Cartesian Decomposition in Data Analysis

Pavel Emelyanov and Denis Ponomaryov

***Abstract*—We consider the Cartesian decomposition of relational data sets, i.e. the problem of finding two or several data sets such that their unordered Cartesian product equals the source set. In terms of relational databases, this means reversing the SQL CROSS JOIN operator. We describe a polytime algorithm for computing a Cartesian decomposition based on factorization of boolean polynomials. We provide an implementation of the algorithm in Transact SQL and discuss some generalizations of the Cartesian decomposition.**

***Index Terms*—Databases, Data Analysis, Partitioning Algorithms**

1. Introduction

The analysis of big data sets of different origin is an important problem of modern theoretical and applied informatics. Detecting the Cartesian property of a data set, i.e. whether it can be given as an unordered Cartesian product of two (or several) data sets appears to be important in at least three out of the six classes of data analysis problems (Data Mining), as defined by the classics in the domain [1], namely anomaly detection, dependency modeling and constructing a more compact data representation. In this paper, we consider the problem of Cartesian decomposition for the relational data model.

Only for the first twenty-five years after Codd had developed his relational data model, more than 100 types of dependencies were described in the literature [2]. Cartesian decomposition underlies the definitions of the major dependency types encompassed by the theory of relational databases. This is because numerous concepts of dependency are based on the *join* operation, which is inverse to Cartesian decomposition. Recall that the *join dependency* is the most common kind of dependencies considered in the framework of the fifth normal form [3]. A relation R satisfies the join dependency $\bowtie (A_1, \ldots, A_n)$ for a family of subsets of its attributes $\{A_1, \ldots, A_n\}$ if R is the union of the projections on the subsets A_i, $1 \leqslant i \leqslant n$. Thus, if A_i are disjoint, we have the Cartesian decomposition of the relation R into the corresponding components–projections.

For the case $n = 2$ the join dependency is known in the context of the fourth normal form under the name *multivalued dependency*. A relation R for a family of subsets of its attributes $\{A_0, A_1, A_2\}$ satisfies the multivalued dependency $A_0 \mapsto A_1$ iff R satisfies the join dependency $\bowtie (A_0 \cup A_1,\ A_0 \cup A_2)$. Thus for each A_0-tuple of values, the projection of R onto $A_1 \cup A_2$ has a Cartesian decomposition. Historically, multivalued dependencies were introduced earlier than join dependencies [4] and attracted wide attention in the literature as they are natural variant thereof.

An important task is the development of efficient algorithms for solving the computationally challenging problem of finding dependencies in data. A lot of research has been devoted to mining functional dependencies (see surveys [5], [6]), while the detection of more general dependencies, like the multivalued ones, has been less studied. In [7], the authors propose a method based on directed enumeration of assumptions/conclusions of multivalued dependencies (exploring the properties of these dependencies to narrow the search space) with checking satisfaction of the generated dependencies on the relation of interest. In [8], the authors employ an enumeration procedure based on the refinement of assumptions/conclusions of the dependencies considered as hypotheses. Notice that when searching for functional dependencies $A \mapsto B$ on a relation R, once an assumption A is guessed, the conclusion B can be efficiently found. For multivalued dependencies, this property is not trivial and leads to the question of efficient recognition of Cartesian decomposition (of the projection of R on the attributes not contained in A). Thus, the algorithmic results presented in this paper can be viewed as a foundation for the development of new methods for detecting the general kind dependencies, in particular, multivalued and join dependencies.

Let us consider the Cartesian product of two relations given in the form of tables:

A	B
x	y
x	z

$\times$

C	D	E
x	u	p
y	u	q
z	v	r

$=$

A	B	C	D	E
x	y	x	u	p
x	y	y	u	q
x	y	z	v	r
x	z	x	u	p
x	z	y	u	q
x	z	z	v	r

B	E	D	A	C
z	q	u	x	y
y	q	u	x	y
y	r	v	x	z
z	r	v	x	z
y	p	u	x	x
z	p	u	x	x

In the first representation of the product result, where the natural order of rows and columns is preserved, a careful

- *A.P. Ershov Institute of Informatics Systems, Novosibirsk State University*
- *E-mail: emelyanov@iis.nsk.su, ponom@iis.nsk.su*

reader can easily recognize the cartesian structure of the table. However, this is not so easy to do for the second representation, where the rows and columns are randomly shuffled, even though the table is small. In the sequel, we only consider the relations having no surrogate key of any kind and assume that the tuples found in the relations are all different.

A highly important topic in database research is data security. For a long time, it was handled through data encryption. The disadvantage of this approach, however, was degradation of query performance. In [9], the authors proposed to use a distributed table storage for relational databases (see also [10]). It is quite evident that in the most general case, taking into account integrity constraints and query efficiency, the problem is computationally hard (see, for example, [11]), therefore, to solve it heuristic algorithms were proposed (see ibid.). It should be noted that in the mid–1980s a similar topic was developed, namely, vertical database partitioning, where the objective was the optimization of operation performance and access parallelization [12], [13].

In [14], [15], the authors described an algorithm for factorization of polylinear polynomials over the finite field of the order 2 (Boolean polynomials) with the time complexity $O(l^3)$, where l is the length of the polynomial given as a string. It is based on checking whether the partial derivatives of some auxiliary polynomial constructed from the original one are equal to zero. Using this algorithm the authors showed that the problem of conjunctive decomposition of boolean functions given in full DNF and positive DNF is solvable in polynomial time. The algorithm was obtained independently of the result of Shpilka and Volkovich presented in [16], which gives a different algorithm of the same complexity. Also, in [17] the author proposed an algorithm with the complexity $O(n^5N)$ for the decomposition of monotone boolean functions given in DNF, where n is the number of variables and N is the number of conjuncts.

The relationship between the problems of cartesian decomposition and factorization of boolean polynomials is easily established. Each tuple of the relation is a monomial of a polynomial, where the attribute values play the role of variables. Importantly, the attributes of the same type are considered as different. Thus, if in a tuple different attributes of the same type have equal values, the corresponding variables are different. `NULL` is also typed and appears as a different variable. For example, for the relation above the corresponding polynomial is

$$\begin{gathered} z_B \cdot q \cdot u \cdot x_A \cdot y_C + y_B \cdot q \cdot u \cdot x_A \cdot y_C + \\ y_B \cdot r \cdot v \cdot x_A \cdot z_C + z_B \cdot r \cdot v \cdot x_A \cdot z_C + \\ y_B \cdot p \cdot u \cdot x_A \cdot x_C + z_B \cdot p \cdot u \cdot x_A \cdot x_C = \\ x_A \cdot (y_B + z_B) \cdot (q \cdot u \cdot y_C + r \cdot v \cdot z_C + p \cdot u \cdot x_C) \end{gathered}$$

In the following, we use this correspondence between relational tables and polynomials.

2. Algorithm for Factorization of Boolean Polynomials

Let us briefly mention the factorization algorithm given in [14], [15]. It is assumed that the input polynomial F has no trivial divisors and contains at least two variables.

1) Take an arbitrary variable x from F.
2) Let $\Sigma_{same} := \{x\}, \Sigma_{other} := \varnothing$, and $F_{same} := 0, F_{other} := 0$.
3) Compute $G := F_{x=0} \cdot F'_x$.
4) For each variable $y \in Var(F) \setminus \{x\}$:
   if $G'_y = 0$ then $\Sigma_{other} := \Sigma_{other} \cup \{y\}$
   else $\Sigma_{same} := \Sigma_{same} \cup \{y\}$.
5) If $\Sigma_{other} = \varnothing$, then output $F_{same} := F, F_{other} := 1$ and stop.
6) Restrict each monomial of F onto Σ_{same} and add every obtained monomial to F_{same} if F_{same} does not contain it.
7) Restrict each monomial of F onto Σ_{other} and add every obtained monomial to F_{other} if F_{other} does not contain it.
8) Check which of the products $(F_{same}+c_1)(F_{other}+c_2), c_1, c_2 = 0, 1$, gives the original polynomial F and output these components.

Notice that for the decomposition of relations of relational databases it suffices to consider the case $c_1 = c_2 = 0$.

3. Implementation of the Decomposition Algorithm in SQL

The decomposition algorithm for relational tables implements the steps of the factorization algorithm described above.

In terms of polynomials, it is easy to formulate and prove the following property: if two variables always appear in different monomials (i.e., there is no monomial, in which they appear simultaneously) then these variables appear in different monomials of the same decomposition component if a decomposition exists. A direct consequence of this observation is that for each relation attribute it is enough to consider just one value of this attribute because the others must belong to the same decomposition component (in case it exists).

In the following, we assume that the input table name is stored in the variable `@TableName`. All auxiliary variables are assumed to be declared.

3.1. Trivial Attribute Elimination

If some attribute of a relation has only one value then we have a case of trivial decomposition. In terms of polynomials, this condition can be written as $F = x \cdot F'_x$. This attribute can be extracted into a separate table. Further we assume that there are no such trivial attributes. Extraction of trivial attributes can be easily done within SQL, so we omit details of its implementation.

3.2. Preliminary Manipulations

At the first step, we need to select a variable x, with respect to which decomposition will be constructed. We need to find two sets of attributes forming the tables as decomposition components. As mentioned above, we can take an arbitrary value of an arbitrary attribute of the table. In the SQL-script given below, the first value of the first table attribute is selected. They are stored in `@FirstName` and `@FirstValue`. Note that the optimal choice of this variable is an interesting problem, which has not been solved satisfactorily yet.

```
SET @Query =
    N'SET @FirstName = '
    +'(SELECT TOP(1) sysTbl.COLUMN_NAME'
    +'   FROM INFORMATION_SCHEMA.COLUMNS sysTbl'
    +'  WHERE sysTbl.TABLE_NAME='''+@TableName+''')'
EXEC sp_executesql @Query,
                   N'@FirstName NVARCHAR(max) out',
                   @FirstName = @FirstName out

SET @Query =
    N'SET @FirstValue = ' +
    +'(SELECT TOP(1) ' + @FirstName
    +'   FROM ' + @TableName + ')'
EXEC sp_executesql @Query,
                   N'@FirstValue NVARCHAR(max) out',
                   @FirstValue = @FirstValue out
```

With the help of the next query, the string `@Columns` is created. It represents table attributes and their aliases corresponding to the product $F_{x=0} \cdot F'_x$ (in the terms of polynomials). The prefixes F and S correspond to $F_{x=0}$ and F'_x. The product constructed below contains information about the distribution of variables between components and is essential for computing decomposition. We note that some string manipulations used in the SQL-code can be DBMS–dependent.

```
SET @Query =
    N'SET @Columns = '
    +'(SELECT '',F.'' + sysTbl.COLUMN_NAME
    +      '' AS F_'' + sysTbl.COLUMN_NAME + '
    +        ''',S.'' + sysTbl.COLUMN_NAME
    +      '' AS S_'' + sysTbl.COLUMN_NAME '
    +'   FROM INFORMATION_SCHEMA.COLUMNS AS sysTbl'
    +'  WHERE sysTbl.TABLE_NAME  = '''
                                  + @TableName +''''
    +'    AND sysTbl.COLUMN_NAME!= '''
                                  + @FirstName +''''
    +' FOR XML PATH(''''))'
EXEC sp_executesql @Query,
                   N'@Columns NVARCHAR(max) out',
                   @Columns = @Columns out
SET @Columns = SUBSTRING(@Columns,2,LEN(@Columns))
```

With the help of the following query, the string @Duplicates is created, which represents a condition (a logical expression) allowing us to reduce the size of the table-product through the exclusion of duplicate rows. In the terms of polynomials, these are the monomials of the polynomial-product with the coefficient 2, which can obviously be omitted in the field of the order 2. Since this table is used for bulk queries, its size significantly impacts performance. An example is given in **Section 3.3**.

```
SET @Query =
    N'SET @Duplicates = '
    +'(SELECT ''AND((T.F_'' + sysTbl.COLUMN_NAME
    +             ''=R.F_'' + sysTbl.COLUMN_NAME
    +  '' AND '' +   ''T.S_'' + sysTbl.COLUMN_NAME
    +             ''=R.S_'' + sysTbl.COLUMN_NAME
    +  '') OR '' +  ''(T.F_'' + sysTbl.COLUMN_NAME
    +             ''=R.S_'' + sysTbl.COLUMN_NAME
    +  '' AND '' +   ''T.S_'' + sysTbl.COLUMN_NAME
    +             ''=R.F_'' + sysTbl.COLUMN_NAME
    +            ''))'''
    +'   FROM INFORMATION_SCHEMA.COLUMNS AS sysTbl'
    +'  WHERE sysTbl.TABLE_NAME  = '''
                                  + @TableName +''''
    +'    AND sysTbl.COLUMN_NAME!= '''
                                  + @FirstName +''''
    +' FOR XML PATH(''''))'
EXEC sp_executesql @Query,
                   N'@Duplicates NVARCHAR(max) out',
                   @Duplicates = @Duplicates out
SET @Duplicates = SUBSTRING(@Duplicates, 4,
                            LEN(@Duplicates))
```

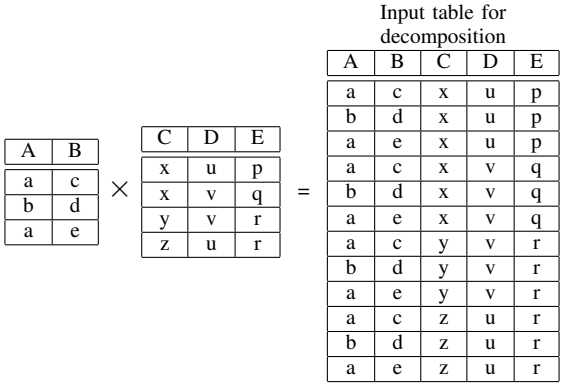
3.3. Retrieval of the 'Sorting Product'

The table–product, which allows for sorting attributes with respect to the component selected, is created in the form `VIEW`. It is worth noting that it can be constructed in different ways. In our case, a conceptually very simple solution has a disadvantage: its Transact SQL implementation works very slowly. The reason is that Transact SQL does not allow us to create a materialized `VIEW` (Oracle implements `CREATE MATERIALIZED VIEW`). We have decided to keep this version due to its simplicity.

```
SET @Query =
    N'CREATE VIEW SortingProduct AS '
    +'WITH Prod AS ('
    +' SELECT ' + @Columns
    +' FROM (SELECT * '
          +'   FROM ' + @TableName
          +'  WHERE ' + @FirstName + ' = '''
                      + @FirstValue + ''') AS F,'
          +'(SELECT * '
          +'   FROM ' + @TableName
          +'  WHERE ' + @FirstName + '!= '''
                      + @FirstValue + ''') AS S)'
    +'SELECT DISTINCT T.* '
    +'  FROM Prod AS T '
    +' WHERE 1 = (SELECT COUNT(1) '
               +'  FROM Prod AS R '
               +' WHERE ' + @Duplicates + ')'
EXEC sp_executesql @Query
```

Below we provide an example of computing a sorting product.

It is easy to see that the table corresponding to the full product is bigger than the original table. In this example, it would contain 32 rows. However, its size can be reduced substantially by applying the filter `@Duplicates`. The `VIEW SortingProduct` contains only 8 rows.



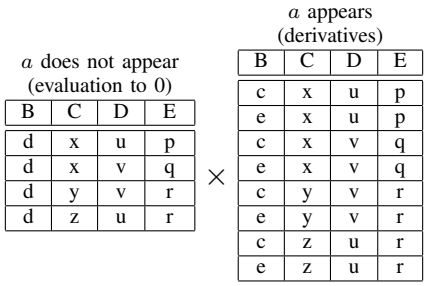


Table of sorting product

F_B	S_B	F_C	S_C	F_D	S_D	F_E	S_E
c	d	x	x	u	u	p	p
c	d	x	x	v	v	q	q
c	d	y	y	v	v	r	r
c	d	z	z	u	u	r	r
e	d	x	x	u	u	p	p
e	d	x	x	v	v	q	q
e	d	y	y	v	v	r	r
e	d	z	z	u	u	r	r

In [15] it was shown that the algorithm can be implemented without an explicit computation of the product $F_{x=0} \cdot F'_x$. Note that computing this product can be expensive for large inputs. For the case of boolean polynomials, the computation is replaced by the bulk evaluation of polynomials having smaller sizes. An interesting question whether one can avoid an explicit computation of the product for the case of relational tables.

3.4. Attribute Partitioning

Recall that in the terms of polynomials the membership of a variable y to a component containing the variable x that was selected at the first step is decided by checking whether the partial derivative of the polynomial $\frac{\partial}{\partial y}(F_{x=0} \cdot F'_x)$ is equal to zero (in the finite field of order 2). It is easy to see that this corresponds to checking whether a variable appears in the monomials in the first or second degree (or is absent at all). Thus, the corresponding SQL–script looks quite simple and the most part of it is taken by the procedure of selecting the attribute values:

```
SET @Query =
    N'SET @ColumnCursor = '
    +'CURSOR FORWARD_ONLY STATIC FOR'
    +'SELECT sysTbl.COLUMN_NAME '
    +'  FROM INFORMATION_SCHEMA.COLUMNS sysTbl'
    +' WHERE (sysTbl.TABLE_NAME  = '''
                                    + @TableName +''')'
    +'   AND (sysTbl.COLUMN_NAME!= '''
                                    + @FirstName +''')'
    +'OPEN @ColumnCursor;'
EXEC sys.sp_executesql
     @Query, N'@ColumnCursor CURSOR OUTPUT',
              @ColumnCursor=@ColumnCursor OUTPUT
FETCH NEXT FROM @ColumnCursor INTO @ColumnName;

WHILE (@@FETCH_STATUS = 0)
BEGIN
   SET @Query =
       N'SELECT 1 '
       +'  FROM SortingProduct,'
             +'(SELECT TOP(1) F_' + @ColumnName
             +'         AS VR'
             +'   FROM SortingProduct) SP'
       +' WHERE ((F_' + @ColumnName + '  = SP.VR) '
       +'   AND  (S_' + @ColumnName + ' != SP.VR))'
       +'    OR ((F_' + @ColumnName + ' != SP.VR) '
       +'   AND  (S_' + @ColumnName + '  = SP.VR))'
   EXEC sys.sp_executesql @Query
   IF (@@ROWCOUNT != 0)
      PRINT @ColumnName
            + ' belongs to the 1st component'  --(1)
   ELSE
      PRINT @ColumnName
            + ' belongs to the 2nd component'; --(2)
   FETCH NEXT FROM @ColumnCursor INTO @ColumnName;
END

CLOSE @ColumnCursor;
DEALLOCATE @ColumnCursor;
```

Note that for simplicity this script just reports the attribute membership (the strings with comments `(1)` and `(2)`). In fact, we need to construct strings corresponding to the lists of the component attributes. Let these strings be `@FirstCompAttrs` and `@SecondCompAttrs`, respectively. If the cycle is completed and it is the case that the string `@SecondCompAttrs` is empty, then the table is not decomposable.

Otherwise, the following simple script completes table decomposition. The resulting tables–components are produced by restricting the source table onto the corresponding component attributes and selecting unique tuples:

```
SET @Query =
    N'CREATE TABLE FirstComponent'
    +'    AS (SELECT DISTINCT ' + @FirstCompAttrs
    +'        '  FROM ' + @TableName + '); '
    +'CREATE TABLE SecondComponent'
    +'    AS (SELECT DISTINCT ' + @SecondCompAttrs
    +'        '  FROM ' + @TableName + '); '
EXEC sp_executesql @Query
```

4. Conclusions

Observe the following property, which says that with high probability a given relation does not have a decomposition into a Cartesian product.

Remark 1. *Let R be a random relation of degree n and cardinality N, which does not contain trivial attributes.*

Then it holds

$$\mathbb{P}(R \textit{ is non–decomposable}) > 1 - \left(1 - \frac{\phi(N)}{N}\right)^n > \\ > 1 - \left(1 - \frac{1}{e^\gamma \ln\ln N + \frac{3}{\ln\ln N}}\right)^n,$$

where ϕ and γ are the Euler function and constant, respectively.

Despite this fact, the algorithm described in this paper is important for the following reasons. First, a method of computing Cartesian decomposition can provide a basis for new approaches to mining general kinds of dependencies, in particular, multivalued ones, since the join dependency underlies their definition. Second, the problem of Cartesian decomposition allows for the following two generalizations and a combination thereof:

– The pure Cartesian product may be "spoiled" by a few additional tuples:

$$P(X,Y) = F(X) \times G(Y) + H(X,Y),$$

where the size of the relation H is significantly smaller, than the size of P. In this case, extracting the "defect" $H(X,Y)$ into a separate table allows for decomposing the principal part of the table.

– Partition of the attributes into disjoint subsets is a rather strong requirement. A relation may prove to be decomposable if one admits the existence of shared attributes in the components (in this case, the notion of decomposition needs to be refined). Hence, we have a decomposition problem either with prescribed shared attributes or with a set of shared attributes that satisfies certain optimality criteria, such as the minimization of the shared attribute set or/and the sizes of components.

Both cases are of interest in the context of distributed data storage, because decomposition of the above mentioned forms has a good structure and can increase the performance of systems. The solution to the Cartesian decomposition problem considered in this paper is a special case and a crucial component in solving the more general problem of decomposition with shared attributes. Therefore, the efficiency of this solution is quite important.

Acknowledgments

The authors sincerely thank Mikhail Bulyonkov for useful discussions and comments on this paper.

References

[1] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *AI Magazine*, vol. 17, no. 3, pp. 37–54, 1996.

[2] B. Thalheim, "An overview on semantical constraints for database models," in *Proceedings of the 6th International Conference on Intellectual Systems and Computer Science*, 1996, pp. 81–102.

[3] C. Date, *Introduction to Database Systems*, 8th ed. Addison–Wesley Longman, Inc., 2004.

[4] R. Fagin and M. Vardi, "The theory of data dependencies: a survey," in *Mathematics of Information Processing: Proceedings of Symposia in Applied Mathematics*. Providence, Rhode Island: AMS, 1986, vol. 34, pp. 19–71.

[5] J. Liu, J. Li, C. Liu, and Y. Chen, "Discover dependencies from data – a review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 2, pp. 251–264, 2012.

[6] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schoenberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: an experimental evaluation of seven algorithms," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 1082–1093, 2015.

[7] M. Yan and A. W.-c. Fu, "Algorithm for discovering multivalued dependencies," in *Proceedings of the 10^{th} International Conference on Information and Knowledge Management (CIKM'01)*. New York, NY, USA: ACM, 2001, pp. 556–558.

[8] I. Savnik and P. Flach, "Discovery of multivalued dependencies from relations," *Intelligent Data Analysis*, vol. 4, no. 3–4, pp. 195–211, 2000.

[9] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu, "Two can keep a secret: A distributed architecture for secure database services," in *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. VLDB Endowment, 2005, pp. 186–199.

[10] M. Gorawski and S. Panfil, "A system of privacy preserving distributed spatial data warehouse using relation decomposition," in *Proceedings on the 4th International Conference on Availability, Reliability and Security (ARES 2009)*. IEEE, 2009, pp. 522–527.

[11] V. Ganapathy, D. Thomas, T. Feder, H. Garcia-Molina, and R. Motwani, "Distributing data for secure database services," *Transactions on Data Privacy*, vol. 5, pp. 253–272, 2012.

[12] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical partitioning algorithms for database design," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 680–710, 1984.

[13] C.-H. Cheng and J. Motwani, "An examination of cluster identification–based algorithms for vertical partitions," *International Journal of Business Information Systems*, vol. 4, no. 6, pp. 622–638, 2009.

[14] P. Emelyanov and D. Ponomaryov, "On tractability of disjoint AND–decomposition of boolean formulas," in *Proceedings of the PSI 2014: 9^{th} Ershov Informatics Conference*, ser. Lecture Notes in Computer Science, vol. 8974. Springer, 2015, pp. 92–101.

[15] ——, "Algorithmic issues of conjunctive decomposition of boolean formulas," *Programming and Computer Software*, vol. 41, no. 3, pp. 162–169, 2015, translated: Programmirovanie, Vol. 41, No. 3, pp. 62-72, 2015.

[16] A. Shpilka and I. Volkovich, "On the relation between polynomial identity testing and finding variable disjoint factors," in *Proceedings of the 37^{th} International Colloquium on Automata, Languages and Programming. Part 1 (ICALP 2010)*, ser. Lecture Notes in Computer Science, vol. 6198. Springer, 2010, pp. 408–419.

[17] J. C. Bioch, "The complexity of modular decomposition of Boolean functions," *Discrete Applied Mathematics*, vol. 149, no. 1-3, pp. 1–13, 2005.

Appendix

Proof of Remark 1

Let c_v be the number of occurrences of an attribute v in the relation R (the set of all v's is the set of different values in R). Since R does not contain trivial attributes, a necessary condition for decomposability is

$$\forall v \ : \ (c_v, N) > 1.$$

Given a random relation R, the attribute values are independent random variables and thus, so are c_v's (their quantities). The probability that for at least one of these n random variables it holds $(c_v, N) = 1$ (i.e. the complement of the event of interest) is equal to

$$1 - \mathbb{P}\left(\bigwedge_v (c_v, N) \neq 1\right) = 1 - \mathbb{P}\left((c_v, N) \neq 1\right)^n = 1 - \left(1 - \mathbb{P}\left((c_v, N) = 1\right)\right)^n.$$

For $1 \leq c_v \leq N$, it holds $\mathbb{P}\left((c_v, N) = 1\right) = \frac{\phi(N)}{N}$. This is a lower bound, since we used the necessary condition.