Chapter 2
Multithreaded PSS-SQL for Searching Databases of Secondary Structures

...; life was no longer considered to be a result of mysterious and vague phenomena acting on organisms, but instead the consequence of numerous chemical processes made possible thanks to proteins.

Amit Kessel, Nir Ben-Tal, 2010 [13]

Abstract Protein secondary structure (PSS), as an organizational level, provides important information regarding protein construction and regular spatial shapes, including alpha-helices, beta-strands, and loops, which protein amino acid chain can adopt in some of its regions. The relevance of this information and the scope of its practical applications cause the requirement for its effective storage and processing. In this chapter, we will see how PSSs can be stored in the relational database and processed with the use of the protein secondary structure-structured query language (PSS-SQL). The PSS-SQL is an extension to the SQL language. It allows formulation of queries against a relational database in order to find proteins having secondary structures similar to the structural pattern specified by a user. In this chapter, we will see how this process can be accelerated by parallel implementation of the alignment using multiple threads working on multiple-core CPUs.

Keywords Proteins · Secondary structure · Query language · SQL · Relational database · Multithreading · Parallel computing · Alignment

2.1 Introduction

Secondary structures are a kind of intermediate organizational level of protein structures, a level between the simple amino acid sequence and complex 3D structure. The analysis of protein structures on the basis of the secondary structures is very supportive for many processes that are important from the viewpoint of biomedicine

D. Mrozek, *High-Performance Computational Solutions in Protein Bioinformatics*, SpringerBriefs in Computer Science, DOI: 10.1007/978-3-319-06971-5_2,

and pharmaceutical industry, e.g., drug design. Algorithms comparing protein 3D structures and looking for structural similarities quite often make use of the secondary structure representation at the beginning as one of the features distinguishing one protein from the other. Secondary structures are taken into account in algorithms, such as VAST [8], LOCK2 [20], CTSS [5], CASSERT [16]. Also in protein 3D structure prediction by comparative modeling [12, 28], particular regions of protein structures are modeled through the adoption of particular secondary structure types of proteins that structure is already determined and deposited in a database. Secondary structure organizational level also shows what types of secondary structure a protein molecule is composed of, what is their arrangement—whether they are segregated or alternating each other. Based on the information proteins are classified by systems, such as CATH [19] and SCOP [18]. All these examples show how important the description by means of secondary structures is.

For scientists studying structures and functions of proteins, it is very important to collect data describing protein construction in one place and have the ability to search particular structures that satisfy given searching criteria. Consequently, this needs an appropriate representation of protein structures allowing for effective storage and searching. The problem is particularly important in the face of dynamically growing amount of biological and biomedical data in databases, such as PDB [4] or Swiss-Prot [3].

At the current stage of development of IT technologies, a well-established position in terms of collecting and managing various types of data reached relational databases [6]. Relational databases collect data in tables (describing part of reality) where data are arranged in columns and rows. Modern relational databases also provide a declarative query language—SQL that allows retrieving and processing collected data. The SQL language gained a great power in processing regular data hiding details of the processing under a quite simple SELECT statement. However, processing biological data, such as protein secondary structures (PSSs), by means of relational databases are hindered by several factors:

- Data describing protein structures have to be managed by database management systems (DBMSs), which work excellent in commercial uses, but they are not dedicated for storing and processing biological data. They do not provide the native support for processing biological data with the use of the SQL language, which is a fundamental, declarative way of data manipulation in most modern relational database systems.
- Processing of biological data must be performed by external tools and software applications, forming an additional layer in the IT system architecture, which is a disadvantage.
- Currently, results of data processing are returned in different formats, like: table-form datasets, TXT, HTML, or XML files, and users must adopt them in their software applications.
- Secondary processing of the data is difficult and requires additional external tools.

In other words, modern relational databases require some enhancements in order to deal with the data on secondary structures of proteins. The possibility of collecting

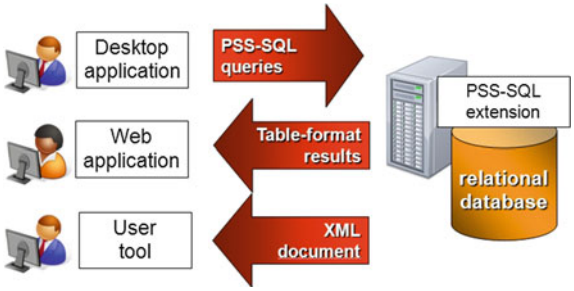

Fig. 2.1 Exploration of protein secondary structures in relational databases using PSS-SQL language. Secondary structure description of protein molecules is stored in relational database. The database management system (DBMS) has the PSS-SQL extension that interprets queries submitted by users. Users can connect to the database from various tools, desktop software applications, and Web applications. They obtain results of their queries in a table-like format or as an XML document

protein structural data in appropriate manner and processing the data by submitting simple queries to a database simplifies a work of many researchers working in the area of protein bioinformatics. Actually, the problem of storing biological data describing biopolymer structures of proteins and DNA/RNA molecules and possessing appropriate query language allowing processing the data has been noticed in the last decade and reported in several papers. There are only a few initiatives in the world reporting this kind of solutions.

For example, the ODM BLAST [23] is a successful implementation of the BLAST family of methods in the commercial Oracle database management system. ODM BLAST extends the SQL language by providing appropriate functions for local alignment and similarity searching of DNA/RNA and protein amino acid sequences. ODM BLAST works fast, but in terms of protein molecules it is limited only to the primary structure. In [9], authors describe their extension to the SQL language, which allows searching on the secondary structures of protein sequences. The extension was developed in Periscope (dedicated engine) and in Oracle (commercial database system). In the solution, secondary structures are represented by segments of different types of secondary structure elements (SSEs), e.g., hhhllleee. In [24], authors show the Periscope/SQ extension of the Periscope system. Periscope/SQ is a declarative tool for querying primary and secondary structures. To this purpose authors introduced new language PiQL, new data types, and algebraic operators according to the defined query algebra PiOA. The PiQL language has many possibilities. In this paper [25], the authors present their extensions to the object-oriented database (OODB) by adding the Protein-QL query language and the Protein-OODB middle layer for requests submitted to the OODB. Protein-QL allows to formulate simple queries that operate on the primary, secondary, and tertiary level.

Finally in 2010, me and a group of researchers from my university (Silesian University of Technology in Gliwice, Poland) developed the PSS-SQL [15, 17, 26, 27], which is an extension to the Transact-SQL language and Microsoft SQL Server DBMS allowing for searching protein similarities on the secondary structure level (Fig. 2.1).

I had the opportunity to be the manager and supervisor of the project, and I have never stopped thinking on its improvement in the following years. New versions of the PSS-SQL consists of many improvements leading to the significant growth of the efficiency of PSS-SQL queries, including:

- parallel and multithreaded execution of the alignment procedure used in the searching process,
- reduction of the computational complexity of the alignment algorithm by using gap penalty matrices, and
- indexing of sequences of SSEs.

The PSS-SQL language containing these improvements will be described in this chapter. In the chapter, we will also see results of performance tests for sample queries in PSS-SQL language and how to return query results as table-like result sets and as XML documents.

2.2 Storing and Processing Secondary Structures in a Relational Database

Searching for protein similarities on secondary structures by formulating queries in PSS-SQL requires that data describing secondary structures should be stored in a database in an appropriate format. The format should guarantee an efficient processing of the data. In PSS-SQL the search process is carried out in two phases, by:

1. Multiple scanning of a dedicated Segment Index for secondary structures.
2. Alignment of found segments in order to return k-best solutions.

All these steps, including data preparation, creating and scanning the Segment Index, and alignment will be discussed in the following sections.

2.2.1 Data Preparation and Storing

The PSS-SQL uses a specific representation of PSSs while storing them in a database.

Let us assume, we have a protein P described by the amino acid sequence (primary structure):

$$P = \{p_i | i = 1, 2, \ldots, n \wedge p_i \in \Pi \wedge n \in \mathbb{N}\}, \tag{2.1}$$

where n is the length of protein amino acid chain, i.e., the number of amino acids, and Π is a set of 20 common types of amino acids.

Secondary structure of protein P can be then described as a sequence of SSEs related to amino acids in the protein chain:

$$S = \{s_i | i = 1, 2, \ldots, n \wedge s_i \in \Sigma \wedge n \in \mathbb{N}\}, \tag{2.2}$$

```
P0A2U6
ADCC_STRPN
Zinc transport system ATP-binding protein adcC OS=Streptococcus pneumoniae GN=adcC PE=3 SV=1

MRYITVEDLSFYYDKEPVLEHINYCVDSGEFVTLTGENGAAKTTLIKASLGILQPRIGKVAISKTNTQGKKLRIAYLPQQIASFNAGFPSTV
YEFVKSGRYPRKGWFRRLNAHDEEHIKASLDSVGMWEHRDKRLGSLSGGQKQRAVIARMFASDPDVFILDEPTTGMDAGSKNEFYELMHHSA
HHHGKAVLMITHDPEEVKDYADRNIHLVRNQDSPWRCFNVHENGQEVGHA

CCCEEECCCEEECCCCCCEEEEEEECCCCCEEEECCCCCCCHHHHHEEEEECCCCCCCCEEEEECCCCCCEEEEEEHHHHHHHHHCCCCCCE
EEEECCCCCCHHHHHCCCCCHHHHHHHHHHHCCCCCCCCCCCCCCCCCHHHHHHHHHHHHCCCCEEEECCCCCCCCCCCCCHHHHHHHHCC
CCCCEEEEEEECCCCCCCCCCCCCEEEEECCCCCCCCEEEECCCCCCCCCC
```

Fig. 2.2 Sample amino acid sequence of *Zinc transport system ATP-binding protein adcC* in the *Streptococcus pneumoniae* with the corresponding sequence of secondary structure elements

id	protID	protAC	name	length	primary	secondary
3294	2DRA_HUMAN	P01903	HLA class II histoco...	254	MAISGVPVLGFFIIAVLMSAQESWAIK...	CCCCCEEECCCEEEHHHHHHHHHHHH...
3296	2ENR_CLOTY	P11887	2-enoate reductase (...	30	MKNKSLFEVIKIGKVEVXXKIXMAVMG...	CCCCCCEEEEECCCCCCCCCCCCEEEE...
3297	2NPD_NEUCR	Q01284	2-nitropropane dioxy...	378	MHFPGHSSKKEESAQAALTKLNSWFPT...	CCCCCCCCCHHHHHHHHHHHCCCCCC...
3299	2PS_SERHY	P48391	2-pyrone synthase (S...	402	MGSYSSDDVEVIREAGRAQGLATILAI...	CCCCCCCEEEEEEHHHHHHHHHEEEEE...
3300	ACSA1_PSEPK	Q88EH6	Acetyl-coenzyme A sy...	653	MSAAPLYPVRPEVAATTLTDEATYKAM...	CCCEEEEECCHHHHHHHHHHHHHHHH...
3302	ACSA2_ACEXY	Q59167	Cellulose synthase 2...	1596	MIYRAILKRLRLEQLARVPAVSAASPF...	CCCHHHHHHHHHHHHHHCEEECCCCE...

Fig. 2.3 Sample relational table storing sequences of secondary structure elements (SSEs) (*secondary* field), amino acid sequences (*primary* field), and additional information of proteins from the Swiss-Prot database. The table (called *ProteinTbl*) will be used in sample queries presented in next sections. Secondary structures were predicted from amino acid sequences using the Predator program [7]

where each element s_i corresponds to a single element p_i, and Σ is a set of secondary structure types. The set Σ may be defined in various ways. A widely accepted definition of the set provides DSSP [10, 11]. The DSSP code distinguishes the following secondary structure types:

- H = alpha helix,
- B = residue in isolated beta-bridge,
- E = extended strand, participates in beta ladder,
- G = 3-helix (3/10 helix),
- I = 5 helix (pi helix),
- T = hydrogen bonded turn, and
- S = bend.

In practice, the set is often reduced to the three general types [7]:

- H = alpha helix,
- E = beta strand (or beta sheet), and
- C = loop, turn or coil.

An example of such a representation of protein structure is shown in Fig. 2.2, where we can see primary and secondary structures of a sample protein recorded as sequences. In such a way both sequences can be effectively stored in a relational database, as it is shown in Fig. 2.3.

Fig. 2.4 Part of the segment table

id	protID	type	startPos	length
67	3	C	0	3
68	3	H	3	23
69	3	C	26	8
70	3	H	34	12
71	3	C	46	3
72	3	E	49	3

2.2.2 Indexing of Secondary Structures

At the level of DBMS, the PSS-SQL uses additional data structures and indexing in order to accelerate the similarity searching. A dedicated *segment table* is created for the table field storing sequences of secondary structures elements. The segment table consists of secondary structures and their lengths extracted from the sequences of SSEs, together with locations of the particular secondary structure in the molecule (identified by the residue number, Fig. 2.4). Then, additional Segment Index is created for the segment table. The Segment Index is a B-Tree clustered index holding on the leaf level data pages from the additional segment table. The idea of using the segment table and segment index is adopted from the work [9]. The Segment Index supports preliminary filtering of protein structures that are not similar to the query pattern. During the filtering, the PSS-SQL extension extracts the most characteristic features of the query pattern and, on the basis of the information in the index, eliminates proteins that do not meet the search criteria. Afterward, proteins that pass the filtering process are aligned to the query pattern.

If we take a closer look at the segment table, we will see that it stores secondary structures in the form that has been described in Sect. 1.3.2. During the scanning of the Segment Index the search engine of the PSS-SQL tries to match segments distinguished in the given query pattern to segments of the index.

2.2.3 Alignment Algorithm

The alignment implemented in the PSS-SQL is inspired by the Smith–Waterman method [21]. The method allows to align two biopolymer sequences, originally DNA/RNA sequences or amino acid sequences of proteins. When scanning a database the alignment is performed for each pair of sequences—query sequence given by a user and a successive, qualified sequence from a database. In PSS-SQL, after performing multiple scanning of the Segment Index (MSSI), a database protein structure S^D of the length d residues is represented as a sequence of segments (see also formulas 1.14 and 1.15), which can be expanded to the following form:

$$S^D = SSE_1^D L_1, SSE_2^D L_2, \ldots, SSE_n^D L_n, \tag{2.3}$$

where $SSE_i^D \in \Sigma$ describes the type of secondary structure (as defined in Sect. 2.2.1), n is the number of segments (secondary structures) in a database protein, $L_i \leq d$ is the length of the ith segment of a database protein S^D.

Query protein structure S^Q, given by a user in a form of string pattern, is represented by ranges, which gives more flexibility in defining search criteria against proteins in a database:

$$S^Q = SSE_1^Q(L_1; U_1), SSE_2^Q(L_2; U_2), \ldots, SSE_m^Q(L_m; U_m), \tag{2.4}$$

where $SSE_j^Q \in \Sigma$ describes the type of secondary structure (as defined in Sect. 2.2.1), $L_j \leq U_j \leq q$ are lower and upper limits for the number of successive SSEs of the same type, q is the length of the query protein S^Q measured in residues, which is the maximal length of the string query pattern resulting from expanding the ranges of the pattern, m is the number of segments in the query pattern.

Additionally, the SSE_j^Q can be replaced by the wildcard symbol '?', which denotes any type of SSE from Σ, and the value of the U_j can be replaced by the wildcard symbol '*', which denotes $U_j = +\infty$.

The advantage of the used alignment method is that it finds local, optimal alignments with possible gaps between corresponding elements. A big drawback is that it is computationally costly, which negatively affects efficiency of the search process carried out against the whole database. The computational complexity of the original algorithm is $O(n * m(n + m))$ when allowing for gaps calculated in a traditional way. However, in the PSS-SQL we have modified the way how gap penalties are calculated, which results in better efficiency.

While aligning two protein structures S^Q and S^D, the search engine of the PSS-SQL calculates the similarity matrix D according to the following formulas.

$$D_{i,0} = 0 \quad \text{for} \quad i \in [0, q], \tag{2.5}$$

and

$$D_{0,j} = 0 \quad \text{for} \quad j \in [0, d], \tag{2.6}$$

and

$$D_{i,j} = max \begin{cases} 0 \\ D_{i-1,j-1} + d_{i,j} \\ E_{i,j} \\ F_{i,j} \end{cases}, \tag{2.7}$$

for $i \in [1, q]$, $j \in [1, d]$, where q, d are lengths of proteins S^Q and S^D, and $d_{i,j}$ is the matching degree between elements $SSE_i^D L_i$ and $SSE_j^Q(L_j; U_j)$ of both structures calculated using the following formula:

$$d_{i,j} = \begin{cases} \omega_+ & \text{if } \; SSE_i^D = SSE_j^Q \wedge L_i \geq L_j \wedge L_i \leq U_j \\ \omega_- & \text{otherwise} \end{cases}, \tag{2.8}$$

where ω_+ is the matching award, and ω_- is the mismatch penalty. If the element SSE_j^Q is equal to '?', then the matching procedure ignores the condition $SSE_i^D = SSE_j^Q$. Similarly, if we assign the '*' symbol for the U_j, the procedure ignores the condition $L_i \leq U_j$.

Auxiliary matrices E and F, called gap penalty matrices, allow to calculate horizontal and vertical gap penalties with the $O(1)$ computational complexity (as opposed to the original method, where it was possible with the $O(n)$ computational complexity for each direction). In the first version of the PSS-SQL, the calculation of the current element of the matrix D required an inspection of all previously calculated elements in the same row (for a horizontal gap) and all previously calculated elements in the same column (for a vertical gap). By using gap penalty matrices we need only to check one previous element in a row and one previous element in a column. Such an improvement gives a significant acceleration of the alignment method, and the acceleration is greater for longer sequences of SSEs and greater similarity matrices D. Elements of the gap penalty matrices E and F are calculated according to the following equations:

$$E_{i,j} = max \begin{cases} E_{i-1,j} - \delta \\ D_{i-1,j} - \sigma \end{cases}, \tag{2.9}$$

and

$$F_{i,j} = max \begin{cases} F_{i,j-1} - \delta \\ D_{i,j-1} - \sigma \end{cases}, \tag{2.10}$$

where σ is the penalty for opening a gap in the alignment, and δ is the penalty for extending the gap, and:

$$E_{i,0} = 0 \;\text{ for }\; i \in [0, q], \quad F_{i,0} = 0 \;\text{ for }\; i \in [0, q], \tag{2.11}$$

$$E_{0,j} = 0 \;\text{ for }\; j \in [0, d], \quad F_{0,j} = 0 \;\text{ for }\; j \in [0, d]. \tag{2.12}$$

The PSS-SQL uses the following values for matching award $\omega_+ = 4$, mismatch penalty $\omega_- = -1$, gap open penalty $\sigma = -1$, and gap extension penalty $\delta = -0.5$.

Filled similarity matrix D consists of many possible paths how two sequences of SSEs can be aligned. Backtracking from the highest scoring matrix cell and going along until a cell with score 0 is encountered allows to find the highest scoring alignment path. However, in the version of the alignment method that is implemented in the PSS-SQL, the search engine finds k-best alignments by searching consecutive

maxima in the similarity matrix D. This is necessary, since the pattern is usually not defined precisely, contains ranges of SSEs or undefined elements. Therefore, there can be many regions in a protein structure that fit the pattern. In the process of finding alternative alignment paths, the alignment method follows the value of the internal parameter MPE (minimum path end), which defines the stop criterion. The search engine finds alignment paths until the next maximum in the similarity matrix D is lower than the value of the MPE parameter. The value of the MPE depends on the specified pattern, according to the following formula.

$$MPE = (MPL \times \omega_+) + (NoIS \times \omega_-), \tag{2.13}$$

where MPL is the minimum pattern length, $NoIS$ is the number of imprecise segments, i.e., segments, for which $L_j \neq U_j$. For example, for the structural pattern $h(10;20)$, $e(1;10)$, $c(5)$, $e(5;20)$ containing α-helix of the length 10–20 elements, β-strand of the length 1–10 elements, loop of the length 5 elements, and β-strand of the length 5–20 elements, the $MPL = 21$ (10 elements of the type h, 1 element of the type e, 5 elements of the type c, and 5 elements of the type e), the $NoIS = 3$ (first, second, and fourth segment), and therefore, $MPE = 81$.

2.2.4 Multithreaded Implementation

In the original PSS-SQL [17], the calculation of the similarity matrix D was performed by a single thread. This negatively affected performance of PSS-SQL queries or, at least, this left a kind of computational reserve in the era of multicore CPUs. In the new version of the PSS-SQL we have reimplemented procedures and functions in order to use all processor cores that are available on the computer hosting the database with the PSS-SQL extension. A part of the work was carried out by B. Socha [22], my associate in this project.

However, the multithreaded implementation required different approach while calculating values of particular cells of the similarity matrix D. Successive cells cannot be calculated one by one, as in the original version, but calculations are carried out for cells located on successive diagonals, as it is shown in Fig. 2.5. This is because, according to Eqs. (2.7), (2.9), and (2.10) each cell $D_{i,j}$ can be calculated only if there are calculated cells $D_{i-1,j-1}$, $D_{i-1,j}$ and $D_{i,j-1}$. Such an approach to the calculation of the similarity matrix is called a *wavefront* [2, 14].

Moreover, in order to avoid too many synchronizations between running threads (which may lead to significant delays), the entire similarity matrix is divided to so-called areas (Fig. 2.6a). These areas are parts of the similarity matrix that have a smaller size $q' \times d'$. Assuming that the entire similarity matrix has the size of $q \times d$, where q and d are lengths of two compared sequences of SSEs, the number of areas that must be calculated is equal to:

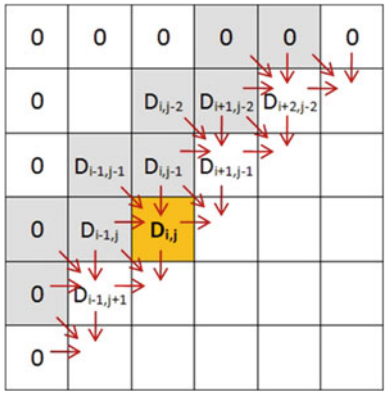

Fig. 2.5 Calculation of cells in the similarity matrix D by using the wavefront approach. Calculation is performed for cells at diagonals, since their values depend on previously calculated cells. *Arrows* show dependences of particular cells and the direction of value derivation

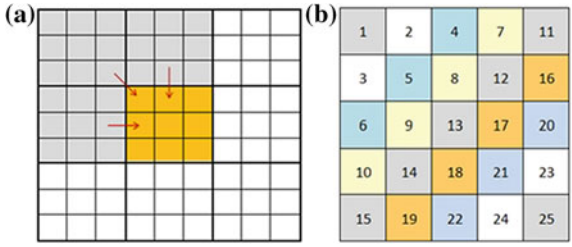

Fig. 2.6 Division of the similarity matrix D into areas (*left*)—arrows show mutual dependencies between areas during calculation of the matrix. (*right*) An order in which areas will be calculated in a sample similarity matrix

$$n_A = \left\lceil \frac{q}{q'} \right\rceil \times \left\lceil \frac{d}{d'} \right\rceil . \tag{2.14}$$

For example, for the matrix D of the size 382×108 and size of the area $q' = 10$ and $d' = 10$, the $n_A = \lceil \frac{382}{10} \rceil \times \lceil \frac{108}{10} \rceil = 39 \times 11 = 429$. Areas are assigned to threads working in the system. Each thread is assigned to one area, which is an atomic portion of calculation for the thread. Areas can be calculated according to the same wavefront paradigm. The area $A_{z,v}$ can be calculated, if there have been calculated areas $A_{z-1,v}$ and $A_{z,v-1}$ for $z > 0$ and $v > 0$, which implies an earlier calculation of the area $A_{z-1,v-1}$. The area $A_{0,0}$ is calculated as a first one, since there are no restrictions for calculation of the area.

In order to synchronize calculations, each area has a semaphore assigned to it. Semaphores guarantee that an area will not be calculated until the areas that it depends on have not been calculated. When all cells of an area have been calculated, the semaphore is being unlocked. Therefore, each area waits for unlocking two semaphores—for areas $A_{z-1,v}$ and $A_{z,v-1}$ for $z > 0$ and $v > 0$. While calculating an area each thread realizes the algorithm, which pseudocode is presented in Algorithm 1.

In Algorithm 1, after initialization of variables (lines 2–4), the thread enters the critical section marked with the *lock* keyword (line 5). Entering the critical section means that a thread obtains the mutual-exclusion lock for a given object. The thread executes some statements, and finally releases the lock. In our case, the thread obtains an exclusive access to the coordinates (z, v) of the area, which should be calculated by calling *GetAreaZ*() and *GetAreaV*() methods (lines 6–7). In the critical section, the thread also triggers the calculation of the (z, v) coordinates of the next area that should be calculated by another thread (line 8). Lines 9–11 determine whether this will be the last area that is calculated by any thread. Upon leaving the critical section, the current thread waits until areas $A_{z-1,v}$ and $A_{z,v-1}$ are unlocked (lines 13–14). Then, based on coordinates (z, v) and the area size in both dimensions, the thread determines absolute coordinates (i, j) of the first cell of the area (lines 15–16). These coordinates are used inside the following two *for* loops in order to establish absolute coordinates (i, j) of the current cell of the area. Figure 2.7 helps to interpret the variables used in the algorithm. The value of the current cell is calculated in line 21, according to formulas (2.5)–(2.7). When the thread completes the calculation of the current area, it unlocks the area (line 24) and asks for another area (lines 25–27).

Algorithm 1 The algorithm for the calculation of an area by a thread

```
1: procedure CALCULATEAREA
2:     z ← 0
3:     v ← 0
4:     boolFinish ← true
5:     lock                                          ▷ starts critical section
6:         z ← GetAreaZ()
7:         v ← GetAreaV()
8:         Calculate (z, v) coordinates of the next area
9:         if calculation successful (i.e., exists next area) then
10:            boolFinish ← false
11:        end if
12:    endlock
13:    Wait for unlocking the area A_{z-1,v}
14:    Wait for unlocking the area A_{z,v-1}
15:    absStart_i ← z * areaSizeZ
16:    absStart_j ← v * areaSizeV
17:    for rel_i ← 0 to areaSizeZ − 1 do
18:        for rel_j ← 0 to areaSizeV − 1 do
19:            i ← absStart_i + rel_i
20:            j ← absStart_j + rel_j
21:            Calculate cell D_{i,j} according to formulas 2.5-2.7
22:        end for
23:    end for
24:    Unlock area A_{z,v}
25:    if ¬boolFinish then
26:        Apply for the next area (enqueue for execution)
27:    end if
28: end procedure
```

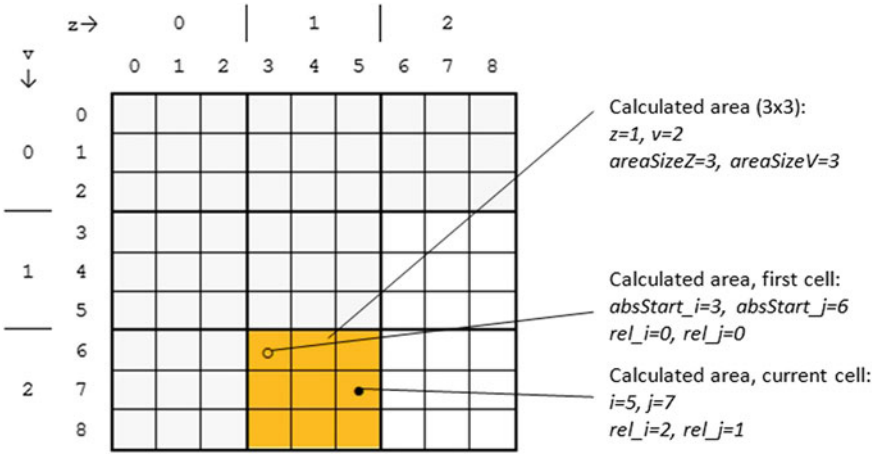

Fig. 2.7 Interpretation of variables used in the Algorithm 1 for the calculated area

The order in which areas are calculated is provided by a scheduling algorithm dispatching areas to threads. For example, the order of calculation particular areas in similarity matrix of the size 5×5 areas is shown in Fig. 2.6b. Such a division of the similarity matrix into areas reduces the number of tasks related to initialization of semaphores needed for synchronization purposes and reduces the synchronization time itself, which increases the efficiency of the alignment algorithm. For the PSS-SQL, the size of the area was set to 3×7 elements (3 for query protein, 7 for database protein) on the basis of experiments conducted by Socha [22].

2.3 SQL as the Interface Between User and the Database

PSS-SQL extends the standard syntax of the SQL language by providing additional functions that allow to search protein similarities on secondary structures. SQL language becomes a user interface (UI) between the user, who is a data consumer, and DBMS hosting secondary structures of proteins. PSS-SQL discloses three important functions for scanning PSSs: *containSequence*, *sequencePosition*, and *sequenceMatch*; all will be described in this chapter. PSS-SQL covers also a series of supplementary procedures and functions, which are used implicitly, e.g., for extracting segments of particular types of SSEs, building additional segment tables, indexing SSEs sequences, processing these sequences, aligning the target structures from a database to the query pattern, validating patterns, and many other operations. PSS-SQL extension was developed in the C# programming language. All procedures were assembled in the ProteinLibrary DLL file and registered for the Microsoft SQL Server 2008R2/2012 (Fig. 2.8).

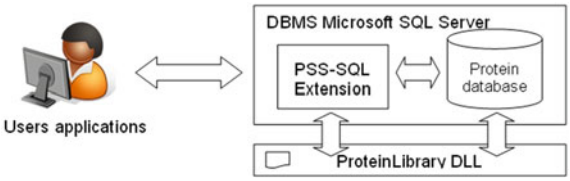

Fig. 2.8 General architecture of the system with the PSS-SQL extension. The PSS-SQL extension is registered in the Microsoft SQL Server DBMS. When the user submits a query invoking PSS-SQL functions (actually, Transact-SQL functions) the DBMS redirects the call to the PSS-SQL extension, which invokes appropriate functions assembled in the ProteinLibrary DLL library, passing appropriate parameters

2.3.1 Pattern Representation in PSS-SQL Queries

While searching protein similarities on secondary structures, we need to pass the query structure (query pattern) as a parameter of the search procedure. In PSS-SQL queries the pattern is represented as in the formula (2.4). Such a representation allows users to formulate a large number of various query types with different degrees of complexity. Moreover, we assumed that query patterns should be as simple as possible and should not cause any syntax difficulties. Therefore, we have defined the corresponding grammar in order to help constructing the query pattern.

In simple words, in PSS-SQL queries, the pattern is represented by blocks of segments. Each segment is determined by its type and length. The segment length can be represented precisely or as an interval. It is possible to define segments, for which the type is not important or undefined (wildcard symbol '?'), and for which the upper limit of the interval is not defined (wildcard symbol '*'). The grammar for defining patterns written in the Chomsky notation has the following form. The grammar is formally defined as the ordered quad-tuple:

$$G_{pss} = \langle N_{pss}, \Sigma_{pss}, P_{pss}, S_{pss} \rangle, \tag{2.15}$$

Σ_{pss} = {c, h, e, ?, *, $\mathbf{N}_+$}
N_{pss} = { <sequence>, <blocks_of_segments>, , <type>, <begin>, <end>, <length>, <integer_greater_than_zero_and_zero>, <undetermined> }
P_{pss} = {
<sequence> ::= <blocks_of_segments>
<blocks_of_segments> ::= | , <blocks_of_segments>
 ::= <type> (<begin>; <end>) | <type> (<length>)
<begin> ::= <integer_greater_than_zero_or_zero>
<end> ::= <integer_greater_than_zero_or_zero> | <undetermined>
<length> ::= <integer_greater_than_zero_or_zero>
<type> ::= c | h | e | ?
<integer_greater_than_zero_or_zero> ::= $\mathbf{N}_+$ | 0
<undetermined> ::= * }
S_{pss} = <sequence>

where the symbols respectively mean: N_{pss}—a finite set of nonterminal symbols, Σ_{pss}—a finite set of terminal symbols, P_{pss}—a finite set of production rules, S_{pss}—a distinguished symbol $S \in N_{pss}$ that is the start symbol.

Assumption: `<begin> ≤ <end>`

The following terms are compliant with the defined grammar G_{pss}:

- `h(1;10)`—representing an α-helix of the length 1–10 elements;
- `e(2;5),h(10;*),c(1;20)`—representing a β-strand of the length 2–5 elements, followed by an α-helix of the length at least 10 elements, and a loop of the length 1–20 elements;
- `e(10;15),?(5;20),h(35)`—representing a β-strand of the length 10–15 elements, followed by any element of the length 5–20, and an α-helix of the exact length 35 elements.

With such a representation of the query pattern, we can start the search process using one of the functions disclosed by PSS-SQL extension.

2.3.2 Sample Queries in PSS-SQL

The PSS-SQL extension provides a set of functions and procedures for processing PSSs. Three of the functions can be effectively invoked from the SQL commands, usually the SELECT statement.

The *containSequence* function verifies if a particular protein or a set of database proteins contain the structural pattern specified as a query pattern. This function returns the Boolean value 1 (true), if the database protein contains specified pattern, or 0 (false), if the protein does not include the pattern.

Sample invocation of the function is shown in Listing 2.1.

```
1 SELECT protID, protAC
2 FROM ProteinTbl
3 WHERE name LIKE '%Escherichia coli%' AND
4     dbo.containSequence(id, 'secondary', 'h(5;15),c(3),?(6),c(1;5)')=1
```

Listing 2.1 Sample query invoking *containSequence* function and returning identifiers of proteins from *Escherichia coli* containing the given secondary structure pattern.

The sample query returns identifiers and accession numbers of proteins from *Escherichia coli* having the structural region containing an α-helix of the length 5–15 elements, 3-element loop, any structure of the length 6 elements, and a loop of the length up to 5 elements (pattern `h(5;15),c(3),?(6),c(1;5)`).

Partial results of the query from Listing 2.1 are shown below.

```
protID       protAC
------------ --------
ACTP_ECOUT   Q1R3J9
ADD_ECOLC    B1IQD2
ADD_ECOLI    P22333
ADEC_ECO24   A7ZTM0
ADEC_ECO57   Q7A9L5
...
```

The *containSequence* function can be used in the SELECT and the WHERE phrase of the SQL SELECT statement. It is also possible to use the function in the WHERE clause of other DML statements, including UPDATE and DELETE, if needed. Detailed description of input arguments of the *containSequence* function is given in Table 2.1.

The *sequencePosition* and *sequenceMatch* functions allow to match the specified pattern to the structure of a protein or a group of database proteins. Pattern searching and matching is performed by multiple scanning of the segment index built on the segment table, followed by the alignment of the found segments. Both functions return a table containing information about the location of query pattern in the structure of each database protein. Both functions differ in the way how they are invoked in PSS-SQL queries.

Sample queries invoking both functions are shown in Listing 2.2. The function accepts the same arguments according to the list presented in Table 2.1. Since they return a table of values, they are nested in the FROM clause of SQL statements (mainly SELECTs, but also possible in some variants of UPDATE and DELETE statements). The use of the CROSS APPLY operator, instead of traditional JOIN, allows to avoid specifying the join condition, shortens the query syntax and, what even more important, improve performance, in the case of complex filtering conditions in the WHERE clause.

```
1  – invoking sequenceMatch and CROSS APPLY
2  SELECT p.protAC AS AC,p.name, s.startPos, s.endPos, p.[primary],
3      s.matchingSeq, p.secondary
4  FROM ProteinTbl AS p CROSS APPLY dbo.sequenceMatch(p.id, 'secondary',
5      'e(1;10),c(0;5),h(5;6),c(0;5),e(1;10),c(5)') AS s
6  WHERE p.name LIKE '%Staphylococcus aureus%' AND p.length > 150
7  ORDER BY AC, s.startPos
8
9  – invoking sequencePosition and standard JOIN
10 SELECT p.protAC AS AC, p.name, s.startPos, s.endPos, p.[primary],
11     s.matchingSeq,  p.secondary
12 FROM ProteinTbl AS p JOIN dbo.sequencePosition('secondary',
13     'e(1;10),c(0;5),h(5;6),c(0;5),e(1;10),c(5)',
14     'p.name LIKE ''%Staphylococcus aureus%'' AND p.length > 150') AS s
15     ON p.id=s.proteinId
16 ORDER BY AC, s.startPos
```

Listing 2.2 Sample query invoking *sequenceMatch* and *sequencePosition* table functions and returning information on proteins from *Staphylococcus aureus* having the length greater than 150 residues and containing the given secondary structure pattern.

These sample queries return Accession Numbers (AC) and names of proteins from *Staphylococcus aureus* having the length greater than 150 residues and structural region containing β-strand of the length from 1 to 10 elements, optional loop up to 5 elements, an α-helix of the length 5–6 elements, optional loop up to 5 elements, a β-strand of the length 1–10 elements and a 5 element loop—pattern `e(1;10),c(0;5),h(5;6),c(0;5),e(1;10),c(5).`

Partial results of the query from Listing 2.2 are shown in Fig. 2.9. Detailed description of the output fields of the *sequenceMatch* and *sequencePosition* functions is given in Table 2.2.

Table 2.1 Input arguments of PSS-SQL functions

Argument	Description
@proteinId[a]	Unique identifier of a protein in the database table that contains sequences of SSEs (e.g. *id* field in case of the *ProteinTbl*)
@columnSSeq	Database field containing sequences of SSEs of proteins (e.g. *secondary*)
@pattern	Query pattern represented by a set of segments, e.g., h(2;10), c(1;5),?(2;*)
@predicate[b]	An optional, simple, or complex filtering criteria that allow to limit the list of proteins that will be processed during the search, e.g.,: *length* < 150

[a]except *sequencePosition*
[b]only *sequencePosition*

AC	name	startPos	endPos	primary	matchingSeq	secondary
Q2FJ31	Alcohol dehydrogenase OS=S...	177	199	MRAAVVTKDHKVSIEDKK...	eeeehhhhhheeeeeeccc...	CCCEEEECCCCCCCCCCCC...
Q2FJ31	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCCCCCCCCC...
Q2G0G1	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCCCCCCCCC...
Q2YSX0	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCEEECCCCC...
Q5HI63	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCEEEHHHHH...
Q6GJ63	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCEEEHHHHH...
Q7A742	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCEEEHHHHH...
Q99W07	Alcohol dehydrogenase OS=S...	187	218	MRAAVVTKDHKVSIEDKK...	eeeeeeeccccchhhhhcc...	CCCEEEECCCCEEEHHHHH...

Fig. 2.9 Partial results of the sample queries from Listing 2.2 returned as a relational table, returned fields: *AC*—accession number, *name*—molecule name, *startPos*, *endPos*—position, where the pattern starts and ends in the target protein from a database, *primary*—amino acid sequence of the database protein, *matchingSeq*—exact sequence of SSEs, which matches to the pattern defined in the query, *secondary*—sequence of secondary structure elements SSEs of the database protein

Table 2.2 Output table of *sequenceMatch* and *sequencePosition* functions

Field	Description
proteinId	Unique identifier of the protein that contains the specified pattern
startPos	Position, where the pattern starts in the target protein from a database
endPos	Position, where the pattern ends in the target protein from a database
length	Length of the segment that matches to the given pattern
matchingSeq	Exact sequence of SSEs, which matches to the pattern defined in the query

Results of the PSS-SQL queries are originally returned in a tabular form. However, by adding an extra FOR XML clause at the end of the SELECT statement, like in the example in Listing 2.3, produces results in the XML format that can be easily transformed to the HTML web page by using appropriate XSLT transformation file, and finally, published in the Internet. Partial results of the query from Listing 2.3 are shown in Fig. 2.10. An additional function—*superimpose*—that was used in the presented query (Listing 2.3) visualizes the alignment of the matched sequence and the database sequence of SSEs.

Fig. 2.10 Partial results of the query from Listing 2.3

```
<proteins>
  <protein>
    <AC>Q2FJ31</AC>
    <name>Alcohol dehydrogenase OS=Staphylococcus
          aureus (strain USA300)...</name>
    <startPos>177</startPos>
    <endPos>199</endPos>
    <matchingSeq>eeeehhhhhheeeeeeeccccc</matchingSeq>
    <primary>MRAAVVTKDHKVSIEDKKLRALKPGEALVQTEYCGVCH
      TDLHVKNADFGDVTGVTLGHEGIGKVIEVAED... </primary>
    <alignment>CCCEEEECCCCCCCCCCCCCCCCCCCCEEEEECC...
      CCCCEEECCCCCCCCeeeehhhhhheeeeeeccccchhhhhh...
    </alignment>
  </protein>
  ...
</proteins>
```

```
1  SELECT p.protAC AS AC, p.name, s.startPos, s.endPos, s.matchingSeq, p .[primary],   dbo.superimpose
       (s.matchingSeq, p.secondary) AS alignment
2  FROM ProteinTbl AS p CROSS APPLY dbo.sequenceMatch(p.id, 'secondary',
3      'e(1;10),c(0;5),h(5;6),c(0;5),e(1;10),c(5)') AS s
4  WHERE p.name LIKE '%Staphylococcus aureus%'
5      AND p.length > 150
6  ORDER BY AC, s.startPos
7  FOR XML RAW ('protein'), ROOT('proteins'), ELEMENTS
```

Listing 2.3 Sample query invoking *sequenceMatch* table function and returning results as an XML document by using the FOR XML clause.

2.4 Efficiency of the PSS-SQL

The efficiency of the PSS-SQL query language was examined in various experiments. Tests were performed on the Microsoft SQL Server 2012 Enterprise Edition working on nodes of the virtualized cluster controlled by the HyperV hypervisor hosted on Microsoft Windows 2008 R2 Datacenter Edition 64-bit. The host server had the following parameters: 2x Intel Xeon CPU E5620 2.40 GHz, RAM 32 GB, 3x HDD 1TB 7200 RPM. Cluster nodes were configured to use 4 CPU cores and 4GB RAM per node, and worked under the Microsoft Windows 2008 R2 Enterprise Edition 64-bit operating system.

Most of the tests were performed on the database storing 6,360 protein structures. However, in order to compare our language to one of the competitive solutions, some tests were performed on the database storing 248,375 protein structures.

During the experiments, we measured execution times for various query patterns. The query patterns were passed as a parameter of the *sequencePosition* function. Tests were performed for queries containing the following sample patterns:

- SSE1: `e(4;20),c(3;10),e(4;20),c(3;10),e(15),c(3;10),e(1;10)`
- SSE2: `h(30;40),c(1;5),?(50;60),c(5;10),h(29),c(1;5),h(20;25)`
- SSE3: `h(10;20),c(1;10),h(243),c(1;10),h(5;10),c(1;10),h(10;15)`
- SSE4: `e(1;10),c(1;5),e(27),h(1;10),e(1;10),c(1;10),e(5;20)`
- SSE5: `e(5;20),h(2;5),c(2;40),?(1;30),e(5;*)`

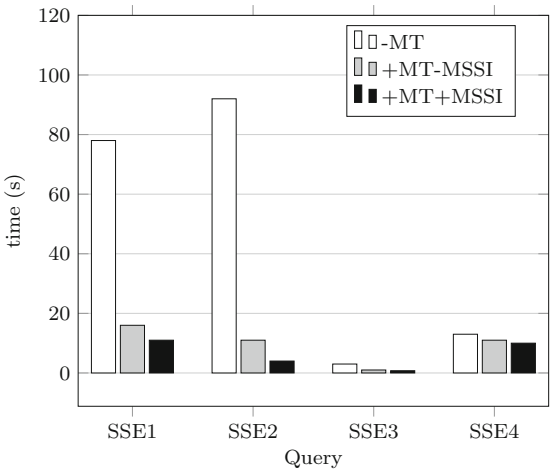

Fig. 2.11 Execution time for various query patterns SSE1–SSE4 and for three variants of the PSS-SQL language: without multithreading (−MT), with multithreading, but without multiple scanning of the Segment Index (+MT−MSSI), with multithreading and with multiple scanning of the Segment Index (+MT+MSSI)

Pattern SSE1 represents protein structure built only with β-strands connected by loops. Pattern SSE2 consists of several α-helices connected by loops and one undefined segment of SSEs ('?' wildcard symbol). Patterns SSE3 and SSE4 have regions that are unique in the database, i.e., `h(243)` in pattern SSE3 and `e(27)` in pattern SSE4. Pattern SSE5 has a wildcard symbol '*' for undetermined length, which slows down the search process.

In order to verify the influence of particular acceleration techniques on the execution times, tests were carried out for the PSS-SQL in three variants:

- without multithreading (−MT),
- with multithreading, but without MSSI (+MT–MSSI), and
- with multithreading and with MSSI (+MT+MSSI).

Results of the tests shown in Fig. 2.11 prove that the performance of +MT–MSSI variant is higher, and in case of SSE1 and SSE2 even much higher, than −MT variant (implemented in original PSS-SQL). For +MT+MSSI we can see additional improvement of the performance. It is difficult to estimate the overall acceleration, because it tightly depends on the uniqueness of the pattern. The more unique the pattern is, the more proteins are filtered out based on the Segment Index, the fewer proteins are aligned and the less time we need to obtain results. We can see it clearly in Fig. 2.11 for patterns SSE3 and SSE4 that have precisely defined, unique regions `h(243)` and `e(27)`. For universal patterns, like SSE1 and SSE2, for which we can find many fitting proteins or multiple alignments, we can observe longer execution times. In such cases, the parallelization and MSSI start playing a more significant role. In these cases, the length of the pattern influences the alignment time—for longer patterns we experienced longer response times. We have not observed any dependency between the type of the SSE and the response time.

However, specifying wildcards in the query pattern increases the waiting period, which is visible for the pattern SSE5 (Fig. 2.12). In Fig. 2.12 for the pattern SSE5, we

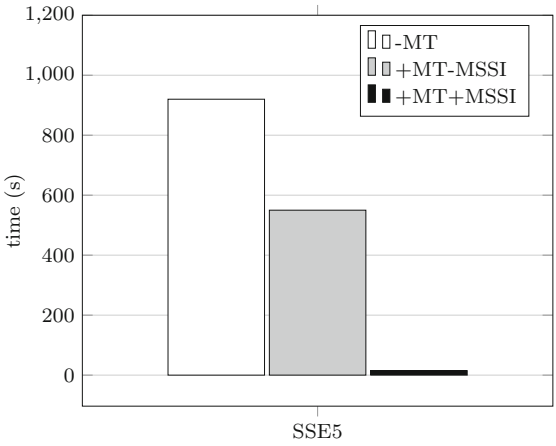

Fig. 2.12 Execution time for query pattern SSE5 for three variants of the PSS-SQL language: without multithreading (−MT), with multithreading, but without multiple scanning of the Segment Index (+MT−MSSI), with multithreading and with multiple scanning of the Segment Index (+MT+MSSI)

can also see how beneficial the use of the MSSI technique can be. In this particular case, the execution time was reduced from 920 s in −MT (original PSS-SQL), and 550 s in +MT−MSSI, to 15 s in +MT+MSSI, which gives 61.33-times acceleration over the −MT variant and 36.67-times acceleration over the +MT−MSSI variant.

2.5 Discussion

PSS-SQL language complements existing relational DBMSs, which are not designed to process biological data, such as PSSs stored as sequences of SSEs. By extending the standard SELECT, UPDATE, and DELETE statements of the SQL language, it provides a declarative method for retrieving, modifying, and deleting records. Records that satisfy the criteria given by a user can be returned in a table-like form or as an XML document, which is easy to display as a Web page. In such a way, the PSS-SQL extension to relational database management systems (RDBMS) provides a kind of domain-specific language for processing PSSs. This is especially important for relational database designers, wide group of biological data analysts, and bioinformaticians.

The PSS-SQL language can be used for the fast classification of proteins based on their secondary structures. For example, systems such as SCOP [18] and CATH [19] make use of the secondary structure description of protein structures in order to classify proteins into classes and families. PSS-SQL can be also supportive in protein 3D structure prediction by homology modeling, where appropriate structure profile can be found based on primary and secondary structure and the secondary structure can be superimposed on the protein of the unknown 3D structure before performing a free energy minimization.

Comparing the PSS-SQL to other languages presented in Sect. 2.1, we can notice that all variants of the PSS-SQL extend the syntax of the SQL. This makes the PSS-SQL similar to PiQL [24], rather than to ProteinQL [25]. ProteinQL was developed for the OODB and relies on its own domain-specific database and dedicated ProteinQL interpreter and translator. As opposed to ProteinQL, both PiQL, and PSS-SQL extend capabilities of RDBMS. They extend the syntax of the SQL language by providing additional functions that can be nested in particular clauses of the SQL commands. However, the form of queries provided by users is different. PiQL accepts query patterns in a full form, like in BLAST [1]—a tool used for fast local matching of biomolecular sequences of DNA and proteins. Query patterns provided in PSS-SQL are similar to those presented by Hammel and Patel in [9]. The pattern defined in a query does not have to be specified strictly. Segments in the pattern can be specified as intervals and they can have undefined lengths. Both languages allow specifying query patterns with undefined types of the SSE or patterns, where some SSE segments may occur optionally. Therefore, the search process has an approximate character, regarding various possible options for segment matching. The possibility of defining patterns that include optional segments allows users to specify gaps in a particular place.

The described version of the PSS-SQL also uses the method of scanning the Segment Index in order to accelerate the search process. The method was adopted from the work of Hammel and Patel [9]. However, after multiple scans of the Segment Index Hammel and Patel used sort-merge join operations in order to join segments from the same candidate proteins and decide, whether they meet specified query conditions or not. The novelty of PSS-SQL is that it relies on the alignment of the found segments. Alignment implemented in PSS-SQL gives the unique possibility of finding many matches for the same database protein and returning k-best matches, matches that in some particular cases can be separated by gaps. These are not the gaps defined by a user and specified by an optional segment, but the gaps providing better alignment of particular regions. This type of matching is typical for similarity searching between biomolecular sequences, such as DNA/RNA sequences or amino acid sequences. Presented approach extends the spectrum of searching and guarantees the optimality of the results according to assumed scoring system.

Despite the fact that PSS-SQL uses the alignment procedure, which is computationally complex, it gained quite a good performance. We have compared the efficiency of the PSS-SQL (+MT+MSSI variant) and language presented by Hammel and Patel for single-predicate exact match queries with various selectivity (between 0.3 and 6 %) using the database storing 248,375 proteins (515 MB for *ProteinTbl*, 254 MB for segment table storing 11,986,962 segments). The PSS-SQL was on average 5.14 faster than *Comm-Seg* implementation, 3.28 faster than *Comm-CSP* implementation, both implemented on a commercial ORDBMS, and 1.84 faster than *ISS-MISS*(1) implementation on Periscope/SQ. This proves, that PSS-SQL compensates the efficiency loss caused by alignment procedure by using the Segment Index. In such a way, the PSS-SQL joins wide capabilities of the alignment process (possible gaps, mismatches, and many solutions), provides optimality and quality of results, and guarantees efficiency of scanning databases of secondary structures.

2.6 Summary

Integrating methods of PSS similarity searching with DBMSs provides an easy way for manipulation of biological data without the necessity of using external data mining applications. The PSS-SQL extension presented in this chapter is a successful example of such integration. PSS-SQL is certainly a good option for biological and biomedical data analysts who want to process their data on the server side. This has many advantages that are typical for such a processing in the client-server architecture. Entire logic of data processing is performed on the database server, which reduces the load on the user's computer. Therefore, data exploration is performed while retrieving data from a database. Moreover, the number of data returned to the user, and the network traffic between the server and the user application, are much reduced.

The use of multithreading allows to utilize the whole capable computing power more efficiently. The PSS-SQL adapts to the number of processing units possessed by the server hosting the DBMS and to the number of cores used by the database system. This results in better performance of the language while scanning huge databases of PSSs. For the latest information on the PSS-SQL, please visit the project home page: http://zti.polsl.pl/dmrozek/science/pss-sql.htm.

Parallelization of calculations in bioinformatics brings tangible benefits and reduces the execution time of many algorithms. In this chapter, we could see one of many examples of such parallelization. For readers that are interested in other examples I recommend the book *Parallel Computing for Bioinformatics and Computational Biology* by Zomaya [29] for further reading. In the next chapter, we will see how a massive parallelization of the 3D structure similarity searching on many-core CUDA-enabled GPU devices leads to reduction of the execution time of the process.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. **215**, 403–410 (1990)
2. Anvik, J., MacDonald, S., Szafron, D., Schaeffer, J., Bromling, S., Tan, K.: Generating parallel programs from the wavefront design pattern. In: Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02), Fort Lauderdale, Florida, April 2002, pp. 1–8 (2002)
3. Apweiler, R., Bairoch, A., Wu, C.H., et al.: Uniprot: the universal protein knowledgebase. Nucl. Acids Res. **32**(Database issue), D115–D119 (2004)
4. Berman, H., et al.: The Protein Data Bank. Nucl. Acids Res. **28**, 235–242 (2000)
5. Can, T., Wang, Y.: CTSS: a robust and efficient method for protein structure alignment based on local geometrical and biological features. In: Proceedings of the 2003 IEEE Bioinformatics Conference (CSB 2003), pp. 169–179 (2003)
6. Date, C.: An Introduction to Database Systems, 8th edn. Addison-Wesley, Reading (2003)
7. Frishman, D., Argos, P.: Incorporation of non-local interactions in protein secondary structure prediction from the amino acid sequence. Protein Eng. **9**(2), 133–142 (1996)
8. Gibrat, J., Madej, T., Bryant, S.: Surprising similarities in structure comparison. Curr. Opin. Struct. Biol. **6**(3), 377–385 (1996)

9. Hammel, L., Patel, J.M.: Searching on the secondary structure of protein sequences. In: Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, 2002, pp. 634–645 (2002)
10. Joosten, R.P., Te Beek, T.A.H., Krieger, E., Hekkelman, M.L., et al.: A series of PDB related databases for everyday needs. Nucl. Acid Res. **39**(Database issue), D411–D419 (2011)
11. Kabsch, W., Sander, C.: Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. Biopolymers **22**, 2577–2637 (1983)
12. Källberg, M., Wang, H., Wang, S., Peng, J., Wang, Z., Lu, H., Xu, J.: Template-based protein structure modeling using the RaptorX web server. Nat. Protoc. **7**, 1511–1522 (2012)
13. Kessel, A., Ben-Tal, N.: Introduction to Proteins: Structure, Function, and Motion, 1st edn. CRC Press, Boca Raton (2010)
14. Liu, W., Schmidt, B.: Parallel design pattern for computational biology and scientific computing applications. In: Proceedings of the 2003 IEEE International Conference on Cluster Computing, pp. 456–459 (2003)
15. Małysiak-Mrozek, B., Kozielski, S., Mrozek, D.: Server-side query language for protein structure similarity searching. In: Human-Computer Systems Interaction: Backgrounds and Applications. Springer, Berlin, Advances in Intelligent and Soft Computing **99**(2), 395–415 (2012)
16. Mrozek, D., Małysiak-Mrozek, B.: CASSERT: a two-phase alignment algorithm for matching 3D structures of proteins. In: Kwiecień, A., Gaj, P., Stera, P. (eds.) Proceedings of 22nd International Conference on Computer Networks, Communications in Computer and Information, Springer-Verlag, CCIS **370**, 334–343 (2013)
17. Mrozek, D., Wieczorek, D., Małysiak-Mrozek, B., Kozielski, S.: PSS-SQL: protein secondary structure—structured query language. In: Proceedings of 32nd Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS 2010, Buenos Aires, Argentina, pp. 1073–1076 (2010)
18. Murzin, A.G., Brenner, S.E., Hubbard, T., Chothia, C.: SCOP: a structural classification of proteins database for the investigation of sequences and structures. J. Mol. Biol. **247**, 536–540 (1995)
19. Orengo, C.A., Michie, A.D., Jones, S., Jones, D.T., et al.: CATH—a hierarchic classification of protein domain structures. Structure **5**(8), 1093–1108 (1997)
20. Shapiro, J., Brutlag, D.: FoldMiner and LOCK2: protein structure comparison and motif discovery on the web. Nucl. Acids Res. **32**, 536–541 (2004)
21. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. 147, 195–197 (1981)
22. Socha, B.: Multithreaded execution of the Smith-Waterman algorithm in the query language for protein secondary structures. MSc thesis, supervised by Mrozek D., Silesian University of Technology, Gliwice, Poland (2013)
23. Stephens, S., Chen, J.Y., Thomas, Sh.: ODM BLAST: sequence homology search in the RDBMS. In: Bulletin of the IEEE Computer Society Technical Committee on Data Engineering (2004)
24. Tata, S., Patel, J.M., Friedman, J.S., Swaroop, A.: Declarative querying for biological sequences. In: Proceedings of 22nd International Conference on Data Engineering, IEEE Computer Society, 2006, pp. 87–98 (2006)
25. Wang, Y., Sunderraman, R., Tian, H.: A domain specific data management architecture for protein structure data. In: Proceedings of 28th IEEE EMBS Annual International Conference, New York City, USA, pp. 5751–5754 (2006)
26. Wieczorek, D., Małysiak-Mrozek, B., Kozielski, S., Mrozek, D.: A metod for matching sequences of protein secondary structures. J. Med. Info. Technol. **16**, 133–137 (2010)

27. Wieczorek, D., Małysiak-Mrozek, B., Kozielski, S., Mrozek, D.: A declarative query language for protein secondary structures. J. Med. Info. Technol. **16**, 139–148 (2010)
28. Yang, Y., Faraggi, E., Zhao, H., Zhou, Y.: Improving protein fold recognition and template-based modeling by employing probabilistic-based matching between predicted one-dimensional structural properties of the query and corresponding native properties of templates. Bioinformatics **27**, 2076–2082 (2011)
29. Zomaya, A.Y.: Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies, 1st edn. Wiley-Interscience, New York (2006)

http://www.springer.com/978-3-319-06970-8

High-Performance Computational Solutions in Protein Bioinformatics

Mrozek, D.

2014, XIX, 109 p. 56 illus., 21 illus. in color., Softcover
ISBN: 978-3-319-06970-8