Acta Cybernetica **21** (2014) 629–653.

Database Slicing on Relational Databases*

Dávid Tengeri† and Ferenc Havasi‡

Abstract

Many software systems today use databases to permanently store their data. Testing, bug finding and migration are complex problems in the case of databases that contain many records. Here, our method can speed up these processes if we can select a smaller piece of the database (called a slice) that contains all of the records belonging to the slicing criterion. The slicing criterion might be, for example, a record which gives rise to a bug in the program. Database slicing seeks to select all the records belonging to a specific slicing criterion. Here, we introduce the concept of database slicing and describe the algorithms and data structures necessary for slicing a given database. We define the *Table-based* and the *Record-based slicing* algorithms and we empirically evaluate these methods in two scenarios by applying the slicing to the database of a real-life application and to random generated database content.

Keywords: database theory, program slicing, database slicing

1 Introduction

Program slicing is an interesting topic in software engineering. This concept was first introduced by Weiser [12, 13, 14] and it has grown into a respectable area of research. The technique can be used to select a small part of the program by choosing the relevant statements from the source code. These statements are interesting to the developer from some point of view called the slicing criterion. For example, a variable may contain an incorrect value because of a bug in the program. We can select those statements which are probably responsible for the computation of the incorrect value stored in the variable and examine them in more detail. This type of slicing is called backward slicing. With program slicing, we can also determine which parts of the program must be retested after the modification

*This study was supported by the European Union and co-financed by the European Regional Development Fund within the project TÁMOP-4.2.1/B-09/1/KONV-2010-0005.

†PhD student, Software Engineering Department of the University of Szeged, Hungary, E-mail: `dtengeri@inf.u-szeged.hu`

‡Assistant lecturer, Software Engineering Department of the University of Szeged, Hungary, E-mail: `havasi@inf.u-szeged.hu`

of a statement in the program source code. This latter type of slicing is called forward slicing.

The program slicing of database-driven applications introduces new problems. The statements responsible for handling data in the database are often not present in the output of a slicing procedure; however, attempts have been made to extend program slicing with database support [1].

Our research question was how we could define slicing on relational databases, and how useful it is in practice.

We defined *database slicing*, which means selecting a small piece of data from the records or tables stored in the database, where each element of the slice is related directly or indirectly to the slicing criterion and this small piece might be an independent part of the database. The slicing criterion is the *Starting Point of Slicing* and the result is a *database slice*. Database slicing may have other applications. For instance, the slice can be used for migration between two or more databases that have the same structure. Slicing can also improve the process of bug finding and testing by selecting those records which probably give rise to a bug, so the developers can then focus on the more relevant parts of the database. Database slicing can also be useful when we wish to understand the structure of the database or to discover connections between the data stored in separate tables. It can be especially useful when a relevant change is necessary in the structure of database (e.g. change the value of the key column) because slicing can help to determine the impact of this change.

Here, we make the following contributions:

- We define the concept of database slicing and the two different type of slicing algorithms concerning relational databases.

- We test the new algorithms in two different scenarios. The first one was a reproducible environment based on the Drupal CMS and the second one was a real-life application developed at out department and used by more than 40 hospitals daily. We measured the memory consumption and execution time of the algorithms in addition to the selection capability.

This paper is organized as follows. Section 2 contains the required background knowledge needed to understand the concepts introduced later like the basic definitions of database theory and program slicing. In Section 3, we present our definitions of database slicing. In Section 4 we provide a detailed description of slicing algorithms, then to compute a database slice, it is necessary to know the dependencies among the tables. Section 5 we present results of our implementation tested on popular PHP-based content management systems, and a real-life application. After giving an outline of some related articles in Section 6, we round off with a brief summary and suggest some possible future directions of study.

2 Background knowledge

Here, we provide the background knowledge needed to understand the theory of database slicing.

2.1 Database

A lot of today's software uses databases to store their data. In most cases, these databases are relational databases. The general basis of a relational database is the one defined by E.F. Codd in 1970 [4, 5].

We will now describe the data in the relational model via relations. These relations store their data in tables that have columns and rows. Moreover, the model contains relational operators that can be used to handle the data stored in the tables. The following definitions are needed for the relational model ([5]):

Domain: The set of data values of similar types. Let $D_1, D_2, \ldots, D_n$ $((n > 0))$ be domains. Then the $D_1 \times D_2 \times \ldots \times D_n$ set contains all possible n-tuples $\{t_1, t_2, \ldots, t_n\}$, where $t_i \in D_i$ for all i.

Attribute: Instead of indices $(1, 2, \ldots, n)$ used in the previous definition, we can use an unordered set where we store not just the domain, but also the unique identifier of each $t_i \in D_i$ element. Hence the elements of $\{t_1, t_2, \ldots, t_n\}$ will be $(A : v)$ pairs, where A is the attribute of the element and v is a value taken from the domain of A.

Relation: This is a subset of $D_1 \times D_2 \times \ldots \times D_n$, where its elements have the same attributes. Each relation has a tabular representation with the following properties:

1. It does not contain duplicate rows (records).
2. The row order is irrelevant.
3. The attribute (column) order is irrelevant.
4. Each element of the table is atomic; that is, the database management system (DBMS) can store its value directly.

Let $R(A : a, B : b, C : c, \ldots)$ be the relation, where A is an attribute whose domain is a, B is an attribute whose domain is b, etc. Here we can ignore the domains for the sake of clarity - hence we will just use $R(A, B, C, \ldots)$. A relation is a base relation if its elements cannot be derived from any other base relation. The base relations will be the data tables in the DBMS.

Relational schema: The description of the database elements via relations.

Key: Let U be a set of relation attributes. The $U - component$ of $t \in R$ is the set of $(A : v)$ pairs obtained by deleting those pairs from t whose attribute is not in U.

There is a set of candidate keys for each relation. Let K be a candidate key of relation R if K is a subset of the attributes of R. Moreover, we will assume that

1. There are no two rows in R with the same $K - component$.
2. If any of the attributes have been deleted from K, then the above statement (the uniqueness property) will not hold.

Primary key: For each relation, one of the candidate keys is selected as a primary key.

Foreign key: Links between tables are not represented by pointers but data values. The association is a reference to the value of the key.

In [4], a foreign key is an attribute or a set of attributes of R which is not the primary key of R, but contains the value of the primary key of another relation S.

Example 1. Let us take a simple example which contains three tables taken from a website. The *users* table stores the users of our site. The *node* table contains the content sent by the users and the *comments* table stores the comments written to the nodes by the users. The relational schema of the database is:

- users(<u>uid</u>, name, pass, mail)
- node(<u>nid</u>, *uid*, title, body, created)
- comments(<u>cid</u>, *nid*, *uid*, comment, timestamp)

The underlined attributes are the primary keys and attributes in Italics are foreign keys. For example, node.*uid* and comment.*uid* reference the value of users.*uid*.

Example 2. The above-mentioned database in tabular form containing some sample data:

Table 1: Users of the sample website.

users			
uid	name	pass	mail
1	admin	452ce338	admin@localhost
2	shatho	ec80ac93	shatho@localhost
3	sleshec	02f27fcf	sleshec@localhost

Table 2: Content sent by users.

node				
nid	uid	title	body	created
1	2	Zelus Esca	Valde roto vicis metuo cau...	2010-01-21
2	3	Hendrerit	Capto oppeto letalis incass...	2010-02-14
3	1	Humo Cam...	Huic neque tincidunt...	2010-02-17
4	3	Praemitt...	Jumentum nobis si loqu...	2010-03-01
5	2	Si Natu D...	Hos luctus quadrum. M...	2010-03-23

Table 3: Comments of the users.

comments				
cid	nid	uid	comment	timestamp
1	1	2	Tego aliquip exerci nimis nunc torq...	2010-02-01
2	1	2	Luctus nisl populus gilvus consequa...	2010-02-02
3	2	1	Inhibeo abbas imputo patria quae. N...	2010-02-14
4	3	3	Jus ibidem roto pertineo lobortis i...	2010-02-21
5	2	3	Qui roto incassum refoveo uxor sing...	2010-03-01
6	2	1	Exputo mauris pertineo vulpes typic...	2010-03-02
7	5	2	Gemino tamen zelus bene quae jus ca...	2010-03-24
8	4	1	Appellatio esca defui abigo suscipi...	2010-03-25

2.2 Program slicing

Finding and fixing bugs in large and complex software systems is not a trivial task. It is often not easy to determine which parts of the program are affected by a bug.

A *program slice*, introduced by Weiser [12, 13, 14], contains the relevant parts of the source code that are important from a slicing point of view. This view is a statement of the program and this point will be called the *slicing criterion*. The process of determining the program slice is called *program slicing*.

Program slice: The program slice S may be obtained from the program P by dropping statements, so S reproduces a part of the functionality of P.
Hence, the slice is a subset of statements of a program P. The elements of this set participate directly or indirectly in computing the value at the slicing criterion.

Slicing criterion: This is the starting point of slicing. The outcome of program slicing will contain those statements which participate in the computation of the value at the slicing criterion. In general, it has the form (line in source code,set of variables) or (input values, line in source code,set of variables). The actual form depends on the type of slicing performed.

There are two types of program slicing, namely static slicing and dynamic slicing. Both slicing methods have two directions - forward and backward slicing -, but they have different purposes in each case.

Static slicing: In static slicing, the only input is the source code of the program so we do not have any dynamic information such as the input data of the program or user events. Weiser generates sets of statements in his algorithm. But Ottensetin and Ottenstein outlined another approach in [10]. They defined the program dependence graph (PDG) and the computation of a program slice is turned into an accessibility problem.

PDG: This is a directed graph where its vertices are the statements of the program and the edges represent the data and control dependencies between the statements. The slicing criteria is a vertex of the graph and the results of slicing contain the vertices that are accessible via the slicing criterion.

There are two directions of static slicing. The difference between them is the direction of graph traversal.

Backward slicing: With backward slicing, we start traversing the graph in a backward direction from the starting point. This slicing method looks for those statements which take part in computing the value of the slicing criterion. The results of slicing can later be used to find bugs in a program.

Forward slicing: The results of a forward slicing will contain those statements that depend on the value of the slicing criterion. This is useful if we wish to determine which parts of the program will be affected by our modification, so we know from the results which parts of the modified program ought to be retested.

Example 3. Static slicing. This example was taken from [6]. It shows a typical static backward slicing of a program. The slicing criterion was the pair (10,product).

Dynamic slicing: The second type of slicing is dynamic slicing, which works with a specific execution of the program, so it depends on the particular input values and user events. Based on this data we can describe an execution history of the program. A single program statement may appear many times in the execution history and in the slice if it is in a loop or in the body of a function that is called several times. This leads to a more precise solution than that for static slicing, so the results of slicing here contain only those dependencies which take part in the computation of the value of the variable given in the slicing criterion.

```
        Original program            | Result of slicing
                                    |
(1)     read(n);                    | read(n);
(2)     i := 1;                     | i := 1;
(3)     sum := 0;                   |
(4)     product := 1;               | product := 1;
(5)     while i <= n do             | while i <= n do
        begin                       | begin
(6)         sum := sum + i;         |
(7)         product := product * i; |     product := product * i;
(8)         i := i + 1;             |     i := i + 1;
        end;                        | end;
(9)     write(sum);                 |
(10)    write(product);             | write(product);
```

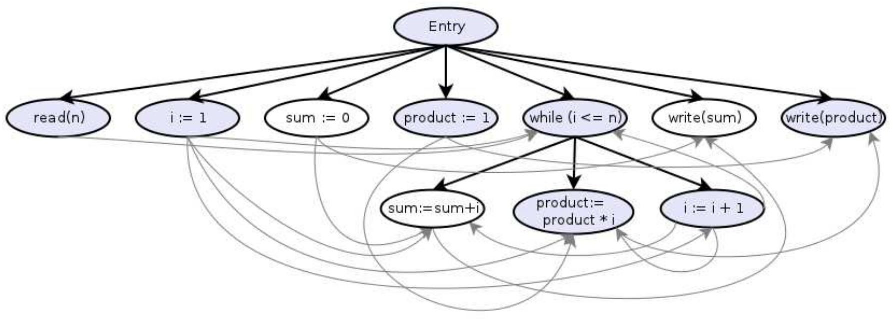


Figure 1: The program dependence graph of the program given in Example 3. The grey vertices are included in the results of the slicing.

Slicing criterion: The slicing criterion now contains an input of the program, so it has the form (x, I^q, V), where x is the input, I in I^q is the line number in the source code, q in I^q is the number of statement in the execution history of the program, and V is a subset of variables.

Backward slicing: With Backward slicing the dynamic case is different from the static case. The direction in the dynamic case will mean the way of computing the slice, not the direction from the statement location. First, we execute and record the execution history. Then we look for the slicing criteria in the history and start the slice backwards from that point. This can produce more precise information for debugging purposes.

Forward slicing: The result can be computed at the time of execution, so this method can save computer resources, as we don't have to store the execution history of the program and start the slicing just after the running of the program.

DDG: The Dynamic Dependency Graph is defined by Agrawal and Horgen in [7]. This graph contains a node for each statement occurrence in the execution history. The slice will contain the reachable nodes from the starting point.

Example 4. An example of dynamic slicing is taken from [6]. Here, the slicing criterion is (n = 2, 9^{14},z).

		1^1 read(n)	
(1)	read(n);	2^2 i := 1	read(n);
(2)	i := 1;	3^3 i <= n	i := 1;
(3)	**while** i <= n **do**	4^4 (i **mod** 2 = 0)	**while** i <= n **do**
	begin	6^5 x := 18	**begin**
(4)	**if** (i **mod** 2 = 0)	7^6 z := x	**if** (i **mod** 2 = 0)
(5)	x := 17;	8^7 i := i + 1	x := 17;
	else	3^8 i <= n	**else**
(6)	x := 18;	4^9 (i **mod** 2 = 0)	;
(7)	z := x;	5^{10} x := 17	z := x;
(8)	i := i + 1;	7^{11} z := x	i := i + 1;
	end;	8^{12} i := i + 1	**end;**
(9)	write(z);	3^{13} i <= n	write(z);
		9^{14} write(z)	
	Example program	Execution history for input $n = 2$	Result of slicing

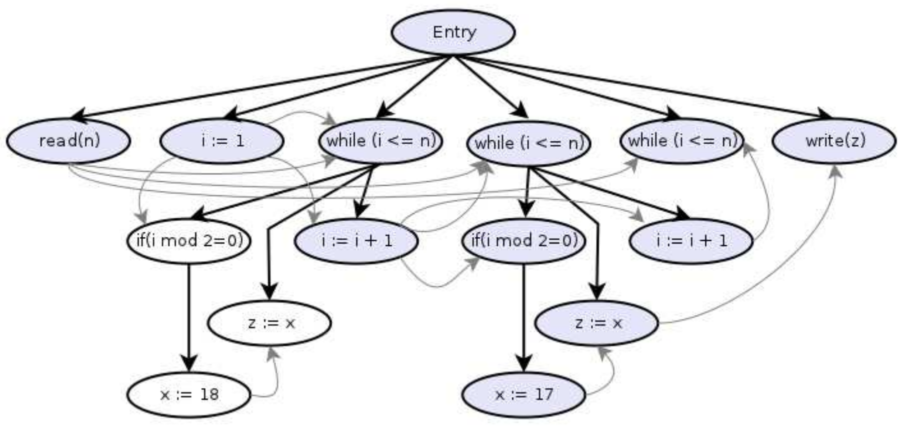


Figure 2: DDG of the program presented in Example 4

Static and dynamic slicing can be combined to compute more relevant slices. For example, in [17] the authors combined static and dynamic information to compute relevant slices. Relevant slices can be used for regression testing. A relevant slice with respect to a variable occurrence includes all statements that may affect the value of the variable. The relevant slices can be used to determine the test cases in a test suite where the modified program may differ from the original one. The authors in [17] first compute the dynamic slice of the program, then they generate an extended program dependence graph. They add two new dependences between statements of the program called conditional dependence and potential dependence. The relevant slice can be computed by combining information obtained from the extended PDG and the dynamic slice of the program. Their algorithm can be used in a variety of real-life applications.

3 Database slicing

Working with large databases and looking for bugs arising from data stored in a large system or moving data from one database to another raises several issues that need to be addressed:

- Which parts of the database do we need to reproduce a bug?
- Which is the smallest, but still coherent piece of the database that contains the relevant data to reproduce a bug? How should we select this part?
- How can we move just a small piece of the data from one database to another if we want to be sure that each relation of the transported data will exist in the target database at the end of the process? How should we select the relevant records from the database?

Consider the following example based on the database defined in Example 1. When the comments are listed on a page the user name of the second comment has disappeared. If a developer wants to reproduce this bug, she or he has to access the complete database. Perhaps the developer has to copy the database to a local computer. It takes a lot of time if the database contains a lot of data and the developer does not need everything from the database to reproduce the bug. The developer needs just one specific comment and all related content.

Database slicing can help the user to address and handle these issues. Here, we define the key concepts of database slicing.

Database slice: A subset of a database. The slice will contain those database records that have a direct or indirect relation with a given point, which will be taken as the starting point of slicing. A slice may contain dangling references or it may be an independent part of the database.

Database slicing: This is the process of slicing a database. The idea behind database slicing was naturally inspired by program slicing. We incorporated the notion of program slicing into the database context.

Starting point of slicing (SPS): We begin the slicing from this point. The results will include information about those database records or tables that have a direct or indirect connection with this point. Inter-table relations are based on foreign keys in relational databases.

3.1 Types of database slicing

In analogy with program slicing, we will define several kinds of slicing and the directions for database slicing. The first two types depend on the basic elements of the slicing.

Table-based slicing: Table-based slicing works with the tables of a database. It is only aware of the database structure and does not know anything about the data stored in the tables. SPS will be a set of database tables in this case and the results will contain those tables whose records have a direct or indirect relation with each other. Table-based slicing uses only the database schema (structures of the tables and its connections) and it does not depend on the actual content of them. Hence we can say it uses only static information, as is the case with static slicing.

Record-based slicing: Record-based slicing works with records instead of tables. It is also aware of the stored data as well as the structure of the database, so its results can be more precise. Here, the SPS is now a set of records. The results will contain those records (taken from separate tables) which are referred to directly or indirectly via foreign keys. With this type of slicing we can select a smaller piece of data from the database. The record-based slicing generates "dynamic" information (the actual content of the database) to be used, just as it does in dynamic program slicing, to produce a more precise result.

In analogy with program slicing, we can define several directions of slicing. These directions differ in the way we take into account the inter-table relations:

Forward slicing (FS): This slice only contains those tables or records which are connected directly or indirectly by foreign keys. The slice starts from the SPS. This direction tells us the name of each table or records that must be exported if we would like to have a coherent set of data related to the starting point.

Backward slicing (BS): Backward slicing can be used for an analysis to find the potential consequences of a change. A backward slice contains those tables or records which are influenced by a modification made at the starting point. This information can be used to check whether this modification harms the data consistency of the database or not. For example, if we change the date of a comment before the node submission date. Afterwards the data becomes inconsistent. The backward slicing can show us the list of tables or records that are influenced by this change at the comment, so we can modify them too if necessary. Another good example is when we have to change the primary key of a record and we need to know all the places in the database where the value of the old key is used as a foreign key.

Backward+Forward slicing (B+FS): A B+FS slice is the union of a forward and a backward slice started from the same SPS. This type of slice computes the scope of the SPS.

Backward and Backward+Forward slicing may contain dangling references. To produce an independent piece of the database for these types of slicing too, a forward slicing must be executed starting from every element in the result set.

Example 5. Table-based database slicing example.
Let us use the database example in Example 2. A database slice in the case of $SPS = node$ is:

- **FS**: $\{node, users\}$, because the *uid* column of *node* table is a foreign key for the *uid* column of the *users* table.

- **BS**: $\{node, comment\}$. *nid* of *comments* references the table *node*, so it has to be added to the result set.

- **B+FS**: $\{node, users, comments\}$. The unioin of the FS and BS slice.

Figure 3 shows the results of these slicings.

Example 6. Row-based database slicing example.
Let us apply the database example given in Example 2.
Row-based database slice in the case of $SPS = node$ table 1st row:

- **FS**: {1st row of *node*, 2nd row of *users*}

- **BS**: {1st row of *node*, 1st and 2nd rows of *comments*}

- **B+FS**: {1st row of *node*, 2nd row of *users*, 1st and 2nd rows of *comments*}

Figure 4 shows the results of these slicings.

4 Computing a database slice

We will now define a key component of database slicing, namely the Table Dependence Graph. The slicing algorithms will use this graph as input. The graph can represent the structure of a database, including the relations between its elements.

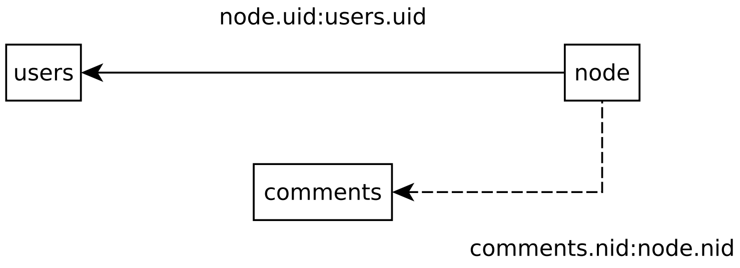
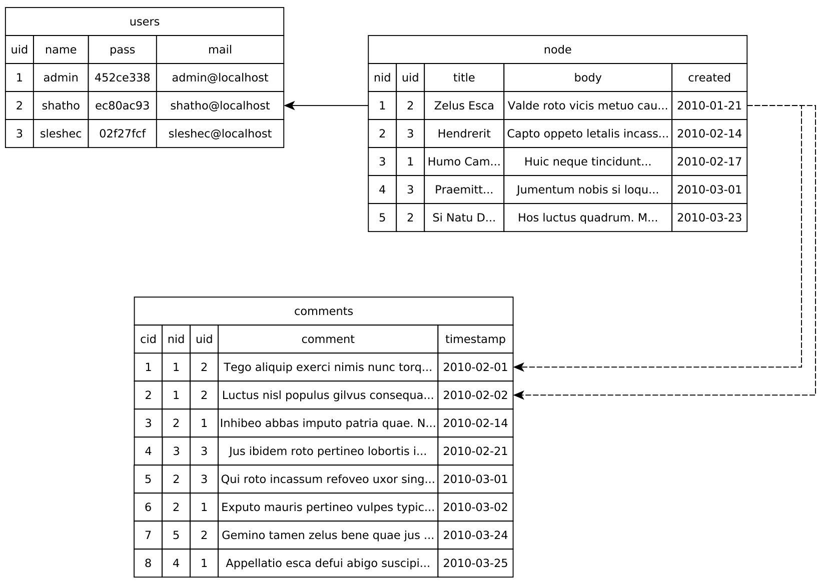


Figure 3: The FS and BS table-based slicing of the database given in Example 5. The result of FS is represented by the solid line and the one with a dashed line represents the result of the BS. The B+FS contains both results.



users

uid	name	pass	mail
1	admin	452ce338	admin@localhost
2	shatho	ec80ac93	shatho@localhost
3	sleshec	02f27fcf	sleshec@localhost

node

nid	uid	title	body	created
1	2	Zelus Esca	Valde roto vicis metuo cau...	2010-01-21
2	3	Hendrerit	Capto oppeto letalis incass...	2010-02-14
3	1	Humo Cam...	Huic neque tincidunt...	2010-02-17
4	3	Praemitt...	Jumentum nobis si loqu...	2010-03-01
5	2	Si Natu D...	Hos luctus quadrum. M...	2010-03-23

comments

cid	nid	uid	comment	timestamp
1	1	2	Tego aliquip exerci nimis nunc torq...	2010-02-01
2	1	2	Luctus nisl populus gilvus consequa...	2010-02-02
3	2	1	Inhibeo abbas imputo patria quae. N...	2010-02-14
4	3	3	Jus ibidem roto pertineo lobortis i...	2010-02-21
5	2	3	Qui roto incassum refoveo uxor sing...	2010-03-01
6	2	1	Exputo mauris pertineo vulpes typic...	2010-03-02
7	5	2	Gemino tamen zelus bene quae jus ...	2010-03-24
8	4	1	Appellatio esca defui abigo suscipi...	2010-03-25

Figure 4: The FS and BS record-based slicing of the database given in Example 6. The solid line represents the results of FS and the dashed lines represent the results of BS. B+FS is the union of these result sets.

4.1 Table-based slicing - Table Dependence Graph

The key part of a slice is the *table dependence graph* (TDG). As we mentioned earlier in the definition of table-based slicing, the results contain the data tables. Hence, the graph will also contain these tables.

Let T be the set of tables and let C_{t_i} be the set of the columns in table t_i for every $t_i \in T$.

TDG: $TDG = (V, E)$, where

- $V = T$, the set of vertices, the tables in the database.
- E, the set of edges:
  - $E \subseteq T \times T$. Let $t_1, t_2 \in T$. The (t_1, t_2) is an edge in E if there is a $c \in C_{t_1}$ which is a foreign key to one of the rows in t_2.
  - Let the label of this (t_1, t_2) edge be "$t_1.c_1 : t_2.c_2$", where $c_1 \in C_{t_1}$ and $c_2 \in C_{t_2}$, and c_1 is a foreign key to c_2. We call t_1 the *referrer table*, t_2 the *referred table*, c_1 the *referrer column* and c_2 the *referred column*.

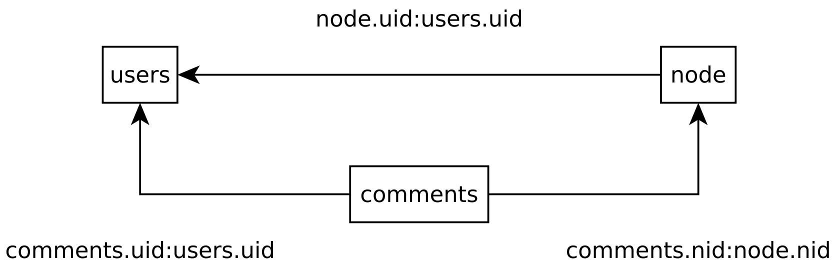


Figure 5: Instance of a table dependence graph based on the database described in Example 2. The vertices of the graph represent the tables.

4.2 Table-based slicing - computing slices

In essence, the slicing algorithm generates the solution of a reachability problem. The key part of the algorithm is the breadth-first search on the TDG. The algorithm requires some input parameters:

- TDG: a table dependence graph.
- SPS: a table of the database, represented by a node of the graph.
- D: the direction of slicing ('FS', 'BS' or 'B+FS').

The algorithm itself is quite simple. We just call $TSLICE\_DO$ with the corresponding parameters, where $TSLICE\_DO$ uses a modified version of the breadth-first search.

The methods $ENQUEUE$, $DEQUEUE$ were taken from [15] and they act on queue data structure.
Figure 6 shows how the table-based slicing works on the graph shown in Figure 5.

Algorithm 1 Method used for table-based slicing

```
// $TDG$: this table dependence graph will be sliced.
// $SPS$: starting point of slicing (set of tables)
// $D$: direction of slicing.
$TSLICE(TDG, SPS, D)$
1: if $D =$'$B + FS$' then
2:     $S1 \leftarrow TSLICE\_DO(TDG, SPS,$ '$F$')
3:     $S2 \leftarrow TSLICE\_DO(TDG, SPS,$ '$B$')
4:     $S \leftarrow S1 \cup S2$
5: else if $D =' FS'$ then
6:     $S \leftarrow TSLICE\_DO(TDG, SPS,$ '$F$')
7: else
8:     $S \leftarrow TSLICE\_DO(TDG, SPS,$ '$B$')
9: return $S$
```

4.3 Record-based slicing - computing slices

Record-based slicing takes a slice of the records of a database. SPS is a record of a table in this case. The result set then contains the corresponding rows. The algorithm uses the TDG of the database to discover the connections among the records. The result set contains row references:

Row reference: A (t, K) pair, where

t means the table node that contains the row, and

K means the list of primary key values of the row.

In Algorithm 4, GREYSET and BLACKSET can be stored either in the memory - as we did in $RSLICE\_DO$ - or they can be stored in a database as well to avoid an overly large memory consumption. In both cases it is sufficient to store just the data present in the row reference - not all the columns of the rows.

5 Experimental results

5.1 Research question to validate

In this section we would like to answer to the following **Research Questions**:

RQ1 How large is the size of the slice (with respect to the size of the database)?

RQ2 How much resources (memory and runtime) are used for computing the slices?

The efficiency of our algorithm is database structure- and data-dependent, so unfortunately we cannot provide a relevant and general answer to these questions. To estimate its efficiency, we looked at two different test scenarios:

Algorithm 2 Table-based slicing algorithm. Modified version of breadth-first search.

```
// TDG: table dependence graph.
// SPS: starting point of slicing (a table node).
// D: direction: 'F' in case of forward or 'B' in case of backward.
$TSLICE\_DO(TDG, SPS, D)$
1: for $\forall u \in V[TDG] - \{SPS\}$ node do
2:     $color[u] \leftarrow WHITE$
3: $color[SPS] \leftarrow GREY$
4: $BLACKSET \leftarrow \emptyset$, $GREYSET \leftarrow \emptyset$
5: ENQUEUE($GREYSET, SPS$)
6: while $GREYSET \neq \emptyset$ do
7:     $u \leftarrow$DEQUEUE($GREYSET$)
8:     if $D =$ 'F' then
9:         $E \leftarrow \{e \in E[TDG] | e$ is an outgoing edge of $u\}$
10:    else
11:        $E \leftarrow \{e \in E[TDG] | e$ is an incoming edge of $u\}$
12:    for $\forall e \in E$ do
13:        if $D =$ 'F' then
14:            $v \leftarrow e.referred\_table$
15:        else
16:            $v \leftarrow e.referrer\_table$
17:        if $color[v] = WHITE$ then
18:            $color[v] \leftarrow GREY$
19:            ENQUEUE($GREYSET, v$)
20:    $color[u] \leftarrow BLACK$
21:    $BLACKSET \leftarrow BLACKSET \cup u$
22: return $BLACKSET$
```

- For the basis of the first scenario, we chose one of the most popular PHP content management system, namely Drupal, and we generated random data with its *devel* module.

- For the basis of the second scenario, we used a real-life Drupal-based application, namely an information system that is employed in more than 40 hospitals in Hungary - and debugging this system was one of the strongest motivations behind our study.

5.2 Main findings of experiments

The implementation of slicing algorithms was written in Java, and tested on a computer with an Intel Core i5 CPU running at 2.8 GHz and 4GB of RAM.

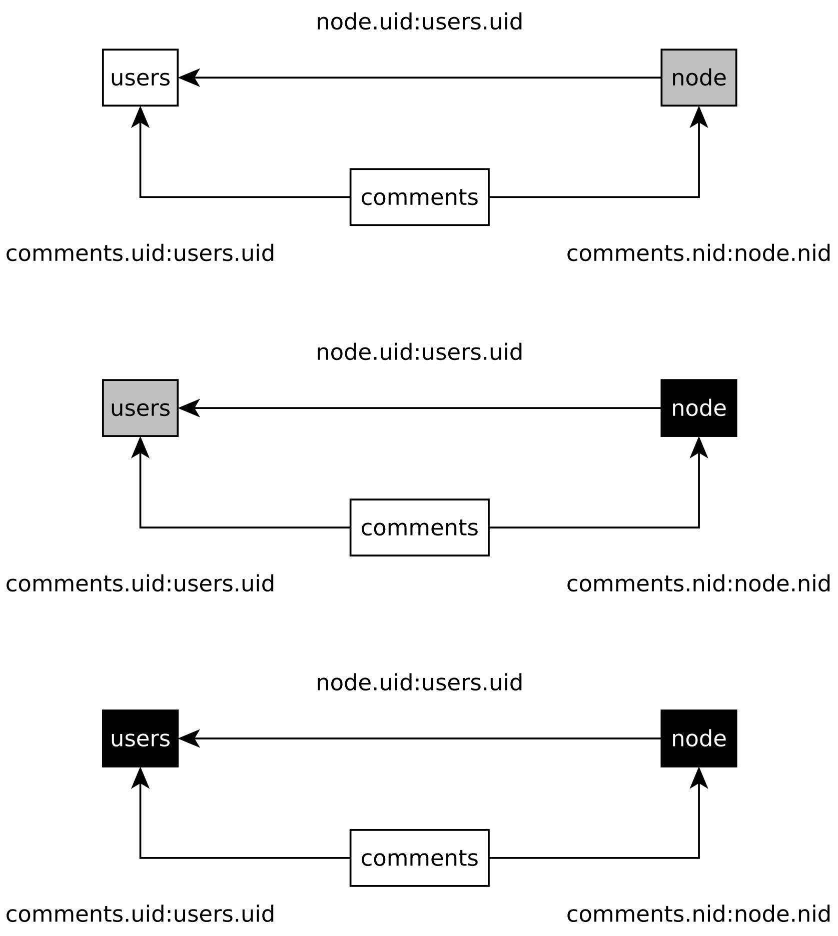


Figure 6: The table-based slicing of a database. The grey nodes have to be visited by the algorithm. The black nodes are those that have been completely processed by the algorithm and are in the result set.

5.2.1 Experiments on random generated data

In our first scenario, we randomly generated the users, the nodes and the comments with the help of *devel* module, which is a powerful helper module for Drupal developers. We used one of its functions (the content generation) to generate random content into the test database on Drupal version 7.15 (base installation with all modules enabled) under the Linux operating system using the Apache Web server and the PostgreSQL database system.

Our test database stored 100 random users and we increased the number of

Algorithm 3 Method used for record-based slicing.

```
// $TDG$: table dependence graph.
// $SPS$: starting point of slicing (a row reference).
// $D$: direction of slicing. (Can be 'FS', 'BS' or 'B+FS'.)
$RSLICE(TDG, SPS, D)$
1: if $D =$'$B + FS$' then
2:     $S1 \leftarrow RSLICE\_DO(TDG, SPS,$'$F$'$)$
3:     $S2 \leftarrow RSLICE\_DO(TDG, SPS,$'$B$'$)$
4:     $S \leftarrow S1 \cup S2$
5: else if $D =$'$F$' then
6:     $S \leftarrow RSLICE\_DO(TDG, SPS,$'$F$'$)$
7: else
8:     $S \leftarrow RSLICE\_DO(TDG, SPS,$'$B$'$)$
9: return $S$
```

nodes for each measurement. Each node had a maximum of 15 comments that were written by randomly chosen users. We selected the central table of Drupal, the *node* table, as our SPS and executed record-based slicings from this point. We measured the execution time and memory consumption of our algorithm.

Table 4 lists the size of each result set for each direction. The minimum number of nodes in the database was 100 and the maximum was 50,000. We were interested in how the size of the result set varied if the database contained more or fewer records. As can be seen in the table, the forward slice always selected 6 records from the database. The reason is that there are only 6 tables that are reachable from the node table by forward slicing; so the size of result sets will be the same, regardless of the number of records in the database. The size of BS and B+FS increased as the database size got bigger, but the size of the result set was about 1 % or less in every case. In the last scenario, when we generated 50,000 nodes via the devel module, the database contained 1,825,106 records, but the B+FS algorithm selected only 9,954 rows. This is only 0.55 % of the database. The result set size of B+FS slicing is always equal to $|FS| + |BS| - 1$, because the SPS is in both the FS and BS result sets, but it is only counted once for B+FS. The SQL dump of the full database was 2,284 MB. The dump of the result set was 13 MB. In the case of Drupal, as the database increased the number of selected rows did not increase much in terms of the number of records in the database.

Figure 7 shows the execution times and memory consumption for each type of slicing and for the two different types of storage used during the execution of the slicing algorithm. Upon examining the diagrams we see that if we use a database storage, the execution time of the algorithm grows with the size of the result set, but the memory consumption is nearly constant. When we used memory storage, the execution time did not change significantly, but the memory consumption increased dramatically as the result set contains more and more row references. The memory consumption was 2,235.43 KB in the case of B+FS slicing

Algorithm 4 Computes record-based slices.

```
// TDG: table dependence graph.
// SPS: starting point of slicing, a row reference.
// D: direction: 'F' in case of forward or 'B' in case of backward.
$RSLICE\_DO(TDG, SPS)$
 1: $BLACKSET \leftarrow \emptyset$, $GREYSET \leftarrow \emptyset$
 2: $ENQUEUE(GREYSET, SPS)$
 3: while $GREYSET \neq \emptyset$ do
 4:     $rf \leftarrow DEQUEUE(GREYSET)$
 5:     $t \leftarrow rf.t$ // table node belongs to row_ref $rf$
 6:     // load row from database belongs to $rf$
 7:     // value of the columns will be accessible by row[column name]
 8:     $row \leftarrow DB\_LOAD\_BY\_ROW\_REF(rf)$
 9:     if $D =$ 'F' then
10:         $E \leftarrow \{e \in E[TDG] | e$ is an outgoing edge of $t\}$
11:     else
12:         $E \leftarrow \{e \in E[TDG] | e$ is an incoming edge of $t\}$
13:     for $\forall e \in E$ do
14:         if $D =$ 'F' then
15:             $rows \leftarrow DB\_LOAD(e.referred\_table,$
                        $e.referred\_column,$
                        $row[e.referrer\_column])$
16:         else
17:             $rows \leftarrow DB\_LOAD(e.referrer\_table,$
                        $e.referrer\_column,$
                        $row[e.referred\_column])$
18:         while $\forall r \in rows$ do
19:             $rowref \leftarrow$ row reference of $r$
20:             if NOT $(IS\_IN(GREYSET \cup BLACKSET, rowref))$ then
21:                 $ENQUEUE(GREYSET, rowref)$
22:     // move rf from GREYSET to BLACKSET
23:     $MOVE(GREYSET, BLACKSET, rf)$
24: return $BLACKSET$
```

on the biggest database using memory storage. The algorithm using database storage required approximately 600 KB to produce the results in each case. As we can see, the memory consumption can be reduced by using a database as a storage, but in our measurements we only managed to save 1,600 KB of memory, which is not so significant. But the execution time of the algorithm that used memory storage, was much faster. It was only 0.91 second, while the one with database storage was 25.2 seconds. So to slice the Drupal database, the memory-based algorithm is more useful in practice.

	Results set size			
Generated content (number of nodes)	FS	BS	B+FS	Records in DB
100	6	55	60	5,846
250	6	113	118	9,260
500	6	140	145	20,031
1,000	6	265	270	36,497
2,500	6	696	701	53,180
5,000	6	1,433	1,438	150,198
10,000	6	2 536	2,541	351,118
50,000	6	9,949	9,954	1,825,106

Table 4: Size of the result set based on the generated content and the type of slicing. FS: Forward slicing, BS: Backward slicing, B+FS: Backward + Forward slicing.

5.2.2 Experiments on real-life application

In the second scenario, instead of using a randomly generated content, we found a real-life application and database to test our algorithm. This system evaluated the blood samples (mb_sample) of children (mb_child) with measurements (mb_result) got with different instruments (mb_machine). Each sample was sampled by a user (mb_employee and users) in an Institute (mb_institute). The system stores some extra information about foreign children (mb_child_foreign).

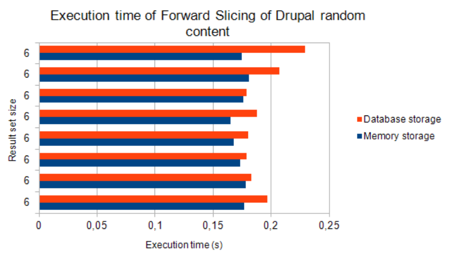
Figure 8 shows the TDG of the application and Table 5 shows the size of each tables. The total number of records stored in the database was 1,270,463.

Table	Number of records
mb_sample	216,870
mb_result	861,027
mb_child	191,988
mb_employee	260
mb_institute	47
mb_machine	10
users	261

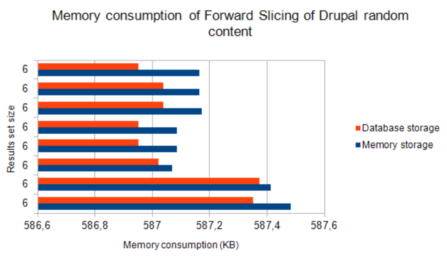
Table 5: Size of the tables in the real life application

As a starting point, we selected different records from different tables. Table 6 contains the results of set sizes of B+FS slicing starting from different tables.

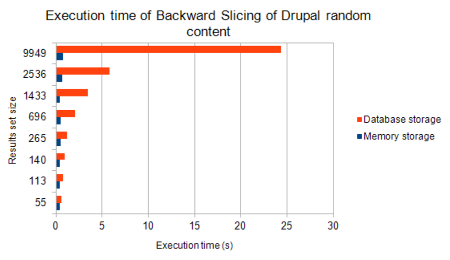
The results are shown in Figure 9. We got similar results (for the execution time and memory consumption) to those for randomly generated content. There is a curious thing in Figure 9a: the execution time of B+FS slicing - with memory storage - starting from the mb_machine table produces the largest result set, but it is much faster than those having a result set that is ten times smaller. The



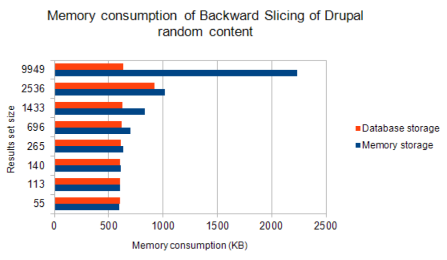
(a)



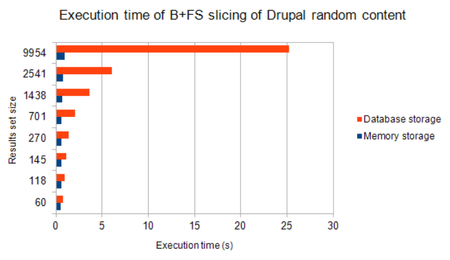
(b)



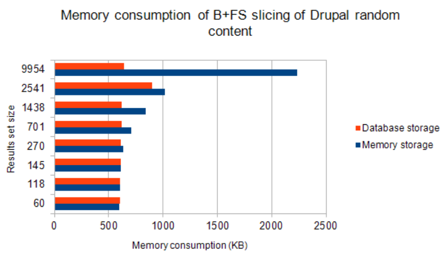
(c)



(d)



(e)



(f)

Figure 7: The execution times and memory consumption for the record-based slicing algorithm in the case of randomly generated contents.

reason for this seems to be that the algorithm has to visit only one table and also the starting table to collect results. This table is the mb_result and the algorithm cannot step backwards from this point. The maximum selection size was 16.35 % when the SPS was in the mb_machine table. Also, the number of records in the rest of the result sets was under 1 % compared to the database sizes.

In summary, the result sets of the database slicing contained less than 1% of the records of the entire database in most of the cases and the maximum selection size was 16.35%. The resource requirements of the algorithm was sufficient in each

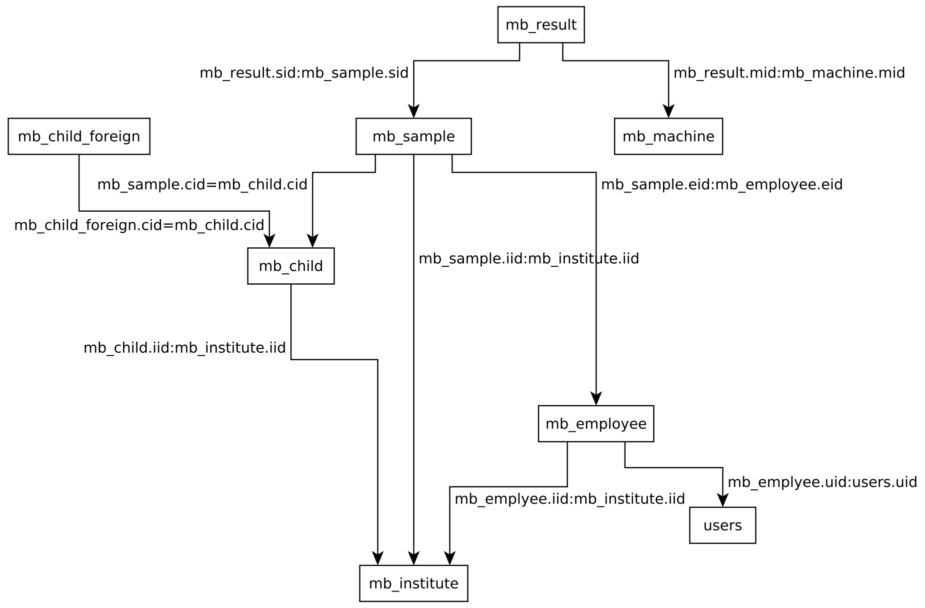

mb_result
mb_result.sid:mb_sample.sid
mb_result.mid:mb_machine.mid
mb_child_foreign
mb_sample
mb_machine
mb_sample.cid=mb_child.cid
mb_sample.eid:mb_employee.eid
mb_child_foreign.cid=mb_child.cid
mb_child
mb_sample.iid:mb_institute.iid
mb_child.iid:mb_institute.iid
mb_employee
mb_emplyee.uid:users.uid
mb_emplyee.iid:mb_institute.iid
users
mb_institute

Figure 8: Table Dependence Graph of our real-life application.

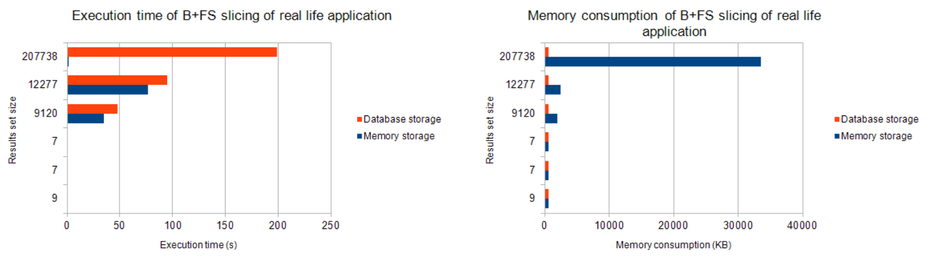
SPS	Results set size
mb_sample	9
mb_result	7
mb_child	7
mb_employee	9,120
mb_institute	12,277
mb_machine	207,738

Table 6: The size of the result set based on the different starting tables.

case.

5.3 Threats to validity

Unfortunately, it cannot be guaranteed, that these kinds of good results can be achieved in our experiments with every application, because the algorithm is database structure- and data-dependent. Hence precise efficiency can vary in practice. The random generation of content in our experiments may not represent the content distribution of a real Drupal-based website, but it involves the most used parts of the Drupal database structure, so it should provide a good picture of how the slicing works in this environment.



(a) (b)

Figure 9: The execution time and memory consumption for the B+FS slicing of our real-life application.

We investigated only Drupal-based applications in our experiments, but our slicing method can be used on every type of application that uses a relational database as its storage because the references among records are defined in the same way as in the case of Drupal.

6 Related work

In Microsoft's patent [8], there is a definition of *slicing of relational databases*, but here they use this concept only in a specific sense. The method they defined is equivalent to our forward record slicing algorithm, and they use it for test data generation. The result of their method is a database that stores a subset of real data taken from the database. They represent the database as a connected graph, based on the schema description of the database. This graph is then used to direct the slicing method and it helps to discover dependencies among records. Their graph is similar to our TDG. It represents the same information as the TDG in our definition. Based on this graph, their algorithm copies the corresponding record from the connected tables too. Their algorithm generates the same results for one record as our forward record-based slicing.

In [16], the authors process the SQL statements of a program to discover interesting inclusion dependencies in the database. Their method works on the execution plans of the SQL queries instead of the raw queries. The execution plan of a query is a description of the steps that the database engine has to perform to retrieve the selected data from the database. The authors define the *NavLog* algorithm which can be used to determine the links between attributes of different relational schemas. This technique can help one to discover the connections among database tables, which we treated as an input to our methods, just as [9].

In [9], the authors provide another approach to finding dependencies in a database. They compare the results of applying reverse engineering techniques to elicit the dependencies among records. The reverse engineering techniques use the source code as input and for handling database-driven applications. This process is called Database Reverse Engineering (DBRE). One of the main tasks of DBRE is to find the data dependencies among fields. The authors found that these dependencies can be discovered in the source code by looking for those places where the records get stored or modified. Program understanding techniques can be used to find these places in the source code.

Program slicing inspired new techniques in other areas, such as model slicing and model transformation slicing.

Model slicing tries to solve problems arising from the large size and complexity of models like UML models of an industrial application. In [11], the authors describe techniques for the slicing of UML models. The slices are performed using model transformations. The transformed models contain only those parts which specify the properties of a subset of the elements of the original model. The authors' work is based on the Object Constraint Language (OCL) of UML, which can be used to describe rules that apply to models. The authors describe how to slice a model based on the constraints applied to the classes and introduce state machine slicing.

In [18], the author defined a general model-based slicing framework that can be used to slice models that meet the necessary requirements. In his article, he demonstrates the framework by slicing a simple UML model. The author introduces a model definition to describe the meaning of any model and demonstrates slicing using this new model. He describes four types of slicing; namely, static slicing, dynamic slicing, conditioned slicing and slicing unions.

In [19], the authors introduce a dynamic backward slicing technique for model transformation programs and the underlying models of the transformation. Their technique relies upon an execution trace of the model transformation program and simultaneously produces slices for the statements of the transformation program and the model elements which affect the slicing criterion.

7 Summary and Future Work

Our research goal was to find out the applicability and usefulness of defining slicing on relational databases. In this paper we presented the following contributions. We introduced the concept of database slicing and its variants: table and record-based slicing, and forward, backward and forward+backward slicing. We provided table-based and record-based slicing algorithms and we evaluated the results using Drupal and a real-life Web-based information system. We learned that it can be useful in practice to help migration, debugging, testing and understanding the software by reducing the size of the database when the dependencies are known, but the result depends on the database structure and the data stored in the tables.

All of the techniques described here just concentrated on the database structures, connections and the corresponding operations and this introduced limitations. We reached a point where it was impossible to significantly improve these slicing techniques by just staying in the database.

In the future, we would like to combine database slicing with the results of program slicing and analysis, just like those in [9]. Another example is given in [1], where the classical PDG-based slicing technique was extended with the capability of handling database operations. However, it should be optimized and could be made more realistic if the dependencies in the database were properly analysed.

References

[1] David Willmor, Suzanne M. Embury, Jianhua Shao, *Program Slicing in the Presence of Database State*. In Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pages 448-452. IEEE Computer Society, 2004

[2] Devel module, URL: http://drupal.org/project/devel

[3] Drupal, URL: http://drupal.org/ 2002.

[4] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 13(6):377-387, June, 1970.

[5] E. F. Codd, *Extending the Database relational Model to Capture More Meaning*. ACM Transactioins on Database Systems, 4(4):397-434, December 1979

[6] Frank Tip, *A Survey of Program Slicing Techniques*. Journal of Programming Languages 3, pages 121-189, 1995.

[7] H. Agrawal and J. R. Horgan, *Dynamic Program Slicing*. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 25(6):246-256, 1990.

[8] Hui Shi, Kenton Gidewall, Marcelo M. De Barros, Chan Chaiyochlarb, Murali R. Krishnan, Robert Irwin Voightmann, Christina Ruth Dhanaraj, *Slicing of relational databases*. Patent. US 7873598. April 2008. URL: http://www.google.com/patents/US7873598

[9] Jean Henrard, Jean-Luc Hainaut, *Database dependency elicitation in database reverse engineering*. In CSMR '01: Proceedings of the Conference on Software Maintenance and Reengineering, pages 11-19. IEEE Computer Society, 2002.

[10] K.J. Ottenstein and L.M. Ottenstein, *The program dependence graph in a software development environment*. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 177-184, 1984.

[11] K. Lano, S. Kolahdouz-Rahimi, *Slicing of UML Models*

[12] M. Weiser, *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigen, Ann Arbor, 1979.

[13] M. Weiser, *Programmers use slices when debugging.* Communications of the ACM, 25(7):446-452, 1982.

[14] M. Weiser, *Program slicing.* IEEE Transactions on Software Engineering, 10(4):352-357, 1984.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill, 2001.

[16] Stéphane Lopes, Jean-Marc Petit, Farouk Toumani, *Discovering interesting inclusion dependencies: application to logical database tuning.* Information Systems, Volume 27, Issue 1, March 2002, Pages 1-19, ISSN 0306-4379, 10.1016/S0306-4379(01)00027-8.

[17] Tibor Gyimóthy, Árpád Beszédes, and István Forgács, *An efficient relevant slicing method for debugging.* In Lecture Notes in Computer Science, 1687:303–321. Springer, 1999.

[18] Tony Clark, *A general model-based slicing framework.* Proceedings of the Workshop on Composition and Evolution of Model Transformations, 2011

[19] Z. Ujhely, Á. Horváth, D. Varró, *Dynamic Backward Slicing of Model Transformations.* International Conference on Software Testing and Validation:1-10 2012.

Received 10th April 2012