**Supplementary materials**

# PatternQuery: Web application for fast detection of biomacromolecular structural patterns in the entire Protein Data Bank

David Sehnal[1,2,3], Lukáš Pravda[1,2], Radka Svobodová Vařeková[1,2], Crina-Maria Ionescu[1], Jaroslav Koča[1,2,*]

[1]CEITEC - Central European Institute of Technology, Masaryk University Brno, Kamenice 5, 625 00 Brno, Czech Republic
[2]National Centre for Biomolecular Research, Faculty of Science, Masaryk University, Kotlářská 2, 611 37 Brno, Czech Republic
[3]Faculty of Informatics, Masaryk University Brno, Botanická 68a, 602 00 Brno, Czech Republic

* To whom correspondence should be addressed. JK: Tel: +420 54949 4947; Fax: +420 54949 2556; Email: Jaroslav.Koca@ceitec.muni.cz

The authors wish it to be known that, in their opinion, the first 2 authors should be regarded as joint First Authors.

**Web address: http://ncbr.muni.cz/PatternQuery**

# Theoretical Background of the PatternQuery Language

## Basic Definitions

### Chemical Sets

In the following definitions we will use different common sets that can be used to describe the chemical properties of molecules. For examples the set **ChemicalElements** $= \{C, N, O, H, \dots\}$, **Bonds** $= \{\text{single}, \text{double}, \dots\}$, or **ResidueNames** $= \{\text{CYS}, \text{HIS}, \dots\}$ (corresponds to the names of amino acids, ligands, etc.).

### Biomolecular Graph

We need to express a molecule including its various properties, like spatial positions of atoms, information about residues in biomolecules, etc. For this purpose, we define the structure called *biomolecular graph*. The structure describes the topology of the molecule as well as its geometry (x, y, and z coordinates of each atom) and its special biomolecular properties (denotation of residues and chains, etc.).

**Definition 1.** *A biomolecular graph is a structure* $(G, P)$ *where*

**G** $= (\text{Atoms}, \text{Bonds})$ *is an undirected graph.* Atoms *is the set of the graph's vertices,* Bonds *is the set of edges.*

**P** $= \{P_i\}$ *is a set of mappings that varies depending on context. Most often, these properties are of interest:*

**atomId** : $\text{Atoms} \to \mathbb{N}$ *Atom identifiers.*

**atomType** : $\text{Atoms} \to \text{ChemicalElements}$ *Describes chemical elements of atoms.*

**bondType** : $\text{Bonds} \to \text{BondTypes}$ *Describes type of the bond.*

**atomPosition** : $\text{Atoms} \to \mathbb{R}^3$ *Maps atom to a position in 3D space.*

**residueId** : $\text{Atoms} \to \mathbb{N}$ *Maps atom to a residue identifier.*

**residueName** : $\text{Atoms} \to \text{ResidueNames}$ *Maps atom to a name of a residue.*

**chain** : $\text{Atoms} \to \text{ChainName}$ *Maps atom to a chain.*

**charge** : $\text{Atoms} \to \mathbb{R}$ *Maps atom to a partial charge (real number).*

*The set of all biomolecular graphs is denoted as* ***BioMolecularGraphs****.*

**Definition 2.** *We say that two biomolecular graphs* $A, B \in$ ***BioMolecularGraphs*** *are isomorphic, with respect to a set of properties* $P = \{P_i : X_i \to Y_i\}$*, denoted* $A \approx_P B$*, if these conditions hold:*

- $G^A \approx G^B$*, i.e. their underlying graphs are isomorphic.*

- $P_i^A(x) \equiv P_i^B(h(x))$ *for each* $x \in X_i$ *and each isomorphism* $h$ *between graphs* $G^A, G^B$*, i.e., the properties are equivalent under all isomorphisms.*

*We say two biomolecular graphs are equal, if they are isomorphic with regards to all defined properties.*

We are particularly interested in analyzing subgraphs of biomolecular graphs which we call *molecular pattern graphs* (or just pattern graphs). Molecular patterns are uniquely determined by a subset of atoms which induces a subgraph in the biomolecular graph.

**Definition 3.** *A molecular pattern graph is defined as* $(G, F)$ *where*

$\mathbf{G} \in \mathbf{BioMolecularGraphs}$ *is a biomolecular graph.*

$\mathbf{F} \subset \mathrm{Atoms}$ *is a subset of biomolecular graph atoms.*

*The set of all molecular pattern graphs is denoted* ***BioMolecularPatterns****. The set of all molecular pattern graphs of a biomolecular graph* $G \in$ ***BioMolecularGraphs*** *is denoted* ***BioMolecularPatterns***$(G) = \{(G, F) \mid F \subset \mathrm{Atoms}_G\}$*. For a pattern* $N = (G, F) \in$ ***BioMolecularPatterns*** *we use the notation* $G_i$ *to denote the* $i$*-th atom in* $F$*.*

**Definition 4.** *Isomorphism/equality is defined for pattern graph similarly to the the biomolecular graph case, with the exception of the relations being restricted to the set of the pattern atoms.*

**Definition 5.** *We define a mapping* $\mathrm{toGraph} :$ ***BioMolecularPatterns*** $\to$ ***BioMolecularGraphs*** *which maps a pattern graph to a biomolecular graph, by restricting the vertices, edges, and property mappings to the pattern atom set:*

$$\mathrm{toGraph} : ((G, P), F) \mapsto (G|_F, \{P_i|_F\})$$

We define two basic operations, one function, and one relation on pattern graphs:

**Definition 6.** *For a biomolecular graph* $G \in$ ***BioMolecularGraphs*** *we define functions* $\mathrm{union}_G$, $\mathrm{intersection}_G$, $\mathrm{distance}_G$, $\mathrm{areConnected}_G$ *using* ***BMP*** *as a shorthand for* ***BioMolecularPatterns*** *as*

$\mathbf{union}_G : \mathbf{BMP}(G) \times \mathbf{BMP}(G) \to \mathbf{BMP}(G),\ ((G, F_1), (G, F_2)) \mapsto (G, F_1 \cup F_2)$

$\mathbf{intersection}_G : \mathbf{BMP}(G) \times \mathbf{BMP}(G) \to \mathbf{BMP}(G),\ ((G, F_1), (G, F_2)) \mapsto (G, F_1 \cap F_2)$

$\mathbf{distance}_G : \mathbf{BMP}(G) \times \mathbf{BMP}(G) \to \mathbb{R},$
$((G, F_1), (G, F_2)) \mapsto \min\{\|\mathbf{atomPosition}(a) - \mathbf{atomPosition}(b)\| \mid a \in F_1 \wedge b \in F_2\}$
*(minimum distance of any pair of atoms from different patterns).*

$\mathbf{areConnected}_G$ *is a relation on* ***BMP***$(G)$ *with all pairs of patterns that are connected by an edge.*

## Describing and Recognizing Structural Patterns

We introduce a (domain specific) language that is defined in this chapter. We will use syntax of the Haskell programming language to illustrate the basic concepts as it allows a straightforward definition of the basic concepts. However, the presented approach is general and easy to implement in almost any modern programming language.

## Building Blocks of the Language

This section provides a general overview and explains basic principles of the language.

**Definition 7.** *A pattern query is a function that maps a biomolecular graph to a set of patterns:* query : $\textbf{\textit{BioMolecularGraphs}} \to 2^{\textbf{\textit{BioMolecularPatterns}}}$.

We define the data types based on the abstract structures **BioMolecularGraphs** (definition 1) and **BioMolecularPatterns** (definition 3). In order to make the code shorter and more readable, we will call these types `Molecule` and `Pattern`. These types can be defined as

```
{- Atom is identified by an integer number.
   Its properties are stored in the Molecule type. -}
data Atom = Atom Integer
data Bond = Bond (Atom, Atom) {- Bond is a tuple of atoms -}
data Molecule = Molecule (
  Set Atom,
  Set Bond,
  <properties>)    {- e.g. atom types, positions, etc. -}
data Pattern = Pattern (Molecule, Set Atom)
```

In the definitions we use the data type `Set` to represent sets. Basic operations on sets are called using the *dot* notation, for example *Set.union* or *Set.singleton* (creates a set with a single element).

The properties of atoms/bonds can be defined as mappings from a `Molecule` and `Atom`/`Bond`. For example, the `atomType` property is defined as

```
atomType :: Molecule -> Atom -> String
atomType (Molecule (_,_,...,atomTypes,...)) a = atomTypes a
```

The equality of types `Molecule` and `Pattern` is defined in the sense of definitions 2 and 4 with respect to properties `atomId` and `atomType`.

The operations on the type `Pattern` from definition 6 also need to be defined. The definition/implementation of these functions is rather straightforward, for example `union` on patterns would be defined as:

```
union :: Pattern -> Pattern -> Pattern
union (Pattern (m,a)) (Pattern (m,b)) = Pattern (m, Set.union a b)
```

The data type corresponding to the definition 7 of a query can then by defined as

```
type Query = Molecule -> Set Pattern
```

**Example 1.** *A query that returns all atoms as patterns is defined as*

```
atomsQuery :: Query
atomsQuery = \mol ->
  map (\a -> Pattern (mol, Set.singleton a)) (atoms mol)
```

*The query is represented as a lambda expression. In the query, each atom is mapped to a singleton set and then coupled with the context molecule.*

The execution of the query is then a simple application of the query function on the input molecular graph:

```
execute :: Query -> Molecule -> Set Pattern
execute query molecule = query molecule
```

We recognize three basic categories of queries:

**Generator queries** Queries that create sets of patterns from molecular graphs.

**Modifier queries** Queries that modify sets of patterns.

**Combinator queries** Queries that combine one or more sets of patterns.

These categories will now be described in more detail.

### Generator Queries

The most basic building block of the language is a generator query. It provides a way of describing basic patterns within a molecular graph, such as atoms or residues.

The general shape of a generator query can be defined as

```
generatorQuery :: (Molecule -> [Pattern]) -> Query
generatorQuery generator =
  \mol -> Set.fromList (
    map (\atoms -> Pattern (mol, atoms)) (generator mol))
```

which splits the input molecular graph into a list of patterns defined by the `generator` argument.

A specific example of a generator query is the `atomsQuery` which returns patterns corresponding to all atoms of a specific type:

```
atomsQuery :: [ElementType] -> Query
atomsQuery elements = generatorQuery (\mol ->
  ( map (\a -> Pattern (mol, Set.singleton a))
  . filter (\a -> elem (atomType mol a) elements))
    {- elem checks if an element is a member of a list -}
  (atoms mol))
```

This query filters the atoms of the input structure by checking if they have the matching element type. Next, each matching atoms is converted to a pattern containing a single atom.

### Modifier Queries

The modifier queries work by mapping each pattern from the result of an *inner query* to a set of patterns. The general template of a modifier query can be defined as

```
modifierQuery :: (Pattern -> Set Pattern) -> Query -> Query
modifierQuery modifier innerQuery = \mol ->
  Set.unions (map modifier (execute innerQuery mol))
```

The way this works is:

- The `innerQuery` is executed.
- The modifier function is then mapped onto each pattern in the set from the previous step. This results in a set of sets of patterns.
- Union is computed on the result of the previous step, resulting in a set of patterns.

An example of a modifier query is the `ambientAtomsQuery` which extends the inner pattern by all atoms within a given radius:

```
ambientAtomsQuery :: Float -> Query -> Query
ambientAtomsQuery radius =
  modifierQuery (\Pattern (g, m) -> Pattern
    (g, m >>= \a -> filter (\b -> distance a b <= radius) (atoms g)))
```

What this does is that for each atom in each pattern, it identifies all atoms within a given radius and then adds them to a single set.

A somewhat special (and a very useful) case of the modifier query is the `filterQuery`:

```
filterQuery :: (Pattern -> Bool) -> Query -> Query
filterQuery filterFunction = modifierQuery (\m ->
  if filterFunction m
    then Set.singleton m
    else Set.empty)
```

Here, the modifier function "tests" the pattern. If the condition is satisfied, a singleton with this single pattern is returned.

Another special kind of a modifier query is the `insideQuery`, that executes a particular query on results from another and combines them into a single query:

```
insideQuery :: Query -> Query -> Query
ambientAtomsQuery what within = \ctx ->
  foldr Set.union Set.empty
    (map (execute ctx . toMolecule) (execute ctx within))
```

## Combinator Queries

The combinator query, as the name implies, combines multiple queries into one. The general template of a modifier query can be defined as

```
combinatorQuery ::
  (Molecule -> [Pattern] -> Maybe Pattern) -> [Query] -> Query
combinatorQuery combinator queries = \mol ->
  (Set.fromList
   . catMaybes {- discard the Nothing results -}
   . map (combinator mol)
   . sequence
   . map (\q -> Set.toList (execute q mol)))
  queries
```

The way this works is the following:

- Each query in `queries` list is executed. The result is converted to list so that the types match.
- Next, the `sequence` function yields all combinations of the queries from the previous step. For example, if the result from step 1 was `[[a, b], [c, d]]`, the function `sequence` returns `[[a, c], [a, d], [b, c], [b, d]]`.
- The function `combinator` is provided with the current context and is applied to each combination from the previous step. The combinator function might fail to create a combination from the input patterns. Therefore, the function returns either a pattern or *nothing*, represented by the `Maybe` type.
- The *nothing* results from the previous step are discarded and the result is converted to a set.

A straightforward example of a combinator query is `combinationsQuery` that yields patterns created as all possible combinations of the underlying patterns and uses the set union as its combinator function:

```
combine :: Molecule -> [Pattern] -> Maybe Pattern
combine mol patterns =
  Just (Pattern (mol, foldr1 (\a b -> union a b) patterns))

combinationsQuery :: Query
combinationsQuery = combinatorQuery combine
```

A more complicated example of a combinator query is the `clusterQuery`, which is an extension of the `combinationsQuery`. In `clusterQuery`, the combinator function first checks if all pairs of patterns are within a given distance of each other and only then combines the patterns into a single one:

```
clusterQuery :: Float -> Query
clusterQuery maxDistance = combinatorQuery (\mol patterns ->
  let
    dist = max [distance a b | a <- patterns, b <- patterns, a /= b]
  in
    if dist > maxDistance then Nothing
    else combine mol patterns)
```

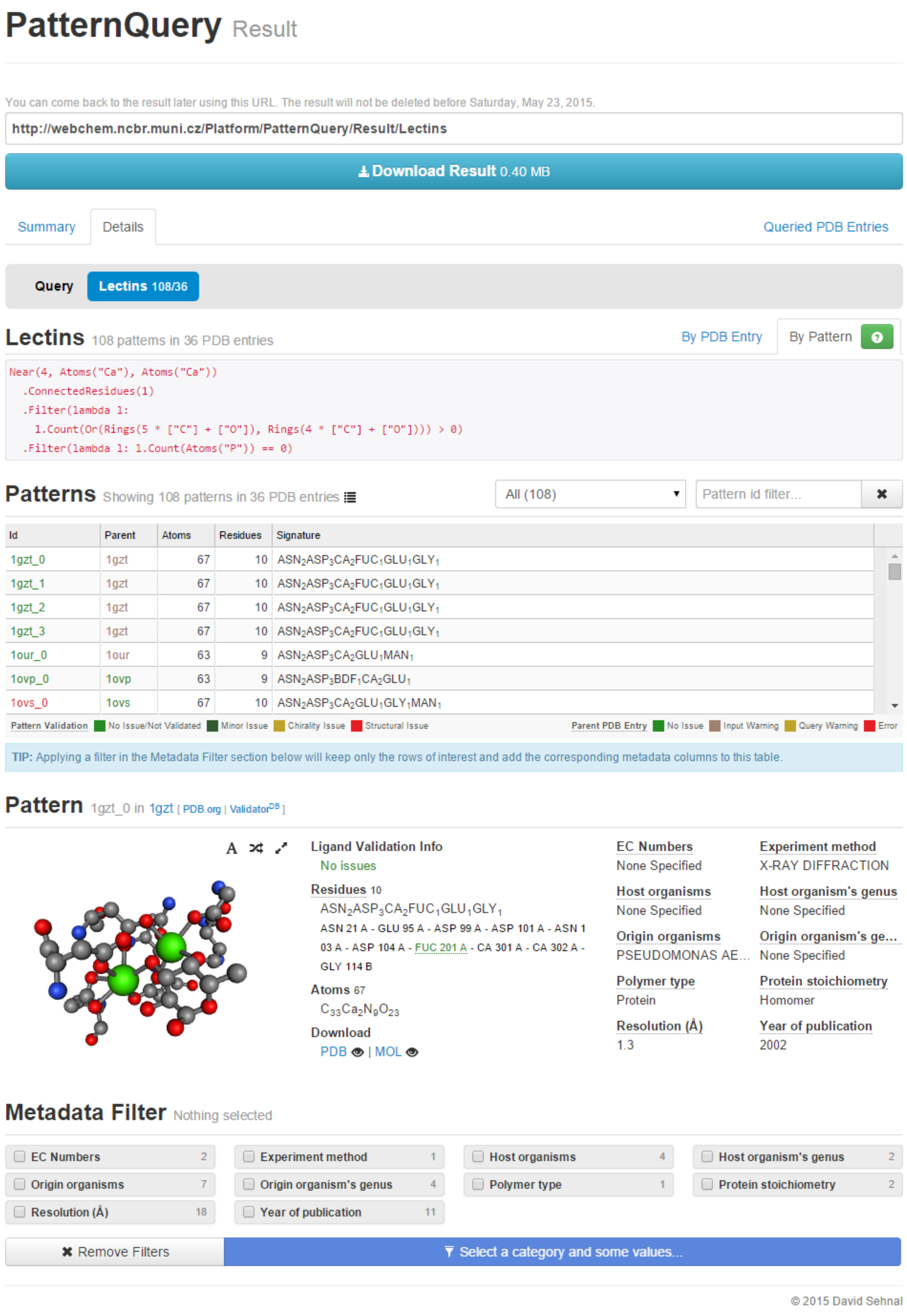
## Notes on Complexity

By the query complexity we mean the time required to execute a query on a biomolecular graph with $N$ vertices. The typical complexities of queries from different categories are given here.

**Generator queries** These queries have typically $O(N)$ (linear) complexity as they mostly consists of checking element symbols of atoms, comparing names of residues, etc.

**Modifier queries** The typical complexity of this type of query is $O(N \log N)$. The query `ambientAtomsQuery` has this complexity if implemented using a space-partitioning data structure such as kD-trees or octree.

**Combinator queries** The combinator queries are the most complicated to execute. Their complexity can be exponential—for example in the case of the `combinationsQuery`. Fortunately, some of the useful queries, such as `clusterQuery`, can be optimized using more sophisticated data structures such as kD-trees to have the complexity $O(N \log N)$ in most practical cases.



# PatternQuery Result

You can come back to the result later using this URL. The result will not be deleted before Saturday, May 23, 2015.

http://webchem.ncbr.muni.cz/Platform/PatternQuery/Result/Lectins

## Lectins 108 patterns in 36 PDB entries

```
Near(4, Atoms("Ca"), Atoms("Ca"))
  .ConnectedResidues(1)
  .Filter(lambda l:
    l.Count(Or(Rings(5 * ["C"] + ["O"]), Rings(4 * ["C"] + ["O"]))) > 0)
  .Filter(lambda l: l.Count(Atoms("P")) == 0)
```

## Patterns Showing 108 patterns in 36 PDB entries

| Id | Parent | Atoms | Residues | Signature |
|---|---|---|---|---|
| 1gzt_0 | 1gzt | 67 | 10 | $ASN_2ASP_3CA_2FUC_1GLU_1GLY_1$ |
| 1gzt_1 | 1gzt | 67 | 10 | $ASN_2ASP_3CA_2FUC_1GLU_1GLY_1$ |
| 1gzt_2 | 1gzt | 67 | 10 | $ASN_2ASP_3CA_2FUC_1GLU_1GLY_1$ |
| 1gzt_3 | 1gzt | 67 | 10 | $ASN_2ASP_3CA_2FUC_1GLU_1GLY_1$ |
| 1our_0 | 1our | 63 | 9 | $ASN_2ASP_3CA_2GLU_1MAN_1$ |
| 1ovp_0 | 1ovp | 63 | 9 | $ASN_2ASP_3BDF_1CA_2GLU_1$ |
| 1ovs_0 | 1ovs | 67 | 10 | $ASN_2ASP_3CA_2GLU_1GLY_1MAN_1$ |

Pattern Validation 🟩 No Issue/Not Validated 🟩 Minor Issue 🟨 Chirality Issue 🟥 Structural Issue

Parent PDB Entry 🟩 No Issue 🟫 Input Warning 🟨 Query Warning 🟥 Error

TIP: Applying a filter in the Metadata Filter section below will keep only the rows of interest and add the corresponding metadata columns to this table.

## Pattern 1gzt_0 in 1gzt [ PDB.org | Validator$^{DB}$ ]

**Ligand Validation Info**
No issues

**Residues** 10
$ASN_2ASP_3CA_2FUC_1GLU_1GLY_1$
ASN 21 A - GLU 95 A - ASP 99 A - ASP 101 A - ASN 103 A - ASP 104 A - FUC 201 A - CA 301 A - CA 302 A - GLY 114 B

**Atoms** 67
$C_{33}Ca_2N_9O_{23}$

**Download**
PDB | MOL

| | |
|---|---|
| **EC Numbers** | **Experiment method** |
| None Specified | X-RAY DIFFRACTION |
| **Host organisms** | **Host organism's genus** |
| None Specified | None Specified |
| **Origin organisms** | **Origin organism's ge...** |
| PSEUDOMONAS AE... | None Specified |
| **Polymer type** | **Protein stoichiometry** |
| Protein | Homomer |
| **Resolution (Å)** | **Year of publication** |
| 1.3 | 2002 |

## Metadata Filter Nothing selected

| | | | |
|---|---|---|---|
| EC Numbers 2 | Experiment method 1 | Host organisms 4 | Host organism's genus 2 |
| Origin organisms 7 | Origin organism's genus 4 | Polymer type 1 | Protein stoichiometry 2 |
| Resolution (Å) 18 | Year of publication 11 | | |

© 2015 David Sehnal

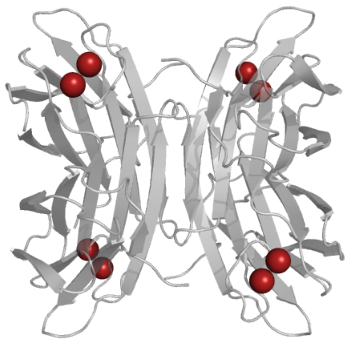
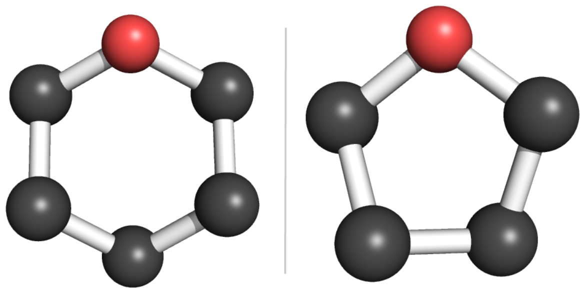
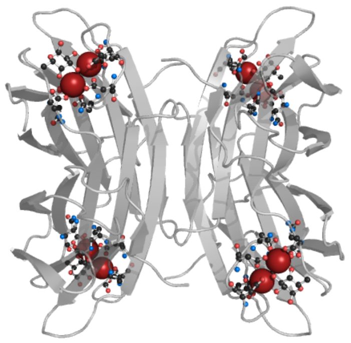
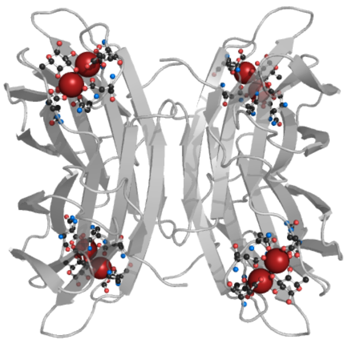
Figure S1: The results page contains a detail about all identified patterns and PDB entries of origin together with metadata details and visualization of each pattern. Color denotation specifies possible structural issues.

## SI Structure Validation

When running a query, the user may request an optional validation of ligands and non-standard residues of size larger than 6 atoms present in the atom patterns. The structure of these residues can be examined on their completeness and correctness using MotiveValidator [1], and the validation report can be immediately reached via ValidatorDB [2]. In brief, ValidatorDB identifies all the residues relevant for validation based on their annotation (PDB residue name, PDB residue ID, and PDB chain). For each residue to be validated, the input for MotiveValidator is made up from the atoms of the residue, and all atoms which are at most 2 bonds away from these. Validation is performed against a suitable structural model from the wwPDB Chemical Component Dictionary [3], and employs SiteBinder [4]. Any structural discrepancies are reported.

## SI Query 1: Workflow of building the query

The results for the Case Study I can be found at:
http://webchem.ncbr.muni.cz/Platform/MotiveQuery/Result/CSI

| 1) Defining two calcium ions close to each other | 2) Defining a ring of either pyranose or furanose, typically associated with sugars |
|---|---|
|  |  |
| `Near(4, Atoms("Ca"), Atoms("Ca"))` | `Or(Rings(5 * ["C"] + ["O"]), Rings(4 * ["C"] + ["O"]))` |
| 3) Defining a pair of calcium ions and the coordinating residues | 4) Filtering only to those patterns which contain a sugar moiety by combining the PQ expressions 2 and 3 |
|  |  |

| | |
|---|---|
| `Near(4,Atoms("Ca"),Atoms("Ca")).`<br>`ConnectedResidues(1)` | `Near(4,Atoms("Ca"), Atoms("Ca")).`<br>`ConnectedResidues(1).`<br>`Filter(lambda l: l.Count(`<br>`Or(Rings(5 * ["C"] + ["O"]),`<br>`Rings(4 * ["C"] + ["O"])))>0)` |

## SI Query 2

Filtering out nucleotides from SI Query 1 (patterns containing a phosphorus atom):

Table S1: Content of the 87 sugar binding site of the Asp*3-Glu-Asn*2-Gly composition

| **Sugar ID** | **Sugar name** | **Counter** |
|---|---|---|
| FUC | $\alpha$-L-fucose | 33 |
| MMA | O1-methyl-mannose | 17 |
| MFU | $\alpha$-L-methyl-fucose | 8 |
| MAN | $\alpha$-D-mannose | 5 |
| ARW | methyl $\beta$-D-arabinopyranoside | 4 |
| GXL | $\alpha$-L-galactopyranose | 4 |
| 2G0 | (2S)-1-[(2S)-6-amino-2-([(2S,3S,4R,5S,6S)-3,4,5-trihydroxy-6-methyltetrahydro-2H-pyran-2-yl]acetylamino)hexanoyl]-N-[(1S)-1-carbamoyl-3-methylbutyl]pyrrolidine-2-carboxamide | 4 |
| A1Q | methyl L-glycero-$\alpha$-D-manno-heptopyranoside | 4 |
| YX0 | [(3E)-3-(1-hydroxyethylidene)-2,3-dihydroisoxazol-5-yl]methyl 6-deoxy-$\alpha$-L-galactopyranoside | 3 |
| LZ0 | [1-(2-oxoethyl)-1H-1,2,3-triazol-5-yl]methyl 6-deoxy-$\alpha$-L-galactopyranoside | 3 |
| FUL | B-L-fucose | 2 |

Table S2: Content of the 12 sugar binding site of the Asp*3-Glu-Asn*2 composition

| **Sugar ID** | **Sugar name** | **Counter** |
|---|---|---|
| MAN | O1-methyl-mannose | 4 |
| MFU | $\alpha$-D-mannose | 3 |
| MMA | O1-methyl-mannose | 2 |
| BDF | B-D-fructopyranose | 1 |
| FUC | $\alpha$-L-fucose | 1 |
| A1Q | methyl L-glycero-$\alpha$-D-manno-heptopyranoside | 1 |

## SI Query Validation 1

Out of the 114 residues relevant for validation, in three cases the structural discrepancy prevented proper validation, and the validated residue was flagged as Degenerate. This was caused by the erroneous annotation of sugar residues in the 1ovs entry. All three MAN residues housed in the binding site are formally decomposed into two separate residues, where the second residue annotated as MAN contains just two atoms, therefore marked as Degenerate. All four 2G0 ligands present in the pectate lyase (3dcq) are incomplete, and finally 2 out of 4 FUC ligands present in the 1oxc entry exhibit incorrect chirality in the C1 carbon. The remaining 105 residues are complete and exhibit correct chirality, resulting in the overall good quality of the queried data.

## SI Query 3

The results for the Case Study II can be found at:
http://webchem.ncbr.muni.cz/Platform/MotiveQuery/Result/CSII

```
RegularMotifs('.{2}C.{2,4}C.{12}H.{3,5}H').
        ConnectedAtoms(1).
```

## SI Query 4

```
RegularMotifs('.{2}C.{2,4}C.{12}H.{3,5}H').
          ConnectedAtoms(1).
          Filter(lambda m:
            m.Find(Atoms('Zn').
              ConnectedResidues(1).
              Filter(lambda n:
                        (n.Count(Residues('Cys')) == 2) &
                        (n.Count(Residues('His')) == 2))).
                SeqCount() > 0)
```

## SI Query 5

```
RegularMotifs('.{2}C.{2,4}C.{3}[F|Y].{5}[AILFPGV].{2}H.{3,5}H').
          ConnectedAtoms(1).
          Filter(lambda m:
            m.Find(Atoms('Zn').
              ConnectedResidues(1).
              Filter(lambda n:
                        (n.Count(Residues('Cys')) == 2) &
                        (n.Count(Residues('His')) == 2))).
                SeqCount() > 0)
```

## SI Query 6

```
RegularMotifs('.{2}C.{2,4}C.{12}H.{3,5}H').
          ConnectedAtoms(1).
          Filter(lambda m:
            m.Find(NotAtoms('C', 'N', 'O', 'P', 'S', 'H', 'Zn').
              AmbientResidues(3).
              Filter(lambda n:
                        (n.Count(Residues('Cys')) == 2) &
                        (n.Count(Residues('His')) == 2))).
                SeqCount() > 0)
```

## Limitations

To overcome the shortcoming for an amount of parallel queries executed so as the amount of returned patterns or atoms, the search can be split into several runs, and for each run the data set to be queried can be specified using metadata filtering or a list of PDB ids. The command-line version of PQ does not include these limitations.

Similar to all services dealing with external data sources, the performance of PQ is dependent on the quality of input structures, in this case the quality of the data set which is queried.

Low resolution structures can contain a variety of structural discrepancies. Therefore making difficult for PQ to properly identify atomic bonds. In order to deal with this potential drawback, we encourage users to use custom structure lists based on useful metadata such as the structure resolution, or a careful selection of PDB ids. For the user's convenience, the PQ results page links to ValidatorDB, which contains validation reports of the completeness and correctness of ligands and non-standard residues larger than 6 atoms.

Finally, there is an initial requirement for the user to learn basic queries and develop the ability to decompose chemical or structural problems into smaller pieces in order to construct a particular query. However, we believe the language learning curve is not too steep, enabling even the inexperienced user to learn the basics fast. In addition, the PQ user manual includes a large number of examples for each of the basic queries, as well as more complex examples for solving particular biologically relevant issues. Last but not least, direct user support for constructing user-defined queries is enabled.

## Performance Overview

This section gives an overview of the "real world" performance of the presented tool. Unless stated otherwise, each computation was run 7 times, the fastest and slowest result was discarded, and the times are shown as the average (the differences in individual times were about 1% in all cases). All computations were run on a machine with Intel i7-3770 @ 3.4GHz processor, 32GB RAM, SSD drive, using Microsoft Windows Server 2008 R2 and .NET Framework 4.5.2 (Your computation can be executed on a different machine, such as: 4xIntel Xeon E5-4610 @ 2.9GHz, 64GB RAM, SSD drive using Microsoft Windows Server 2012 and .NET Framework 4.5.2).

Table S3 gives running times of the PatternQuery application for different queries and input sizes.

| **Query** | **# of PDBs / Size** | **# of Patterns / Size** | **Time** | **Avg. Rate** |
|---|---|---|---|---|
| 1: Empty[1] | 107249 / 118GB | 0 / 0 MB | 29m28s | ~68 MB/s |
| 2: Residues with metals[2] | 107249 / 118GB | 277127 / 160 MB | 30m56s | ~65 MB/s |
| 3: Residues with metals and surroundings[3] | 107249 / 118GB | 276762 / 1 GB | 35m56s | ~56 MB/s |
| 4: Lectins[4] | 7857 / 7.2GB | 108 / 710 KB | 1m50s | ~67 MB/s |
| 5: $C_2H_2$ Zinc Finger motifs[5] | 9699 / 16.1GB | 354 / 471KB | 5m6s | ~54 MB/s |

Table S3: Performance of the PatternQuery application on various queries and datasets. The size is of the input data stored in uncompressed mmCIF format.

The query 1 is an "empty" query to establish the amount of time spent by reading the input from disk and parsing it to create a representation in memory. The results show that the user can expect 2-20% time overhead over the "empty" query based on the complexity of the query and/or the result size. The implementation of the application allows to easily query only relevant data to dramatically reduce the query time as shown by queries 4 and 5.

[1] `Atoms(' ')`, executed on the entire PDB.org archive as of 16.3.2015.

[2] `Atoms('Zn', 'Fe', 'Ca', 'Mg').ConnectedResidues(0)`, executed on the entire PDB.org archive as of 16.3.2015.

[3] `Atoms('Zn', 'Fe', 'Ca', 'Mg').ConnectedResidues(0).AmbientResidues(5)`, executed on the entire PDB.org archive as of 16.3.2015.

[4] `Near(4, Atoms('Ca'), Atoms('Ca')).ConnectedResidues(1).Filter(lambda l: l.Count(Or(Rings(5 * ['C'] + ['O']), Rings(4 * ['C'] + ['O']))) > 0).Filter(lambda l: l.Count(Atoms('P')) == 0)`, executed on entries with calcium atom(s) in the PDB.org archive as of 16.3.2015.

[5] `RegularMotifs('.{2}C.{2,4}C.{3}[F|Y].{5}[AILFPGV].{2}H.{3,5}H').ConnectedAtoms(1)`, executed on entries with zinc atom(s) in the PDB.org archive as of 16.3.2015.

# Bibliography

[1] Vařeková,R.S., Jaiswal,D., Sehnal,D., Ionescu,C.M., Geidl,S., Pravda,L., Horský,V., Wimmerová,M. and Koča,J. (2014) MotiveValidator: Interactive web-based validation of ligand and residue structure in biomolecular complexes. *Nucleic Acids Res.*, 42, W227–33.

[2] Sehnal,D., Svobodová Vařeková,R., Pravda,L., Ionescu,C.-M., Geidl,S., Horský,V., Jaiswal,D., Wimmerová,M. and Koča,J. (2015) ValidatorDB: database of up-to-date validation results for ligands and non-standard residues from the Protein Data Bank. *Nucleic Acids Res.*, 43, D369–D375.

[3] Sen,S., Young,J., Berrisford,J.M., Chen,M., Conroy,M.J., Dutta,S., Di Costanzo,L., Gao,G., Ghosh,S., Hudson,B.P., et al. (2014) Small molecule annotation for the Protein Data Bank. *Database (Oxford).*, 2014, 1–11.

[4] Sehnal,D., Vařeková,R.S., Huber,H.J., Geidl,S., Ionescu,C.-M., Wimmerová,M. and Koča,J. (2012) SiteBinder: an improved approach for comparing multiple protein structural motifs. *J. Chem. Inf. Model.*, 52, 343–59.