Representing a Relational Database as a Directed Graph and Some Applications

Radoslav Radev
Faculty of Mathematics and Informatics, Plovdiv University 'Paisii Hilendarski', Plovdiv, Bulgaria
4001 Plovdiv, Bulgaria, Chemshir str. 6, ap. 8
+359 889 472 575
radoslav_radev@gbg.bg

ABSTRACT

In this paper we represent the relational database as a directed graph, regarding tables as nodes and foreign keys as edges. We also make several definitions to describe the dependencies between tables. Then algorithms are proposed for identifying all tables dependent upon a certain table and ordering them in a specific way so every table to appear before all tables that depend on it in the sequence. After that we define dependencies between records and represent a set of records also as a graph. In the end several applications of the algorithms are shown as well as practical problems that could be solved by them.

Categories and Subject Descriptors

E.1 [**Data Structures**]: *Graphs and networks, Records*

General Terms

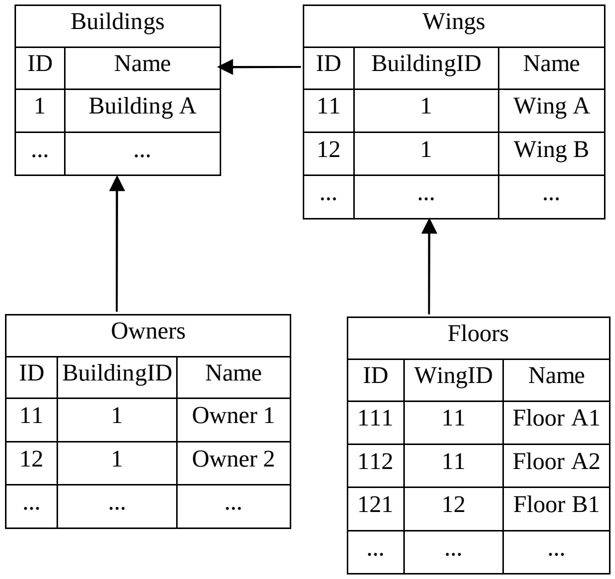
Algorithms, Design, Languages.

Keywords

relational database, foreign key, directed graph.

1. INTRODUCTION

The point of departure of this paper is a complete practical problem. Let us have a record (we will call it *base record*) in a relational database table (let us call it *base table*) and let other tables have foreign keys to this table. In these other tables there are records with foreign key values corresponding to the primary key value of the chosen record, i.e. these records “point” to the base record. Let us call them *directly dependent* records, because they depend on the existense of the base record – if the base record does not exist in the database, they also cannot exist because of the foreign key constraint. In turn there are other tables and records in them “pointing” to these dependent records, and they are in turn dependent on them, so these records are *indirectly dependent* on the base record. In this way we have a set of tables dependent on the base table through foreign keys and a set of records in them depedent on the base record (directly or indirectly). Here is a simple example (Figure 1):



Buildings

ID	Name
1	Building A
...	...

Wings

ID	BuildingID	Name
11	1	Wing A
12	1	Wing B
...	...	...

Owners

ID	BuildingID	Name
11	1	Owner 1
12	1	Owner 2
...	...	...

Floors

ID	WingID	Name
111	11	Floor A1
112	11	Floor A2
121	12	Floor B1
...	...	...

Figure 1

Table *Buildings* is our base table and record *Building A* is our base record. Table *Owners* has a foreign key (on the column *BuildingID*) and two records directly dependent on the base record - *Owner 1* and *Owner 2*. It is the same with table *Wings*, but the dependent records are *Wing A* and *Wing B*. Table *Floors* does not have a foreign key to the table *Buildings*, but is has a foreign key to the table *Wings* (on the column *WingID*). Records *Floor A1* and *Floor A2* depend on record *Wing A*, and record *Floor B1* depend on the record *Wing B*. But because records *Wing A* and *Wing B* depend on the base record *Building A*, we can say that records *Floor A1*, *Floor A2* and *Floor B1* are indirectly dependent on the base record.

The problem we want to solve is how to find all such directly or indirectly dependent records in the database and to order them in such a way that every record appears *after* all records it depends on. For example, such an order for Figure 1 is: *Building A, Owner 1, Owner 2, Wing A, Wing B, Floor A1, Floor A2, Floor B1*.

For the purpose first we will represent the relational database as a directed graph (Chapter 2). After that we will describe algorithms for idenfifying the dependent tables and ordering them (Chapter 3). In the end we will give examples of some practical applications of the solution (Chapter 5).

2. REPRESENTING A RELATIONAL DATABASE AS A DIRECTED GRAPH

Every relational database can be represented as a directed (oriented) graph $G = (T, F)$, where:

- $T = \{T_1, T_2, ..., T_N\}$ is the set of all tables in the database, i.e. every table is regarded as a node in the graph.
- $F = \{F_1, F_2, ...F_M\}$ is the set of all foreign keys defined in the database, i.e. every foreign key is regarded as a directed edge $Fx = (Ta, Tb)$, where $Fx \in F$ and $Ta, Tb \in T$.

The foreign key in a relational database identifies a column or a set of columns in a referencing/child table that refer to a column (or a set of columns) in the referenced/parent table. But every column also could be nullable or not nullable. So let us make the following definitions:

- A foreign key is a *single-column* foreign key, if it maps a single column in the referencing/child table to a single column in the referenced/parent table.
- A foreign key is a *multi-column* foreign key, if it maps more than one column in the referencing/child table to more than one column in the referenced/parent table.
- A single-column foreign key is a *nullable* foreign key if the column in the referencing/child table is nullable. Otherwise, the foreign key is *not nullable*.
- A multi-column foreign key is a *nullable* foreign key if all columns in the referencing/child table are nullable. Otherwise, the foreign key is *not nullable*.

So we can devide the set F into two subsets:

- $F_N = \{Fx\}$, where Fx is a nullable foreign key, i.e. F_N is the set of all nullable foreign keys in the database.
- $F_{NN} = \{Fx\}$, where Fx is a not nullable foreign key, i.e. F_{NN} is the set of all not nullable foreign keys in the database.

Because one foreign key is either nullable or not nullable, it is true that:

- $F_N \cup F_{NN} = F$
- $F_N \cap F_{NN} = \varnothing$

Let us make some more definitions:

- We will call $Tx \in T$ *dependent* on table $Ty \in T$ if and only if a walk exists between Tx and Ty.
- We will call $Tx \in T$ *directly dependent* on table $Ty \in T$ if and only if there exists $Fx \in F$, where $Fx = (Tx, Ty)$.
- We will call $Tx \in T$ *indirectly dependent* on table $Ty \in T$ if and only if a walk exists between Tx and Ty, but there does not exists $Fx \in F$, where $Fx = (Tx, Ty)$.
- We will call $Tx \in T$ *strongly dependent* on table $Ty \in T$ if and only if Tx is dependent on Ty and there exists a walk between Tx and Ty in which at least one of the edges is a not nullable foreign key.
- We will call $Tx \in T$ *weakly dependent* on table $Ty \in T$ if and only if Tx is dependent on Ty and every walk between Tx and Ty consists only of nullable foreign keys.
- We will call $Tx \in T$ *indepedent* on table $Ty \in T$ if and only if a walk does not exist between Tx and Ty.

A table $Tx \in T$ is dependent on itself if and only if there exists a directed cycle in the graph G that includes the node Tx.

Let us choose a table $Ts \in T$. Then let us define:

- $P_{Ts} = \{Tx\}$, $Tx \in T$, where Tx is dependent to Ts, i.e. P_{Ts} is the set of all tables dependent on table Ts.
- $\overline{P_{Ts}} = \{Tx\}$, $Tx \in T$, where Tx is not dependent to Ts, i.e. $\overline{P_{Ts}}$ is the set of all tables not dependent on table Ts.

If Ts is a self-referencing table, i.e. there exists a foreign key from Ts to Ts, then $Ts \in P_{Ts}$. Otherwise $Ts \notin P_{Ts}$.

But for defining our algorithm below it will be more convenient always to include Ts in the set of dependent tables, so let us define two more sets:

- $P^*_{Ts} = \{Tx\}$, $Tx \in T$, $Tx \equiv Ts$ or $Tx \in P_{Ts}$, i.e. P^*_{Ts} is the set of all tables dependent on table Ts including table Ts itself even if it does not depend on itself.
- $\overline{P^*_{Ts}} = \{Tx\}$, $Tx \in T$, $Tx \neq Ts$ and $Tx \in \overline{P_{Ts}}$, i.e. $\overline{P^*_{Ts}}$ is the set of all tables indepedent on table Ts, but without table Ts itself.

Because of the definitions there immediately follows:

- $P_{Ts} \cup \overline{P_{Ts}} = T$
- $P_{Ts} \cap \overline{P_{Ts}} = \varnothing$
- $P^*_{Ts} \cup \overline{P^*_{Ts}} = T$
- $P^*_{Ts} \cap \overline{P^*_{Ts}} = \varnothing$

Accordingly we can devide F in the following manner:

- $F_{Ts} = \{Fx\}$, $Fx \in F$, $Fx = (Ta, Tb)$, $Ta, Tb \in P^*_{Ts}$, i.e. F_{Ts} is the set of all foreign keys between the tables in P^*_{Ts}.
- $\overline{F_{Ts}} = \{Fx\}$, $Fx \in F$, $Fx = (Ta, Tb)$, $Ta \in \overline{P^*_{Ts}}$ or $Tb \in \overline{P^*_{Ts}}$, i.e. $\overline{F_{Ts}}$ is the set of all foreign keys, for which the referencing/child table and/or the referenced/parent table do not belong to P^*_{Ts}.

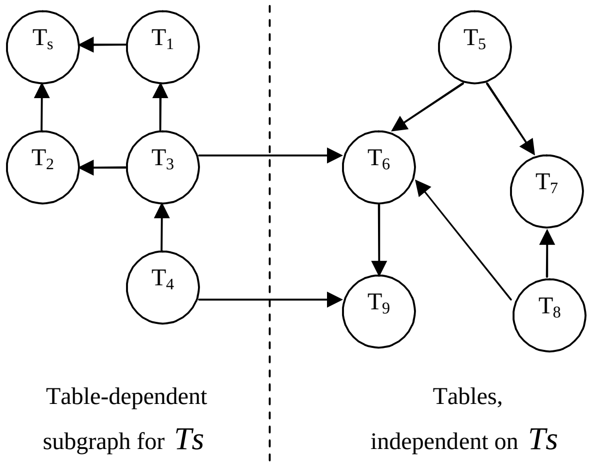
Again, it is true that:

- $F_{Ts} \cup \overline{F_{Ts}} = F$
- $F_{Ts} \cap \overline{F_{Ts}} = \varnothing$

In this way we can create a subgraph from the graph G:

- $G_{Ts} = (P^*_{Ts}, F_{Ts})$, i.e. a subgraph of all tables dependent on table Ts and the foreign keys between them.

We will call such a subgraph *table-dependent subgraph* for Ts.



T_s
T_1
T_2
T_3
T_4
T_5
T_6
T_7
T_8
T_9
Table-dependent subgraph for Ts
Tables, independent on Ts

Figure 2

But we can define also one specific subset of F_{Ts}:

- $F^*_{Ts} = \{Fx\}, Fx \in F_{Ts}, Fx \in F_{NN}$, i.e. F^*_{Ts} is the set of all not nullable foreign keys in F_{Ts}.

Then we can define one more subgraph:

- $G^*_{Ts} = (P^*_{Ts}, F^*_{Ts})$, i.e. a subgraph of all tables dependent on table Ts and the not nullable foreign keys between them.

We will call this graph *not nullable table-dependent subgraph* for Ts.

3. IDENTIFYING AND ORDERING THE DEPENDENT TABLES

3.1 Definition of the Task

We want to order all tables dependent on the base table Ts in such a way, that every table appears *after* all tables it depends on. In other words, every table appears *before* all tables that depend on it. This is needed for the solutions of the practical problems described in the Chapter 5 of this paper.

We can define the task as follows:

We seek an ordered set $PP_{Ts} = \{T_0, T_1, .., T_k\}$, for which is true that:

- For $\forall x \in [0, k] \Rightarrow Tx \in T^*_{Ts}$.
- For every table $Tx \in T_{Ts}$ $\exists y \in [0, k]$, for which $Tx \equiv Ty$, i.e. every table dependent on Ts appears in the set PP_{Ts}.
- $\exists x \in [0, k]$, for which $Ts \equiv Tx$, i.e. Ts appears in the set PP_{Ts}.
- If $x \neq y \Rightarrow Tx \neq Ty$, for $\forall x, y \in [0, k]$, i.e. every table appears only once in the set PP_{Ts}.
- For $\forall x, y \in [0, k], x < y$, Tx is independent on Ty.

It is possible that such a set PP_{Ts} does not exist. This is the case when in the graph G_{Ts} there exists at least one cycle. Practically it means that in the database there exists at least one table, that is dependent on itself.

Also, in general case, PP_{Ts} is not unique. There could be many ordered sets PP_{Ts} satisfying the conditions above. In this paper we will limit our efforts on finding one such set.

3.2 Identifying Dependent Tables

First we have to identify all tables dependent on Ts. We can do this with any graph-traversal algorithm [8], starting at Ts as a root. For this reason it was needed to represent the relational database as a directed graph, which we did in the Chapter 2. Once we have done this, we can perform an algorithm for traversing the graph, such as breadth-first search or depth-first search [8].

The only important thing we must have in mind is not to forget to invert the edges when performing these algorithms. Because when we start from node Ts we have to visit first all nodes that are directly connected to it (see Figure 2). For this reason we have to regard the edges as inverted. Once we have identified all tables dependent on Ts, i.e. the set P_{Ts} (respectively the set P^*_{Ts}), we can continue with checking table-dependent graph for cycles.

3.3 Checking the Table-Dependent Graph for Cycles

Once we have identified the tables in P_{Ts}, we can easily identify the foreign keys between them, i.e. the set F_{Ts} and construct the graph G_{Ts}. But if there are cycles in G_{Ts} then there is no ordered set PP_{Ts} satisfying the conditions listed above. So we first have to check if there are cycles in G_{Ts}. This can easily be done with any algorithm for cycle detection, for example depth-first search. [7]

3.4 Ordering the Dependent Tables

Once we have identified G_{Ts} and have assured that there are no cycles in it, we can continue with elaborating an algorithm for finding an ordered set PP_{Ts}.

3.4.1 Algorithm for Ordering Dependent Tables

Here are the steps of the proposed algorithm:

1. First we have all tables in P_{Ts} unordered. In the next steps of the algorithm we will assign a number to each table – its order in PP_{Ts}.

2. For each still unordered table in P_{Ts} check if all tables it depends on, if any, are already ordered. If so, we can order the table in the series, i.e. to assign it the next order number. If not, we cannot do that – so continue with another table.

3. Repeat step 2. until all tables are ordered. This is guaranteed if there are no cycles of foreign keys. If we have not ensured that, we may fall into endless loop here. Then we have to remeber whether at least one table has been removed in step 2. If not, there have remained only tables that are parts of cycles.

Here the algorithm is described with object-oriented pseucode with comments in order to illustrate best the main idea:

```
class Table
{
    ForeignKey[] ForeignKeys;
    int OrderIndex = 0; // initially table is not ordered
}


class ForeignKey
{
    Table FromTable;
    Table ToTable;
}


int N; // tables count
// we suppose tables and foreign keys are properly initialized
Table[] tables = new Table[N];

// here we count how many tables we have already ordered
int tablesWithOrderFound = 0;

// Subroutine to check if we can order a table, which is possible
// only if all tables this table depends on are already ordered.
bool CanOrderTable(Table t)
{
    foreach (ForeignKey foreignKey in t.ForeignKeys)
    {
        // if there is a foreign key to a still not
        // ordered table, we cannot order this neither
        if (foreignKey.ToTable.OrderIndex == 0)) return false;
    }

    // we are here if all tables that this table depends on are
    // already ordered - we can order this table also
    return true;
}


void Main()
{
    while (tablesWithOrderFound < N)
    {
        // we must know if at least one table is ordered in this
        // iteration of the while cycle; otherwise we are stuck
        // because of cycles of foreign keys
        bool wasTableOrderedInThisIteration = false;

        for (int i = 0; i < N; i++)
        {
                // if table is already ordered - just skip it
                if (tables[i].OrderIndex > 0) continue;

                if (CanOrderTable(tables[i]))
                {
```

```
                tables[i].OrderIndex = ++tablesWithOrderFound;

                wasTableOrderedInThisIteration = true;
            }
        }

        if (!wasTableOrderedInThisIteration)
        {
            // we have cycle(s) of foreign keys
        }
    }

    if (tablesWithOrderFound == N)
    {
        // all tables are ordered correctly
    }
}
```

We will call this algorithm *Algorithm for ordering dependent tables*.

3.4.2 Algorithm for Ordering Strongly-Dependent Tables

The algorithm above can be modified to the case of strongly-dependent tables, i.e. to regard only not nullable foreign keys. The reasons for this will become clear in Chapter 5 of this paper about its practical applications.

So we seek for an ordered set $PP_{Ts}^* = \{T_0, T_1, .., T_k\}$, for which we change only the last condition defined in the beginning of Chapter 3. Instead of:

- For $\forall x, y \in [0,k], x < y$, Tx is not dependent on Ty.

we want:

- For $\forall x, y \in [0,k], x < y$, Tx is not strongly-dependent on Ty.

This means that whenever we have a nullable foreign key we can just ignore it as not creating dependancy at all.

In this case the step about checking for cycles must be done with the *not nullable table-dependent subgraph* $G_{Ts}^* = (P_{Ts}^*, F_{Ts}^*)$.

We will call this modified algorithm *Algorithm for ordering strongly-dependent tables*.

Here is the object-oriented pseudocode:

```
class Table
{
    ForeignKey[] ForeignKeys;
    int OrderIndex = 0; // initially table is not ordered
}

class ForeignKey
{
    Table FromTable;
    Table ToTable;
    bool Nullable;
}

int N; // tables count

// we suppose tables and foreign keys are properly initialized
Table[] tables = new Table[N];

// here we count how many tables we have already ordered
int tablesWithOrderFound = 0;

// Subroutine to check if we can order a table, which is possible
// only if all tables this table depends on are already ordered or
// the foreign keys to the still not ordered tables are nullable.
bool CanOrderTable(Table t)
{
    foreach (ForeignKey foreignKey in t.ForeignKeys)
    {
        // if there is a not nullable foreign key to a still not
        // ordered table, we can not order this neither
        if (!foreignKey.Nullable &&
          (foreignKey.ToTable.OrderIndex == 0)) return false;
    }

    // we are here if all tables that this table depends on are
    // already ordered or the foreign keys to the still not ordered
    // tables are nullable - so we can order this table also
    return true;
}

void Main()
{
    while (tablesWithOrderFound < N)
    {
        // we must know if at least one table is ordered in this
        // iteration of the while cycle; otherwise we are stuck
        // because of cycles of not nullable foreign keys
        bool wasTableOrderedInThisIteration = false;

        for (int i = 0; i < N; i++)
        {
                // if table is already ordered - just skip it
                if (tables[i].OrderIndex > 0) continue;

                if (CanOrderTable(tables[i]))
                {
                        tables[i].OrderIndex =
                ++tablesWithOrderFound;

                        wasTableOrderedInThisIteration =
                true;
                }
        }

        if (!wasTableOrderedInThisIteration)
        {
                // we have cycle(s) of NN foreign keys
        }
    }

    if (tablesWithOrderFound == N)
    {
        // all tables are ordered correctly
    }
}
```

4. RECORD DEPENDENCY

We can extend the concept of dependent tables to that of dependent records. Let us make the following definitions:

- *Record A* in *Table A* is *dependent* on *Record B* in *Table B* if and only if *Record A* cannot exists in the database without *Record B*, i.e. that would violate at least one foreign key violation constraint. That may happen only if *Table A* is strongly dependent on *Table B*.
- *Record A* in *Table A* is *directly dependent* on *Record B* in *Table B* if and only if *Table A* is directly dependent on *Table B* and the foreign key values of *Record A* match the candidate key values of *Record B*. For example, in Figure 1 record *Wing A* is directly dependent on record *Building A*.
- *Record A* in *Table A* is *weakly dependent* on *Record B* in *Table B* if and only if *Record A* is dependent on *Record B*, but is not strongly dependent. For example, in Figure 1 record *Floor A1* is weakly dependent on record *Building A*.
- *Record A* in *Table A* is *independent* on *Record B* in *Table B* if and only if *Record A* can exists in the database without *Record B*, i.e. that would violate no foreign key violation constraint. The important thing here is that it does not mean that *Table A* is independent on *Table B*. For example, in Figure 1 record *Floor A1* is independent on record *Wing B* although table *Floors* is dependent on table *Wings*.

In this way a dependency graph could be constructed for the records, similar to the table-dependent graph. We will call it *record-dependent graph*. Here is an example for the record-dependent graph of record *Building A* in Figure 1:

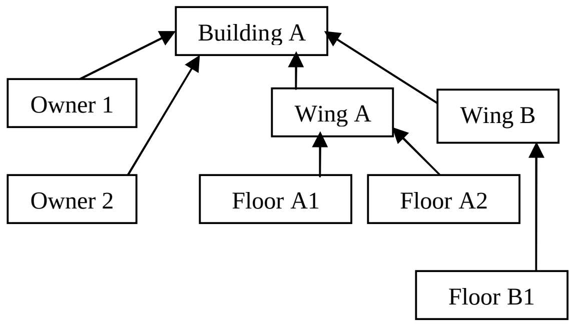


Figure 3

5. PRACTICAL APPLICATIONS

On the basis of the above representations and algorithms there could be solved some practical problems in the domains of relational databases, object-relational mapping (OR/M) and enterprise applications.

5.1 Inserting Records with OR/M Framework

Many OR/M frameworks use *Unit of Work* design pattern [4]. Among the examples are some of the most popular OR/M frameworks today – Microsoft's *Entity Framework* [2], Hibernate for Java [1] and NHibernate [5] for .NET. These frameworks allow a set of entities to be added to the *Object Context*, which is responsible for inserting them into the database in a single transaction. This means that the tables corresponding to the entities must be identified from the OR/M metadata and corresponding insert statements must be generated and executed against the database in a correct order, so that a foreign key violation not to occur.

It also important that during the process of inserting the records no foreign key constraint must be disabled or deactivated, because if an error occurs in the insert process the database must stay in a consistent state, i.e. with all foreign keys active and consistent. This is very critical especially in data warehouse environment [3] with a heavy traffic and a lot of parallel connections to the database. In this case disabling or deactivating foreign key constraints for a period of time might end in an expected behaviour and, even worse, the data in the database could be in such a state that enabling/activating them could be impossible because of wrong or corrupted data.

As a general rule, every record must be inserted after all records it depends on are already inserted, in order to ensure that all foreign key constrants are satisfied. But there are exceptions of this rule. For example, if a foreign key is nullable, i.e. it allows *NULL* values to appear in the referencing/child table column(s). Then a record could be inserted *before* another record it depends on, but with *NULL* foreign key values. That is only possible if the corresponding foreign key is nullable.

For example, if we want to insert the records in Figure 1 and the foreign key between tables *Floors* and *Wings* is nullable, we can implement the following sequence:

1. Insert record *Floor A1* with *NULL* value for *WingID* column.

2. Insert record *Wing A1*.

3. Update record *Floor A1* and set its *WingID* value to 11.

In this way we can find a workaround for the problem described in Chaper 3.3. If there are cycles in the table-dependent graph and the ordered set PP_{Ts} does not exists, we can use the order PP^*_{Ts} and for every nullable foreign key, for which the referencing/child table is *before* the referenced/parent table in the sequence, first to insert the records with *NULL* foreign key values and to update later with the correct ones.

A possible problem that may occur here is that of violating some check or unique constraint defined on the foreign key column(s) in the table. For example, if an unique constraint is defined on column *WingID* in table *Floors* and there is already a record in the table *Floors* with *WingID = NULL*. Then step 1. will fail because of the unique key violation.

So we can propose the following algorithm in order to find the best possible solution and to cover most various cases:

1. Identify all entities that are to be inserted by the Object Context.

2. Take the corresponding database tables from the OR/M framework metadata.

3. Choose any table found in step 2. This will be our Ts table.

4. Try to find PP_{Ts} with the algorithm for ordering dependent tables described in Chapter 3.4.1. If there exists PP_{Ts} :

4.1. For every table in the sequence found:

4.2. Execute the insert statements against the database.

5. If PP_{Ts} does not exist, try to find PP^*_{Ts} with the algorithm for ordering strongly-dependent tables described in Chapter 3.4.2. If found, then:

5.1. For every table in the sequence:

5.2. For every record to be inserted:

5.3. Generate an insert statement with *NULL* foreign key values for every foriegn key, for which the referenced/parent table is *before* the record's table.

5.4. Execute the statement against the database.

5.5. Remember the record and when the referenced/parent record is inserted later, update it with the proper foreign key values.

6. Because PP_{Ts} / PP^*_{Ts} may not include all tables found in step 2, repeat steps from 3 to 5 until all insert statements are executed.

Different variations and optimizations of the algorithm above are also possible, but they are outside the scope of this paper. Here we want to describe only the general idea.

5.2 Cloning a Record Along with Dependent Records

Another task is that of cloning a record in a relational database along with all records dependent on it. It is similar to that of inserting a set of records with OR/M framework described in Chapter 5.1. The steps needed for solving this task are the following:

1. Starting with the base record we want to clone, choose its table as base table.

2. Find all tables dependent to it as described in Chapter 3.2.

3. Retrieve from the database all records in these tables that are dependent on the base record. Start with the directly dependent records, then for everyone of them take its directly dependent records and so on.

4. Use the algorithm for inserting the set of records described in Chapter 5.1, but choose as base table the one taken in step 1.

Here are two real-world scenarios that can often be found in enterprise applications:

5.2.1 Backup Records

In an enterprise application it is often the case that we have a record with many records related to it. For example, an order with order items, or a hospital with rooms, devices, nurses and so on. Imagine we are to make an important operation with such a record that will affect many of the related records. For the sake of sustainability we want to "backup" this record along with all records related to it, so if the operation fails and leaves the initial records in a inconsistent state (in theory that should not happen, but sometimes in practise this cannot be ensured), we will have a valid copy of them. In this case "backup" means to create a copy of the base record and all records related to it.

5.2.2 Template Records

Many enterprise applications provide the functionality of creating data by template. For example, in a hospital or a hotel, usually there are several predefined templates of rooms with devices and cabling plans. When you want to add a new room into the database, you can select such a template and to have a lot of data inserted automatically instead of adding manually records for every device, cabling paths, furniture, stuff, etc. After that you can update this automatically generated records if needed.

For implementing such a functionality it is extemely useful to have the template records into the database and to have a general mechanism for "cloning" them, creating the "real" records on the basis of the template ones. Of course, this can be implemented individually for every specific situation but a general and universal solution is much preferrable in terms of time of development, quality assurance and maintenance.

5.3 Deleting a Record Along with Dependent Records

The task of deleting a database record along with all its dependent records can be regarded as inverted to that of cloning them. Here again we have to retrieve all dependent records, but to invert the order found in the process of cloning. Then we can safely delete them one by one from the database without violating foreign key constraints.

Again, in the case of cycles of foreign keys, we can first update the record's foreign key values with *NULL* values – in this way we "break" the dependancy between records – and after that to delete them safely. The possible problems with violating check or unique key constraints, mentioned in Chapter 5.1, are valid here too.

This task can be accomplished if all foreign keys connecting the dependent tables to the table of the record to be deleted are cascade. In practise that is not always possible, for example because:

- The RDMS does not support cascade foreign keys.
- There are restrictions of the cascade foreign keys in the RDMS. For example, in Microsoft SQL Server a table cannot appear more than one time in a list of all the cascading referential actions that are started by either a DELETE or an UPDATE statement. [6]
- We do not want to have cascade foreign keys because of security or sustainability reasons.
- We work with a legacy database that must not be changed.

6. Conclusion

A practical solution of the problem described in Chapter 5.2 was successfully implemented in the Research and Development department of the Belgium company Televic. A functionality was implemented on the base of the content in this paper that allows cloning of any record in a relation database along with all records dependent on it. The technologies used were:

- .NET Framework 4.0
- Entity Framework 4.0
- SQL Server 2008

7. ACKNOWLEDGMENTS

This paper is partially supported by Plovdiv University NPD grant NI13-FMI (2013).

8. REFERENCES

[1] Bauer, C., and King, G. 2004. Hibernate in Action. Manning Publications Co.

[2] Castro, P., Melnik, S., and Adya, A. 2007. ADO.NET Entity Framework: raising the level of abstraction in data programming, *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ACM New York, NY, USA, 1070-1072. DOI= http://doi.acm.org/10.1145/1247480.1247609

[3] Devlin, B. 1996. Data Warehouse: From Architecture to Implementation. Addison-Wesley Longman Publishing Co.

[4] Fowler, M. 2002. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 184-194.

[5] Kuate, P.H., Harris, T., Bauer, C., and King, G. 2009. NHibernate in Action. Manning Publications Co.

[6] Shapiro, J. 2006. Microsoft SQL Server 2005: the complete reference. McGraw-Hill Osborne Media.

[7] Tucker, A. 2006. Applied Combinatorics. Hoboken: John Wiley & sons, 49.

[8] Valiente, G. 2002. Algorithms on Trees and Graphs. Springer.