in Figure 2. Sales are made by an individual employee and each sale is from a particular region. The database definition generated by Designer/2000 from this model includes a table for each entity, as shown in Figure 3.

Often there is a need to tune the database definition to improve database performance or to simplify programming tasks. Typical ways in which database definitions can be tuned for performance reasons include denormalization, horizontal or vertical table splitting, and inclusion of derived data. These types of modifications are often associated with the term *physical database design* [4, 5, 9].

Conceptual entities can be split into multiple tables in the database. For example, a database designer might choose to perform a horizontal split of the *sale* table from Figure 3 if there is a need to separate the sales that are made in Canada from those made in the United States. The new tables that are created in the database should not be new entities in the ER model, since the *sale* entity already represents them conceptually.

Developers may also modify the database definition at this point because they prefer alternative relational constructs to those used by the standard conversion method (e.g. using triggers instead of foreign keys to maintain referential integrity).

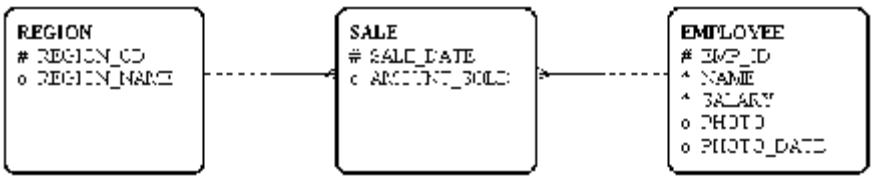

Figure 2: An example ER model

```
CREATE TABLE employee ( emp_id      NUMBER(4,0)  NOT NULL,
                        name        VARCHAR2(30) NOT NULL,
```

```
                        salary      NUMBER(7,2)  NOT NULL,
                        photo       LONG         NULL,
                        photo_date  DATE         NULL);

CREATE TABLE region ( region_cd   CHAR(2)      NOT NULL,
                      region_name VARCHAR2(20) NULL);

CREATE TABLE sale ( emp_id      NUMBER(4,0)  NOT NULL,
                    region_cd   CHAR(2)      NOT NULL,
                    sale_date   DATE         NOT NULL,
                    amount_sold NUMBER(10,2) NULL);

ALTER TABLE employee ADD PRIMARY KEY (emp_id);

ALTER TABLE region ADD PRIMARY KEY (region_cd);

ALTER TABLE sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE sale ADD FOREIGN KEY (region_cd) REFERENCES region(region_cd);

ALTER TABLE sale ADD CHECK (amount_sold >0);
```

Figure 3: A generated database definition

The result is shown in Figure 1 as a ***tuned database***, which we define as a database that may differ from the generated database but is still consistent with the requirements reflected in the current conceptual data model. We also define any modification to the generated database that results in a tuned database as a ***tuning transformation***.

Figure 4 defines an example tuned database, which differs in three ways from the generated database:

1. The company would like to keep track of the sales in Canada and the United States in separate tables (to make reporting and maintenance easier). The *sales* table has been horizontally split into two tables: *can_sale*, and *usa_sale*. The check constraint defined on the *amount_sold* column in the *sale* table is propagated to the *can_sale* and *usa_sale* tables.
2. Check constraints are added to the *can_sale* and *usa_sale* tables to ensure that correct values for *region_cd* are maintained in each table.
3. The *photo* ffield~~ield~~ has been moved into a separate table named *employee_photo*. This is a vertical split of the employee table. (Moving LONG fields to separate tables can help to reduce storage space requirements.)

Figure 4 is consistent with the conceptual requirements defined in Figure 2.

Once a database is available, developers can begin programming and testing with sample data. As development proceeds, changes to the data requirements almost invariably crop up. Ideally these changes should be captured in the ER model and propagated to the database definition. There are two reasons, however, why this is often not done.

First, if the ER model is updated and a new database definition is generated, the new database definition does not reflect any tuning transformations that have been applied to the existing database. This can force a considerable amount of rework. It can even be difficult to determine what tuning

transformations are required—for example, effort is required to compare Figure 4 with Figure 3 to determine which types of tuning were previously applied.

Second, individual changes are often relatively minor, such as adding a new column to a table. The amount of work required to change the database directly can be much less than that required to begin the process by changing the ER model. For these reasons it is common for the database definition to evolve over time and to become inconsistent with the requirements documented in the ER model. This is shown in Figure 1 as an ***altered database***. Figure 5 shows how our example database might be altered to reflect a newly recognized need to maintain the company to which the sale was made. The text that differs from Figure 4 is shown in bold. The unchanged portions of Figure 4 are summarized in italics (to save space and reduce redundancy in Figure 5).

If the conceptual data model remains inconsistent with an altered database, the model provides little value as documentation. A key component of RTE is the periodic use of automated reverse engineering to create an up-to-date ER model based on the altered database. Designer/2000 and ERWin (mentioned above) are examples of tools that support automated database reverse engineering.

Unfortunately, an altered database often has characteristics that are problematic for the current automated reverse engineering tools [2, 7, 8]. Some of the most common characteristics result from the types of tuning transformations mentioned above. As an example related to denormalization, if a single denormalized database table represents two conceptual entities, this table will reverse engineer to only a single entity. Conversely, a table that has been split vertically or horizontally should be represented conceptually as a single entity but reverse engineering results in two or more entities (e.g. *can_sale* and *usa_sale* in Figure 4 conceptually represent *sale* in Figure 2). A derived column should not appear as a conceptual attribute, but it will in a reverse engineered ER model. Triggers used to maintain referential integrity should (but do not) appear as relationships in reverse engineered models.

The risks associated with automated reverse engineering can also be demonstrated by generating a database from an ER model, then reverse engineering the unchanged database. The result can differ in significant ways from the original model. For example (based on Designer/2000), an original ER model with one many-to-many relationship between two entities will be forward engineered to three

```
CREATE TABLE employee ( emp_id      NUMBER(4,0)  NOT NULL,
                        name        VARCHAR2(30) NOT NULL,
                        salary      NUMBER(7,2)  NOT NULL,
                        photo_date  DATE         NULL);

CREATE TABLE employee_photo ( emp_id      NUMBER(4,0)  NOT NULL,
                              photo       LONG         NULL);

CREATE TABLE region ( region_cd   CHAR(2)      NOT NULL,
                      region_name VARCHAR2(20) NULL);

CREATE TABLE can_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL);

CREATE TABLE usa_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL);

ALTER TABLE employee ADD PRIMARY KEY (emp_id);

ALTER TABLE employee_photo ADD PRIMARY KEY (emp_id);

ALTER TABLE employee_photo ADD FOREIGN KEY (emp_id)
      REFERENCES employee(emp_id);

ALTER TABLE region ADD PRIMARY KEY (region_cd);

ALTER TABLE can_sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE usa_sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE can_sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE usa_sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE can_sale ADD FOREIGN KEY (region_cd)
      REFERENCES region(region_cd);
```

```
ALTER TABLE usa_sale ADD FOREIGN KEY (region_cd)
      REFERENCES region(region_cd);

ALTER TABLE can_sale ADD CHECK (amount_sold >0);

ALTER TABLE usa_sale ADD CHECK (amount_sold >0);

ALTER TABLE can_sale ADD CHECK (region_cd IN ('NB','NS','ON','BC'));

ALTER TABLE usa_sale ADD CHECK (region_cd IN ('ME','NH','NY','CA'));
```

Figure 4: A tuned database definition

```
(Insert the CREATE TABLE statements from Figure 4 for employee,
employee_photo, and region here)

CREATE TABLE can_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL,
                        company_id NUMBER(4,0)   NOT NULL);

CREATE TABLE usa_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL,
                        company_id NUMBER(4,0)   NOT NULL);

CREATE TABLE company ( company_id   NUMBER(4,0)   NOT NULL,
                       name         VARCHAR2(30)  NOT NULL,
                       address      VARCHAR2(100) NULL,
                       in_region_cd CHAR(2)       NOT NULL);

(Insert all constraints from Figure 4 here)

ALTER TABLE company ADD PRIMARY KEY (company_id);

ALTER TABLE can_sale ADD FOREIGN KEY (company_id)
      REFERENCES company(company_id);

ALTER TABLE usa_sale ADD FOREIGN KEY (company_id)
      REFERENCES company(company_id);

ALTER TABLE company ADD FOREIGN KEY (in_region_cd)
      REFERENCES region(region_cd);
```

Figure 5: An altered database definition

tables, then reverse engineered to three entities and two relationships. When a one-to-many relationship is forward engineered, only one of the two "mandatory versus optional" constraints for the relationship is retained. The other is always assigned by default to be "optional" during reverse engineering, regardless of the original model. Descriptive relationship names are typically lost, and are replaced with less descriptive foreign key identifiers. Automated reverse engineering provides a diagram depicting how the relational tables are organized, but often does not ensure that conceptual data requirements are reflected accurately.

An alternative solution is to institute strict manual procedures for maintaining consistency between the ER models and the evolving database definition [7]. For example, all modifications to the database definition can be recorded as they are made, and then applied in bulk at periodic synchronization times to the ER model. There are, however, two problems with such an approach. First, the process is manual and therefore error-prone. It is possible to forget to record one or more changes to the database, or to change the ER model in a way that is inconsistent with the database. Second, the process is labor-intensive, which means compliance can be problematic when budgets and/or timeframes are tight. The focus in such situations is often on completing development of the system, rather than on maintaining the conceptual documentation.

For the reasons discussed above, maintaining up-to-date conceptual data models is often neglected as commercial development projects proceed. This paper proposes a new approach that ensures data models are always up-to-date and reduces the required effort.

2. An Alternative Approach

A proposed approach that allows database definitions to evolve using forward (rather than reverse) engineering is shown in Figure 6. The following discussion illustrates how database development proceeds based on this approach.

Database development begins in the same manner as with the process in Figure 1: an ER model captures an initial view of data requirements. At some point the model is forward engineered to SQL DDL code, which is used to create a database. Some tools allow the database to be generated directly from the model without generating SQL but this does not affect the process shown in Figure 6.

A key difference between RTE and the proposed approach is the manner in which performance tuning is applied to the database. In RTE, there is an assumption that the definition of the database (captured in the form of SQL code) can change over time. The SQL code generated from the ER models is updated with tuning transformations and alterations for new requirements.

With our approach, the SQL code generated from the ER models is never modified. Any modifications made to the database definition following SQL code generation are recorded separately. (The importance of this separation is related to alterations for new requirements, which are discussed in Section 2.2.)

2.1 Tuning

Assume, for example, that a database has been created from the SQL code (Figure 3) generated from Figure 2. Also assume we wish to tune this database so that it ends up in a state consistent with Figure 4. This can be accomplished by creating a script that modifies the database definition. Figure 7 provides one way to specify (using SQL) the tuning transformations for the example database. Other

similar SQL specifications for the same tuning transformations are possible. Figure 7 corresponds to the icon labeled *SQL code (to accomplish tuning tran's)* in Figure 6.

Figure 6: An alternative process

```
-- Tuning #1: Split the sales table horizontally

CREATE TABLE can_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL);

ALTER TABLE can_sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE can_sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE can_sale ADD FOREIGN KEY (region_cd)
      REFERENCES region(region_cd);

ALTER TABLE can_sale ADD CHECK (amount_sold >0);

CREATE TABLE usa_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL);

ALTER TABLE usa_sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE usa_sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE usa_sale ADD FOREIGN KEY (region_cd)
      REFERENCES region(region_cd);

ALTER TABLE usa_sale ADD CHECK (amount_sold >0);

-- Remove the sale table

DROP TABLE SALE;

-- Tuning #2: Add check constraints

ALTER TABLE can_sale ADD CHECK (region_cd IN ('NB','NS','ON','BC'));

ALTER TABLE usa_sale ADD CHECK (region_cd IN ('ME','NH','NY','CA'));

-- Tuning #3: Split the employee table vertically

CREATE TABLE employee_photo ( emp_id      NUMBER(4,0)  NOT NULL,
                              photo       LONG         NULL);

ALTER TABLE employee_photo ADD PRIMARY KEY (emp_id);
```

```
ALTER TABLE employee_photo ADD FOREIGN KEY (emp_id)
      REFERENCES employee(emp_id);

ALTER TABLE employee DROP COLUMN photo;
```

Figure 7: Using SQL for tuning transformations

Database tuning can be performed in an incremental fashion, creating new SQL scripts and applying them to the database as tuning needs are determined. At any point in time the SQL generated from the ER model and the tuning SQL (e.g. Figures 3 and 7, respectively) can be executed to create a new database instance.

It is important that only tuning transformations (as defined in Section 1) are applied directly to the database. Other changes to the database definition should be made to the ER model instead. This is dependent on the ability of database developers to determine which transformations will retain consistency between the database and the current ER model, and which will not. For developers familiar with ER modeling and conversion to relational databases, however, this is a relatively straightforward determination.

One potential problem is that some types of tuning transformations can be lengthy when specified using SQL. As a result, the effort involved in creating Figure 7 may be more than that required to modify Figure 3 directly into Figure 4. Some developers may be tempted to modify the SQL generated from the ER model, rather than recording tuning transformations separately as the proposed approach requires. Also, in the absence of comments, it can be difficult to determine which SQL statements are intended to accomplish a single tuning modification.

To aid developers in this regard, we introduce a concise notation for specifying tuning transformations. The idea is to specify commonly-used tuning transformations in as concise and convenient a manner as possible.

The three types of tuning transformations for which concise statements are most useful are *split horizontal*, *split vertical* and *denormalize*. Figures 8, 9, and 10 provide the syntax for these three statement types.

The *split horizontal* tuning transformation adds one or more new tables with the same structure, indexes, check constraints and referential integrity constraints as an existing table. The new tables will each hold a subset of the rows that the original table would normally hold. This can be useful, for example, to enable geographic distribution of data or to increase query performance in some circumstances. The original table is not automatically dropped as there are cases where the original table may still be required. Figure 11 provides an example split horizontal statement, which specifies that two tables (*can_sale* and *usa_sale*) are to be created based on the *sale* table.

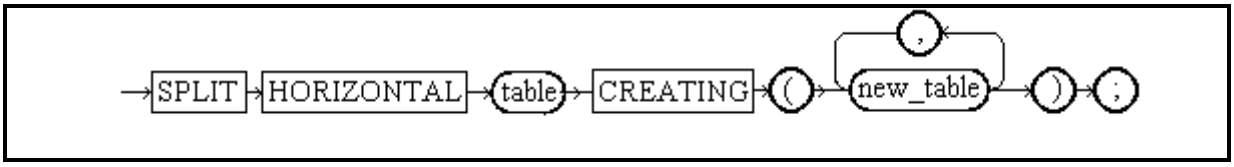

Figure 8: Syntax of the split horizontal statement

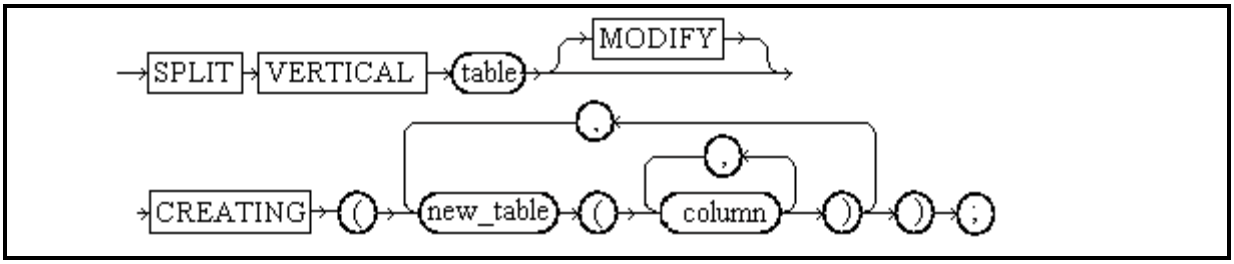

Figure 9: Syntax of the split vertical statement

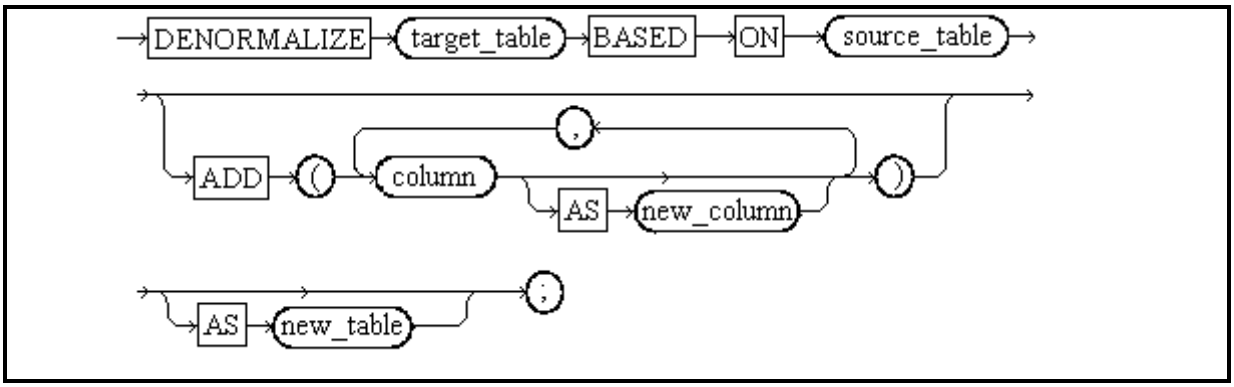

Figure 10: Syntax of the denormalize statement

The *split vertical* tuning transformation distributes the columns of an existing table to form one or more new tables. The statement specifies which columns to include in the new tables. Primary key columns from the original table are added automatically to the new tables along with any unique keys, check constraints, foreign keys, and indexes defined on the specified columns in the original table. The specified columns are removed from the original table (excluding primary key columns) if and only if the MODIFY keyword is included. A vertical split can be used, for example, to control storage space requirements or to enable geographic distribution of subsets of a database.

The example split vertical statement in Figure 11 removes the *photo* field from the *employee* table and places this field in a new table named *employee_photo*. The *emp_id* field is also placed in the new table automatically, since *emp_id* is the primary key of the *employee* table.

The *denormalize* tuning transformation allows columns to be replicated from one existing table (the "source") to another (the "target"), transforming the latter table into a weaker normal form. The source table is not altered. This type of transformation is typically done to enhance ease and speed of data retrieval, often sacrificing ease and speed of data updates.

The denormalize statement allows a subset of the source table columns to be replicated and optionally renamed in the target table. If no columns are specified, then all non-primary-key columns in the source table are replicated in the target table. The primary key columns of the source table are excluded since these columns are assumed to already be represented in the form of a foreign key in the target table. If an AS clause is included in a denormalize statement, then the target table is not altered. Instead, a new table is created and includes all columns from the target table as well as the specified columns from the source table.

If any uniqueness constraints, check constraints or indexes are defined on the source table for the fields to be replicated, then these definitions are not replicated in the denormalized table. There are instances where such definitions can make sense in the source table but result in conflicts in the resultant table. For example, *region_name* could be a column with all unique values in the *region* table, but if this

column were replicated into the *sale* table then a given region name would appear in multiple rows and a uniqueness constraint would not make sense.

An example of the denormalize statement is provided in Section 2.2.

```
-- Tuning #1: Split the sales table horizontally

SPLIT HORIZONTAL sale
      CREATING (can_sale, usa_sale);

-- Remove the sale table

DROP TABLE SALE;

-- Tuning #2: Add check constraints

ALTER TABLE can_sale ADD CHECK (region_cd IN ('NB','NS','ON','BC'));

ALTER TABLE usa_sale ADD CHECK (region_cd IN ('ME','NH','NY','CA'));

-- Tuning #3: Split the employee table vertically

SPLIT VERTICAL employee
      CREATING (employee_photo(photo));
```

Figure 11: Concise tuning transformations

A prototype "pre-processor" tool has been developed that expands statements of the types defined in Figures 8 to 10 into SQL statements to accomplish the desired transformations. For example, the statements in Figure 11 are converted by the pre-processor into the SQL shown in Figure 7. This illustrates the savings in effort that are possible using the proposed approach. For example, specifying a single split horizontal statement in Figure 11 replaces the need to specify the first ten SQL statements in Figure 7.

The current implementation of the tool uses information about the generated database in the DBMS data dictionary while interpreting tuning transformation statements. For example, the tool needs to know what columns are in the *sale* table to create the SQL code for the horizontal split in Figure 7. This reliance on the data dictionary means that the tool can expand Figure 11 into Figure 7 only after the SQL in Figure 3 has been executed to create a database. (An alternative implementation might, for example, use the ER model stored in the CASE tool repository instead of the database definition in the data dictionary.) The SQL statements in Figure 7 can then be executed to transform the database into a tuned database. This is consistent with the process shown in Figure 6.

There are types of tuning transformations commonly discussed by database design texts [4, 5, 9] for which no new statement types are required, since these transformations can be accomplished in a concise fashion using SQL (e.g. adding derived data columns to existing tables). In our running example, Figure 11 includes three SQL statements; one DROP TABLE statement and two ALTER TABLE statements. The DROP TABLE statement is included because the *can_sale* and *usa_sale* tables replace the need for the original *sale* table, and the horizontal split transformation does not include removal of the original table. The ALTER TABLE statements add constraints that make sense only after the horizontal split and so are included as part of the tuning transformations. Any SQL statements in the input to the pre-processor tool are simply copied with no changes to the SQL file produced as output by the tool.

The ordering of the statements in Figure 11 is significant. For example, the split horizontal statement in Figure 11 must be completed before the drop table statement can be executed.

2.2 Alterations

New data requirements often arise after a tuned database has been completed. As the proposed approach in Figure 6 shows, new requirements are captured by altering the ER model (which should be archived first). Figure 12 provides an altered ER model for our example database, reflecting the new need to maintain the company to which the sale was made and the region in which the company head office is located.

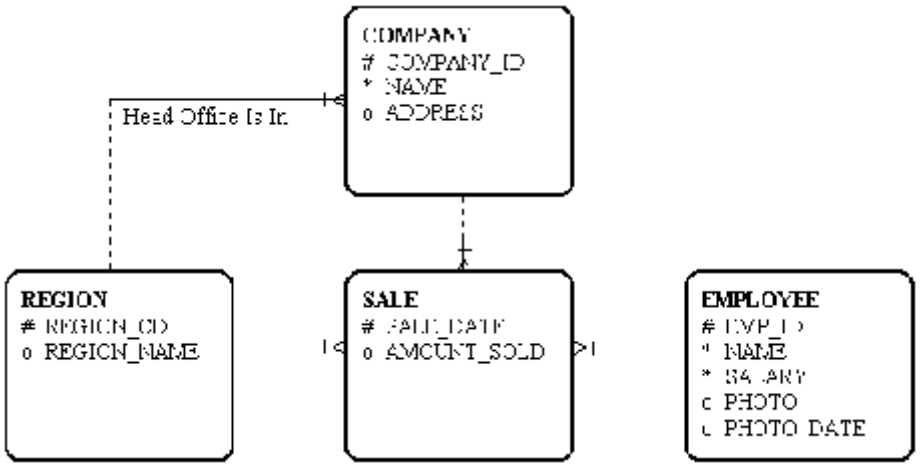

Figure 12: Altered ER model

Since the tuning transformations (Figure 11) are specified separately from the SQL generated from the ER model, the tuning transformations are not lost when the altered ER model is used to generate new SQL code. The existing tuning transformations might not, however, be consistent with the new SQL code. For example, a tuning transformation might refer to a table for which the corresponding entity was deleted from the altered ER model. In our example, the tuning transformations specified in Figure 11 are still consistent with Figure 12. There may be, however, a need to tune the newly created *company* table. Figure 13 provides an example of altered concise tuning transformation specifications. The *company* table is denormalized by placing the *region_name* field from the *region* table into the *company* table.

Figure 14 shows the SQL code generated from Figure 13 by the prototype pre-processor tool. To generate Figure 14, the tool uses the data dictionary for the database generated from Figure 12. Figure 14 can then be executed against this database to create a new tuned database consistent with the altered ER model. This corresponds to the icon labeled *tuned database #2* in Figure 6.

The bolded code in Figure 14 shows the differences between this result and the auto-generated SQL in Figure 7 (prior to the alterations). Note that even though the split horizontal statement is the same in Figures 11 and 13, the SQL code generated from this statement is different in Figures 7 and 14. In Figure 14 the new foreign key from *sale* to *company* is propagated to the *can_sale* and *usa_sale* tables. This illustrates how the approach proposed in this paper can reduce the amount of effort required to update a database definition as new requirements are added to the ER model.

3. Conclusions and Future Work

The new process proposed in this paper provides an alternative to round-trip engineering. A database can be tuned for performance reasons and altered for new requirements while maintaining the consistency between the database and the data model. The effort required to maintain this consistency is reduced because there is never a need to make changes at both the ER model and database levels to reflect new requirements. Instead, the database definition is captured completely by the combination

```
-- Tuning #1: Split the sales table horizontally

SPLIT HORIZONTAL sale
      CREATING (can_sale, usa_sale);

-- Remove the sale table

DROP TABLE SALE;

-- Tuning #2: Add check constraints

ALTER TABLE can_sale ADD CHECK (region_cd IN ('NB','NS','ON','BC'));

ALTER TABLE usa_sale ADD CHECK (region_cd IN ('ME','NH','NY','CA'));

-- Tuning #3: Split the employee table vertically
```

```
SPLIT VERTICAL employee
      CREATING (employee_photo(photo));

-- Tuning #4: Denormalize company table by including region_name
DENORMALIZE company
   BASED ON region
        ADD (region_name AS in_region_name);
```

Figure 13: Altered concise tuning transformation specifications

of the ER model together with the tuning specifications. Changes need to be maintained only at this level. The pre-processor tool for concise tuning transformations described in this paper has the potential to be a useful utility for CASE tools and relational database management systems.

Database triggers are often used to enforce referential integrity and check constraints. Triggers can be implemented in many different ways and therefore represent an opportunity for future work to apply the concise tuning transformations to database triggers.

In addition, application code and database views have not yet been considered. Views and code may refer to tables that have been denormalized or split, so extending the preprocessor to revise views and application code would be useful.

There is also further potential for automation within the context of the given approach. Experience may show additional types of tuning transformations that can benefit from a concise notation. Automated help for altering tuning transformations based on the altered ER model is possible (e.g. flagging a tuning transformation for a table or column that no longer exists).

Finally, interactions between the proposed approach and the related problem of updating sample data when a new database version is created remain to be investigated.

```
-- Tuning #1: Split the sales table horizontally

CREATE TABLE can_sale ( emp_id      NUMBER(4,0)  NOT NULL,
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL,
                        company_id  NUMBER(4)    NULL);

ALTER TABLE can_sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE can_sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE can_sale ADD FOREIGN KEY (region_cd)
      REFERENCES region(region_cd);

ALTER TABLE can_sale ADD FOREIGN KEY (company_id)
      REFERENCES company(company_id);

ALTER TABLE can_sale ADD CHECK (amount_sold >0);

CREATE TABLE usa_sale( emp_id      NUMBER(4,0)  NOT NULL,
```

```
                        region_cd   CHAR(2)      NOT NULL,
                        sale_date   DATE         NOT NULL,
                        amount_sold NUMBER(10,2) NULL,
                        company_id  NUMBER(4)    NULL);

ALTER TABLE usa_sale ADD PRIMARY KEY (emp_id, region_cd, sale_date);

ALTER TABLE usa_sale ADD FOREIGN KEY (emp_id) REFERENCES employee(emp_id);

ALTER TABLE usa_sale ADD FOREIGN KEY (region_cd)
      REFERENCES region(region_cd);

ALTER TABLE usa_sale ADD FOREIGN KEY (company_id)
      REFERENCES company(company_id);

ALTER TABLE usa_sale ADD CHECK (amount_sold >0);
```

Figure 14: SQL generated from Figure 13 based on altered database (Part 1 of 2)

```
-- Remove the sale table
DROP TABLE SALE;

-- Tuning #2: Add check constraints

ALTER TABLE can_sale ADD CHECK (region_cd IN ('NB','NS','ON','BC'));

ALTER TABLE usa_sale ADD CHECK (region_cd IN ('ME','NH','NY','CA'));

-- Tuning #3: Split the employee table vertically

CREATE TABLE employee_photo( emp_id      NUMBER(4,0)  NOT NULL,
                             photo       LONG         NULL);

ALTER TABLE employee_photo ADD PRIMARY KEY (emp_id);

ALTER TABLE employee_photo ADD FOREIGN KEY (emp_id)
      REFERENCES EMPLOYEE (emp_id);

ALTER TABLE employee DROP COLUMN photo;

-- Tuning #4: Denormalize company table by including region_name

ALTER TABLE company ADD (in_region_name VARCHAR2(20) NULL);
```

Figure 14: SQL generated from Figure 13 based on altered database (Part 2 of 2)

REFERENCES

1. Anderson, C. and Wendelken, D. *The Oracle Designer/2000 Handbook*, Addison Wesley Longman, 1997.
2. Blaha, M. and Premerlani, W. Observed Idiosyncrasies of Relational Database Designs. In *Proc. Second Working Conf. on Reverse Engineering* (Toronto, 1995), 116-125.
3. Chen, P.P. The entity-relationship model - toward a unified view of data, *ACM TODS* 1(1) (1976) 9-36.
4. Connolly, T., Begg, C. and Strachan, A. *Database Systems: A Practical Approach to Design, Implementation and Management*, Addison Wesley Longman, 1999.
5. Elmasri, R. and Navathe, S. *Fundamentals of Database Systems (third edition)*, Addison Wesley Longman, 2000.
6. ERWin web site: http://www.cai.com/products/platinum/appdev/erwin-ps.htm, March 2000.
7. Maciaszek, L.A. Process Model for Round-Trip Engineering with Relational Database. In *Proc. 11th Conf. of the Information Resources Management Association* (Anchorage, Alaska, May 2000), to appear.
8. McAllister, A.J. Techniques For Reverse Engineering a Medical Database. In *Proc. Third Working Conf. on Reverse Engineering* (Monterey, California, November 1996), 121-130.

9. Ramakrishnan, R. *Database Management Systems*, WCB/McGraw-Hill, 1998.