Chapter 3

Pattern Recognition

3.1 Pattern Matching

The idea behind this approach is the following: during an evolution step of the system, the user issues SQL commands against the database modifying the schema according to the new needs. In order to be effective the commands must be issued in a natural order; for instance if the user wants to copy the data from one table to a new one, he must create the table and then perform the copy of the data from the old table to the new one.

The natural order of the statement is the same accepted from each DBMS; if a user tries to issue a command to copy data from one table to a non-existing one, the command fails. Moreover when executing a command, if it involves more than one table then is always possible to get a reference of each table from the statement. This means, related to the example in the previous paragraph that when a user wants to copy data from one table to another, the copy command must always include a reference to the table the data are being copied. These implicit dependencies are the base for the *dependencies graph*.

3.1.1 Dependencies Graph

The idea behind the *Dependencies Graph* is to represent the dependencies between the statements with the dependencies between the related tables. The Dependencies Graph stores the dependencies (links), between the tables in the statements. The set of nodes and the set of edges represent respectively the set of tables extracted from the statements and the dependencies between the tables. The graph is directed and each edge connects a table to the table it is dependent on. The label on the edge is the name of the operation that generated those edges.

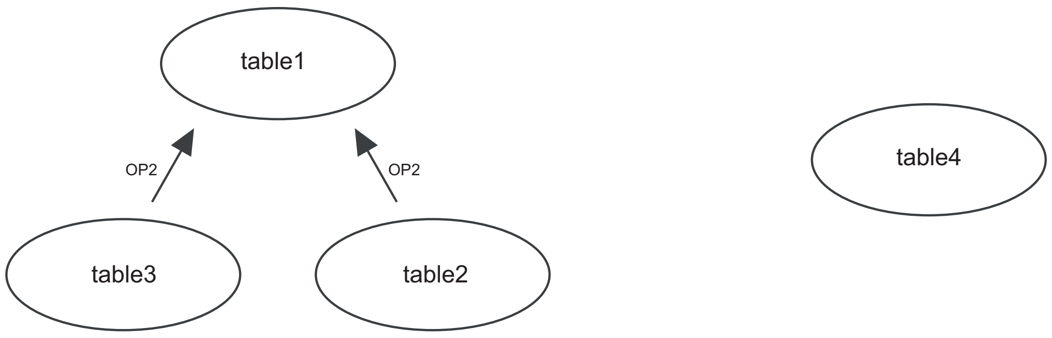
Every time the system reads a statement from the input, it firstly explores the structure retrieving the affected tables and then adds each table to the graph. If a statement affects more than one table this will generate a dependencies between those tables. Once the input has been completely read, the graph will contain a complete representation of the dependencies between the statements.

The assumption done during this step is that the system reads the set of statements from an input file without any user interaction. In order to get the correct result and create the dependencies graph the statements must be read in a batch. Subgraphs of the dependencies graph will be the input to the pattern matching phase. Before explaining in detail how the subgraphs are created starting from the dependencies graph, the following examples help to clarify the creation process of the dependencies graph.

1. The system reads from the input file the following statements:
   - operation1 table A
   - operation2 table A from table B, table C
   - operation3 table C;
   - operation4 table D;
2. Then explore the structure of each statement retrieving the affected tables. The result from the operation performed on the previous set is as follows:
   - $operation1 = \{A\}$
   - $operation2 = \{B \longrightarrow A, C \longrightarrow A\}$
   - $operation3 = \{C\}$
   - $operation4 = \{D\}$
3. Then the graph is created executing the following operations:
   - each table (A,B,C,D) is added to the graph
   - the edge B-A is added
   - the edge B-C is added

Figure 3.1 represents the resulting graph: an edge connects table3 and table2 to table1 while table4 does not have any dependencies towards any other table of the graph. The isolation of the table4 is the key to explain the subgraph extraction.

Figure 3.1: Dependencies Graph

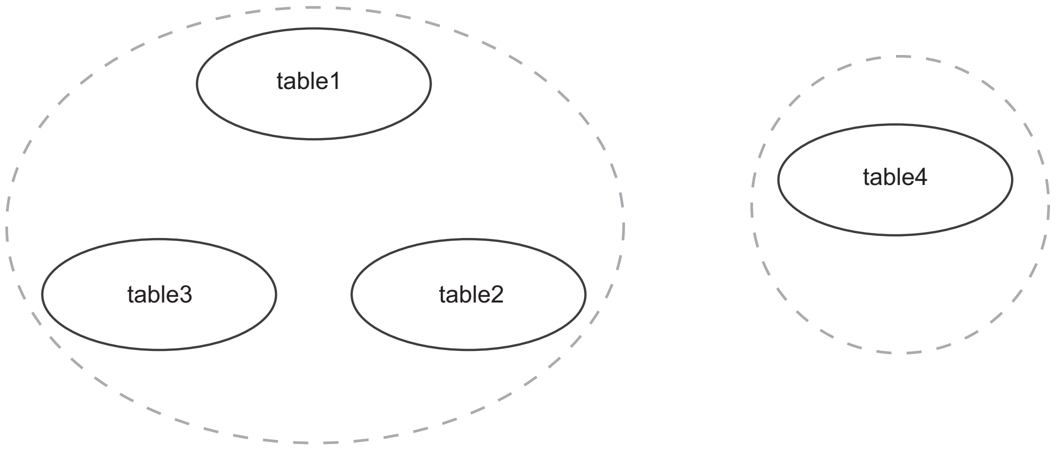


3.1.2 Connected Components

The *Connected Component* subgraph is an intermediate representation used to isolate the groups of independent tables and facilitate the exploring and statement retrieval. Once the Dependencies Graph has been created, the system traverses it seeking for connected subgraphs. Each subgraph represents a set of connected tables.

If a table has no dependencies in the graph, it means that only one statement affects that table or otherwise more statements affect the same table without involving any other. These statements can be considered independent and each statement is translated independently from the others, without any check against the patterns.

Figure 3.2: Connected Components



To start the translation, each node of the graph is explored and the statements associated to all the tables in the current node are retrieved and grouped together. In the following phase the system will try to match the retrieved batch of statement against any of patterns defined in the next section.

The following example extracted from the evolution of the Ensembl database between the revisions 224 and 226 helps to understand how the dependencies graph and the connected components mechanisms work. The patch released to evolve the database schema includes among others the following statements (a numeric reference has been assigned to each statement):

```
(2) ALTER TABLE gene ADD description text;
(4) UPDATE gene g, gene_description gd SET g.description =
gd.description WHERE gd.gene_id = g.gene_id;
(5) DROP TABLE gene_description;
(10) CREATE TABLE transcript_supporting_feature ( transcript_id
int(11) DEFAULT '0' NOT NULL, feature_type
enum('dna_align_feature','protein_align_feature'),
feature_id int(11) DEFAULT '0' NOT NULL,
UNIQUE all_idx (transcript_id,feature_type,feature_id),
KEY feature_idx (feature_type,feature_id));
```

The first operation performed by the system is the extraction of the tables affected from each statement as follows:

- $statement(2) = \{gene\}$
- $statement(4) = \{gene\_description \longrightarrow gene\}$
- $statement(5) = \{gene\}$
- $statement(10) = \{transcript\_supporting\_feature\}$

The tables extracted generate the dependencies graph in figure 3.1.2

From the previous dependencies graph are are identified the two connected components as from figure 3.1.2

```
Component 1: {gene, gene_description}
Component 2: {transcript_supporting_feature}
```

The second component has only one statement (statement 10) associated with it and the statement can be easily recovered and translated. The statement that identifies the first component are statement (2),(4),(5) as follows:

Figure 3.3: Dependencies Graph Example 1

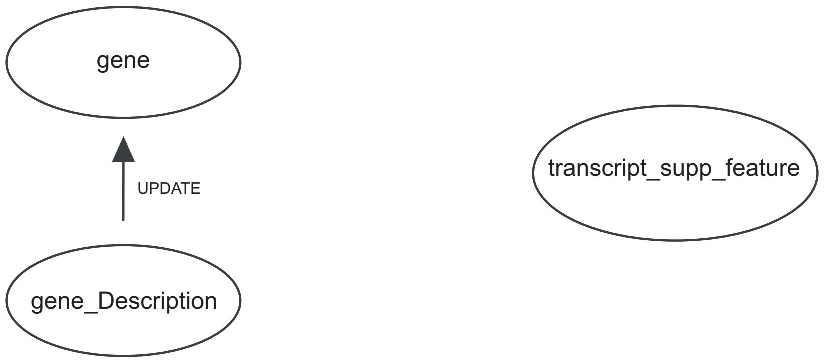
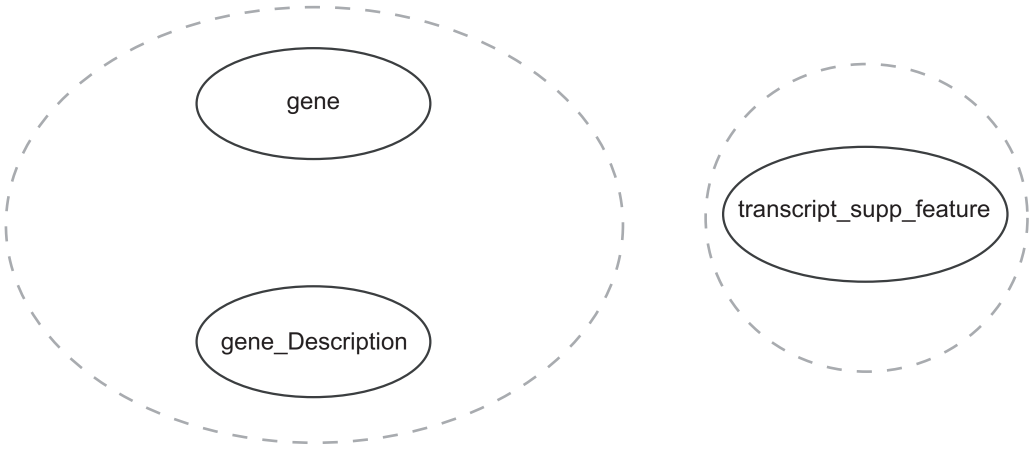


Figure 3.4: Connected Components Example 1



```
ALTER TABLE gene ADD description text;
UPDATE gene g, gene_description gd SET g.description =
gd.description WHERE gd.gene_id = g.gene_id;
DROP TABLE gene_description;
```

These statements will be matched against the pattern defined in the system trying to obtain a possible translation in terms of SMO language of the evolution undergone by the system.

3.1.3 Patterns

A pattern is a sequence of SQL statements that describe a common operation on the schema, modeled on the SMOs language [4]. The purpose of each pattern is to identify the most common sequences of SQL commands used by a DBA to obtain a specific result or evolution of the schema, and define an automatic way to recognize them. This means that if the same operation has been executed with a different sequence of SQL operation, the system is not able to recognize and translate that sequence. More important that other patterns can be defined and added to the application, improving its translating capabilities and making the system extensible and maintainable.

The SQL2SMO translator defines 5 patterns:

1. *Join Pattern type 1*
2. *Join Pattern type 2*
3. *Merge Pattern*
4. *Partition Pattern*
5. *Decompose Pattern*

The main SQL2SMO module is responsible for matching a given batch of SQL statements against each of the defined patterns. The pattern-matching mechanism is accomplished in two steps:

1. pattern-defined criteria analysis;

2. translation.

The first step verifies that the batch of statements follows some patter-defined criteria, while the second one performs the real translation. If the first step succeeds the system can continue with the second, otherwise it aborts the execution on the current batch.

The first step is in turn composed of two sub-steps:

1a. label correspondence

1b. pattern-constraint.

The label correspondence verifies that the SQL commands in the current batch are the same defined in the pattern, while the second step checks if the relations between the statements (table dependencies, etc ...) are the same defined for the pattern. If the first check fails, no errors are raised from the application: once has been verified the current batch does not conform the given pattern, the system tries to match the batch with the next pattern. Once a batch of statements has been identified with a pattern, the system produces a message informing the user of the success of the matching. If any error occurs during the translation, the batch is discarded and the batch execution terminates.

During the translation step, the system tries to translate the sequence of SQL commands into one or more SMOs. This step performs other kind of checks and it is possible that some errors are raised during the attempt of translation.

The translation-safe behaviour of the system is an important design choice: giving the system strict check and translation constraints, allows its use without worrying of the possible side-effects. Any error occurred during the translation is reported to the user, and every time the system can't perform the translation, it aborts its execution, without forcing the translation. As a consequence, every time a translation is generated, its formal accuracy is guaranteed.

3.1.4 Join Pattern Type I

The *Join Pattern type 1* has been designed, over the SMO JOIN, following one of the two most common ways to perform a join in the SQL language. It is defined by the following SQL statements:

- CREATE TABLE a
- INSERT into a (SELECT (..) FROM b)
- DROP TABLE b* (*optional)

This *Join Pattern type 1* cover the need of joining two tables into a new one. The user, performing this operation, creates a new table ready to host the joined data, selects the desired columns from the source tables according to the new table's signature and migrates the data from the old tables to the new one. Optionally either one or both source tables can be dropped after the data migration.

In section 3.1.3 was discussed how the pattern matching mechanism is organized in two sub-phases: matching the given statements against some predefined criteria and translating the pattern. The first phase of the matching is in turn divided in two sub-phases. As the first sub-phase is trivial and consists only in a correspondence check between the statements label and the pattern labels, this and the following sections will mainly focus on illustrating the second phase of the matching and the translation.

Before stating the correspondence between the given set of statements and the *Join Pattern type 1* the system performs the following checks:

- the tables in the INSERT statement must be more than one;
- the table in the CREATE and INSERT statements must be the same
- if a DROP statement is in the sequence, the table dropped must be one of the tables involved in the join

Once executed the previous operations, the system is ready to translate the statements. The following section explain in details some crucial operations performed by the system during the translation.

drop table

If one or more tables are dropped after the data migration, the system performs a copy of the tables which are not dropped and works for the remaining part of the translation process on the copy. This behaviour derives from the SMO JOIN definition that consumes both the source tables once the join is completed. If the tables remain in the schema after the join, then is necessary to copy the tables before performing the join.

columns and table rename

The columns from the INSERT clause and the SELECT clause are compared looking for any possible renaming needed. If all columns are selected using the *start* operator, the check is performed against all the columns of the table. The system does not perform any check whether the signature of the columns is the same as this would make the SQL statement fail when issued against the DBMS. Each matching pair of columns is compared and if the name of the source and destination column differ, a RENAME SMO is generated.

As the join SMO is defined creating a new table in the schema, the name given to the newly created table is the name of the table in the CREATE TABLE statement

partition analysis

The INSERT statement could involve all the data in the source tables or only a portion of those data. Selecting a portion of the data means using a *WHERE* condition in the SELECT statement, partitioning the source according to condition.

The system manage this situation looking for any condition inside the WHERE clause involving a column and a scalar. The condition can either be of $<=, >=, =$. It then generates a PARTITION SMO, dividing the partitioned tables in two subtables: the useful part, to which the system assigns the name *part_1* is kept; the *part_2* is dropped. The partition condition for the part_1 is the same available in the WHERE condition, while the condition for the part_2 is reversed accordingly.

The following example extracted from the evolution of the Mediawiki database between the revisions 17217 and 17244 helps to understand the *Join Pattern type 1* mechanism.

```
(1) CREATE TABLE redirect (rd_from int(8) unsigned NOT NULL,
rd_namespace int NOT NULL default '0',rd_title varchar(255));

(2) INSERT INTO redirect (rd_from,rd_namespace,rd_title)
SELECT pl_from,pl_namespace,pl_title
FROM pagelinks, page
WHERE pl_from=page_id AND page_is_redirect=1;
```

The goal of the evolution step is to migrate the record containing redirected-page information to the newly created table *redirect*. Once verified all the pattern conditions are satisfied, the system produces the following output in terms of SMO:

1. As both the source tables are still available in the schema after the evolution, the system performs a copy of those tables

   ```
   COPY TABLE page INTO page_copy;
   COPY TABLE pagelinks INTO pagelinks_copy;
   ```

2. it then delete with a PARTITION operation the unnecessary row from the page table

   ```
   PARTITION TABLE page_copy INTO page_copy_part_1 WITH
   page_is_redirect = 1, page_copy_part_2 WITH
   page_is_redirect <>1
   ```

3. it then performs a join between the two table to obtain the desired redirect table

   ```
   JOIN TABLE page_copy_part1, pagelinks_copy INTO redirect
   WHERE pagelinks_copy.pl_from=page_copy_part_1.page_id
   ```

4. it finally drops the unnecessary columns from the destination table. As the join between the page and pagelinks tables result in a table with more column than the redirect table, the system drop the column that shouldn't be available in the final table.

   ```
   DROP COLUMN page_restriction FROM page;
   DROP COLUMN page_counter FROM page;
   DROP COLUMN page_random FROM page;
   DROP COLUMN page_restriction FROM page;
   DROP COLUMN (...) FROM page;
   ```

5. If any of the destination columns had had a different name from the selected ones, the system would have rename that column.

3.1.5 Join Pattern Type II

The *Join Pattern type 2* has been designed, over the SMO JOIN, covering the second most common way to perform a join in the SQL language. It is defined by the following SQL statements:

- ALTER TABLE a ADD COLUMN c
- UPDATE TABLE a,b SET a.c = b.c WHERE (..)
- DROP TABLE b* (*optional)

The *Join Pattern type 2* covers the need of moving a table column from a source to a destination table. This can be either explained with the need of creating a foreign key column in the destination table, or with a database restructuring which involves the elimination of a table and the migration of one ore more columns to another one. In both the situations the user modify the destination table adding a column to host the new data and then update the column with the data from another table. Eventually if the source table is no longer needed, the user can drop it.

It is important to note how the evolution described in the *Join Pattern type 2* can be easily brought back to the *Join Pattern type 1*İnstead of adding the new columns to the table, it would have been possible to create a new table with the referenced column, move the data to the table joining the source and destination table and selecting only the desired column and eventually drop the source table.

Before stating the correspondence between the given set of statements and the *Join Pattern type 2* the system performs the following checks:

- the table of the ADD COLUMN statement is the destination table of the UPDATE;
- the column of the SET clause in the UPDATE is the same column of the ALTER TABLE ADD COLUMN;
- the source of the SET clause is a column; a number is not allowed;
- if a DROP statement is present, the table involved is the source table of the UPDATE.

Once executed the previous operations, the system is ready to translate the statement. The following section explain in details some crucial operations performed by the system during the translation.

drop table

The drop table phase works in the same way as for the join type 1.

partition analysis

The partition analysis starts from the work discussed in the previous section: the table involved in the WHERE condition, if present, is partitioned according to the method explained for the previous join. The main difference involves the useless part of the partition, the sub-tables that is dropped in the *Join Pattern type 1*İn this case, as what really happens is a data migration from the source table to the destination table, the partition not involved in the join, will not be discarded, but will be merged back to the joined table after the join execution.

In order to be merged, the part_2 must have the same signature of the other part, that after executing the join has increased by one its number of columns. The column added to the part_1 must be added also to the part_2 before performing the merge.

The following example extracted from the evolution of the Mediawiki database between the revisions 7649 and 7653 helps to understand the *Join Pattern type 2* mechanism.

```
(1) ALTER TABLE ipblocks ADD ipb_by_text varchar(255)
binary NOT NULL default '';
(2) UPDATE ipblocks p, user u SET ipb_by_text = user_name
WHERE p.ipb_by = u.user_id AND ipb_by != 0;
```

The goal of the evolution step is to modify the ipblocks table adding the column ipb_by_text with the information contained in the user_name column of the table user. Once verified all the pattern conditions are satisfied, the system produces the following output in terms of SMO:

1. As the source table user is still available in the schema after the evolution, the system performs a copy of the table

   ```
   COPY TABLE user INTO user_copy;
   ```

2. it then split the table destination table separating the rows that must be updated from the others. As made clear by the UPDATE command, the rows involved in the update are the ones with the ipb_by column different from zero.

```
PARTITION TABLE ipblocks INTO ipblocks_part_1 WITH
ipb\_by <> 0, ipblocks_part_2 WITH
ipb\_by = 0
```

3. it then performs a join between the two table to obtain the desired redirect table

   ```
   JOIN TABLE ipblocks_part1, user_copy INTO ipblocks
   WHERE ipblocks_part_1.ipb_by = user_copy.user_id
   ```

4. it then add the new column to the partition 2 of the ipblocks table to adequate the table's signature

   ```
   ADD COLUMN ipb_by_text varchar(255) INTO ipblocks_part_2;
   ```

5. finally merges back the two two part of the ipblocks table to reconstruct the original table

   ```
   MERGE TABLE ipblock_part_1, ipblocks_part_2 INTO ipblock
   ```

3.1.6 Merge Pattern

The merge pattern is composed of an INSERT and eventually a DROP statement. The relations between the two statements are as follows:

- if a table is dropped, it must be the source table in the INSERT statement
- the source and destination table must have the same signature

signature comparison

The SMO MERGE works comparing the tables' signature: if the signature are the same, than the tables can be merged, otherwise the merge fails; In order to compare the tables signature, the system analyzes the columns from the SELECT and INSERT INTO clause distinguishing the following situations: *no* means that no columns are selected in the clause, in other word the *star (*)* operator was used; *yes* means that a list of columns was used in the clause.

1. INSERT *yes*, SELECT *yes*;

2. INSERT *yes*, SELECT *no*;
3. INSERT *no*, SELECT *yes*;
4. INSERT *no*, SELECT *no*;

In every situation the system retrieves the columns used in the statement and compares their signature. The most interesting cases are the first and the second, when a selection of the destination columns occurs. In these cases the system must insert the data from the source table only in the columns selected by the user. In order to perform the task the system works as follows:

- copies the destination table
- drops the columns not involved in the insertion
- drops the columns from the source table not involved in the selection
- merges the source and destination table

Then the table obtained from the previous steps must be joined back with the original table to add the columns not involved in the merge operation which will get a null value for every row inserted from the source table.

3.1.7 Decompose Pattern

The Decompose Pattern, according to the SMO DECOMPOSE operator, manage the decomposition of a table into two subtables. The decomposition is carried out splitting the table's column into two destination tables. The pattern is composed by the following statements:

- CREATE TABLE a
- CREATE TABLE b
- INSERT INTO a SELECT (c1,c2,c3) FROM c
- INSERT INTO b SELECT (c1,c4,c5) FROM d
- DROP TABLE c* (*optional)

The first two statements are responsible for creating the two destination tables to accept the data from the source tables, while the two INSERT statement transfer the data from the source to the destination tables. A possible DROP TABLE statement drops the migrated table from the schema.

The relations between the statements are as follows:

- the destination tables from the INSERT statement must match the created tables
- the source table in both the INSERT statement must be the same
- if a DROP TABLE statement is present, the table dropped must be the source table
- at least one column in both the INSERT statement must be available

double partitioning

The double partitioning arose when the partition condition of the two INSERT statement is different and there is the need to move the data from the source table to the destination table according to different conditions. The two tables are partitioned accordingly and the DECOMPOSE smo is applied to the correspondent partition.

If the partition condition is the same in both the INSERT statements, then the source table is partitioned only once and the decomposition is applied following the columns selection given by the user.

3.1.8 Partition Pattern

The Partition pattern, according to the correspondent SMO PARTITION operator, manages a horizontal partition of the tuples from the input table. The statements composing the pattern are:

- CREATE TABLE a
- INSERT INTO a SELECT (...) FROM c WHERE condition
- DELETE FROM c WHERE condition

The fisrt statement creates the table that will host the moved data from the source input table, the second transfer from the input to the destination table. The DELETE statement deletes the moved tuples from the input table.

The relation between the statements are as follows:

- the created table and the destination table coincide
- the DELETE statement refers to the source table in the INSERT statement
- the table being partitioned is the source table
- the DELETE and INSERT statement must contain the same partition condition

columns rename

The renaming of the columns follows the same procedure detailed in section 3.1.4

partition analysis

The partition analysis follows the same procedure detailed for the Join Pattern type one in section 3.1.4.

3.2 PowerSet Computation

The mechanism explained in sections 3.1.1 and 3.1.2 explain how the system derives from the evolution step the set of statement to match against the pattern and produce the corresponding translation. It may happen that the set of statements retrieved by the system is too large to be compared with a pattern and the sequence of commands is too complex to be translated. The system wouldn't be able to propose a translation to the user, leaving him the burden to migrate the evolution step towards the SMO language. In these situation would be helpful to have a sort of *feedback* returned by the system, a guideline for the migration of the script in the SMO language, helping the user in the translation.

In addition, when the sequence of statements is too large or complex, does not necessary mean that a translation cannot be derived for that sequence or that all the statements are involved in the same operation. From the design of the *Dependencies Graph* in section 3.1.1 can be gathered that whenever the output of an evolution operation is the input for the successive operation on the schema (meaning the output tables involved in the first sub-step are used as input in the following sub-step), this would derive in a unique connected components. All the statements in the components will be tested against the pattern and none of the comparison will succeed. It is clear however that, if the operations performed in the two sub-steps were considered individually, the output could have been different and the system could have obtain a translation.

The solution to the discussed problems is the *Power set* computation. The power set of a set of statement S, is the set of all the subset of S. Whenever the batch-to-pattern comparisons fails, then the system computes the power set of the set of statement, extracting each possible subsequence of statements from the current batch, and performs a new batch-to-pattern comparison for any of the subsequences identified.

From the definition of the power set derives that given n elements in the set, the power set contains 2^n elements.

The following example continues the one reported in section 3.1.1 and 3.1.2, considering all the statement really released in the evolution of the Ensembl database between the revisions 224 and 226. It helps to clarify the results obtained with the power set mechanism. The statements in the script are as follows (a numeric reference has been assigned to each statement):

```
(1) ALTER TABLE gene CHANGE type biotype VARCHAR(40) NOT NULL
default 'protein_coding';
(2) ALTER TABLE gene ADD description text;
(3) ALTER TABLE gene ADD source VARCHAR(20) NOT NULL
default 'ensembl';
(4) UPDATE gene g, gene_description gd SET g.description =
gd.description WHERE gd.gene_id = g.gene_id;
(5) DROP TABLE gene_description;
(6) ALTER TABLE transcript ADD biotype VARCHAR(40) NOT NULL
DEFAULT 'protein_coding';
(7) ALTER TABLE transcript ADD description text;
(8) UPDATE transcript t, xref x SET t.description =
x.description WHERE t.display_xref_id = x.xref_id;
(9) UPDATE transcript t, gene g SET t.biotype = g.biotype
WHERE g.gene_id = t.gene_id;
(10) CREATE TABLE transcript_supporting_feature ( transcript_id
int(11) DEFAULT '0' NOT NULL, feature_type
enum('dna_align_feature','protein_align_feature'),
feature_id int(11) DEFAULT '0' NOT NULL,
UNIQUE all_idx (transcript_id,feature_type,feature_id),
KEY feature_idx (feature_type,feature_id));
```

The first operation performed by the system is the computation of the dependencies graph as follows:

From the previous isolation graph are identified the two connected subgraph as from figure 3.2

```
Component 1: {gene, gene_description, transcript, xref}
Component 2: {transcript_supporting_feature}
```

The second component, as was in the previous example, has only one statement (statement 10) associated with it and the statement can be easily recovered and translated into a CREATE TABLE SMO. The statement that

Figure 3.5: Dependencies Graph Powerset Example

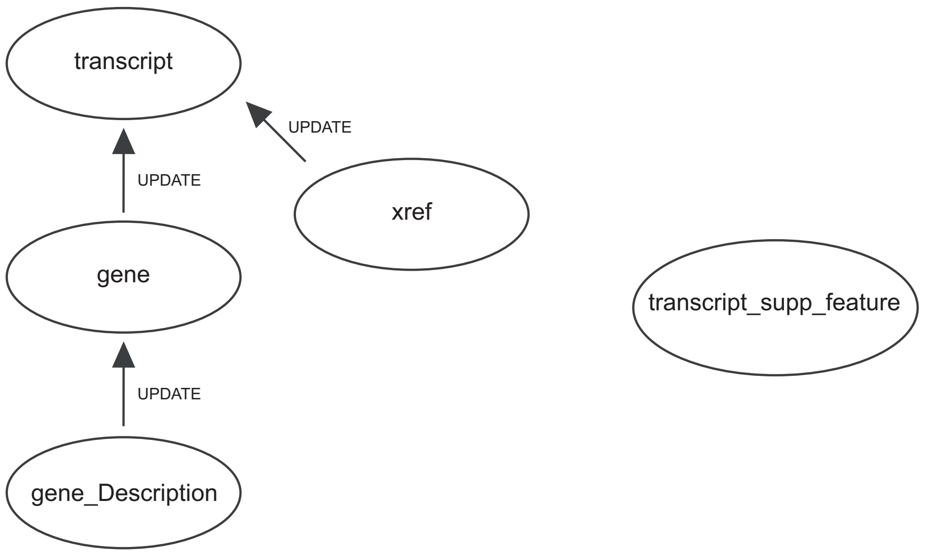
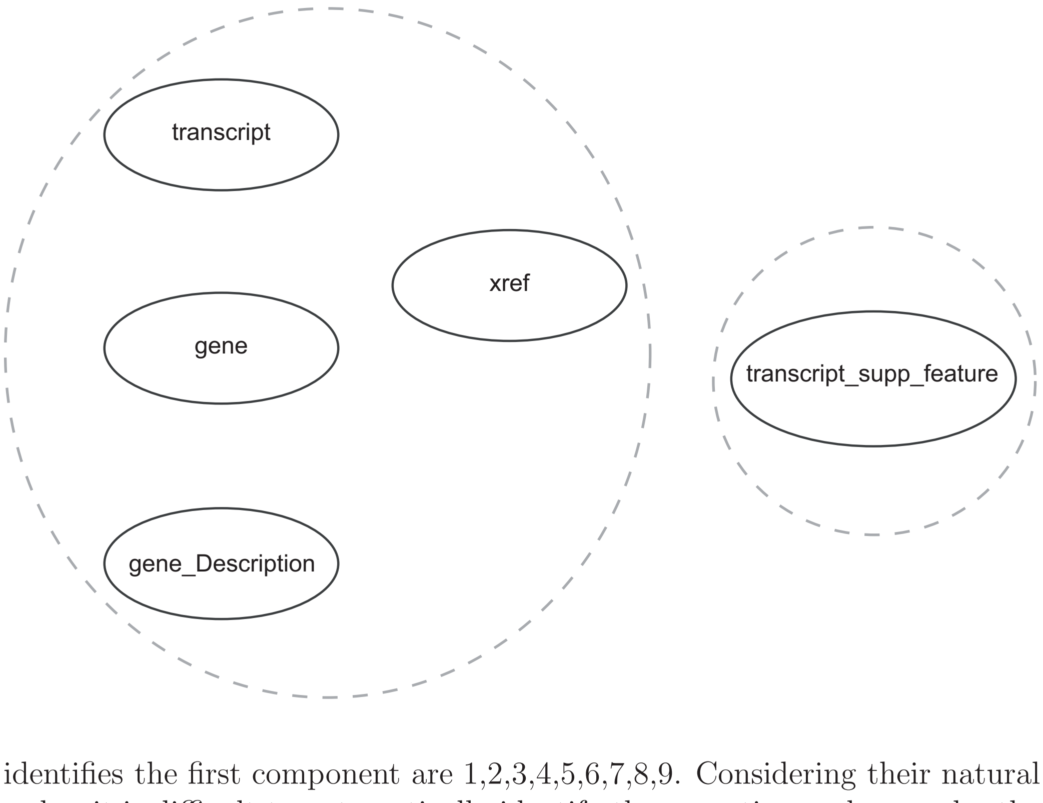


Figure 3.6: Connected Components Powerset Example



identifies the first component are 1,2,3,4,5,6,7,8,9. Considering their natural order, it is difficult to automatically identify the operation undergone bu the schema. The power-set computation, allow to identify 3 fixed patterns inside the batch of statement. The first one is composed by the statements 2,4,5 as follows

```
ALTER TABLE gene ADD description text;
UPDATE gene g, gene_description gd SET g.description =
gd.description WHERE gd.gene_id = g.gene_id;
DROP TABLE gene_description;
```

The subset of statement is identified and translated as a Join pattern of type 2 as discussed in section 3.1.5

The second pattern identified in the power-set is composed by the statements 7 and 8, while the last pattern is composed by the statements 6 and 9. Both the sequences are identified and translated as a join pattern of type 2. From the previous analysis appears how the statement not involved in any pattern are the statement 1 and 3. These represent an isolated operation over the schema and can be translated accordingly as two ADD COLUMN SMOs.

As expressly emphasized in the previous examples, if a correspondence is found, the system propose a possible translation to the user, informing that the user that he proposed translation is not 100% accurate as derived from a guess made by the system within a batch of multiple statements. The system in addition displays the statement that could not be translated, giving the user the possibility to change the SMO output and working on his own SQL translation.

This discussed mechanism not only improve the system translating capabilities, but also helps the user getting acquainted with the SMO language as allows the understanding of more complicate evolution steps.

3.3 Experimental Result

In order to test the accuracy of the proposed solutions, the system has been tested over the two system introduced in section 1.5.3. The complexity and the reality-modeled derivation of the two system, made Mediawiki and Ensembl two perfect testbeds for the SQL2SMO application.

The results summarized in table 3.1 and 3.2 demonstrate the validity of the model proposed and how the SQL2SMO system can facilitate the learning process of PRISM SMO language and the migration towards the PRISM framework succeeding in the translation of almost 98% of the schema evolution steps.

Table 3.1: Toolsuite detailed results

	Wikipedia	**Ensembl**
evolution steps analyzed	248	28
relevant evolution steps analyzed	142	24
SQL instructions	226	174
steps correctly translated	131	17
steps partially translated	9	7
steps not translated	1	0
generated SMOs	319	
generated SMOs (pattern-matched)	26	

Table 3.2: Toolsuite aggregated results

	Wikipedia	**Ensembl**
Translation Rate (fully translated)	92.25%	70.83%
Translation Rate (plus partially translated)	98.59%	100%