A Query Language for Biological Networks

Ulf Leser

Department for Computer Science, Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12485 Berlin, Germany
leser@informatik.hu-berlin.de

Abstract

Many areas of modern biology are concerned with the management, storage, visualization, comparison, and analysis of networks. For instance, networks are used to model signal transduction and metabolic pathways, gene regulation, and interaction of molecules in general. A large number of databases have emerged that collect and provide information on cellular networks and protein interaction. However, most users and applications are not concerned with entire databases, but search for specific subsets of the data. For these purposes, it is essential to be able to describe the desired subnetwork as specific as necessary and as simple as possible. Despite the increased importance of network data in biology, there still exists no proper language for describing and retrieving specific parts of a network.

In this paper, we introduce the pathway query language (PQL) for retrieving specific parts of large, complex networks. The language is based on a simple graph data model with extensions reflecting properties of biological objects. PQL queries match arbitrary subgraphs in the database based on node properties and paths between nodes. PQL is a powerful language, being able to express graph isomorphism. A specific feature is that the result of a query is de-coupled from the matched subgraph. Thus, a query may require a certain structure in the database to exist, but return a different subgraph. Furthermore, the result of a PQL query itself is a graph and can be used in further queries, which allows for query composition, query nesting, and graph views, features well known from relational databases.

PQL is easy to learn for everybody with a basic knowledge of SQL. It is implemented on top of a relational database. A query is compiled into a stored procedure which returns the resulting graph in temporary tables. All computation is performed by relational queries, thus exploiting the capabilities of modern database systems in terms of query optimization and memory management. The code is for free available from the author.

1 Introduction

Many areas of modern molecular biology deal with data that is structured in forms of networks. Metabolic pathways, signal transduction pathways, and networks of gene regulation are naturally modeled as graphs. Systems Biology tries to model the entire network of interacting molecules that forms cells and, ultimately, all living beings [JTA+00; Kit01]. In these networks, nodes typically represent biological entities such as enzymes, genes, or compounds, and edges represent some form of chemical interaction or relationship. The availability of such network data is currently increasing rapidly, both due to enhanced experimental techniques such as two-hybrid systems [LS00] or tandem affinity purification [RSR99] and due to improved prediction methods [MYC+02; Sjo04;

vMHJ+03]. Ontologies, such as the MGED ontology for describing microarray experiments [SP03], the GeneOntology for gene function, biological processes, and sub-cellular locations [GO01], or the ontology of molecular function from EcoCyc [Kar00], form large directed, acyclic graphs where nodes are concepts and edges represent semantic relationships between concepts. The "tree of life", as for instance provided in the NCBI taxonomy database, is represented by a tree with nodes representing different species and edges standing for an evolutionary relation. Protein networks are extracted from publications, based either on natural-language processing [BHV02; FKKR01; HPL05; MXE01; PRJ00] or simply on co-occurrence [JLKH01]. In these graphs, nodes are terms extracted from the literature, and edges mean, for instance, that two proteins often appear together in a sentence. In a wider sense, also the chemical structure of molecules can be modeled as a graph, with atoms forming nodes and chemical bonds forming edges. This is, for instance, used to find common sub-structures in a given set of proteins [NK03; vHNM00].[1]

Driven by the current interest in systems biology, a large number of databases have emerged which collect and provide data on interactions of biological entities [BBH03; HMP+04; JTV+03; KGK+04; KRS+02; KNO+03a; SMS+04]. At the time of writing, an up-to-date list of "pathway data resources" lists not less than 153 entries[2]. At the same time, the networks under analysis are becoming larger and larger. While isolated pathways rarely have more than a hundred components and can thus be examined manually, many of the network types described above are far bigger: At the time of writing, the GeneOntology has more than 16.000 nodes, the NCBI taxonomy database has more than 120.000 nodes, the STRING database [vMHJ+03] of predicted functional associations between proteins has more than 200.000 associations[3], KEGG [KGK+04] has more than 17.000 pathways of which more than 250 are reference pathways, the BIND database [BBH03] stores more than 110.000 protein interactions and the DIP database [SMS+04] approximately 50.000. The largest current network is probably the PubGene database [JLKH01], which extracted more than 6 million associations from the scientific literature[4].

Clearly, users can only review small fractions of such networks at a time. However, users are usually highly specific about the information they are interested in. Typical examples of queries against a metabolic pathway database are (see Section 4 for more examples):

- Find all reactions involving a certain substance
- Find all paths, i.e., chains of reactions, connecting two given substances
- Find the shortest path between two substances that include a third substance
- Given a set of molecules, extract the connected subgraph that contains all these elements and has the least number of nodes.

Despite the necessity for complex queries, current pathway databases support only very simple queries. Mostly, searching the database is restricted to full text search of node names; sometimes, it is also possible to search paths between two given nodes. We believe that such databases should support a much broader range of queries, such as searching complex patterns of nodes and edges and selecting biological objects based on functional annotation. Further, we believe that there is a strong need for a declarative language to clearly and succinctly specify queries on biological networks. In the current state, all non-trivial operations on networks must be hard-coded by the database devel-

---

[1] For a more complete list of applications of graphs and networks in biology, see the Graph Data Management page of Frank Olken at http://pueblo.lbl.gov/~olken\graphdm/graphdm.htm.
[2] See http://cbio.mskcc.org/prl.
[3] And approximately 44 times as much when also lower confidence scores are included.
[4] See http://www.pubgene.com/statistics.htm.

oper and are directly dependent on the specific data model and hence non portable. In this paper, we propose a language matching these requirements of increased expressibility and declarativity: PQL, the pathway query language.

PQL is a declarative language whose syntax is similar to SQL. Any person having a basic knowledge of SQL will quickly be able to write PQL queries. PQL queries operate on a simple graph data model that is a generalization of many more specific data models; therefore, we believe that PQL can be easily adopted for a broad range of existing systems. The result of a PQL query itself is a graph, which offers possibilities for nesting and composing PQL queries, in the same manner as this is possible for relational queries. Despite its syntactic simplicity, PQL is a powerful language capable of expressing graph isomorphism. Implementing PQL on top of a relational database is quite straightforward, which eases its porting to different systems and databases. We shall describe an implementation based on Oracle.

However, PQL is not capable of complex graph operations such as the computation of spanning trees or the detection of graph cores. In our understanding, such analysis should remain to specialized applications using highly-tuned data structures. Instead, we designed PQL to be a robust, general-purpose graph query language.

We think that this proposal can have many positives influences on the field:

- Designing about a language implicitly forces researchers to think about the requirements that exist for querying pathways. This discussion apparently has not yet started in the community, despite many papers mentioning various types of queries. In Section 4 we will review some of those.
- A properly defined language can be used by many interaction and pathway databases, thus reducing the amount of duplicated work. Furthermore, users and developers need to learn only one language and can use this language on their favorite database, instead of having to cope with a multitude of proprietary languages.
- A query language acts as an interface between applications and databases. PQL should be seen as a proposal for an interface between pathway applications and pathway databases. Having a clear interface fosters the development of database-independent methods for pathway analysis. Network algorithms, user interfaces, and visualization tools may use this language and thus become more easily portable to other databases[5].
- Having a clear semantics of queries helps to integrate data from heterogeneous repositories, since the same query can be shipped to and computed on different databases.
- A clearly defined language is a proper target for further research into query optimization, i.e., algorithms for executing network queries as fast as possible. Query optimization is a necessary requirement as soon as large networks are used which cannot be analyzed fully in main memory any more.

This paper introduces a query language for pathway data, and is not intended to support a concrete data model for biological pathways[6]. The abstract model we define in Section 2 is rather generic and can be implemented in many different ways. One such way is described in Section 4.5, but other representations may be more appropriate depending on the concrete application. We believe that PQL can be implemented on top of all models that can be mapped into a graph.

[5] In the same spirit, the definition of SQL has lead to greater independence of applications from database vendors.
[6] See for instance the lively discussions in the BioPAX mailing list on pathway data models (http://www.biopax.org).

The implementation of PQL we provide in this paper is prototypical. Little effort was put into optimization, and no benchmarks are provided. In our tests, even complicated PQL queries were generally computed in a few seconds. The main reason for this performance is a high degree of (costly) pre-computation. Other implementations are possible, for instance moving all computations into main memory (with advantages in performance and disadvantages in scalability for large networks), or dropping the pre-computation (with disadvantages in performance but advantages in space consumption).

The rest of this paper is organized as follows. Section 2 defines the data model that PQL is based upon, divided into a general part and a part for modeling biological properties of nodes. Section 3 defines syntax and semantics of PQL and is the heart of this paper. Section 4 discusses a number of exemplary queries taken from publications on pathway databases and systems and describes how (if at all) they can be expressed in PQL. Section 4.5 sketches our prototypical implementation. Section 6 discusses related work. Finally, Section 7 presents possible and planned extensions of PQL, and Section 8 concludes.

2 PQL Data Model

We introduce the PQL data model in two steps. First, we discuss only the basic model based on graphs. In Section 2.2 we add biological properties to the basic model.

Although PQL is a query language for biological pathways, our definitions actually do not mention the term "pathway", as it is actually not well-defined. In general, a pathway is considered as a set of molecules and chemical reactions that perform some function, such as the ATP synthesis or the MAPK signaling pathway. However, the borders of such a pathway, i.e., which molecules and reaction pertain to a pathway, usually are not precisely defined but subject to personal preferences. Also, the level of detail of a pathway depends on the application in mind, especially whether or not ubiquitous small molecules should be included (cofactors such as water or carbon dioxide). Some database organize networks in pathways (see e.g. KEGG or Reactome [JTV+03]), while others only store interactions as basic elements and leave it to the user to group them into pathways. PQL follows the later strategy.[7]

2.1 Basic Data Model – Graphs

The basic PQL data model is a graph.

Definition 1. (Graph data model)

A graph G is a tuple (V, E) with vertices (nodes) V and directed edges E, where

- Nodes fall into two disjoint sets I (interactions) and M (molecules), i.e., $V=I\cup M$ and $I\cap M=\emptyset$
- Edges connect nodes, i.e., $E\in V\times V$
- Edges must not connect two molecules, i.e., $\forall(n_1, n_2) \in E: n_1\in I \wedge n_2\in I$
- Edges must not connect an interaction to itself, i.e., $\forall (n_1, n_2) \in E: n_1\neq n_2$

[7] See Section 7.2 for a discussion on how pathways – virtual groupings – could be incorporated into PQL.

- Edges are unique and directed, i.e., there may be at most one edge (n_1, n_2) from node n_1 to node n_2, but there may also exist another edge (n_2, n_1) from n_2 to n_1

■

Note that G need not be connected, i.e., it may fall into several, unconnected subgraphs or even isolated nodes.

The biological interpretation of such a graph is the following. G represents a network of substances and reactions. Nodes from M are biochemical entities, such as proteins, metabolites, or genes. Each node from I stands for a interaction, which can represent a gene expression, a chemical reaction with products and substrates, or the formation of a compound from different proteins (restrictions of interactions to species, pathways, tissues, and cellular locations are discussed later). Edges may connect either (1) a molecule to an interaction, meaning that the molecule is necessary for the interaction to happen, (2) an interaction to a molecule, meaning that the molecule is a product of the interaction, (3) an interaction to an interaction. The latter situation arises when the first interaction influences the second, such as an enzymes inhibiting (first interaction) the catalytic effect (second interaction) of another enzyme. Interactions may involve any number of molecules either as input or as output, and molecules may be connected to any number of interactions.

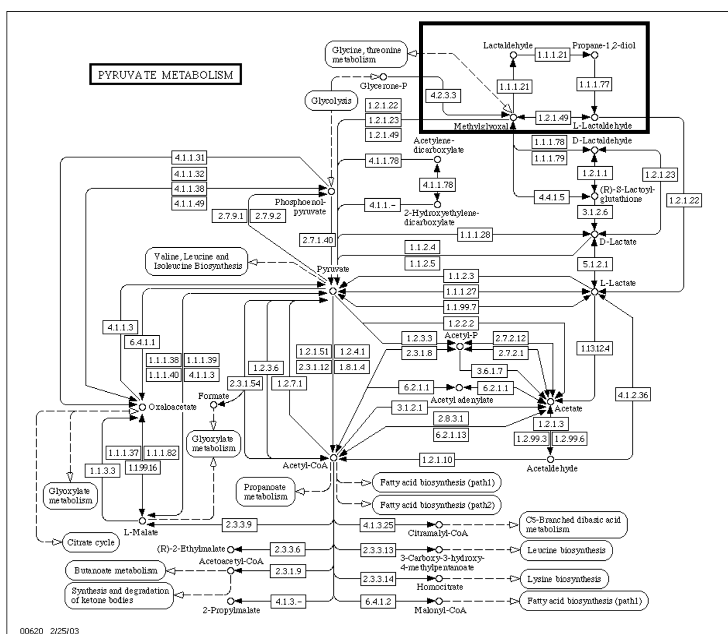
By definition, no node may have an edge to itself, which implies that every cycle in a PQL graph contains at least two nodes. First, edges from a molecule to a molecule are meaningless given our interpretation, and therefore edges emerging from a molecule must point to an interaction. Second, edges from an interaction to itself are semantically meaningless and explicitly forbidden. If we want to model that a certain reaction inhibits itself, this is properly represented by two interactions A and B, where A represents the reaction and B represents the inhibition, and edges exist from A to B and from B to A.

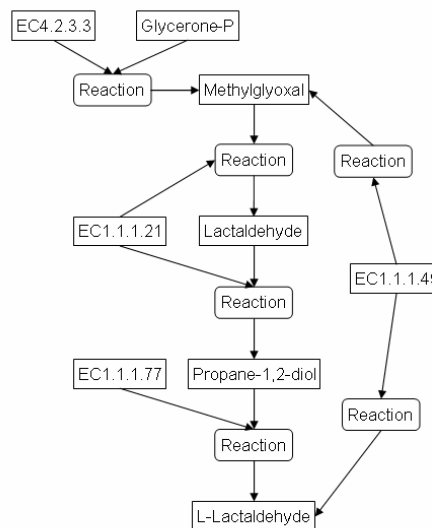
The PQL data model is quite general. Similar models are used in most pathway databases, such as aMAZE [vHNM00], KEGG [KGK+04], or Reactome [JTV+03]. We give two examples of how biological data can be represented in our model.

First, consider the pyruvate metabolism. Figure 1a) show the original pathway as presented in KEGG. Rectangles represent reactions with enzyme classification number, small circles represent compounds, and rectangles with round edges represent other pathways. Dashed arrows connect a compound with a pathway, meaning that the compound is also involved in the other pathway. Other arrows connect either a reaction to a compound, meaning that the reaction generates the compound as a product, or a compound to a compound via a reaction, meaning that the first compound is a necessary input for the reaction that generates the second compound.

Figure 1b) shows a small fraction of the KEGG pathway in the PQL model. Enzymes and compounds are represented as molecules, and each arrow is transformed into an interaction with products and, if present, substrates.

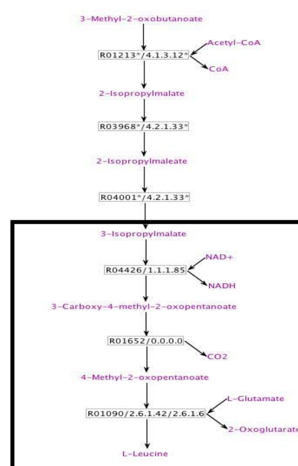
As another example, consider the leucine biosynthesis in yeast. Figure 2a) shows this pathway as represented in the aMAZE database. Here, purple text represents metabolites, and rectangles stand for reactions, denoted by a reaction ID and the EC number of the enzyme catalyzing this reaction. In our model, enzymes and metabolites are molecules and reactions are interactions. Replacing each rectangle with a node for the reaction connected to one node for each participating enzyme translates this representation into our data model (Figure 2b).

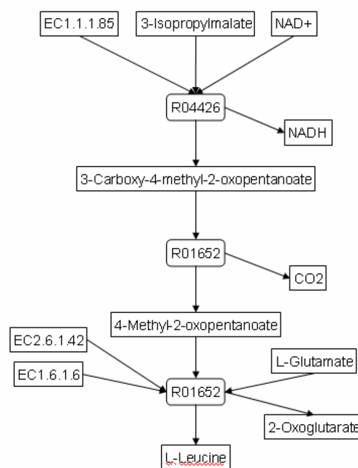




(a) (b)

Figure 1. (a) Pyruvate metabolism as shown in KEGG. (b) Representation of the reactions in the upper right corner of the KEGG pathway in the PQL data model. Molecules are boxes, interactions are rounded boxes.





(a) (b)

Figure 2. (a) Leucine biosynthesis in yeast taken from aMaze. (b) Lower part of the leucine biosynthesis represented in the PQL model. Molecules are boxes, interactions are rounded boxes.

There is a subtle difference in the graphical representation of pathways in KEGG and aMAZE, respectively. In both cases it is not clear whether all incoming edges, i.e., reactions, of a compound are necessary to yield this compound, or whether the incoming edges are alternative reactions with the same product. The semantics in both cases is defined by default only: In KEGG, substrates are alternatives. For instance, there are two ways that Methylglyoxal can be produced, either by a reaction involving enzyme EC1.1.1.49, or by a reaction involving EC4.2.3.3 and Glycerone-P. In aMAZE, all

substrates are required for a reaction to happen (and all products are generated). In the PQL model, such differences are made explicit by properly placing interaction nodes.

2.2 Biological Properties

PQL is a query language for biological networks. The nodes in a PQL database are biological entities or biochemical interactions. These need to be described further to allow biological rather than purely abstract queries. The PQL data model therefore defines a small set of properties of nodes and edges that are important for many biological applications.

Definition 2. (Biological types and functions)
A DAG is a directed, acyclic graph with a single root element. We define the following additional data structures:

- Let `TYP` be a DAG of concepts defining biological entity types.
- Let `FUNC` be a DAG of concepts defining functional descriptions of biological entities.

■

Using TYP, nodes are classified according to a type hierarchy. On the level below root, there are currently two nodes, i.e., 'molecule' and 'interaction'. Below those, different types of molecules, such as 'gene', 'polypeptide', 'enzyme', and 'protein', and different types of interaction, such as 'inhibition', 'catalysis', or 'activation', are defined. Using FUNC, molecules are annotated with information about their function, for instance using the vocabulary of the Gene Ontology [GO01]. Using these data types, we define nodes and edges in more detail.

Definition 3. (Node properties)
Consider a PQL database G=(V, E) consisting of the set of interactions I, the set of molecules M, the set of nodes $V=I \cup M$, and the set of edges E. We define the following properties for each node $v \in V$:

- `Name: STRING`. A human-readable name for v.
- `ID: INT`. An identifier for v that is unique in V.
- `Type: TYP`. Defining the type of v.
- `Function: set-of-FUNC`. Defining the functions of v.

■

Note that each node may have only one type, but many functional annotations. We currently make no difference between molecules and interactions in the set of properties used to describe them. TYPE can be seen as a graph representation of the class hierarchies used in object-oriented pathway databases such as aMaze [vHNM00]. Clearly, many extensions to this model are possible. For instance, molecules and interactions could be grouped into pathways, pathways could be annotated with the tissues they occur in, interactions could be annotated with kinetic parameters, molecules could be annotated with links to external databases, and interactions and molecules could be annotated with the cellular compartment they occur in. Some of these extensions are discussed in Section 7.2. However, the current focus of PQL is on matching structures in networks. We therefore leave extensions to the data model for future work.

3 Pathway Query Language

The purpose of PQL is the extraction of subgraphs with certain properties from a large network. The power of the language is determined by the types of subgraph properties it can express. Imagine a graph with n nodes and m edges. A simple query could extract all nodes with a certain name, which can be implemented in O(log(n)) as a binary search over a sorted list of all node labels. A more complex query could extract the shortest path from a fixed node n_1 to a fixed node n_2, which is possible in $O(n^2)$ for dense graphs using Dijkstra's algorithm. Finally, even more complex queries could ask for the minimal spanning tree of the graph, which requires O(m*log(n)) [CLRS03], or all pairs of nodes for which all paths connecting the two nodes have at least one intermediate node in common. The later would search for pathways that can be knocked out by suppressing a single reaction or eliminating a single molecule from the system.

PQL allows for medium complex queries. More precisely, it can extract subgraphs that are characterized by node and edge properties and by existence and properties of the paths they contain. Thus, PQL goes beyond the search options available in most current interaction databases, but is not capable of computing properties of the entire graph. This restriction has the advantage that PQL queries can be computed relatively efficiently (see Section 3.5). Again, this design choice can be compared to the design choices underlying SQL. In SQL, queries containing a fixed number of joins can be computed, but it is not possible to formulate and hence compute recursive queries, which are for instance necessary to describe trees, part-of-lists, or graphs. This restriction was introduced purposefully in the design of SQL to ensure tractability of query answering.

We introduce PQL in four steps. The first three assume cycle-free graphs. Section 3.1 discusses syntax and semantics of PQL. We introduce expressions describing conditions on paths in Section 3.2. In Section 3.3 we describe how the subgraph returned by a PQL query is determined. Finally, in Section 0 we explain the semantics of PQL queries in graphs containing cycles. Section 3.5 finishes this chapter with a note on the complexity of PQL.

3.1 Basic Syntax and Semantics

PQL queries resemble the syntax of SQL, though the semantics is quite different. As a SQL query, a PQL query has three parts – a SELECT clause, a FROM clause, and a WHERE clause. As SQL, where queries have relations as input and generate a relation as output, PQL queries are evaluated on a graph and result in a graph.

Definition 4. (PQL general syntax)

A PQL query has the following syntax:

```
SELECT   subgraph-specification
FROM     node-variables
WHERE    node-condition-set
```

where

- *subgraph-specification* is an expression defined in Section 3.3.
- *node-variables* is a list of variables.
- *node-condition-set* is a set of *node-conditions* connected by AND or OR and with possible parenthesis.

- A *node-condition* is either a *simple node-condition*, a *path-expression* (see Section 3.2), or the negation of a node-condition, expressed with a preceding NOT.
- A *simple node-condition* is a condition of the form '`A.attribute op B.attribute`', '`A = B`', '`A.attribute op String`', '`A HASFUNC node-function`', or '`A ISA node-type`', where A and B are node variables, `op` is either '=', '<', or '>', attribute is either '`ID`' or '`name`', `String` is an arbitrary string, `node-function` is an element of `FUNC`, and `node-type` is an element of `TYP`.[8]

Simple conditions of the first three types are sometimes called *normal* conditions.

■

Intuitively, during query evaluation node variables are bound to nodes of the database graph such that all *node-conditions* in the WHERE clause evaluate to TRUE. The query result is constructed from these variable bindings according to the *subgraph_specification* of the SELECT clause. Note that binding of node variables, which essentially means matching parts of the graph, does not directly determine the subgraph returned. For now, we assume that the SELECT clause is a "*" and that it returns all nodes from the subgraph matched by the FROM and WHERE clause.

Consider the pathway shown in Figure 2b. The following query returns a graph consisting of two nodes, namely molecules '`3-Isopropylmalate`' and '`EC1.1.1.85`':

```
SELECT    *
FROM      A, B
WHERE     A.name = '2-Isopropylmalate' AND
          B.name = 'EC1.1.1.85'
```

No edges are returned; these would have to be specified by special operations in the SELECT clause (see Section 3.3). In the following, we shall use symbols A, B, C … for node variables and X, Y, Z for actual nodes from the underlying database (bound to variables).

Definition 5. (Node conditions)

A *simple node-condition* is interpreted as a Boolean function. Given an assignment of node variables A (and B) to nodes, this function returns:

- If the condition has the form '`A.attribute op B.attribute`', it returns `TRUE` if the value of `attribute` of the node assigned to A is equal to, greater than, or less than (depending on `op`) the value of `attribute` of the node assigned to A, and otherwise `FALSE`.
- If the condition has the form '`A=B`', it returns `TRUE` if A and B are assigned to the same node, and otherwise `FALSE`.
- If the condition has the form '`A.attribute op String`', it returns `TRUE` if the value of `attribute` of the node assigned to A is equal, grater than, or less than `String` (depending on `op`), and otherwise `FALSE`.
- If the condition has the form '`A ISA node-type`', it returns `TRUE` if the node assigned to A has as type either `node-type` or any successor of `node-type` in `TYP`.
- If the condition has the form '`A HASFUNC node-function`', it returns `TRUE` if the set of functions associated with the node assigned to A contains `node-function` or any predecessor of `node-function` in `FUNC`.

[8] We omit obvious extensions such as the definition of numeric comparisons or the introduction of string functions such as `contains() or leftstring()` for brevity.

A *node-condition-set* is interpreted as a Boolean function returning the value of the Boolean expression formed by its constituent node conditions.

■

We now exactly define the result of a PQL query. This will be achieved in two stages (see also Figure 4). The first stage computes the *match graph* of the PQL query. The match graph can be thought of as a subset of the nodes of the underlying database. It is computed solely based on the node variables and the WHERE clause of the query. The second step, which is defined in Definition 12, will turn the match graph into a result graph based on specifications given in the SELECT clause of a PQL query.

Definition 6. (PQL basic semantics, match graph)

Let P be a PQL query on a database `G=(V, E)` with node variables $N=\{A_1, A_2, \ldots, A_n\}$ and a node-condition-set C. C is a Boolean formula over N. Let f be a function that assigns to each node variable a node from G, i.e., $f: N \mapsto V$.

- f is called a *binding function* (or *assignment*) for N if `C(f(N))=true`.
- The set of all binding functions for N is called the *match set* of P, written `match-set(P)`.
- The *match graph* of P, written `match-graph(P)`, is defined as:

$$match-graph(P) = \bigcup_{f \in match-set(P)} \{(a,x) \mid f(a) = x\}$$

■

Intuitively, this definition can be described follows. Query evaluation considers each *node variable* in the FROM clause. For each of these variables, all possible assignments of the variable to nodes of the graph are determined for which the conditions of the WHERE clause mentioning only this variable evaluates to TRUE. Node variables are equally assigned to molecules and interactions, if not specified otherwise in the WHERE clause. Once all possible bindings are computed for each node variable, the Cartesian product of these sets is computed. From this set, all instances are removed for which the entire WHERE clause evaluates to FALSE, thus enforcing conditions that include more than one node variable (such as a condition '`A.name=B.name`'). Next, all distinct assignments (node variables to database nodes) from the remaining elements of the Cartesian product are combined to form the match graph.

Thus, the set of nodes in the match graph is always a subset of the nodes of the underlying database. The match graph does not contain any edges. In Section 3.3 we shall construct the result of a PQL query from its match graph. For now, we simply consider the result of the query to be the set of database nodes in the match graph of the query, symbolized by a '*' in the SELECT clause.

Note that the semantics of PQL is different to the semantics of SQL queries. In a SQL query, the result is computed from the Cartesian product of all rows of all tables in the FROM clause, removing those elements for which the WHERE clause evaluates to FALSE. In the result of the SQL query the rows from the Cartesian product are preserved, though columns (attributes) might be removed, added, or changed in their value. In contrast, the concrete combinations of bindings of different node variables that together fulfill the WHERE clause are not preserved in the match graph of a PQL query. The match graph simply consists of all bindings present in the filtered Cartesian product "thrown together" into a flat, duplicate-free list of nodes (and bindings). However, viewed from a more abstract point of view, PQL queries closely resemble the intuitive meaning of a SQL query: A

SQL query operates on tables and produces a table (a set of rows), in the same manner as a PQL query operates on a graph and produces a graph (a set of nodes) – and not a set of graphs.

Consider the following queries to be evaluated on the graph in Figure 2b:

```
SELECT    *                              SELECT *
FROM      A, B                           FROM   A, B
WHERE     A.name='ADP' AND               WHERE  A.name='R04426' AND
          B.name='EC1.1.1.85'                   (B.name='NAD+' OR B.name='CO2')
```

The left query returns the empty graph as no binding can be found for node variable A. The right query returns the nodes 'R04426', 'NAD+', and 'CO2', since (only) the bindings (A→'R04426', B→'NAD+') and (A→'R04426', B→'CO2') adhere to the conditions in the WHERE clause.

The input to a PQL query is hence a graph, and the result is also graph[9]. Therefore, nesting of PQL queries and definition of PQL views is feasible. We shall discuss these options in more detail in Section 7.1.

3.2 Path Expressions

Until now, PQL queries are not more powerful than searching keywords in a database of node names. We shall now enrich the language with possibilities to match subgraphs in PQL by using path expressions. We introduce path expression by an example. Consider the following query:

```
SELECT    *
FROM      B, C, D
WHERE     D.name='L-Lactaldehyde' AND B ISA 'Enzyme' AND B[-2]D
          AND B[-*]C[-*]D AND C.name='Lactaldehyde'
```

Intuitively, a condition of the form 'X[-n]Y' with X and Y being node variables means that there must exist a path between X and Y in the matching subgraph. This path may have arbitrary length if n is '*', and must be of length n if n is a number. Hence, the above query matches a subgraph of four nodes A, B, C, and D such that D has name '*L-Lactaldehyde*', B is an enzyme, there exists a path of length 3 between node B and D, and there exists a path of arbitrary length from B through C to D where C must have the name '*Lactaldehyde*'. Thus, the query tries to find instances of the graph given in Figure 3 in the underlying network.

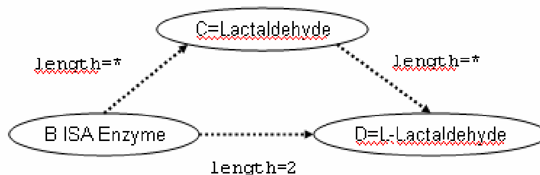


Figure 3. Graphical representation of the query given in the text. Dashed lines represent paths.

Assigning node variables to nodes works as follows (consider network given Figure 1b). First, node D must be the node '*L-Lactaldehyde*'. Then, B could be bound to 'Propane-1,2-diol', 'EC1.1.1.17', or 'EC1.1.1.49', since from those nodes there exists a path of length 2 to D; however, only the latter two are enzymes. Since there must exists a path from B through C to D where C must have name 'Lactaldehyde', B must be enzyme 'EC1.1.1.49'.

[9] This graph so far does not contain any edges.

We now formally define paths and path expressions. Note that we require all paths to be cycle-free. We discuss cycles in graphs and paths in Section 0.

Definition 7. (Paths)

A *path* in a graph `G=(V,E)` is a set of nodes $(v_1, \ldots, v_k)$, $k>1$, $v_i \in V$ such that

- $\forall i<k: (v_i, v_{i+1}) \in E$
- $\nexists i,j$ with $i \neq j$ and $v_i = v_j$.

The *length of a path* $p=(v_1, \ldots, v_k)$ with k nodes is k-1, i.e., the number of edges on the path.

■

Definition 8. (Path expressions)

A *simple path expression* is an expression of the form '`A[- op n]B`' where A and B are node variables and

- `n` is '*' and `op` is '=' or
- `n` is a positive integer and `op` is either "=", ">", or "<"

A *path expression* is a concatenation of simple path expressions such that two subsequent simple path expressions share one node variable. White spaces may be omitted. Further, op may be omitted in which case it defaults to '='.

■

Definition 9. (Semantics of path expressions)

A simple path expression '`A[- op n]B`' is interpreted as a Boolean function on the assignment of the node variables it contains. It returns

- `TRUE`, if there exists a path p from the binding of A to the binding of B such that
    - If n='*', p may have arbitrary length,
    - If n is an integer, p must have length equal to, greater than, or lesser than n, depending on `op`.
- `FALSE` otherwise.

A path expression is also interpreted as a Boolean function. It returns `TRUE` if all its constituent simple path expressions return `TRUE`.

■

Note that an edge from a node to itself is forbidden (see Definition 1 in Section 2.1). Thus, the following query always computes an empty graph, independent of the underlying database:

```
SELECT *
FROM A, B
WHERE A[-1]B AND A=B
```

Also note that path expressions are existential conditions. They require that certain paths exist between the nodes assigned to node variables, but they do not require that all such paths adhere to the stated conditions.

Path expressions hence express conditions on the existence of paths between nodes assigned to node variables. Each path expression considers a path between two nodes, but since a PQL query may contain arbitrary many path expressions and since path expressions may share node variables,

many types of subgraphs can be expressed. However, path expressions are not capable of describing all possible subgraphs, thus restricting the expressive power of PQL. For instance, we cannot state a condition on the length of the shortest path between two nodes, and we cannot state that different paths have the same length. Thus, queries of the form "Compute subgraphs with nodes A, B, C, D such that there is a path between A and B and between C and D of the same length" cannot be expressed.

3.3 The SELECT Clause – Specifying the Output

So far we assumed that the SELECT clause is always a '*', thus returning the set of all bindings of node variables. This is not sufficient, since we are interested in finding subgraphs and not just isolated nodes. These subgraphs may stretch further than the nodes in the match graph, for instance including the vicinity of matched nodes up to a certain range. Furthermore, a query often only requires some nodes to exists, but does require to retrieve the concrete bindings. The SELECT clause of a PQL query must therefore be capable of selecting certain nodes from the match graph, to add other nodes, and to add edges taken from the underlying database.

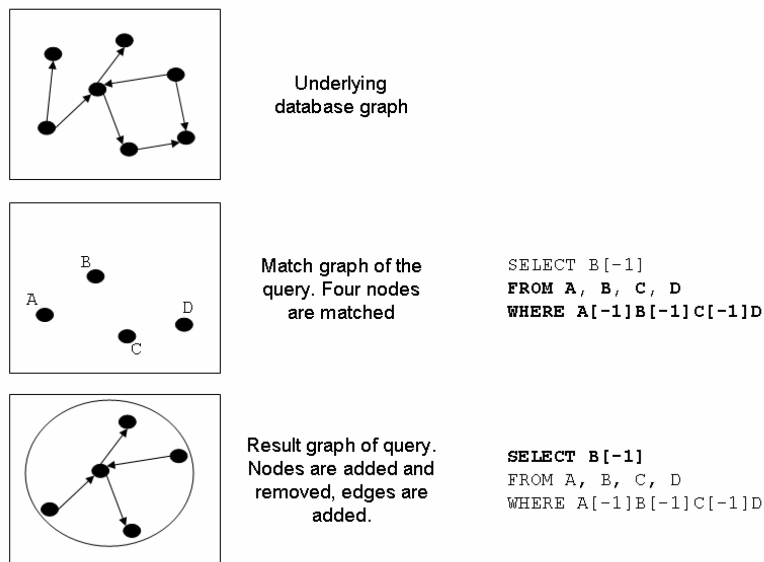


Figure 4. Evaluation of a PQL query happens in two stages. First, the match graph is computed using the node variables and the WHERE Clause. Next, the result graph is computed using the match graph and the SELECT clause. The syntax of the SELECT function is defined later.

We fulfill these requirements using a variety of different expressions in the `subgraph-specification`, i.e., the SELECT clause of a PQL query, called *select functions*. All select functions of a query together compute the *result graph*, i.e., the query result, from the match graph. The match graph depends on the underlying database, the node variables, and the conditions in the WHERE clause; select functions determine the result graph by adding nodes and edges to the match graph or by removing uninteresting bindings (see Figure 4).

As usually, we first define the syntax of the SELECT clause before we describe its semantics.

Definition 10. (SELECT clause, select functions)

A `subgraph-specification` of a PQL query P is a comma separated, non-empty list of the following elements, called *select functions*:

- '*'.
- Any node variable from P.
- An expression of the form '`A[- op n]B`', where either op is '=' and n is '*' or n is a positive integer and op is one of the following: '=', '>', '<'.
- An expression of the form '`A[-s]B`' or '`A[-l]B`' where s and l are exactly these constants.
- An expression of the form '`A[-n]`', where n is a positive integer greater than 0.

White spaces may be omitted. If S contains two expressions '`A[-` op_1 n_1`]B`' and '`B[-` op_2 n_2`]C`', this can be abbreviated with '`A[-` op_1 n_1`]B[-` op_2 n_2`]C`'.

■

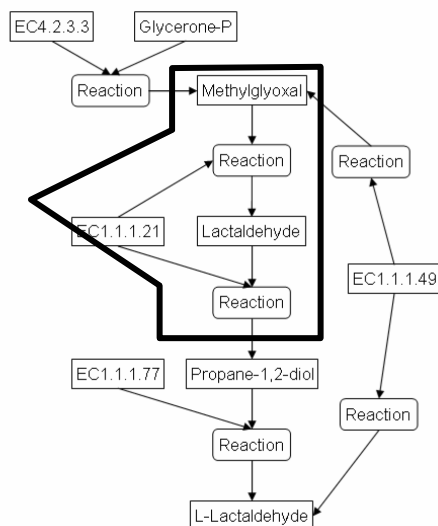


Figure 5. The marked area is computed by the left hand query given in the text. The edge between '`Lactaldehyde`' and the following reaction node is not returned.

Hence, the following are valid PQL queries:

```
SELECT    A[-2]
FROM      A, B
WHERE     A[-<5]B AND
          B.name='Propane-1,2-diol'
          AND A ISA 'enzyme'
```

```
SELECT   A, B
FROM     A, B
WHERE    A[-<5]B AND
         B.name='Propane-1,2-diol'
         AND A ISA 'enzyme'
```

The left query returns the vicinity of radius 2 (all nodes that can be reached from a node or that reach that node by a paths of length at most 2; formal definition below) around all bindings of A, where a path from enzyme A to node '`Propane-1,2-diol`' of length smaller than 5 must exist. Bindings of B are not returned, if they are not in the vicinity of radius 2 around bindings of A and if there is no path shorter than 5 from '`Propane-1,2-diol`' to itself, in which case both variable A and B would be bound to '`Propane-1,2-diol`'. Computing this query on the graph of Figure 1b) binds A to '`EC1.1.1.21`' (the other enzymes are more than 5 steps away from '`Propane-1,2-diol`'), and re-

turns the subgraph between the reaction node after ‘Methylglyoxal’ and ‘Propane-1,2-diol’ minus the edge between ‘Lactaldehyde’ and the following reaction node (see Figure 5). The right query returns a graph consisting of unconnected nodes; evaluating the query on the example from Figure 1b) yields the nodes ‘Propane-1,2-diol’ (bound to B) and ‘EC1.1.1.21’ (bound to A).

The semantics of the different select functions is precisely defined in the following.

Definition 11. (Vicinity)

Let v be a node of graph G=(V,E). The vicinity of v with radius k is the graph (V’, E’) with V’⊆V and E’⊆E, computed as follows. Initially, V’={v} and E’={}. Then,

- add all nodes to V’ reachable from v by a path shorter or equal in length than k; also, add all edges on the paths to E’,
- add all nodes to V’ for which a path to v exists that is shorter or equal in length than k; also, add all edges on the paths to E’,

■

Definition 12. (Result graph)

Let P be a PQL query evaluated on a graph G=(V,E) with subgraph-specification S and match graph M. The *result graph* G_P of P is constructed as follows. Initially, G_P is an empty graph. Then, nodes and edges from G are subsequently added to G_P according to the elements of S and the bindings of M. All elements of S are considered subsequently:

- If the element is a ‘*’, then all nodes in M are added to G_P.
- If the element is a node variable A, then all bindings of A in M are added to G_P.
- If the element is of the form ‘A[- op n]B’, then
  - If op is ’=’ and n is ‘*’, then all paths from any binding of A in M to any binding of B in M are added to G_P.
  - If n is a positive integer and op is one of the following: ‘=’, ‘>’, ‘<’, then all paths from any binding of A in M to any binding of B in M are added to G_P which have length equal to, less than, or greater than n (depending on op).
- If the element is of the form ‘A[-s]B’ (‘A[-l]B’), then the shortest (longest) path from any binding of A in M to any binding of B in M is added to G_P. If more than one shortest or longest path exists, they are all added to G_P.
- If the element is of the form ‘A[-n]’, then the vicinity of each binding of A in M of radius n is added to G_P.

Adding a path to G_P implies adding all edges and nodes on this path to G_P. Each element, node or edge, is at most added once to G_P. Once all expressions in S have been considered, G_P is returned as the result graph of P.

■

There is an important difference in the paths specified in the SELECT clause and those specified in the WHERE clause. The latter formulate existential conditions, while the former specify that all existing paths are included in the result graph. Consider the following example (see Figure 5):

```
SELECT A[-*]B
FROM A, B
WHERE A.name='EC1.1.1.49' AND B.name= 'L-Lactaldehyde' AND A[->4]B
```

A and B will be bound to nodes '`EC1.1.1.49`' and '`L-Lactaldehyde`' since there exists a path from A to B longer than 4 steps (including "`Methylglyoxal`"). The direct path, which is only two edges long, is not considered for the WHERE clause. But both paths between A and B are included in the result. It is currently not possibly to return only those paths that fulfill the path expressions in the WHERE clause[10].

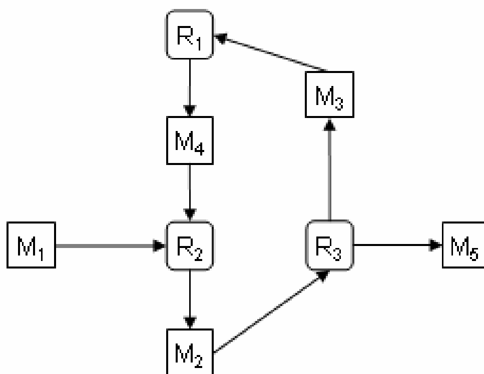


Figure 6. A network containing a cycle.

3.4 Networks with Cycles

So far we ignored cycles in the network, as we considered only cycle-free paths. However, biological networks often contain cycles, such as feedback and feed-forward loops [YSK+04][11]. Therefore, PQL queries must handle cycles in a defined manner. We first argue why we have forbidden cycles in paths. Later, we will analyze in detail the consequences of this design decision.

Consider the (cyclic) network given in Figure 6, and assume that paths are allowed to contain cycles. Since the cycle in the network can be traversed an arbitrary number of times in any path touching it, there are pairs of nodes between which an infinite number of paths exist, such as nodes M_1 and M_5. In consequence, the result of the following query on this graph is undefined using the current definition of the '`-*`' select function, as it would require the computation of an infinite number of paths (though there exists a finite representation):

```
SELECT A[-*]B
FROM A, B
WHERE A.name='M1' AND B.name='M5'
```

There are essentially two ways to deal with cycles in PQL.

[10] In the example, specifying '`A[-l]B`' in the SELECT clause would yield only the desired (longest) path, but this is no general solution.

[11] Note that all motifs (combinations of 2 to 4 proteins or genes having a specific connection pattern wrt. to protein-protein interaction and transcriptional regulation) described in this reference can be found using PQL using path expressions of length 2 and two different types of interaction nodes (interaction and regulation).

- Restrict paths to cycle-free paths as in Definition 7. Hence, graphs may contain cycles, but paths not. Using this approach, the above query is well-defined and returns the graph consisting of nodes M_1, R_2, M_2, R_3, and M_5, and all edges between these nodes. Not all paths are returned, though.
- Allow cycles in paths, i.e., change Definition 7 and adapt the definition of path expressions accordingly. For instance, the definition of the select function '`A[-*]B`' (Definition 12) could be rephrased into: "*Add to G_P the minimal number of edges and nodes such that all paths between A and B in G are also present on G_P.*". With this definition, the above query would return the entire graph of Figure 6. In the same manner, the definitions of the other select functions would need to be changed (with some caution required for conditions including length comparisons).

PQL adopts the first possibility for mainly two reasons:

- In any network with "too large" a number of cycles, many PQL queries containing select functions with path expressions would return an unreasonable large fraction of the network. With increasing number and length of cycles (more precisely: with an increasing number of nodes involved in at least one cycle), the chances rise that a path between any pair of nodes touches a node that is part of a cycle. In this case, the cycle and all its constituents immediately become a part of the result, although it might be completely uninteresting for the query at hand. However, most biological pathways are essentially cycles, especially if the fact is considered that biochemical reactions usually may happen in both directions. Any reversible reaction introduces a cycle.
- Cycle-free paths are much more efficient to deal with during the computation of a query result. For instance, the modified definition "*Add to G_P the minimal number of edges and nodes such that all paths between A and B in G are also present on G_P.*" is much more costly to fulfill than the original condition. The efficiency of our current implementation (see Section 4.5) depends on pre-computing all cycle-free paths in the network. This is impossible if cyclic paths are allowed because the number of paths is infinite even if only one cycle exists.

Hence, PQL operates on graphs with arbitrary cycles, but considers only cycle-free paths for query evaluation. We now analyze the impact of this design choice in more detail, first discussing path expressions in the WHERE clause and then path expressions in the SELECT clause.

3.4.1 Impact of Cycles on Path Expressions in the WHERE clause

Usually, path expressions in the WHERE clause are not affected by cycles, because they are by definition existentially qualified, i.e., they evaluate to TRUE if there exists at least one path obeying the given constraints. For any pair of nodes A and B with A≠B the following holds: If there exists a path between A and B including a cycle, there must also exist a path without a cycle. Hence, path expressions usually evaluate to the same result with or without cycles. The only exception are expressions of the form '`A[-*]A`'. Consider the graph in Figure 6 and the following query:

```
SELECT *
FROM A
WHERE A.name='M2' AND A[-*]A
```

The condition evaluates to TRUE if cyclic paths are allowed and to FALSE otherwise. Since cyclic paths are not allowed in PQL, the query – somewhat counter-intuitively – returns the empty graph. However, it is easy to reformulate the query to achieve the desired result. The following query finds the path $p=(M_2,R_3,M_3,R_1,M_4,R_2,M_2)$ and hence returns a non-empty result.

```
SELECT A
FROM A, B
WHERE A.name='M2' AND A[-*]B[-*]A
```

3.4.2 Impact of Cycles on Path Expressions in SELECT clause

Please note again the difference between the semantics of path expressions in the WHERE clause, which implicitly are existentially qualified, and the semantics of path expressions in the SELECT clause, which are intended to return all qualifying paths. Given the preceding discussion, this is achieved only partly. Path expressions in the SELECT clause only return qualifying cycle-free paths.

As in the previous section, there is a simple workaround. Consider the following query on the network in Figure 6:

```
SELECT A[-*]A
FROM A
WHERE A.name='M2'
```

This query returns the empty graph, because the only path from node M_2 to itself is cyclic. But the following query is semantically identical and perfectly computable in PQL:

```
SELECT A[-*]B[-*]A
FROM A, B
WHERE A.name='M2'
```

This example also shows that the result graph of a PQL query can contain cycles.

3.5 Note on the Complexity of PQL Queries

The *graph homomorphism problem* is the problem of finding a mapping m from the nodes of a graph G to the nodes of a graph G' such that for each edge (n_1, n_2) of G, there exists an edge (m(n_1), m(n_2)) in G'. The *graph isomorphism problem* is the problem of finding a mapping m from the nodes of a graph G into the nodes of a graph G' such that m and m^{-1} are homomorphisms. The *subgraph homomorphism problem* is the problem of, given two graphs G and G', finding a subgraph G'' of G' and a mapping m from G to G'' such that m is a homomorphism. Finally, the *subgraph isomorphism problem* is the problem of, given two graphs G and G', finding a subgraph G'' of G' and a mapping m from G to G'' such that m is an isomorphism.

Clearly, PQL can express subgraph homomorphism and subgraph isomorphism problems. Using path expressions, a query can describe an arbitrary graph and ask whether a subgraph in the database exists that is homomorph to the query graph[12]. Therefore, query evaluation is theoretically exponential in the size of the network. In general, all possible assignments of node variables of the query to nodes of the underlying database are enumerated. However, this is also true for SQL. Consider the following SQL query containing a self-join:

```
SELECT *
FROM sometable A1, sometable A2
```

If table 'sometable' has n tuples, then the query enumerates n*n result tuples. A query containing m self-joins of this form enumerates n^m tuples as a result. Thus, the complexity of SQL queries as well is exponential in the size of the database. This is in general not regarded as a problem, because

[12] Homomorphism problems can be turned into isomorphism problems by requiring that matching nodes are different, i.e., by adding conditions of the form "A != B".

"normal" queries add conditions on the result set that restrict the result size and that can be used for query optimization. We shall see in Section 4.5 that this also holds for PQL queries.

4 Exemplary Biological Queries

To evaluate whether or not PQL is sufficiently expressive for biological questions, we analyzed a number of publications on metabolic database with respect to the types of queries they mention as being crucial or typical. We describe queries taken from those publications and give their equivalent in PQL, if it exists. This should also help to better understand the strengths and limitations of PQL, and serves as a guideline for future extensions (see Section 7).

4.1 Queries from aMaze

The aMaze system "is a WorkBench for the representation, management, annotation and analysis of information on networks of cellular processes: genetic regulation, biochemical pathways, signal transductions"[13]. Underlying the aMaze system is a database for storing and retrieval biological processes, described in [vHNM00][14].

aMaze currently supports a number of simple queries, such as "*Retrieve compounds by name*" or "*Retrieve catalysed reactions by polypeptide*"[15] that are trivially expressed in PQL. Furthermore, [vHNM00] lists a number of more complex queries as being important for the analysis of biological networks. In the following, we quote some of these queries and study their formulation in PQL.

- *"Find all metabolic pathways that convert compound A into compound B in less than X steps."*

  In PQL we can formulate a query searching for all paths between compounds A and B within X steps. However, we cannot express the semantics of "converts". If "converts" is seen as a type of an interaction, one could think of a query searching for all paths between two compounds A and B such that all edges in this path are of type "convert". Such queries are not possible in PQL.

- "*Find all genes whose expression is directly or indirectly affected by a given compound.*"

  If we assume the meaning of "affect" as being identical to "interact", the question can be answered using the following PQL query:

  ```
  SELECT B
  FROM A, B
  WHERE A.name='L-Glutamate' AND A[-*]B and B ISA 'gene'
  ```

- "*In the complete set of metabolic reactions, find all feedback loops including a given compound.*"

  Assume the given compound is '`Methionine`'. Then the question can be answered by the following PQL query that computes all loops containing '`Methionine`':

[13] See http://www.amaze.ulb.ac.be/.
[14] Actually, our work was largely inspired by this project.
[15] See http://www.ebi.ac.uk/research/pfbp/.

```
SELECT A[-*]B[-*]A
FROM A, B
WHERE B.name='Methionine' AND A[-*]B[-*]A
```

- "*In a defined biochemical pathway, find all feedback loops*."

  We cannot restrict a query to a given pathway, since pathways are not defined in PQL. Apart from this, the question is a generalization of the previous question and can be formulated as the following PQL query, returning the graph containing all nodes and edges involved in a loop:

  ```
  SELECT A[-*]B[-*]A
  FROM A, B
  WHERE A[-*]B[-*]A
  ```

- "*Subgraph extraction. Here the user specifies a set of nodes (...) and prompts the system to extract the (…) sub-graph that interconnect each pair of seed nodes via the smallest number of individual links.*"

  PQL is capable of such queries if the number of seed nodes is fixed at query time. Note that the query does not ask for the minimal spanning tree of the set of nodes, something that PQL could not compute. Assume we have four seed nodes A, B, C, and D. Then, the question would be answered by the following query:

  ```
  SELECT A[-s]B, A[-s]C, A[-s]D, B[-s]C, B[-s]D, C[-s]D
  FROM A, B, C, D
  WHERE A[-*]B[-*]C[-*]D
  ```

  Note that the size of the query grows quadratic in the number of seeds, as one expression for each pair of seed nodes must be given in the SELECT clause.

- "*Find all processes that lead from node A to node B in less than Max steps, and more than Min steps*."

  This query cannot readily be expressed in PQL because it is not possible to give a maximum (N) and a minimum (M) on the length of paths between two nodes. The following query does not compute the desired result:

  ```
  SELECT A[-*]B
  FROM A, B
  WHERE A[->M]B AND A[-<N]
  ```

  Instead, the query computes all nodes A and B for which exists at least one path between them longer than M and at least one path shorter than N. Conceptually, the following query would be necessary (see also Section 7.1):

  ```
  SELECT A[-*]B
  FROM A, B
  WHERE A[->M]B
  IN (SELECT A[-*]B
      FROM A, B
      WHERE A[-<N]B)
  ```

4.2 Queries in TopNet

TopNet is an application for analyzing biological networks, especially in the context of gene expression studies. In [SHZ04], some typical queries are described. We skip the simple queries and discuss

only the following question: "(…) *We look for kinases that are directly connected to both Fus3 and Kss1. Fus3 and Kss1 must be connected via at most one additional protein to a transcription factor that regulates genes that are differentially expressed in the knockout experiment*".

Clearly, we cannot exactly express this query in PQL since PQL does not address gene expression intensities in its model[16]. Furthermore, we must reformulate the question, since TopNet has no explicit representation of reactions. Instead, molecules are directly connected via edges representing an interaction. We compensate for this difference by replacing each edge with an edge, a reaction node, and another edge. Then, the answer to this question can be computed as follows:

```
SELECT A[-*]B[-*]A
FROM A, B, C, D, E, F, G
WHERE A.name='Fus3' AND B.name='Kss1' AND
      ((A[-2]F[-2]C AND F ISA protein) OR (A[-2]C)) AND
      ((B[-2]G[-2]C AND G ISA protein) OR (B[-2]C) AND
      C ISA 'transcription-factor' AND C[-1]D[-1]E AND
      D ISA 'regulation' AND E ISA 'gene'
```

The query is possibly best explained by annotating the entities in the textual description with there variables: "Fus3 (A) and Kss1 (B) must be connected via at most one additional protein (F for A and G for B) to a transcription factor (C) that regulates (D) genes (E) …".

4.3 EcyCyc Queries

EcoCyc [KRS+02] and BioCyc are databases for storing genomic and metabolic information[17]. The following queries are examples of pathway computations taken from [Kar00]. In these questions, the term "enzyme" stands for an enzyme number and hence rather for a reaction than for a molecule. To fit our model, we replace each "enzyme" in a question with an enzyme (molecule) and an interaction.

- *"Find all enzymes for which ATP is an inhibitor."*

  The following PQL query finds the desired enzymes[18] (see Figure 7 a for illustration):

  ```
  SELECT A
  FROM A, B, C, D
  WHERE A ISA 'enzyme' AND D.name='ATP' AND A[-1]B AND D[-1]C[-1]B
        AND B ISA 'reaction' AND C ISA 'inhibition'
  ```

- "*Find all proteins that autophosphorylate*."

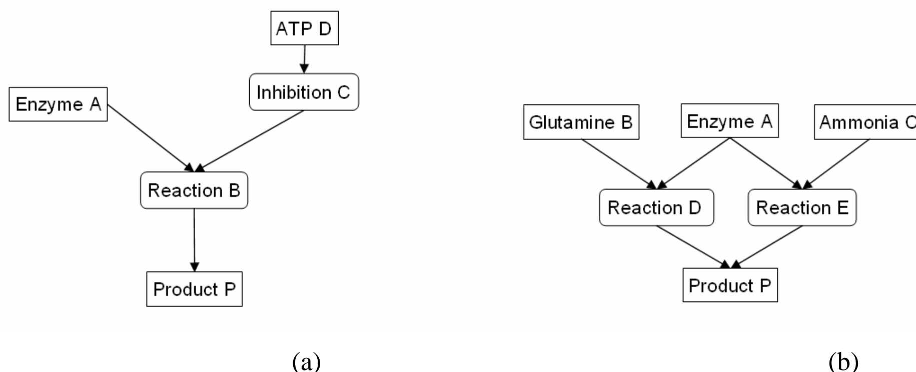
  We believe that this question has two meanings. First, the question could be meant to ask for all enzymes that can phosporylate itself. Second, we question could be meant to ask for all enzymes that can phosporylate itself and that change their behavior due to the phosphorylation. In PQL, we can express the first meaning but not the second. For expressing the latter, PQL would first need a way to differentiate between states of an enzyme (phosphorylated or not), and, second, a method to identify reactions as identical independent of their substrates and products. While the

[16] Although extending the model to specific types of nodes would not be too difficult.

[17] See http://biocyc.org/ and http://ecocyc.org/, respectively.

[18] Since enzymes are usually used synonymous for the reactions they catalyze, the original formulation is slightly ambiguous. We assume that we are looking for reactions catalyzed by an enzyme and inhibited by ATP. If we were looking for enzymes directly inhibited by ATP, this inhibition needs to be modeled in the graph as an interaction between ATP and the enzyme. Querying such subgraphs is trivial.

first is possible by using the TYPE hierarchy, the second cannot be expressed in PQL directly (see next example).



(a) (b)

Figure 7. (a) Enzymes that are inhibited by ATP. (b) Enzymes accepting ammonia or glutamate as substrate.

- "*Find all enzymes that accept glutamine and ammonia as alternative substrates*."

  Translated in our model, the question searches for reactions that are catalyzed by an enzyme and that additionally require either glutamine or ammonia as substrates. That query cannot be computed in PQL because there is no way to match reactions as being identical. We can formulate that two reactions have some identical substrates and products, but we cannot specify that the sets of substrates and products are identical. We explain the problem using an example. Assume we want to find all reactions that (a) produce a product P, (b) are catalyzed by an enzyme A, and (c) require as additional input either Glutamine (B) or Ammonia (C) (see Figure 7b). The following query answers this question:

```
SELECT A
FROM A, B, C, D, E, P
WHERE A ISA 'enzyme' AND D ISA 'reaction' AND E ISA 'reaction' AND
      B[-1]D AND A[-1]D AND A[-1]E AND C[-1]E AND D[-1]P AND E[-1]P
```

  However, this does not exclude reactions D and E from having additional inputs or outputs. This exclusion cannot be formulated in PQL. The only way to compute the desired results is to first execute the following query:

```
SELECT D[-1], E[-1]
FROM A, B, C, D, E, P
WHERE A ISA 'enzyme' AND D ISA 'reaction' AND E ISA 'reaction' AND
      B[-1]D AND A[-1]D AND A[-1]E AND C[-1]E AND D[-1]P AND E[-1]P
```

  After retrieving the complete result, the additional constraint must be enforce by the application program.

4.4 Queries according to Schaefer's survey

In a recent survey on pathway databases [Sch04], Schaefer lists a number of queries on pathway databases. These queries are especially interesting for two reasons. First, they highlight many queries that PQL cannot answer. Second, they nicely highlight a conceptual gap between queries in the sense of PQL and queries (or question) in the sense of a biologist. Note that PQL, like any query language,

is only a tool that can make analysis of biological network easier – but it cannot solve biological problems. We will discuss each example in detail.

- *"Retrieve a given pathway."*

  Such a query is not supported in PQL since pathways are not modeled (see Introduction of Chapter 2 and discussion in Section 7.2.

- *"Retrieval of a set of predefined pathways, merging sets of interactions and joining interactions on molecular identity."*

  PQL cannot answer this query because it does not know about pathways. Consider a similar query that would require the retrieval and union of a set of subgraphs, each defined by a PQL query. This requires the merging of two or more subgraphs into one. It is not immediately clear how this merging should look like:
    - One possibility is the union of the subgraphs, where common elements are merged.
    - A second possibility is the intersection of both subgraphs, in which case the result only contains those elements that are present in both subgraphs, for instance to compute the common elements in two orthologous pathways.
    - A third possibility computes the difference of both subgraphs – computing those elements of two orthologous pathways that are missing in the other one could be helpful for finding enzyme candidates not yet discovered in a genome.

  Each of these graph merging operations requires the identification of identical elements in different subgraphs. This operation needs to be exactly defined. For instance, using only the name of enzymes could merge two enzymes from different species; furthermore, it requires that all elements are named using the same convention, which is not the case if networks are combined from different data sources. Taken together, we see that an implementation of the above questions requires a number of concretizations to the semantics of the question.

- *"Retrieval of all interactions that involve any of a set of molecular species as immediate participant."*

  This can be expressed in PQL using the vicinity operation with a radius of 2 (radius 1 only retrieves nodes representing interactions, but not the interacting molecules).

- *"Retrieval of all successors and/or predecessors of a set of interactions."*

  Again, the vicinity operator answers such a query.

- *"Retrieval of a connected graph that includes a set of specified interactions."*

  Such queries cannot be computed using PQL because the order of elements in a path expression, and thus the structure of the matched subgraph, is fixed with the query. For instance, PQL cannot with one expression find all paths containing a set of more than two enzymes in arbitrary order, even if start node and end node are fixed. In Section 7.1 we discuss a way to include such conditions, including the computation of minimal spanning trees (the connected graph having the minimal number of edges) into PQL.

- *"Pruning molecular species from a set of interactions … to abstract the essence of a process by discarding details … [such as] ubiquitous cofactors …"*

  Clearly, such filter operations cannot be formulated in PQL. Two possible ways of approaching this functionality, one using the MINUS operation on graphs and one introducing an explicit filter function extending the SELECT clause, are presented in Section 7.1 and 7.2, respectively.

Requested functionality	**Expressible in PQL?**
Recursive path queries	Yes, using path expressions.
Path existence queries	Yes.
Transitive reduction Given a graph G, find the minimal subgraph G' of G such that the transitive closure of G' and G are identical	No.
Subgraph isomorphism and subgraph homomorphism	Yes (see Section 3.5).
Find connected components	No. PQL queries cannot address a variable number of nodes.
Shortest path between two nodes	Yes.
Graph difference, intersection, and union	No, but see Section 7.
Graph composition This is identical to graph union with a special kind of node matching.	No, but see Section 7.
Largest common subgraph of two graphs	No.
Least common ancestor Given N nodes, find the node K such that there are paths from K to any of those nodes and the sum of the lengths of those paths is minimal under all possible K.	No. PQL cannot even compute the least common ancestor for two nodes. One can construct a query that enumerates all common ancestors, but the minimality condition cannot be expressed, nor can PQL queries sort the result on the length of paths.
Neighborhood queries	Yes.
Queries with conditions on the number of paths between nodes Note: These are not mentioned by Olken, but necessary for the following type of queries.	No.
K-Core queries Given graph G, compute a subgraph G' of G such that all nodes in G' have a degree of at least K.	No. PQL cannot count paths (see previous query type), and PQL cannot address a variable number of nodes.
Frequent subgraphs Computes the most frequent subgraphs in a set of graphs.	No.

Table 1. Selected requirements for graph computations in biology taken from [OK04].

4.5 Olken's List of Graph Data Management Capabilities

The following is a list of graph queries taken from a web page on biological graph data management maintained by Frank Olken and Kevin Keck[19]. We have omitted queries we consider as graph analysis rather than graph queries (frequent subgraphs, graph majority, approximate graph matching, Clusters-of-Genes queries).

5 Implementation of PQL

We have implemented PQL on top of the relational database system Oracle Server V9.2[20]. The implementation consists of three parts: (a) a relational model for storing a PQL graph, (b) a module that performs certain pre-computations on the data to improve the performance of PQL queries, and (c) a compiler for PQL queries.

A PQL query is compiled into a PL/SQL stored procedure. This procedure finds all possible bindings given the conditions of the query and computes the result graph using the select functions in the SELECT clause. Since the result of a PQL query is a graph, it cannot simply be returned as the result of a procedure. We could encode the result into a single table[21] and return this table, i.e., implement the query as a table function[22], but table functions are currently only supported by commercial database systems, thus impeding the portability of PQL. Therefore, the result graph is currently stored in two tables. If PQL queries are used in client programs, we envisage that these tables are read and turned into a graph representation by the middleware.

5.1 Relational Data Model

The relational data model for storing a PQL graph is straight-forward and given in Figure 8. Nodes are stored in the `Node` table, edges in the `Edge` table. For each edge the two connected nodes are referenced using foreign keys; the direction is encoded in the names of the respective attributes. Annotations of nodes are stored in the `Function` and `Type` tables. `Type` is connected via a foreign key to `Node`, since each node may have only one type; the m:n relationship between `Node` and `Func` is stored in the bridge table `FuncNode`. Since both the type and the function hierarchies are DAG's, the structure of the concepts is encoded in two separate tables. The requirement that `TYP` and `FUNC` are acyclic is not enforced by the model, but must be ensured elsewhere, for instance by triggers.

---

[19] See http://pueblo.lbl.gov/~olken/graphdm/graphdm.htm.

[20] At the time of writing, the implementation was incomplete missing the following features: (1) Only AND may be used in the WHERE clause; OR, NOT, or parenthesis are not allowed. (1) ISA and HASFUNC are not implemented. (3) Complex path expressions are not recognized by the parser.

[21] The table could consist of two attributes `node_from_id` and `node_to_id`; tuples would represent edges and nodes in one, and unconnected nodes could be represented by a tuple with a `NULL` as second element.

[22] See Oracle documentation, for instance at: http://www.oracle.com/technology/documentation/database10g.html.

5.2 Pre-computation of Paths

Evaluating a PQL query has two phases: Computing the match graph and computing the result graph. Computing the match graph conceptually requires that assignments of node variables to nodes are enumerated and tested. We push as much as possible of the work into the database, i.e., express PQL operations using SQL operations. Conditions on node names and node IDs are efficiently supported by any relational database by using indices. Therefore, the efficiency of computing a match graph is dominated by two factors: (a) the cost of evaluating path expressions, i.e., finding and enumerating paths in the network, and (b) the cost of computing the ISA and HASFUNC operations, i.e. traversing a DAG from a node to the root. We here only discuss the first factor, since the second case – computing paths in DAGs – is a special case of the first case. To speed up the evaluation of path expressions, the PQL compiler assumes that information about all cycle-free paths in the graph has been stored in some helper tables before query execution starts. With these tables, path expressions and DAG operations can be answered in roughly logarithmic time[23]. In our implementation, this information is computed by a stored procedure.

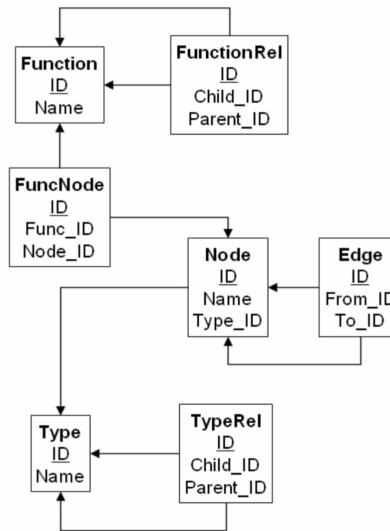


Figure 8. Basic relational data model. Boxes are tables, the bold name is the table name, and primary keys are underlined. Arrows indicate foreign-key relationships between tables.

Information about paths is stored in two tables: `Path` and `Connection`. `Connection` represents all directed connections of given length between nodes. If between two nodes several connections exist, each different length of the path forming the connection is represented by one tuple. Thus, not only the existence of a connection is stored – this would speed up only path expressions of the form '`A[-*]B`' – but also the length of the connection. Using the length, path expressions in the WHERE clause of the form '`A[->3]B`' or '`A[-4]B`' can be evaluated by a single lookup. `Connection` already contains more information than the transitive closure of the graph.

[23] We pre-compute all paths. Assuming an index on start and end node, testing for the existence of a particular path is then possible in O(p), where p is the number of paths in the graph. Of course, main memory issues must also be taken into account.

However, `Connection` does not help in computing select functions, as these always require the inclusion of all paths and its nodes into the result graph. To support those queries, the `Path` table stores all cycle free paths in the graph including all nodes they contain and the positions of each node in the path. Consider a query selecting the vicinity of a node variable A with radius r. All edges and nodes selected by this function can be computed quickly by computing the union of (a) all paths and their components starting in A and having length shorter than or equal to r and (b) all paths and their components ending in A and having length shorter than or equal to r.

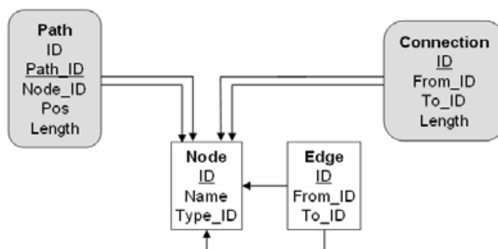


Figure 9. Path information is stored in two tables: Path and Connection.

Computing all cycle-free paths is achieved by iteratively computing paths of increasing length. The algorithm starts by computing paths of length 2 as combination of two edges (Recall that paths of length 1 are not possible in the PQL model; see Section 2.1). Given all paths of length n, if searches for all possible extensions of such a path by one further edge that does not introduce a cycle. Finding these extensions is achieved by a single SQL command, joining the `Edge` table with the `Path` table and filtering for the length of the paths and the cycle-freeness condition. The `Connection` table is generated from the final `Path` table with an additional SQL command.

Computing all paths is a rather costly operation, as the number of paths explodes with a growing number of edges. Consider for instance the graph in Figure 6. It contains 8 nodes, 8 edges, and 43 paths (36 in the cycle and 7 additional involving M_1 and M_5) – although in this graph there is only one (cycle-free) path for each pair of node. However, computing all paths is only necessary once for the database. Our current implementation, run on a laptop computer, computes all ~208.000 paths between the ~16.000 nodes and ~ 23.000 edged of the GeneOntology in approximately 5 minutes on a two processor database server[24] (see Section 6.2 for a short discussion of other algorithms). Computing 856092 paths (where longest path found had length 20) and 103040 connections for approximately 10% of the KEGG database (2552 nodes with 3660 edges) took about 70 minutes. Thus, precomputing paths is feasible for average sized networks. We are currently investigating faster methods for path computation to scale to larger networks.

5.3 The PQL Compiler

The PQL compiler takes a PQL query, parses it, and translates it into a PL/SQL procedure which computes the result graph of the query and stores it into two tables. We will not describe the compiler itself in detail, but concentrate on the generated code. This code has two sections, according to

[24] The GeneOntology originally consists of three isolated DAGs, representing ontologies for cellular locations, biological processes, and molecular functions, respectively. We connected them using a new root node. GO is cycle-free, but nodes are often reachable by multiple paths.

the two stages of evaluating a PQL query. In the first section, the match graph is computed and stored in a temporary table. In the second stage, this table is used to compute the result graph according to the SELECT clause.

Our current implementation has not yet been optimized in any way, but implements a straightforward translation of a PQL query into a series of SQL queries. We do not exploit relationships of path expressions in the SELECT and the WHERE clause, have not studied the optimal set of indexes[25], and have not yet thought about the optimal order of operations. Future work will consider optimizing PQL queries.

5.3.1 Phase 1: Computing the Match Graph

The match graph of a PQL query is the union of all assignments of nodes that appear in valid bindings of nodes to node variables. It only depends on the conditions in the WHERE clause. The match graph is computed in a number of steps.

In the first step, a temporary table is constructed and filled with all bindings (see Definition 6). Hence, the table stores all combinations of assignments of nodes to node variables that fulfill the WHERE clause. This table is computed by a single SQL query. We only consider the translation of path expressions, as HASFUNC and ISA conditions follow the same schema and other conditions are trivially transformed. If the PQL query contains complex path expressions, those are first broken into simple path expressions. Consider the following PQL query:

```
SELECT    *
FROM      A1, A2, …, An
WHERE     A1[-op1 n1]A2 AND A2[-op2 n2]A3 AND … AND An-1[-opn nn]An
```

The query is translated into the following SQL command computing all valid bindings:

```
SELECT    A1.node_id , A2.node_id …, An.node_id
FROM      node A1, node A2, …, node An
WHERE     EXISTS (SELECT id
                  FROM connection C
                  WHERE C.from_id=A1.id AND C.to_id=A2.id and C.len op1 n1) AND
          EXISTS …
```

The query joins two instances of the `node` table to the `Connection` table and checks whether the condition on the length of the connecting path are fulfilled. As a concrete example, consider the following PQL query:

```
SELECT    *
FROM      A,B,C
WHERE     A[-6]C AND A[-*]B AND a.name='spindle'
```

It is translated by the PQL compiler into the following SQL query:

[25] Note that this is a non-trivial question since many of the tables required for executing a PQL query have mixed read/write operations.

```
SELECT    a.id, b.id, c.id
FROM      node a, node b, node c
WHERE EXISTS (SELECT p.id
              FROM connection p
              WHERE p.from_id = a.id AND p.to_id = c.id AND p.length = 6) AND
      EXISTS (SELECT p.id
              FROM connection p
              WHERE p.from_id = a.id AND p.to_id = b.id) AND
      a.name = 'spindle'
```

Thus, the entire WHERE clause of a PQL query can be cast into a single SQL query on the `node` and `connection` tables using only joins and simple comparisons. This offers optimal conditions for the RDBMS query optimizer to find the best execution plan.

The table computed so far stores all bindings. Because this table has one column per node variable, it cannot be easily joined to other tables. Another temporary table storing exactly the match graph of the PQL query is created in the second step.

5.3.2 Phase 2: Computing the Result Graph

In the second phase, the match graph, stored in a temporary table, is used to compute all nodes and edges as required by select functions. For each select function, a piece of code is compiled into the stored procedure that fills the according nodes and edges into the result tables. Let `mnode` by the table storing the match graph, and let `rnode` and `redge` be the node and edge table of the result graph Select functions are implemented in the following way:

- A “*” simply copies all tuples from `mnode` to `rnode`.
- An select function of the form “A” , where A is a node variable, copies all values for this node variable from `mnode` to `rnode`.
- A select function containing a path first finds all paths from the start node to the end node obeying the conditions on the path length (if present). All nodes on any such path are inserted into `rnode`[26], and all edges on any such path are inserted into `redge`.
- A vicinity function is treated in a similar manner as a path function. First, all paths from the start node with a length within the desired radius are computed, and the appropriate nodes and edges are inserted into the result graph. In a second step, the same is performed for all paths ending at the start node having a suitable length.
- A shortest or longest path function is also treated as a path function with an additional condition on the path length.

All insertions are performed such that duplicates in the result graph tables are avoided. As a simple optimization, all single node select functions are ignored when a “*”is found in the SELECT clause, as this could only produce duplicates.

6 Related Work

To the best of our knowledge, this is the first suggestion for a query language directly targeted at biological networks. None of the pathway and interaction databases we have examined so far has a

[26] Note that this often inserts nodes that are not in `mnode`.

query language. With discuss the different areas of related work in separate sections. For an overview of applications and problems of graph management in biology, see the web page by Olken et al. [OK04].

6.1 Pathway Databases

BIND [BBH03], DIP [SMS+04], and IntAct [HMP+04] have only keyword search and no notion of paths or subgraphs. Reactome [JTV+03] and Kinase Pathway Database [LLT03] have an additional module for finding paths between two given nodes. In both cases it is not clear whether or not paths must lie entirely within one pathway or may cross pathways.

[SHZ04] describes the TopNet system for pathway visualization and analysis. The authors mention a XML based pathway query language, but give too few details for a decent comparison. Apparently, a network can be searched by nodes, their properties, and paths between the nodes. TopNet queries have no result construction phase, but return all maximal cliques of nodes that are matched by the query conditions. The exact power of the language remains unclear, and the authors do not mention cycles. Another commercial system for managing pathway data, but without any advanced methods for searching large pathway databases, is Pathway Assist from Ariadne Genomics[27]. Other tools for visualizing and editing pathways are e.g. Genmapp [SDS+03], PaVESy [LWSK04], and ProViz {Iragne, 2005 #1132}. None has any built-in query capabilities.

The pathway database system [KNO+03b; KNO+03a] is a complete system for modeling, visualizing, and editing pathways. It is based on hypergraphs, i.e., edges are reactions connecting sets of nodes (molecules). The data model is very rich, including pathways, species, reaction properties, etc. Reactions and pathways are assigned to a pathway, and queries always need to give the pathway on which they should be evaluated. The system supports canned queries for simple selection based on node or edge properties, simple paths (only two ends), and neighborhood queries. Queries can be posed using graphical user interface or as web service. They are always evaluated in memory, i.e., the entire database is loaded into main memory at startup [KNO+03b]. A similar system is PathDB[28], built around the ISYS system for application integration in the life sciences [SFT+01]. Queries are parameterized functions that do not go beyond the capabilities of the previously described system.

Finally, EcoCyc [KRS+02] is a pathway database based on a frame-based knowledge representation and reasoning systems. Arbitrary queries may be programmed in LISP, PERL, or Java, but no query language is provided.

6.2 Graph Databases

Graph databases have been studied in the database for many years. Despite this long period, no "standard" graph query language exists today. Güting proposed a data model and query language for graph data in spatial applications [Gue94]. The data model is an extension of an object-oriented model with special class types for links and paths. Queries are sequences of operations that match and select objects, links, and paths. The primary result of a query is a value, an object, or a list of objects. This result can be reformatted into the desired output by using a special "rewrite" operation.

[27] See http://www.ariadnegenomics.com.
[28] See http://www.ncgr.org/pathdb.

Query evaluation partly depends on pre-computed (and explicitly modeled) paths, but has also operators such as shortest path which compute paths at run time. However, the query language was only defined in fragments. Compared to PQL, GraphDB has a much richer data model and query language, but queries are very complex to formulate, and the additional possibilities of the data model do not seem to be useful for life science applications. Furthermore, PQL is readily available and based on (today) standard database systems, while no implementation is available for GraphDB.

A similar work to Güting is the system proposed by Sheng et al. [SOO99]. It also extends an object-oriented data model with special types for edges, paths, and graphs. The proposed query language is an extension of OQL with operations for asserting various conditions on paths, such as first or last node, path containment, and path composition. Queries are translated into O-algebra for query optimization. The authors describe a numbering scheme for paths based on Huffmann codes that essentially results in pre-computing all paths, but only works for rooted graphs. From the paper it is not clear how cycles in the graph are handled; given their main application area, i.e., multimedia databases, it is likely that cycles are not allowed. Furthermore, no implementation is available. In [AS92], an algebra for a query language with path expressions (called "hyperwalks") is defined. It is not clear how this algebra could be used for query optimization; further, there are subtle differences between hyperwalks and result graphs in PQL regarding the connectivity of the graph. GraphLog [CM90] is a visual graph query language whose expressions are translated into rules of a logic program. It allows a limited form of negation and recursion and is equally expressive as linear stratified DATALOG. Extensions exist for aggregation summarization along paths. Thus, GraphLog is considerable more expressive than PQL. However, GraphLog only returns tuples of matched nodes, not graphs, and cannot easily be implemented on top of a relational database.

Some commercial database management systems support network-type data. For instance, Oracle has recently released an extender for Network Data Management as part of the spatial extender [Ora03]. With this extender, networks of various types can be generated and managed. Access is either through a PL/SQL or a Java API. There is no notion of network queries, though analysis functions include neighborhood queries, shortest-path computation, and minimal spanning trees. In [DCES04], Oracle also reports on an experimental system for querying ontologies (read: DAGs) in the Oracle database. The system mainly consists of a set of user-defined functions and a special index structure for computing ancestral relationships. Similar to PQL, the implementation is based on the pre-computed transitive closure of the DAG. It is unclear whether their approach can be extended to graphs.

The current PQL implementation is based on a pre-computation of all paths in the graph using a rather simple algorithm. Faster algorithms for computing the transitive closure of graphs in a relational database are for instance described in [AJ87]. However, one must be careful whether or not these algorithms can treat cycles in the graph. Furthermore, for PQL it is not sufficient to compute the set of all connected nodes (hence the transitive closure), but we also need all paths connecting two nodes. Thus, many of the optimizations for transitive closure, which are based on pruning already existing connections, cannot be applied in our setting. However, we are investigating possibilities to adapt such algorithms to achieve faster pre-computation time in large graphs.

6.3 Semi-structured Data and XML

In [MMM97], Mendelzon at el. describe WebOQL, a language for queries in the World Wide Web. WebOQL allows regular expressions over links between web pages, which is similar to path expressions in PQL. Of course, the way of computing a query result with WebOQL is rather different from

our method given the nature of the Web. Regular path expressions are also used in many query languages for semi-structured data (see e.g. [Bun97]; the most prominent language today is XQuery), but these languages operate on trees (ordered or unordered), not graphs. This is also true for query optimization methods for semi-structured data, such as data guides [GW97]. Query languages for semi-structured data also introduced result restructuring, i.e., the ability to construct the result from the matched nodes [CDSS98]. Note that PQL does not allow for restructuring, as the result returned by a PQL query is always a subgraph of the database graph, though not necessarily identical to the match graph.

The GeneSeek (now BioMediator) system [MSH+02] uses a query language that is also named PQL. The language is a generalization of the StruQL query language for semi-structured data [FFLS97]. GeneSeek aims at developing a query system to allow integrated access to a set of dynamically changing biological data sources. These sources are modeled as a graph, where nodes represent the classes of the integrated databases and edges symbolize cross-references between these classes. Queries connect objects in classes following the edges. For this purpose, a query only needs to specify the start and end classes of a request, and the GeneSeek system will find all paths that connect these classes and that thus form potential ways to answer the query. GeneSeek allows to specify conditions on the paths using a form of logic formulas. However, GeneSeek uses graph traversal for query planning and only operates on metadata (schema information), while PQL actually models the data as a graph and operates directly on this graph.

Recently, the semantic web has drawn much attention. In the vision of the semantic web, resources such as web pages are semantically described using the resource description framework (RDF[29]). A single RDF annotation is a triplet of a resource, a property name, and a value, which can again be a resource. Schemas for RDF annotations are defined using RDFS, a rich, object-oriented data model including multiple inheritance of classes and properties. A set of RDF descriptions can be viewed as a graph with resources and values as nodes, properties as edges, and the vocabulary of the node and edge labels being determined by a RDFS schema[30]. Consequently, query languages for RDF need to include graph traversal facilities. One example of a RDF query language is RQL, a typed, fully closed functional query language in the spirit of OQL [KAC+02]. RQL has variables that range over class and property instantiations, a rich set of functions for traversing type hierarchies, nested queries, and aggregate functions over heterogeneous sets. However, the intention of RQL is completely different from PQL. RQL queries select objects and classes, while PQL queries select subgraphs. As a consequence, RQL has no notion of path expressions for traversing networks of linked object. The only points where graph traversal is embodied in the semantics of RQL are functions for traversing class hierarchies, such as `subClassOf`. In the current RQL implementation, such functions are evaluated on the transitive closure of the class model.

7 Possible Extensions to PQL

PQL is a proposal for a pathway query language that was designed for biological questions on network data. The functionality built into PQL was largely influenced by a review of papers that describe requirements for retrieval capabilities in this domain. However, there are many extensions

[29] See http://www.w3.org/RDF.

[30] A proper representation of RDF needs to employ a richer model; for instance, the possibility of specialization hierarchies over property types implies that specialization edges point to edge labels, not nodes.

which could enrich the power of PQL. In the following, we give a brief overview of important or reasonable extensions. The list is neither ordered in terms of importance nor complexity. We divide the list in three sections, one concentrating on adding functionality in terms of graph matching and analysis, one for extending the options for determining the result graph from the match graph, and one focusing on the addition of more biological concepets.

We excluded extensions that are geared towards usability of graph query languages. Undoubtedly, the most important such feature is visualization. Note that PQL queries are perfectly suited to be specified using a graphical tool.

7.1 Extending Graph Matching and Analysis Capabilities

PQL queries are evaluated on graphs and produce graphs. Therefore, PQL queries could be chained using set operators such as `UNION`, `MINUS`, or `INTERSECT`. For instance, the following is a suggestion for a query returning the entire network minus all nodes that are not enzymes, thus leaving only a network of interacting enzymes[31]:

```
SELECT A[-*]
FROM A
MINUS
SELECT A
FROM A
WHERE NOT( A ISA ENZYME)
```

When implementing this functionality, a couple of issues have to be considered. First, edges in the result of the first query have to be removed when the corresponding nodes are returned by the second query. Second, to preserve the connectivity structure of the graph, new edges have to be created replacing paths of which some nodes have been removed. Third, it has to be defined how nodes and edges are mapped onto each other when result graphs are merged by the set operation. This could be achieved by requiring a unique ID for each node; then, nodes are considered identical if their ID is identical, and edges are considered identical if they connect identical nodes. Alternatively, the user could specify the identifying attribute with the query, using a form such as "`MINUS (ON NAME)`". A very important biological application of such features are queries that compute all nodes and edges present in a certain pathway in one species but not in the same pathway in another species (see also next Section).

Due to the "graph in – graph out" property of PQL, views can be defined in a natural way. For instance, the above query could be encapsulated in a view:

```
CREATE PQLVIEW ENZYME AS
  SELECT A[-*]
  FROM A
  MINUS
  SELECT A
  FROM A
  WHERE NOT(A ISA ENZYME)
```

Using this view as the bases for a new PQL query would restrict this query to the network of only enzymes. Note that, if the current implementation is used, all such views have to be materialized since paths have to be pre-computed.

[31] The select function "`A[-*]`"is intended to return the entire database. This function is not yet included in PQL.

The introduction of path variables is another important extension to PQL. Path variables are variables that range over paths, not nodes. Path variables allow the definition of a new class of path conditions, including concurrent requirements on the minimum and maximum length, conditions stating that all nodes in a path must be of a certain type, or conditions requiring that paths contain a certain set of nodes in whatever order. Further extensions include negation (e.g. "compute all paths that do not contain a certain enzyme"), counting (e.g. "compute the number of paths between A and B"), and path expressions with FOR-ALL semantics (e.g. "compute all pair of nodes for which all paths connecting those nodes exceed a certain length").

7.2 Extending Result Construction (SELECT clause)

As became clear in Section 4.4, filtering of results is very important to allow concise, human-readable output. To this end, PQL should be extended by a filter function in the SELECT clause which, applied to a SELECT expression, removes from the result of the expression all elements as specified in the filter. For instance, the following query could compute the sub-network of the underlying graph consisting only of enzymes:

```
SELECT filterOn(A[-*], enzyme)
FROM A
```

Defining this function requires to solve the node and edge identify problem and the edge-path replacement problem (see above). Furthermore, one can imagine extending PQL with graph analysis functions implemented as plug-in procedures. Imagine a query such as:

```
SELECT span-tree(A,B)
FROM A, B
WHERE A.name='MEKK1' and B.name='MEK1'
```

This query would first compute the match graph of the query (binding nodes to molecules named "MEKK1" or "MEK1"), and then compute the minimal spanning tree in the graph that contains all bound nodes. All functions operating on a single graph and computing a single graph could be applied, including transitive reduction, K-core queries, or the finding of connected components of a graph. Caution must be taken that users are aware of the possible run-time implications.

7.3 Biological Properties

Biological objects are currently only described superficially in the PQL data model. Many attributes are missing, depending on the application in mind: position and promoter/inhibitor sequences of genes and for regulatory networks, biochemical properties of molecules for metabolic pathways, compartment and state information for signal transduction networks, links to phrases, sentences, and abstracts for literature networks, or 3D structural information for networks of atoms in protein structure research. Most of these pieces of information can be added easily without effects on the core model of PQL.

One particular important type of data missing in PQL is information about pathways, species, tissues, and cellular compartments. Reactions are often studied and described only in the context of a specific pathway that performs a certain function. Examples include glycolysis, the citrate cycle, the pentose phosphate pathway, and the sucrose metabolism. Pathways may involve hundreds of biological entities and reactions and entities are often contained in multiple pathways. Pathways may be species-specific or may occur in many different species, possible with slight derivation. Reactions

may occur only in certain tissues or in all tissues, and the existence and speed of reactions may also depend on the solution, i.e., to the physical environment. In the same manner, occurrence of biological entities such as mRNA or proteins may be species-, tissue-, or cell compartment specific. Being able to select entities and reactions based on the tissues they occur in or based on the pathway they are involved in is an important functionality for biological pathway queries. Introducing this functionality into PQL raises a number of questions that are subject to current research.

8 Conclusions

We have described PQL, a language for querying biological networks. The power of PQL comes from mainly three sources: The adoption of a native biological data model, the ability to define conditions on paths, and a variety of SELECT functions to give a detailed specification of the desired result constructed from the matching subgraphs. Taken together, these features allow for pathway queries that are much more expressive than those of existing systems.

Our analysis of requirements for pathway query languages has clearly shown that there is a strong need to define more clearly than possible with current languages the exact nature of a query against a biological network. Though we consider PQL as a step into this direction, our requirements analysis also revealed that further extensions are necessary. We discussed some of the extensions we plan for the near future. Since PQL is not restricted to pathway data, requirements stemming from neighboring fields, such as interaction networks extracted from the literature, should also be considered.

Our current focus for PQL is on the definition of an appropriate, expressive, and simple query language for querying biological networks. Not much effort has yet been invested into providing an efficient implementation for PQL. As stated in Section 5.2, our current implementation for path precomputation computes all paths in the GeneOntology in approximately 5 minutes, showing that precomputing is feasible for networks of this size. However, even if we would use faster algorithms for the pre-computation, we cannot pre-compute all paths in a network of all interactions between genes and proteins extracted from Medline, one of our envisaged applications. In such cases, evaluation of PQL queries needs to use recursive or hybrid methods for path enumeration.

Another line of necessary improvement is an efficient and complete implementation of PQL. The current implementation should only be considered as a prototype to support language definition and further development. It lacks a decent parser, implements only a subset of PQL, has not been optimized with respect to memory and time consumption, and must be equipped with a graphical interface for query specification and result visualization.

We hope that PQL is a starting point for a common effort to the development of query languages for biological network data. Repeating our plea from the introduction, we think that such a discussion is badly needed to support exchange of network data, allow for a concise specification of data to be extracted from a pathway database, and to reduce duplication of work in different systems. We believe that a declarative query language should be a natural component of any pathway database.

References

[AJ87] Agrawal, R. and Jagadish, H. V. (1987). "Direct Algorithms for Computing the Transitive Closure of Database Relations". 13th Conference on Very Large Databases, Brighton, UK. pp 255-266.

[AS92] Amann, B. and Scholl, M. (1992). "Gram: A Graph Data Model and Query Language". European Conference on Hypertext Technology, Milan, Italy. pp 201-211.

[BBH03] Bader, G. D., Betel, D. and Hogue, C. W. (2003). "BIND: the Biomolecular Interaction Network Database." *Nucleic Acids Res* **31**(1): 248-50.

[BHV02] Blaschke, C., Hirschman, L. and Valencia, A. (2002). "Information Extraction in Molecular Biology." *Briefings in Bioinformatics* **3**(2): 1-12.

[Bun97] Buneman, P. (1997). "Semistructured Data". 16th ACM Symposium on Principles of Database Systems, Tuscon, Arizona. pp 117-121.

[CDSS98] Cluet, S., Delobel, C., Simeon, J. and Smaga, K. (1998). "Your mediators need data conversion". SIGMOD, Seattle, Washington. pp 177-188.

[CM90] Consens, M. P. and Mendelzon, A. O. (1990). "GraphLog: a Visual Formalism for Real Life Recursion". ACM PODS. pp 404-416.

[CLRS03] Cormen, T. H., Leiserson, C. E., Rivest, D. L. and Stein, C. (2003). "Introduction to Algorithms". Cambridge, MIT Press.

[DCES04] Das, S., Chong, E., Eadon, G. and Srinivasan, J. (2004). "Supporting Ontology-based Semantic Matching in RDBMS". 30th Conference on Very Large Databases (VLDB04), Toronto, Canada. pp 1054-1065.

[SDS+03] Doniger, S. W., Salomonis, N., Dahlquist, K. D., Vranizan, K., Lawlor, S. C. and Conklin, B. R. (2003). "MAPPFinder: using Gene Ontology and GenMAPP to create a global gene-expression profile from microarray data." *Genome Biology* **4**(1-R7).

[FFLS97] Fernandez, M. F., Florescu, D., Levy, A. Y. and Suciu, D. (1997). "A Query Language for a Web-Site Management System." *SIGMOD Record* **26**(3): 4-11.

[FKKR01] Friedman, C., Kra, P., Yu, H., Krauthammer, M. and Rzhetsky, A. (2001). "GENIES: a natural-language processing system for the extraction of molecular pathways from journal articles." *Bioinformatics* **17 Suppl 1**: S74-82.

[GO01] GeneOntology Consortium, T. (2001). "Creating the gene ontology resource: design and implementation." *Genome Research* **11**(8): 1425-33.

[GW97] Goldman, R. and Widom, J. (1997). "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases". 23rd International Conference on Very Large Databases, Athens, Greece. pp 436-445.

[Gue94] Güting, R. H. (1994). "GraphDB: Modeling and Querying Graphs in Databases". 20th Conference on Very Large Databases, Santiago de Chile. pp 297-308.

[HPL05] Hakenberg, J., Plake, C. and Leser, U. (2005). "Optimizing Syntax Patterns for Discovering Protein-Protein Interactions". ACM Symposium on Applied Computing (SAC05), Santa Fe, US (to appear).

[HMP+04] Hermjakob, H., Montecchi-Palazzi, L., Lewington, C., Mudali, S., Kerrien, S., Orchard, S., Vingron, M., Roechert, B., Roepstorff, P., Valencia, A.*, et al.* (2004). "IntAct: an open source molecular interaction database." *Nucleic Acids Res* **32 Database issue**: D452-5.

[JLKH01] Jenssen, T. K., Laegreid, A., Komorowski, J. and Hovig, E. (2001). "A literature network of human genes for high-throughput analysis of gene expression." *Nat Genet* **28**(1): 21-8.

[JTA+00] Jeong, H., Tombor, B., Albert, R., Oltvai, Z. N. and Barabasi, A. L. (2000). "The large-scale organization of metabolic networks." *Nature* **407**(6804): 651-4.

[JTV+03] Joshi-Tope, G., Vastrik, I., Gopinath, G. R., Matthews, L., Schmidt, E., Gillespie, M., D'Eustachio, P., Jassal, B., Lewis, S., Wu, G., *et al.* (2003). "The Genome Knowledgebase: a resource for biologists and bioinformaticists." *Cold Spring Harb Symp Quant Biol* **68**: 237-43.

[KGK+04] Kanehisa, M., Goto, S., Kawashima, S., Okuno, Y. and Hattori, M. (2004). "The KEGG resource for deciphering the genome." *Nucleic Acids Res* **32 Database issue**: D277-80.

[Kar00] Karp, P. D. (2000). "An Ontology for Biological Function Based on Molecular Interactions." *Bioinformatics* **16**(3): 269-285.

[KRS+02] Karp, P. D., Riley, M., Saier, M., Paulsen, I. T., Collado-Vides, J., Paley, S. M., Pellegrini-Toole, A., Bonavides, C. and Gama-Castro, S. (2002). "The EcoCyc Database." *Nucleic Acids Res* **30**(1): 56-8.

[KAC+02] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D. and Scholl, M. (2002). "RQL: a declarative query language for RDF". WWW Conference, Honolulu, Hawaii, USA. pp 592-603.

[Kit01] Kitano, H., Ed. (2001). "Foundations of Systems Biology", MIT Press.

[LLT03] Koike, A., Kobayashi, Y. and Takagi, T. (2003). "Kinase pathway database: an integrated protein-kinase and NLP-based protein-interaction resource." *Genome Res* **13**(6A): 1231-43.

[KNO+03b] Krishnamurthy, L., Nadeau, J., Ozsoyoglu, G., Ozsoyoglu, M., Schaeffer, G., Tasan, M. and Xu, W. (2003). "Pathways database system: an integrated set of tools for biological pathways". ACM Symposium on Applied Computing (SAC). pp 96-102.

[KNO+03a] Krishnamurthy, L., Nadeau, J., Ozsoyoglu, G., Ozsoyoglu, M., Schaeffer, G., Tasan, M. and Xu, W. (2003). "Pathways database system: an integrated system for biological pathways." *Bioinformatics* **19**(8): 930-7.

[LS00] Legrain, P. and Selig, L. (2000). "Genome-wide protein interaction maps using two-hybrid systems." *FEBS Lett* **480**(1): 32-6.

[LWSK04] Ludemann, A., Weicht, D., Selbig, J. and Kopka, J. (2004). "PaVESy: Pathway Visualization and Editing System." *Bioinformatics* **20**(16): 2841-4.

[MXE01] Marcotte, E. M., Xenarios, I. and Eisenberg, D. (2001). "Mining literature for protein-protein interactions." *Bioinformatics* **17**(4): 359-63.

[MYC+02] Mellor, J. C., Yanai, I., Clodfelter, K. H., Mintseris, J. and DeLisi, C. (2002). "Predictome: a database of putative functional links between proteins." *Nucleic Acids Res* **30**(1): 306-9.

[MMM97] Mendelzon, A. O., Mihaila, G. A. and Milo, T. (1997). "Querying the World Wide Web." *Journal on Digital Libraries* **1**: 54-67.

[MSH+02] Mork, P., Shaker, R., Halevy, A. and Tarczy-Hornoch, P. (2002). "PQL: A Declarative Query Language over Dynamic Biological Schemata". Proceedings of the Annual Symposium of the American Medical Informatics Association (AMIA), San Antonio, US. pp 533-537.

[NK03] Nijssen, S. and Kok, J. (2003). "Efficient Frequent Query Discovery in FARMER". Proceedings of PKDD 2003, Cavtat, Croatia, Springer, LNAI 2838. pp 350-262.

[OK04] Olken, F. and Keck, K. (2004). "Biopathways Graph Data Manageer (BGDM)", Lawrence Berkely National Laboratory: Web page; see http://pueblo.lbl.gov/~olken/graphdm/graphdm.htm.

[Ora03] Oracle Corp. (2003). "Oracle Database 10g - Network Data Model".

[PRJ00] Proux, D., Rechenmann, F. and Julliard, L. (2000). "A pragmatic information extraction strategy for gathering data on genetic interactions." *Proc Int Conf Intell Syst Mol Biol* **8**: 279-85.

[RSR99] Rigaut, G., Shevchenko, A., Rutz, B., Wilm, M., Mann, M. and Seraphin, B. (1999). "A generic protein purification method for protein complex characterization and proteome exploration." *Nat Biotechnol* **17**(10): 1030-2.

[SMS+04] Salwinski, L., Miller, C. S., Smith, A. J., Pettit, F. K., Bowie, J. U. and Eisenberg, D. (2004). "The Database of Interacting Proteins: 2004 update." *Nucleic Acids Res* **32 Database issue**: D449-51.

[Sch04] Schaefer, C. F. (2004). "Pathway databases." *Ann N Y Acad Sci* **1020**: 77-91.

[SOO99] Sheng, L., Ozsoyoglu, M. and Ozsoyoglu, G. (1999). "A Graph Query Language and Its Query Processing". 15th International Conference on Data Engineering, Sydney, Australia. pp 572-581.

[SFT+01] Siepel, A., Farmer, A., Tolopko, A., Zhuang, M., Mendes, P., Beavis, W. and Sobral, B. (2001). "ISYS: A Decentralized, Component-Based Approach to the Integration of Heterogeneous Bioinformatics Resources." *Bioinformatics* **17**(1): 83-94.

[Sjo04] Sjolander, K. (2004). "Phylogenomic inference of protein molecular function: advances and challenges." *Bioinformatics* **20**(2): 170-9.

[SHZ04] Sohler, F., Hanisch, D. and Zimmer, R. (2004). "New methods for joint analysis of biological networks and expression data." *Bioinformatics* **20**(10): 1517-21.

[SP03] Stoeckert, C. J. and Parkinson, H. (2003). "The MGED ontology: a framework for describing functional genomic experiments." *Comparative and Functional Genomics* **4**(1): 127-132.

[vHNM00] van Helden, J., Naim, A., Mancuso, R., Eldridge, M., Wernisch, L., Gilbert, D. and Wodak, S. J. (2000). "Representing and analysing molecular and cellular function using the computer." *Biol Chem* **381**(9-10): 921-35.

[vMHJ+03] von Mering, C., Huynen, M., Jaeggi, D., Schmidt, S., Bork, P. and Snel, B. (2003). "STRING: a database of predicted functional associations between proteins." *Nucleic Acids Res* **31**(1): 258-61.

[YSK+04] Yeger-Lotem, E., Sattath, S., Kashtan, N., Itzkovitz, S., Milo, R., Pinter, R. Y., Alon, U. and Margalit, H. (2004). "Network motifs in integrated cellular networks of transcription-regulation and protein-protein interaction." *Proc Natl Acad Sci U S A* **101**(16): 5934-9.