Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003".
S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Alexander Hinneburg, Wolfgang Lehner

Database Support for 3D-Protein Data Set Analysis

Erstveröffentlichung in / First published in:
15th International Conference on Scientific and Statistical Database Management.
Cambridge, MA, 09.-11.07.2003. IEEE, S. 161-171. ISBN 0-7695-1964-4
DOI: https://doi.org/10.1109/SSDM.2003.1214977

Diese Version ist verfügbar / This version is available on:
https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-787878

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003".
S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

Database Support for 3D-Protein Data Set Analysis

Alexander Hinneburg
Institute of Computer Science
University of Halle, Germany
hinneburg@informatik.uni-halle.de

Wolfgang Lehner
Database Technology Group
Dresden University of Technology, Germany
lehner@inf.tu-dresden.de

Abstract

The progress in genome research demands for an adequate infrastructure to analyse the data sets. Database systems reflect a key technology to organize data and speed up the analysis process.

This paper discusses the role of a relational database system based on the problem of finding frequent substructures in multi-dimensional protein databases. The specific problem consists of producing a set of association rules regarding frequent substructures with different lengths and gaps between the amino acid residues of a protein. From a database point of view, the process of finding association rules building the base for a more in-depth analysis of the data material is split into two parts. The first part performs a discretization of the conformational angle space of a single amino acid residue by computing the nearest neighbour of a given set of representatives. The second part consists in adapting a well-known association rule algorithm to determine the frequent substructures. Both steps within this comprehensive analysis task requires substantial support of the underlying database in order to reduce the programming overhead at the application level.

1 Introduction

The progress in genome research generates a tremendous need for an adequate infrastructure to store and efficiently analyse the underlying data sets. Although most of the current data sets are stored in flat files, database technology may help to organize and speed up the overall analysis process by exploiting existing database functionality. This paper focusses on the role of a relational database system referring to a project finding frequent substructures in protein data sets. The motivation of the project stems from the observation that the number of known protein sequences grows very fast. However, the known three-dimensional structures of proteins are lower in the order of magnitudes than the number of known protein sequences because the underlying X-ray protein analysis is time consuming and, ultimately, very expensive.

Therefore, there is a strong desire to derive the 3D-structure directly from the protein sequence. The standard technique, protein homology modeling, is limited to proteins, for which proteins with known 3D-structures and similar sequences exist. Unfortunately, for most proteins with known sequences and unknown 3D-structures, homology modeling is not possible, so that the alternative way to learn about the behaviour and/or functionality of the proteins from the structure is based on the analysis of substructures. The analysis process of our sample project application consists of the two separate steps of discretization of the multi-dimensional dihedral angle space and the generation of frequent item sets. Both steps need to be supported by the underlying database system in order to achieve an efficient analysis process and increase the acceptance of database systems in the biotechnology research community.

Contribution of the Paper The paper is logically divided into two parts. One thread of the paper addresses the modeling perspective of the specific application and outlines the mapping of the protein structures based on a data model consisting of a sequence of vectors of dihedral angles to a relational scenario. The second thread discusses the operational issue and focusses on the database optimization perspective. The bottom line of the paper emphasizes two major issues: on the one hand, we strongly believe that the current relational database technology does not yet reflect a perfect data storage and data analysis platform. On the other hand, we propose extensions to a relational database engine which might be exploited by a huge variety of data mining (or knowledge discovery in general) applications.

Structure of the Paper The following section outlines current techniques to analyse 3D protein data from the application point of view. Additionally, our proposed representation of 3D protein structure information within the dihedral angle space model is presented. Section 3 focusses on the first step of the data analysis process to find similar

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003". S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

substructures: picking the nearest representative for each single amino acid residue. Solutions using currently existing features of the relational database technique show that there is a demand for a more specific database support. The comparison of multiple methods illustrates that only minimal extensions may result in a huge benefit. In the same vein, we show that the generation of frequent item sets (as the second step in performing the analysis process) is not efficiently implementable relying only on existing technology.

2 Methods and Models for Protein Analysis

This section outlines current methods for 3D protein data analysis as well as the dihedral angle space model to present protein information within our application project.

2.1 Current Methods of 3D Protein Data Analysis

In recent years, a good number of high-resolution X-ray structures of proteins stored in the Protein Data Bank (PDB, http://www.pdb.org) have become available. Statistical methods have been applied to the PDB to extract knowledge about the conformational behaviour[1] of the smallest protein substructures, the amino acid residues. Amino acid side chain conformations have been studied, for example, by [4, 8]. These studies led to side chain rotamer libraries, which consist of a list of discrete conformations with a weight corresponding to their frequency in the PDB. Since the PDB contains a multitude of high-resolution structures, it was also possible to determine rotamer preferences depending on the backbone conformation. A backbone-dependent side chain rotamer library for example was developed by [5].

Based on this idea, a number of weak correlations of rotamer distributions and secondary structures have been found by [11]. The effectiveness of the backbone-dependent rotamer libraries has been shown by [3] for homology modeling and by [9] for NMR and X-ray structure refinement.

Although the idea of using rotamer libraries has already been applied successfully in the past, only a small fraction of their potential has been revealed. The main limitation of the available rotamer libraries is the fact that they describe only correlations within the conformation of the smallest possible substructure of a protein, i.e. for a residue. However, for the better understanding of 3D structures, it is highly desirable to examine larger substructures consisting of multiple residues, so that the relationship of a residue to its neighbours in the sequence can be understood. In [7], the basic idea of finding those kinds of frequent substructures

[1]Conformation of a protein or a substructure means its 3D-structure

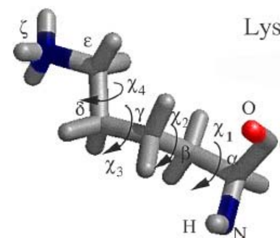


Figure 1. Sample representation of the amino acid Lysine with torsion and side chain angles ([2]).

within a protein data set is explained thoroughly. Based on this idea, the requirements regarding the support of database technology are discussed in the remainder of the paper.

2.2 The Dihedral Angle Model

Within the backbone-dependent rotamer library developed by [10] for each residue type, a probability distribution of the side chain angle χ_1 is calculated for each node on an equidistant grid in the 2D (ϕ, ψ)-space. The distributions of χ_2, χ_3 and χ_4 only depend on the previous side chain dihedral angle. Figure 1 shows the sample amino acid lysine with four torsion axes within its side chain, and the associated dihedral angles (χ_1 to χ_4) of the side chain. From an analysis point of view, the main problem is detecting global relationships consists in the derivation of a probability distribution based on more than one single angle.

In our application framework, we transform the description of the conformation of amino acid residues into data points of a multi-dimensional dihedral angle space. If P denotes a set of protein sequences and A is the set of all 20 natural amino acids, a single sequence $p \in P$ may be formally described as a sequence of linked amino acid residues a, i.e.:

$$p \in P,\ p = a_1 a_2 \ldots, a_l,\ a_i \in A, i = 1, \ldots, l.$$

The main benefit performing all analysis steps within the representation of the dihedral angle model is that (a) the 3D-structures of the conformations are fully reflected and (b) the representation is invariant against rotation and translation. Formally, a single protein $p \in P$ is represented in the dihedral angle model as a sequence of vectors of dihedral angles.

$$p \in P,\ D(p) = s_1, s_2, \ldots s_l,$$
$$s_i \in [-180, 180)^{d_i}, i = 1, \ldots, l$$

In this representation, D reflects the mapping of the original 3D coordinates to dihedral angle values and d_i is the number of dihedral angles for the associated amino acid residue

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003". S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

a_i consisting of the 3D position denoted by ϕ, ψ, ω and the optional side chain angles χ_1 to χ_4. Since the backbone part of a residue is equal for all residue types, the backbone angles (s_1, s_2, s_3) have to be present. Depending on the specific residue type, the remaining angle values represent the optional side chain angles χ_1 to χ_4.

$$s_i = \begin{cases} (\phi, \psi, \omega) & ; d_i = 3 \\ (\phi, \psi, \omega, \chi_1, \ldots, \chi_{d_i - 3}) & ; 3 < d_i \leq 7 \end{cases}.$$

For example, the amino acid glycine is represented only by its backbone information without any additional side chain angles, i.e. $d_i = 3$. Argenine on the other hand exhibits four side chain angles, so that d_i results to 7.

The corresponding relational mapping of the protein information consists of a single relational table. Each tuple represents a single amino acid residue consisting of the 3-letter identifier, three mandatory (i.e. NOT NULL) and $d_i - 3$ floating number attributes. An additional attribute denotes the dimensionality of the amino acid residue, i.e. reflecting the number of NOT NULL dihedral angle values.

Based on this relational representation, the first step in analysing the 3D protein structures in the introduced dihedral angle model consists in the discretization of the conformational angle space according to the observed data distribution. After assigning a representative angle vector to each individual residue, the second step encompasses the order-preserving generation of frequent item sets. The algorithmic procedures and the relational implementation will be discussed.

3 Data Discretization

The main goal of our application process is the detection of similar and frequent substructures. While the generation of frequent item sets and the necessary extension of the underlying database system are discussed in the following, this section focuses on the problem of finding similar amino acid residues. From an application point of view, we apply a cluster algorithm within the multi-dimensional dihedral angle space. The clustering process results in each amino acid residue being assigned to a given prototype. To achieve the highest degree of similarity, the "nearest" prototype is required to be the representative of a single residue.

For our scenario, the prototypes are pre-determined either by random sampling, i.e. randomly picking a number of data points and using it as prototype, or by applying more advanced algorithms like k-means. Further details can be found at [7].

3.1 Nearest Neighbour Rule

From the application point of view, the conformational space of each of the 20 natural amino acid residues is mapped onto a set of prototypes within the discretization step. From an implementational point of view, the data points in the multi-dimensional dihedral space are mapped onto predefined prototypes using the nearest neighbour rule. In the context of the dihedral angle model, the distance between two points in the multi-dimensional space is computed with the shortest path measures using the Euclidean distance. Due to the continuous semantics of an angle space, an adaptation to the regular distance computation is necessary because of the wrap-around effect at the point $(-180, +180)$. Figure 2 illustrates the effect of the wrap-around at the borders of a multi-dimensional angle space using a two-dimensional plane. More formally, the distance between two data points in an d-dimensional angle space is computed by the following formula:

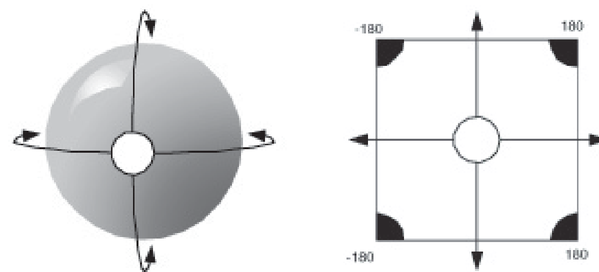


Figure 2. Wrap-Around Effect in the Dihedral Angle Model

$$x, y \in [-180, 180)^d, \; dist(x, y) = \sqrt{\sum_{i=1}^{d} \begin{cases} |x_i - y_i|^2, & |x_i - y_i| \leq 180 \\ (360 - |x_i - y_i|)^2, & \text{else} \end{cases}}$$

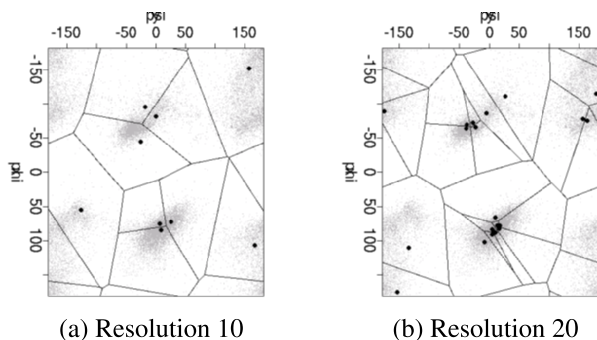
In SQL, the equation computing the Euclidian distance between two points in a multidimensional dihedral angle space may be easily achieved by using *case* statements to cover the wrap-around effect and to deal with the variable number of angles, i.e. with the flexible dimensionality of the amino acid residues. For the sake of simplicity, the scalar function *edist(x,y)* is introduced for a small sample scenario.

For a five-dimensional angle space $(\phi, \psi, \omega, \chi_1, \chi_2)$, the *edist()* function would be defined as follows:

```
CREATE FUNCTION edist(
 x1 FLOAT, x2 FLOAT, x3 FLOAT, x4 FLOAT,
 x5 FLOAT, y1 FLOAT, y2 FLOAT, y3 FLOAT,
 y4 FLOAT, y5 FLOAT)
RETURNS FLOAT
AS
  CASE WHEN ABS(x1-y1)>360
    THEN (ABS(x1-y1)-360)*(ABS(x1-y1)-360)
```


Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003". S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977



(a) Resolution 10 (b) Resolution 20

Figure 3. Discretizations of the conformational space of glycine with different resolutions.

```
    ELSE (ABS(x1-y1))*(ABS(x1-y1))
  END +
  CASE WHEN ABS(x2-y2)>360
    THEN (ABS(x2-y2)-360)*(ABS(x2-y2)-360)
    ELSE (ABS(x2-y2))*(ABS(x2-y2))
  END +
  CASE WHEN ABS(x3-y3)>360
    THEN (ABS(x3-y3)-360)*(ABS(x3-y3)-360)
    ELSE (ABS(x3-y3))*(ABS(x3-y3))
  END +
  CASE WHEN x4 IS NOT NULL
    THEN CASE WHEN ABS(x4-y4)>360
      THEN (ABS(x4-y4)-360)*(ABS(x4-y4)-360)
      ELSE (ABS(x4-y4))*(ABS(x4-y4))
      END
    ELSE 0
  END +
  CASE WHEN x5 IS NOT NULL
    ... -- analogously to x4
  END
AS dist
```

The effect of the discretization is shown for the conformational space of glycine in figure 3. Figure 3(a) shows the discretization with resolution 10 and part (b) with a resolution of 20 prototypes. All data points (plotted in grey) in the same cell are assigned to the black plotted prototype. Note that because of the wrap-around effect, the cells at a border are connected with the opposite part.

3.2 Computing the Nearest Neighbor

The most obvious way to compute the nearest neighbour for each conformation of the current amino acid residue is to iterate over the content of the database. An inner loop over all potential prototypes is then required to identify the nearest prototype, i.e. the cluster to which the current amino acid residue belongs to. Although this algorithm might be easily implemented at the application level, each point of the multi-dimensional angle space must be retrieved from the database, compared with the potential prototypes, and the result must be written back into the database, i.e. every tuple must be updated to record the associated prototype.

Another option would be exploiting the capabilities of the underlying database system and implementing the nearest neighbour search using SQL. Since no data transfer between the database system and the application at the client is necessary, this approach looks promising IF the database system is able to provide the necessary operational environment.

Based on the two tables *PConformation*, holding the raw data converted and imported directly from the PDB, and *PPrototypes*, reflecting the set of predefined prototypes, we discuss three different methods computing the nearest neighbour directly inside the database system.

3.3 The Pivot-Method

The Pivot-Method is based on the idea that the set of prototypes is transformed, so that the resulting table holds only one single tuple with n-times the original set of columns, where n is the number of prototypes.

Supposed we have three different prototypes within a five-dimensional angle space (ϕ, ψ, ω, χ_1, χ_2). A view defining the transformed table would be specified as follows:

```
CREATE VIEW PrototypesPivot AS
SELECT p1.id AS id1, p1.phi AS phi1,
  p1.psi AS psi1, p1.omega AS omega1,
  p1.chi1 AS chi11, p1.chi2 AS chi12,
  p2.id AS id2, p2.phi AS phi2,
  p2.psi AS psi2, p2.omega AS omega2,
  p2.chi1 AS chi21, p2.chi2 AS chi22,
  p3.id AS id3, p3.phi AS phi3,
  p3.psi AS psi3,p3.omega AS omega3,
  p3.chi1 AS chi31, p3.chi2 AS chi32
FROM PPrototypes p1, PPrototypes p2,
  PPrototypes p3
WHERE p1.pid = 1 AND p2.pid = 2 AND
  p3.pid = 3
```

Although pivoting the prototype table looks terribly inefficient, the self-join of the prototypes addresses exactly a single tuple, so that an existing index on the prototype ID reduces the overhead of the join to the concatenation of the prototype tuples.

The selection of the nearest neighbour is then expressed as the Cartesian product of the protein conformation table with the single tuple of the view *PrototypesPivot* followed by the *case* statement picking the prototype ID with the minimum Euclidean distance. For the running example, the statement would be as follows:

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003".
S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

```
SELECT p.id, p.phi, p.psi, p.omega,
  p.chi1, p.chi2,
  -- id of amino acid residue
  x.id, x.mindist
  -- id of prototype with min distance
FROM (SELECT phi, psi, omega, chi1, chi2,
  least(edist(phi, psi, omega, chi1, chi2,
    phi1, psi1, omega1, chi11, chi12),
    edist(phi, psi, omega, chi1, chi2,
      phi2, psi2, omega2, chi21, chi22),
    edist(phi, psi, omega, chi1, chi2,
      phi3, psi3, omega3, chi31, chi32))
    as mindist
  FROM PConformations, PrototypesPrivot) x,
       Prototypes p
WHERE p.phi = x.phi AND p.psi = x.psi AND
  p.omega = x.omega AND p.chi1 = x.chi1 AND
  p.chi2 = x.chi2
```

It is worth mentioning here that the $least()$ scalar function like the $min()$ aggregation function produces the minimal value of the given set of parameters but does not preserve the position of the minimal value, so that a rejoin with the original *Prototypes* table is necessary to retrieve the ID of the nearest prototype for each tuple, i.e. amino acid residue, produced by the inner view. Moreover, the $least()$ operator is not part of the SQL standard but provided by a huge number of database systems, like Oracle, or must be implemented as UDF. Since the function is CPU-bound, the execution cost may be neglected.

3.4 The SelfJoin-Method

The self-join method is basically equivalent to the pivot method; it picks the minimal distance from prototypes stored vertically, i.e. as tuples within a table, and not horizontally after performing the pivot operation.

Analogously to the pivot method, a view is defined as a shortcut in the core SQL statement to determine the nearest neighbour. The view *PrototypeAssign* computes the Euclidean distance for each single amino acid residue and every possible prototype candidate. It is worth mentioning here that no restriction in the *where* clause is feasible to reduce the size of the temporary result.

```
CREATE VIEW PrototypeAssign AS
  SELECT pc.id as pc_id, pc.phi, pc.psi,
    pc.omega, pc.chi1, pc.chi2,
    --  id of amino acid residue
    pt.id as pt_id,
    edist(pc.phi, pc.psi, pc.omega,
      pc.chi1, pc.chi2, pt.phi, pt.psi,
      pt.omega, pt.chi1, pt.chi2) AS dist
    -- id of prototype with distance
  FROM ProteinConformation pc, Prototypes pt
```

Picking the combination with the minimum distance may be accomplished by referring to the predefined view in two separate phases. Within an inner query for each amino acid residue, the minimal distance is generated using a $groupby$ by operator and the $min()$ aggregation function. Unfortunately, the information which prototype (i.e. the prototype id) is lost in this step and must be "added" by a re-join in the outer query. The re-join again requires the view definition. The correct prototype ID (and the angle values) are retrieved by picking the minimal distance prototype for each amino acid residue.

```
SELECT pa.pc_id, pa.phi, pa.psi, pa.omega,
  pa.chi1, pa.chi2,
  --  id of amino acid residue
  pa.pt_id, pa.dist
  -- id of prototype with min. distance
FROM (SELECT pc_id, min(dist) AS dist
  FROM PrototypeAssign GROUP BY pc_id) pc,
  -- each amino acid with min. distance
  prototypeassignment pa
WHERE  pc.pc_id = pa.pc_id
  AND  pc.dist = pa.dist;
```

3.5 The MINPOS() Method

The self-join method would be very nice and intuitive if the problem of losing the position of the minimal values could be eliminated. If an adequate operator were available, the re-join could be avoided and the query could be executed much more efficiently. As a consequence, we propose an extension to the set of aggregate functions by introducing the $minpos()$ and $maxpos()$ function. Both functions are associated to the $min()$ / $max()$ function and return the value of a column P if the value of an aggregate column X holds the minimal/maximal value within an aggregate group. If multiple tuples exhibit the minimal/maximal value, the $minpos()$ / $maxpos()$ function returns randomly one of the valid solutions. For example, consider the following table with three columns A, X, and P and six tuples.

A	X	P
A1	5	1
A1	3	2
A1	7	3
A2	2	1
A2	8	2
A2	2	3

The application of $minpos()$ and $maxpos()$ can be illustrated using the following query:

```
SELECT A, MIN(X), MINPOS(X,P),
  MAX(X), MAXPOS(X,P)
FROM R <<table above>>
GROUP BY A
```

Obviously, the query returns two rows (one for each group). The $minpos()$-function returns the values of column P of the tuple holding the smallest X value within each group, i.e.:

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003".
S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

```
    A MIN(X) MINPOS(X,P) MAX(X) MAXPOS(X,P)
----- ------ ----------- ------ -----------
   A1      3           2      7           3
   A2      2           3      8           2
```

Using the extension, the discretization step could be executed more efficiently by the following query based on the view *PrototypeAssign*:

```
SELECT pc.id, pc.phi, pc.psi, pc.omega,
  pc.chi1, pc.chi2,
  --  id of amino acid residue
  pa.pt_id, pa.dist
  -- id of prototype with min. distance
FROM (SELECT pc_id, MIN(dist) AS dist,
    MINPOS(dist, pt_id) AS pt_id
  FROM   prototypeassignment
  GROUP BY pc_id) pa,
  -- each amino acid id with min.
  -- distance and prototype id
  ProteinConformation pc
 WHERE pa.pc_id = pc.id;
```

It is worth mentioning here that the join with the *ProteinConformations* table can be avoided if the result does not require the individual angles of all amino acid residues.

4 Generating Order-Preserving Frequent Item Sets

Based on the original data set with the associated nearest neighbour, we can focus on the derivation of association rules to discover frequent substructures within the protein data set. In a first step, transaction sets are generated, so that the order of the amino acid residues is preserved. A second step encompasses the generation of the frequent item sets applying well-known algorithms like APriori. Again, the database perspective will be discussed. In this context, a new operator supporting the generation process of the item sets will be introduced.

4.1 Generating the Transaction Data Set

After the discretization of the conformational spaces by assigning the nearest prototype to each amino acid residue, the vector sequences, describing the conformations of the protein structures $p \in P$, are transformed into sequences over a discrete alphabet. The basic idea of the approach is to replace each residue conformation by the prototype identifier, which stands for a set of similar 3D conformations of the same residue type. Formally, the following mapping function C from the residue conformations into the set of prototype identifiers CI is given:

$$p \in P,\ D(p) = s_1, \ldots, s_{l_p},\ \forall j \in \{1, \ldots, l_p\} : \\ C(s_j) = c_j,\ c_j \in CI$$

For a more convenient notation, we write for $p \in P$:

$$C(D(p)) := C(s_1), \ldots, C(s_{l_p}) = c_1, \ldots, c_{l_p}.$$

The transformation of the sequences is exemplified below using a short subsequence of the protein with the PDB code *1slf B*.

```
GLU  -61  -44  179   54   81   53 -> GLU66
ALA  -54  -51  177                -> ALA1
GLY  -54  -35  178                -> GLY71
ILE  -77  -37 -175  -65  176      -> ILE96
THR  -62  129  177  -62           -> THR168
GLY  153 -166  178                -> GLY72
THR -105  137 -177  -59           -> THR164
TRP -135  154  175  -61   76      -> TRP173
TYR -137  151  179  -65  -85      -> TYR184
ASN -107 -178 -174   74    8      -> ASN24
```

The naive application of algorithms finding frequent subsequences [13, 14] is not feasible due to the following two reasons: first of all, the subsequences with gaps of different sizes between the elements (residues) are matched as equal. This makes no sense in the context of protein 3D structures, since the relative distances between the residues are in general important for the whole 3D structure. Second, the classical application of well-known algorithms like APriori to generate frequent item sets and derive association rules based on confidence and support [1, 12] do not obey the order of the residues.

To make the APriori algorithm sensitive to the order of the residues, we additionally consider the positions of the single amino acid residues and generate a new set of transactions as the input for the APriori algorithm. To generate the transactions, a window of the maximum length of a frequent substructure is moved over the sequences. In each step, the position inside the window is added to the letters, consisting of residue type and prototype identifier. This transformation is exemplified in figure 4 for a window of four residues using the same sequence as in the previous example.

The resulting transaction data set is analyzed by the APriori algorithm generating interesting relationships based on the notion of frequent item sets. In this context, frequent denotes that an item set appears more than $supp_{min}$ times in the transaction set. Each found frequent item set corresponds to a 3D substructure of multiple residues, which appears at least $supp_{min}$ times in the given 3D protein sequences. From the frequent item sets, general association rules are derived, which uncover unknown implications. Association rules can be seen as a kind of logic rule, which

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003".
S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

```
    GLU66  ALA1  GLY71  ILE96  THR168  GLY72  THR164  TRP173  TYR184  ASN24
T1 1GLU66 2ALA1 3GLY71 4ILE96
T2        1ALA1 2GLY71 3ILE96 4THR168
T3              1GLY71 2ILE96 3THR168 4GLY72
T4                     1ILE96 2THR168 3GLY72 4THR164
T5                            1THR168 2GLY72 3THR164 4TRP173
T6                                    1GLY72 2THR164 3TRP173 4TYR184
T7                                           1THR164 2TRP173 3TYR184 4ASN24
...
```

Figure 4. Example for the generation of transactions from a sequence.

is true with a certain probability. To restrict the set of possible association rules, a minimum confidence $conf_{min}$ is required.

Since the relative positions of the residues are directly encoded in each item, only subsequences with the exactly same order of residues are matched. This approach also allows gaps in the sequences. However, in contrast to general frequent sequences, the lengths of the gaps in the matching subsequences all have the same size, which is important in this application context. Details regarding the process and experimental results without the use of an underlying database system can be found in [7]

4.2 Generating Frequent Item Sets inside the Database

The necessary prerequisite to efficiently produce frequent item sets inside the database consists in the existence of adequate operators which - on the one hand - allow the user to easily specify application specific tasks and - on the other hand - enable the system to apply internal optimization strategies. In this section, we show that the pure grouping functionality (extended by the *cube()* operator family, [6]) is not sufficient to support the application of data mining algorithms

The relational mapping of our sample protein conformation scenario is rather straightforward. The generated transactions are stored within a single table with four columns denoting the position of the amino acid residues inside the sliding window defining the substructure and a running number as the transaction ID:

TID	A1	A2	A3	A4
T1	1GLU66	2ALA1	3GLY71	4ILE96
T2	1ALA1	2GLY71	3ILE96	4THR168
T3	1GLY71	2ILE96	3THR168	4GLY72
T4	1ILE96	2THR168	3GLY72	4THR164
T5	1THR168	2GLY72	3THR164	4TRP173
T6	1GLY72	2THR164	3TRP173	4TYR184
T7	1THR164	2TRP173	3TYR184	4ASN24
...				

Computing item sets and weighting them with the necessary support and confidence is the main crucial operation in computing association rules. For the n-th generation, an item set reflects all n-combinations for a given set of grouping attributes. With the number of attributes reasonably low (.i.e. short subsequences), we may use the *grouping sets()* clause to specify the necessary combinations within a single query. For example, with four attributes $A1$, $A2$, $A3$, and $A4$, we may issue the following grouping condition:

```
GROUP BY GROUPING SETS ((A1,A2),(A1,A3),
                        (A1,A4),(A2,A3),
                        (A2,A4),(A3,A4))
```

For larger subsequences, i.e. columns contributing to the computation of item sets, the specification of all permutation combinations can be a tedious task. Another alternative is to issue a *cube()* operator over the given set of parameters and eliminate the unwanted grouping combinations in a *having* clause referring to the *grouping()* columns. For the ongoing example with four attributes, we yield the following expression:

```
GROUP BY CUBE(A1,A2,A3,A4)
HAVING NOT(
-- exclude 0-cardinality combinations
(GROUPING(A1) = 1 AND GROUPING(a2) = 1 AND
 GROUPING(a3) = 1 AND GROUPING(a4) = 1)
-- exclude 1-cardinality combinations
OR (GROUPING (A1) = 1 AND GROUPING(A2) = 1
                      AND GROUPING(A3) = 1)
OR (GROUPING (A1) = 1 AND GROUPING(A2) = 1
                      AND GROUPING(A4) = 1)
OR (GROUPING (A1) = 1 AND GROUPING(A3) = 1
                      AND GROUPING(A4) = 1)
OR (GROUPING (A2) = 1 AND GROUPING(A3) = 1
                      AND GROUPING(A4) = 1))
-- exclude 3- and 4-cardinality combinations
...
```

Although this would return the required 2-itemset combinations, the specification becomes tedious with an increasing number of columns. Moreover, the computation of all possible combinations is certainly not the most efficient way of computing the required item sets.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003". S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

With this motivation as a background, we propose the *GROUPING COMBINATIONS()* operator, generating all k item sets for a given set of n grouping columns. The syntax integrates seamlessly into the SQL grouping extensions of *cube()*, *rollup()*, and *grouping sets()*:

```
GROUP BY GROUPING
         COMBINATIONS((A1,A2, ... ,An), k)
```

This expression may be seen as a shortcut for the *grouping sets* expression with k over n item sets each with a cardinality of k, i.e.:

```
GROUP BY GROUPING SETS((A1,A2,...,Ak),
                         (A1,A2,...,Ak-1,Ak+1),
                          ...)
```

The basic idea of the grouping combinations operator is generating all possible grouping combinations with a given cardinality of k. Special cases must be considered regarding the value of k and the number of attributes.

- For $k = n$, the *grouping combinations* operator returns a single set with all grouping columns given as a first parameter, e.g.:

  ```
  GROUP BY GROUPING
        COMBINATIONS((A1,A2, ..., An), n)
  ```

  is equal to

  ```
  GROUP BY GROUPING SETS((A1,A2, ..., An))
  ```

  is equal to a regular

  ```
  GROUP BY A1,A2, ..., An
  ```

- For $k = 1$, the operator corresponds semantically to a grouping sets with n single sets, each consisting of exactly a single grouping column. More formally:

  ```
  GROUP BY GROUPING
           COMBINATIONS((A1,A2, ...,An), 1)
  ```

  is equal to

  ```
  GROUP BY GROUPING
           SETS((A1), (A2), ..., (An))
  ```

- For $k = 0$, the proposed *grouping combinations* operator produces a single grouping combination encompassing all tuples of the underlying table, e.g.

  ```
  GROUP BY GROUPING
           COMBINATIONS((A1,A2, ...,An), 0)
  ```

  is equal to a query without any explicit *group by* clause, but an aggregation function in the *select* clause.

The *grouping combinations* operator extends the set of *group by* operators proposed within the OLAP context. The generation of frequent item sets for a given generation, i.e. cardinality of the grouping combinations, however is not supported by these operators.

5 Cost Estimations

This section analyses the different methods to compute the cluster and to find the frequent item sets from a performance point of view. In a first step, our underlying cost model is outlined, followed by costing the different alternatives to assign the nearest prototype to each amino acid residue.

5.1 General Cost Model

To estimate the cost, we rely on the linear cost model taking the size of the data stream between relational plan operators to compare the costs of different query plans. The size of the data stream is the result of the number of attributes multiplied with the cardinality of the table.

To evalutate the selection predicates, we consider a full table scan with the exception of a single tuple access exploiting the existence of an index structure. For the join operator, we may choose between a nested loop and a hash join. The first alternative implies a multiplication of the data stream cardinalities and is omitted in favour of a hash join. For justifying this assumption, we rely on the fact that a machine has enough capacity to keep the prototype table in main memory. A hash join would read the smaller table during the build phase and the larger table during the probe phase, so that we end up in adding the cardinalities of the data streams to compute the cost of a join operator. Due to our 1:N-relationship of prototypes and conformations, the cardinality of the resulting data stream after a join equals the cardinality of the larger table.

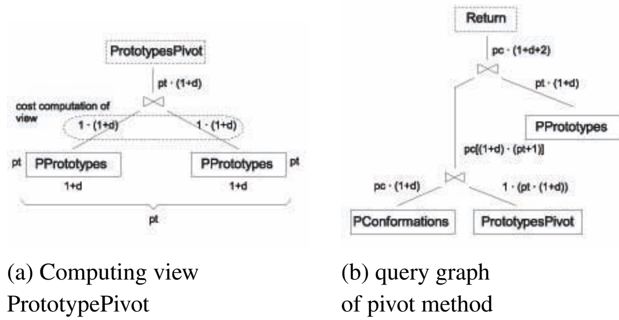
At last, we consider the size of the input stream as the cost of a group by operator, because every tuple has to be taken into account and assigned to the corresponding group. With this assumption, we again consider a hash-based implementation so that we may neglect the sorting cost in case of a sort-based group by implementation. The return operator finally simulates the cost delivering the result to the client program (or in detail: storing the result set temporarily within the SQLDA of a database system). In our example, the *PConformations* table has $1 + d$ attributes (with d as the number of maximum angles) and a cardinality of pc. Similarly, the *Prototypes* table has the same number of attributes with a cardinality of pt. Based on this assumption, the cost computing the following query would be computed as follows:

```
SELECT pc.phi, pc.psi, pc.omega, pc.chi1,
       pc.chi2 pt.omega, pt.chi1, pt.chi2
FROM PConformations pc, prototypes pt
WHERE pc.phi = pt.phi
  AND pc.psi = pt.psi
```

The scan and the join of both tables would yield an overhead of $pc * 6 + pt * 6$ (with $d = 5$). The return operator

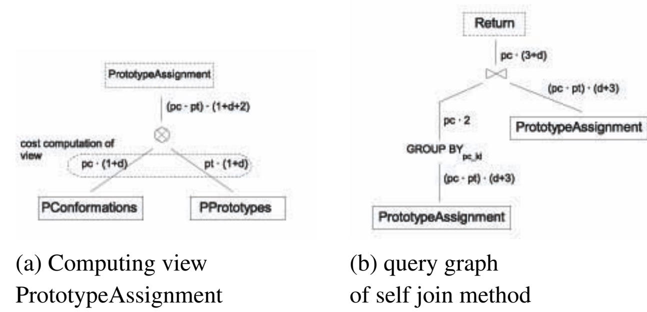
Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003".
S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977



(a) Computing view PrototypePivot (b) query graph of pivot method

Figure 5. pivot method query graphs



(a) Computing view PrototypeAssignment (b) query graph of self join method

Figure 6. self-join method query graphs

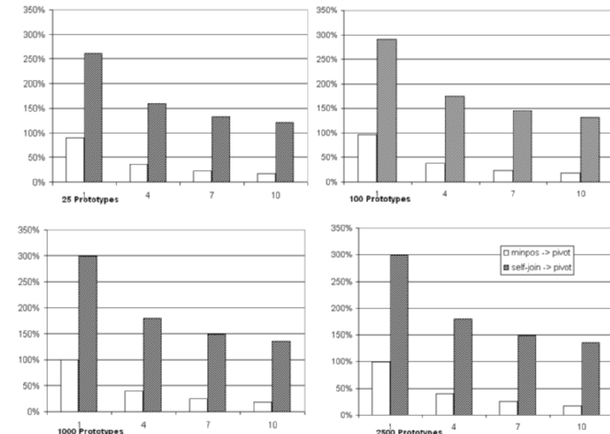


Figure 7. cost reduction scenario

reflecting the produced columns of the select clause contributes with a value of $pc * 8$ to the overall cost resulting in $14 * pc + 6 * pt$.

5.2 Comparing different Methods

The cost of computing the nearest neighbour using the pivot method may be split into the computation of the cost for the view definition to perform the pivot of the prototype information and the core select statement using the pivot view. As already mentioned, the $pt - 1$-ary join of the prototype table is easily optimized by the existence of an index of the prototype ID. In this case, each join partner has the size of $1*(1+d)$ yielding $pt*(1+d)$ as the overall cost computing the view (without any return operator). The query itself joins the view with the *PConformations* table. Reading the result of the view corresponds to one single tuple with $pt * (1 + d)$ columns. The access of the *PConformations* table has the size of $pc * (1 + d)$. The outer query adds to the overall costs with reading the result of the inner query $(pc * (1 + d) * (pt + 1))$ and accessing the *Prototypes* table $(pt * (1 + d))$. The return operator finally consumes a data stream of cardinality pc with $d + 3$ attributes.

The cost of discretization using the self-join method can be computed in three steps. The first step considers the computation of the view $PrototypeAssignment$ which requires the access to the two tables $ProteinConformation$ $(pc * (1 + d))$ and $Prototypes$ $(pt * (1 + d))$. The resulting data stream, which has to be read for further processing yields (due to the semantics of the Cartesian product) to $(pc * pt) * (1 + d + 2)$. The second step addresses the inner query consuming exactly the size of the view and producing a data stream of $pc * 2$ for only two columns. The outer query reads the result of the inner query and the result of the view $((pc * pt) * (3 + d))$. At last, the return operator has costs the size of the join operator yielding $pc * (d + 3)$.

In a similar way, the cost for the $minpos()$ method can be computed. Instead of the join with the $PrototypeAssigment$ view in the outer query, the $ProteinConformation$ table with a cost of $pc * (d + 1)$ is referenced. Additionally, the view must be executed only once, further reducing the execution costs.

Table 1 summarizes the partial and total cost for all different methods computing the prototype for each amino acid residue. Since this tabular and formular-based representation does not give any hint about the best strategy, figure 7 gives a scenario with four different dimensions (i.e. $d = 1, 4, 7, 10$) and 5000 amino acid residues. The scenario shows the resulting performance gain of the pivot method compared to $minpos()$ and selfjoin method with 25, 100, 1000, and 2500 prototypes. Within the proposed cost model, the pivot method yields a slightly lower total cost than the $minpos()$ method because the $pt - 1$-ary self-join is considered extremely cheap. However, due to the dependency of the total cost from the number of prototypes, the pivot method can not be considered a feasible solution for real applications with a reasonable high number of prototypes. Compared to the self-join method, the $minpos()$ method yields a substantial cost reduction.

6 Summary and Conclusion

This paper introduces the problem of finding frequent substructures in protein data sets. The analysis process is split into two parts. The first step consists in finding the

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "15th International Conference on Scientific and Statistical Database Management, Cambridge, MA, 2003". S. 161-171. ISBN 0-7695-1964-4
https://doi.org/10.1109/SSDM.2003.1214977

method	step	cost	# of database operations
pivot method	view execution:	$pt * (1 + d)$	# of database operations:
	inner query:	$pt * (1 + d) + pc * (1 + d)$	$(pt + 1)$ joins
	outer query:	$pc * (1 + d) * (pt + 1) + pt * (1 + d) + pc * (d + 3)$	
	total cost:	$pt * (3 + 5d) + pc * (5 + 3d) + pc * pt * (1 + d)$	
self-join method	view execution:	$2 * (pc * (1 + d) + pt * (1 + d))$	# of database operations:
	inner query:	$(pc * pt) * (3 + d)$	1 join
	outer query:	$pc * 2 + (pc * pt) * (3 + d) + pc * (3 + d)$	2 cross product
	total cost:	$pt * (2 + 2d) + pc * (7 + 3d) + pc * pt * (6 + 2d)$	1 group by
minpos() method	view execution:	$pc * (1 + d) + pt * (1 + d)$	# of database operations:
	inner query:	$(pc * pt) * (3 + d)$	1 join
	outer query:	$pc * 2 + pc * (1 + d) + pc * (3 + d)$	1 cross product
	total cost:	$pt * (1 + d) + pc * (7 + 3d) + (pc * pt) * (3 + d)$	1 group by

Table 1. Cost comparison of the different methods

nearest prototype in the multi-dimensional dihedral angle space. To accomplish this task, the data sets are brought into a relational schema and a method is proposed to compute the minimal distance considering the wrap-around effect in the angle space. Three different methods to find an associated prototype inside the database systems are compared. A minimal SQL extension (*minpos()*/*maxpos()*) function results in much more efficient query execution plans. The second step of generating frequent item sets to detect frequent substructures within the amino acid sequences, requires substantial SQL extension. A new operator (as a new member of the OLAP grouping function operators) is introduced. This operator is a generic tool and may be exploited by a huge set of data mining applications. To summarize, a database system, used to efficiently analyse huge data volumes, requires additional support from the technology perspective. The required extension range from minimal UDFs like our proposed *minpos()*/*maxpos()* functions to more complex operators like our proposed *grouping combinations* operator. If (and only if) the database community provides this kind of functionality, the acceptance of database systems in the biotechnology community will increase in the near future.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[2] S. Bohl, M. Dinkelacker, J. Griese, and S. Schrader. Highly adaptable amino acid side chain rotamer library in pdb coordinates. In *Workshop in Computational Biology at the Plant Biochemistry Department of the Albert-Ludwigs-Universitt Freiburg, Germany*, 2002.

[3] M. Bower, F. Cohen, and R. Dunbrack. Homology modeling with a backbone-dependent rotamer library. *J. Mol. Biol.*, 267:1268–1282, 1997.

[4] R. Chandrasekaran and G. Ramachandran. Studies on the conformation of amino acids. xi. analysis of the observed side group conformations in proteins. *Int. J. Pept. Prot. Res.*, 2:223–233, 1970.

[5] R. Dunbrack and F. Cohen. Bayesian statistical analysis of protein side-chain rotamer preferences. *Protein Sci.*, 6:1661–1681, 1997.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 152–159. IEEE Computer Society, 1996.

[7] A. Hinneburg, M. Fischer, and F. Bahner. Finding frequent substructures in 3d-protein databases. In *Workshop on Bioinformatics at the 19th International Conference on Data Engineering*. IEEE Computer Society, 2003.

[8] M. James and A. Sielecki. Structure and refinement of penicillo-pepsin at 1.8 a resolution. *J. Mol. Biol.*, 125:299–361, 1983.

[9] J. Kuszewski, A. Gronenborn, and G. Clore. Improving the quality of nmr and crystallographic protein structures by means of conformational database potential derived from structure databases. *Protein Sci.*, 5:1067–1080, 1996.

[10] S. C. Lovell, J. M. Word, J. S. Richardson, and D. C. Richardson. The penultimate rotamer library. *Proteins: Struct Funct Genet*, 40:389–408, 2000.

[11] M. MCGregor, S. Islam, and M. Sternberg. Analysis of the relationship between side-chain conformation and secondary structure in globular proteins. *J. Mol. Biol.*, 198:295–310, 1987.

[12] R. Srikant and R. Agrawal. Mining generalized association rules. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, Switzerland*, pages 407–419. Morgan Kaufmann, 1995.

[13] M. J. Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, Bethesda, Maryland, USA, November 3-7, 1998*, pages 68–75. ACM, 1998.

[14] M. Zhang, B. Kao, C. L. Yip, and D. W.-L. Cheung. Ffs - an i/o-efficient algorithm for mining frequent sequences. In *Knowledge Discovery and Data Mining - PAKDD 2001, 5th Pacific-Asia Conference, Hong Kong, China, April 16-18, 2001, Proceedings*, volume 2035 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2001.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden