Cost-based Optimization of Graph Queries

Silke Trißl
Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
trissl@informatik.hu-berlin.de

ABSTRACT

Many applications require efficient management and querying of graph structured data. For example, Systems Biology studies metabolic pathways and gene regulation networks modeled as directed graphs. These graphs consist of tens of thousands of molecules and interactions between them. To get a better understanding of these networks biologists need to query the networks and extract information.

In this paper we propose a framework for cost-based optimization of graph queries in relational database management systems. The result of graph queries is a set of subgraphs that is selected from the data graph based on conditions on nodes and paths posed in the query. We present the pathway query language as syntax to express graph queries. We show how to utilize techniques of classical cost-based query optimization to optimize graph queries. To handle graph specific predicates, such as the existence of paths, we propose new operators. In addition we describe two implementations of path operators in more detail and give an overview of future work.

1. INTRODUCTION

Graphs become more and more important in many areas such as management of XML documents with XPointers [19] or working in the semantic web, which builds on RDF, a graph-based data model and on graph-based query languages such as RQL [13]. In our own research we mostly work with data from the Life Science domain. In every living cell there exist complex mechanisms involving components, such as DNA, proteins, and chemical compounds that are responsible for the functioning of the cell. It is now commonly acknowledged that further progress in understanding the functioning of a cell can only be achieved if the interplay of these components, organized in networks, is understood [3].

In [10] van Helden and colleagues identified several important questions on biological networks. For instance, the question "find all processes that lead from node A to node B in less than max steps and more than min steps" or "find all genes whose expression is directly or indirectly affected by a given compound". If we assume that the given compound is *'glucose'* we can express the second question using the pathway query language (PQL) proposed by Leser in [15]. Figure 1 shows the PQL query for that question. The `FROM`-part specifies the queried network, the `LET`-part the node and path variables, and the `WHERE`-part places conditions on node and path variables.

```
SELECT B
FROM network
LET node A, node B, path P
WHERE B ISA gene
  AND A ISA compound
  AND A.name = 'glucose'
  AND P.path = A[-*]B
```

Figure 1: A PQL query to find all genes that are directly or indirectly affected by glucose.

Typical biological networks, such as gene regulation, protein-protein interaction [21], or metabolic networks [12] are currently in the range of tens of thousands of nodes. This number will increase in the future as more and more organisms are studied [2].

We assume that the graphs as well as all annotations for nodes and edges are stored in a relational database management system (RDBMS). Therefore, graph queries need to be executed inside a RDBMS. Mannino and Shapiro [17] gave an overview of different approaches to extend SQL to answer graph queries. Most of the presented approaches either rely on depth-first traversal of the graph or the precomputation of the transitive closure. Consens and Mendelzon [5] presented a graph query language that is inspired by formal grammar. The queries are evaluated using depth-first search. Computing the transitive closure or traverse the graph at query time only work for small graphs as we showed in [23]. Therefore, there is a need to find other methods to evaluate graph queries in a RDBMS.

In this paper we propose to use ideas from classical query optimization to optimize graph queries in a RDBMS. The reminder of this paper is organized as follows. In the next section we describe classical query optimization in RDBMS and its applicability to graph query optimization and related work. In Section 3 we describe two implementations of path operators and in Section 4 we discuss future steps. Section 5 concludes the paper.

Proceedings of SIGMOD2007 Ph.D. Workshop on Innovative Database Research 2007 (IDAR2007), June 10, 2007, Beijing, China.
.

2. QUERY OPTIMIZATION

The aim of query optimization is to minimize the time to execute a given query. Many relational database management systems use cost-based query optimization [9]. To optimize graph queries we use ideas and techniques of classical cost-based query optimization. We therefore first describe the classical query optimization in RDBMS and then explain how to use these techniques for graph query optimization.

2.1 Classical Query Optimization

Query optimization in RDBMS is well established, for reviews see [11, 4]. We will now give a short introduction to classical cost-based query optimization, which is displayed in Figure 2.

Given an SQL query we first have to parse the query to generate the *parse tree*. In the next two steps the actual query optimization is done. In the rewriting phase the initial parse tree is transformed into an equivalent, but hopefully more efficient parse tree, e.g., by resolving views or nested loops.

In the planning step the query planner uses the rewritten parse tree and searches for the optimal, i.e., cheapest query plan. To identify the cheapest plan the query planner first transforms the parse tree to all possible *logical query plans* using available operators of relational algebra given in the logical operator space. The logical query plan contains only operations, e.g., table access and join, but not the actual implementation of the operations. These are given by the physical plan space, providing for example different join methods, like the nested loop or sort-merge join. Using these implementations the query planner may produce several *physical query plans* for one logical query plan. The query planner can determine the cheapest plan by using the cost model, which contains a *cost function* for every implementation of available operators. A cost function takes the sizes of initial tables and intermediate results as well as data access methods into account and produces a cost estimate for every step of the physical query plan. The accumulated costs are the overall cost of the physical query plan.

The query planner returns the best physical query plan, which is then translated into executable code by the code generator. This code is given to the query processor that actually executes the query and returns the result.

These methods can also be applied for the optimization of graph queries in relational database systems as we will show in Section 2.3.

2.2 Related Work

Graph queries can be answered either by traversing the graph at query time or by using precomputed information. Leser proposed in [15] to evaluate a PQL query using precomputed information. To efficiently execute the queries they precompute and store all paths of a graph. As the number of paths grows exponentially in the number of nodes and edges the precomputation is only feasible for very small graphs. Typical biological networks, e.g., the metabolic network of KEGG [12], which contains about $15,000$ nodes, can not be treated that way but different strategies must be used. Eckman & Brown [8] proposed to query graphs using a commercial RDBMS. They stored the graph as data type, for which type-specific operators can be defined. This has the advantage, that the execution of the query is entirely done inside the RDBMS. But on the downside, they state that in their current implementation the graph must fit into main-memory, which makes it inapplicable for biological networks. Sohler & Zimmer [20] proposed ToPNet that allows to specify graph queries in XML format. To evaluate the query they use a main-memory based algorithm, which also makes this approach inapplicable to large biological networks.

Cost-based query optimization is also known in the context of XML query processing. McHugh & Widom [18] describe a framework for optimizing queries on XML documents. They proposed three different index structures that can be used to evaluate the existence of a path in an XML document. Based on the estimation of intermediate result sizes the optimizer should find an optimal physical query plan. Wu and colleagues showed in [25] that the dynamic programming strategies applied in classical query optimization to find the optimal physical query plan are not equally well applicable to optimize queries on XML documents using their proposed index structure. They proposed and evaluated four other algorithms based on dynamic programming that were adapted for the XML setting. These algorithms find a nearly optimal query plan by applying different heuristics during the enumeration of the different physical query plans.

2.3 Graph Query Optimization

In this section we describe how to use techniques from the classical query optimization to optimize graph queries. As we want to query graphs we first define graphs using notation from Corman et al. [6] and then describe the pathway query language [15]. A graph $G = (V, E)$ is a collection of nodes V and edges E. Given a graph G, a *path* p is a sequence of nodes that are connected by directed edges. The *length of a path* is the number of directed edges in the path. We assume that graphs are stored as a collection of nodes and edges in an RDBMS. The information on nodes includes a unique identifier and possibly additional annotation. Edges are stored as binary relationship between two nodes, i.e., as adjacency list.

The pathway query language (PQL) provides a syntax to query graphs and retrieve specified subgraphs. PQL consists of four different parts – a `FROM`-part that specifies the graph, a `LET`-part, where all node and path variables are defined, a `WHERE`-part that sets conditions on node and path variables, and a `SELECT`-part that specifies the shape of the returned subgraph. An exemplary PQL query is given in Figure 1.

Conditions in the `WHERE`-part are interpreted as a Boolean function. Conditions on node variables include restrictions on attributes and assigned concepts of nodes. Concepts of nodes are organized in hierarchies of concepts. Given an assignment of nodes in the network to a query variable `A` of type `node` every condition can evaluate to `TRUE` or `FALSE`. The nodes from the network for which all conditions of the node variable evaluate to `TRUE` are called *bindings* for the node variable. In analogy, given an assignment of paths in the network to the query variable `P` of type `path` every path in the network for which all conditions, e.g., start node, end node, or path length evaluate to `TRUE` are called *bindings* for the path variable.

To evaluate the graph query – as in the classical query optimization – we first parse the query and generate the parse tree. Using the parse tree we can create the logical query plan using operations defined in the logical plan space.

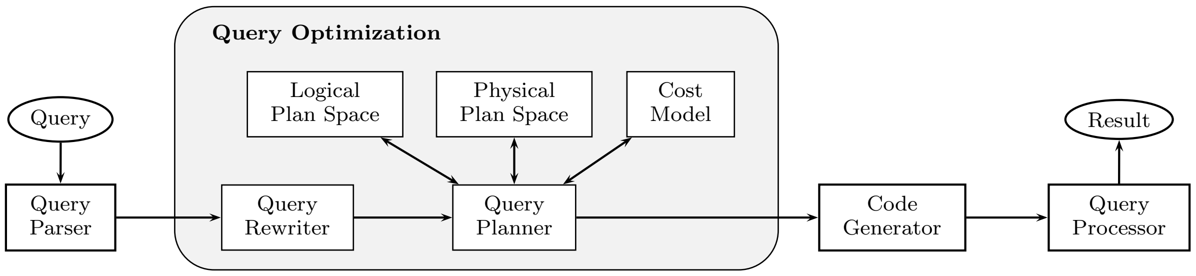


Figure 2: Cost-based query optimization in RDBMS.

As the logical plan space does not contain any operators for evaluation of node and path conditions we first have to define new operators in the logical plan space.

2.3.1 Logical Plan Space

Nodes. Node variables may contain conditions on attributes of nodes. Attributes of nodes in our model are stored together with the nodes in the node table. To evaluate the conditions on attributes of nodes we therefore can use the selection operator (σ) defined in the logical plan space from the classical query optimization.

Node variables can also have conditions on concepts assigned to nodes. The concepts are organized in a hierarchy of concepts, which forms a directed, acyclic graph (DAG) with a single root element. Every child concept is a specialization of its parent concept. For example, to describe nodes in biological networks we use a `TYPE` hierarchy, where below the root concept there are currently the concepts 'molecule' and 'interaction'. Below those, different types of molecules, such as 'gene','protein', or 'compound', and different types of interactions, such as 'inhibition', 'catalysis', or 'activation', are defined. Therefore, if we query for nodes that are associated with the concept 'molecule' we also have to return nodes that are associated with a successor concept of 'molecule' in the `TYPE` hierarchy.

For the evaluation of assigned concepts we have to define a new operator, the *hierarchy operator,* χ. The hierarchy operator takes as input restrictions on concepts of hierarchies and returns the selected concept(s) together with all its successor concepts.

Paths. In PQL we can specify conditions on the start node, end node and length of the path for a path variable. In the logical plan space of the classical query optimization there are no operations for the evaluation of the existence of paths. To generate the logical query plan we therefore have to specify the *path existence operator*, ϕ. This operator can be considered as a special form of a join operator. In the classical query optimization given two input sets, R and S, the join operator returns the set of pairs that can be formed. In analogy we can define the *path existence operator,* ϕ.

DEFINITION 1 (PATH EXISTENCE OPERATOR, ϕ). *Let* `A` *and* `B` *be two node variables and let* `P` *be a path variable in the query. Let* `P` *contain the condition that a path from* `A` *to* `B` *exists and let V be the set of nodes bound to* `A` *and W be the set of nodes bound to* `B`*.* `A` ϕ `B` *returns the set of node pairs (v, w) for which a path from $v \in V$ to $w \in W$ exists.*

In the same line we can define the *path length operator*, i.e., an operator that only returns a pair of nodes if there exists at least one path that is shorter, longer, or between given values.

Figure 3 shows a possible logical query plan for the PQL query from Figure 1 using the newly defined operators for graph queries. Note, this plan first determines the bindings of both node variables by applying the selection operator on the node table and the hierarchy operator on the `TYPE` hierarchy. The resulting bindings of node variables are then used as input for the path existence operator.

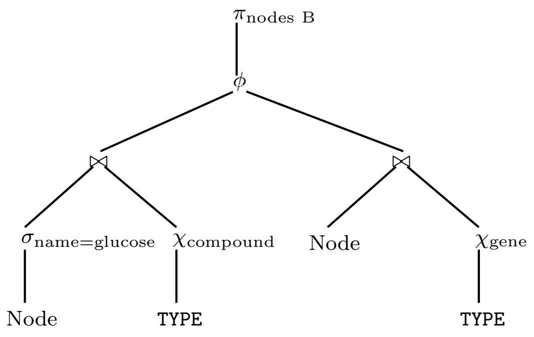


Figure 3: The logical query plan for the query given in Figure 1

2.3.2 Physical Plan Space

The physical plan space contains the actual implementation of logical operators described in the last section.

The hierarchy operator, χ. The hierarchy operator selects a concept and all successors of that concept in a hierarchy. To retrieve the selected concept we can use the implemented selection operator. But to retrieve all successors of a concept we either have to traverse the hierarchy of concepts using depth- or breadth-first search [6] or use precomputed information, such as the transitive closure that stores all successors of a concept [1] or pre-/ postorder labeling [22].

The path existence operator, ϕ. Given bindings of nodes in the graph to node variables in the query the path existence operator returns the set of node pairs for which a path exists. We can evaluate the existence of paths by applying either

- depth-first recursive search,
- breadth-first recursive search,
- index-based search using GRIPP, or
- index-based search using other techniques.

For large graphs we prefer index-based searches over recursive search strategies as these require time linear in the number of nodes and edges of the graph. For a thorough evaluation see [23]. To evaluate reachability queries efficiently on large graphs we developed GRIPP (GRaph Indexing based on Pre- and Postorder numbering) [24], which we will present in the next section as one physical implementation of the path existence operator.

3. IMPLEMENTING A PATH OPERATOR

The path existence operator requires as input bindings of nodes from the graph to node variables in the query. The operator returns the set of node pairs for which a path exists. Given two nodes v and w, the path existence operator evaluates the function $reach(v, w)$. This function returns `TRUE` if a path between the nodes exists, and otherwise `FALSE`.

We developed GRIPP (GRaph Indexing based on Pre- and Postorder numbering) [24] to answer such queries efficiently. In the following we explain briefly how to create the GRIPP index and show how to efficiently answer reachability queries for a single pair of nodes. We then propose a method to answer set-oriented queries, i.e., given a node v and a set of nodes W, which nodes $w \in W$ are reachable from v.

3.1 GRIPP - Graph Indexing

The GRIPP indexing strategy is based on the pre- and postorder labeling, which was originally described for tree structured data [7]. In the pre- and postorder numbering scheme each node in the tree receives a pre- and postorder value. Both values are assigned according to the order in which the nodes are visited during a depth-first traversal of the tree. The preorder value v_{pre} is assigned as soon as node v is encountered during the traversal. The postorder value v_{post} is assigned after all successor nodes of v have been traversed.

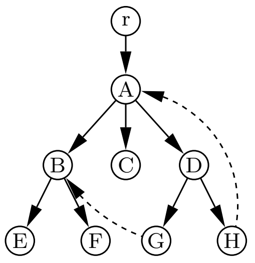
A table of all nodes with their assigned pre- and postorder values forms an index with which reachability queries in trees can be answered with a single query. If w is reachable from v, w must have a higher preorder and lower postorder value than v. If we use only one counter for the pre- and postorder values all successor nodes w of v must lie within the borders given by the pre- and postorder values of v, i.e., $[v_{pre}, v_{post}]$. This condition can be evaluated in an RDBMS with a single query.

As soon as nodes have more than one incoming edge this technique does not work anymore. We therefore developed GRIPP to index cyclic, possibly unrooted graphs [24].

To create the GRIPP index we perform a depth-first search over a graph G and a given order of child nodes. During that traversal every node in G receives at least one pre- and postorder value. The node together with the pre- and postorder value and instance type is stored as *instance* in the index table $IND(G)$. Figure 4 shows a graph G and the resulting index table $IND(G)$. If node v is traversed for the first time, we create a *tree instance* of v in $IND(G)$ and traverse child nodes of v. For any successive traversal of v we add a *non-tree instance* in $IND(G)$ and do not traverse the child nodes of v in G.

The GRIPP index structure resembles a rooted tree, the *order tree,* $O(G)$. In $O(G)$ every non-tree instance is a leaf node, while tree instances can be inner or leaf nodes. The order tree $O(G)$ for the index table $IND(G)$ from Figure 4 is displayed in Figure 5.

In GRIPP every node has as many instances in $IND(G)$ as this node has incoming edges in G, i.e., we have as many instances in $IND(G)$ as we have edges in G. In [24] we created the GRIPP index for generated random and scale-free graphs with up to 5 million nodes and 10 million edges. For



(a) Graph, G.

node	pre	post	type
r	0	21	tree
A	1	20	tree
B	2	7	tree
E	3	4	tree
F	5	6	tree
C	8	9	tree
D	10	19	tree
G	11	14	tree
B	12	13	non-tree
H	15	18	tree
A	16	17	non-tree

(b) Index table, $IND(G)$.

Figure 4: Graph G and its GRIPP index table $IND(G)$. Solid lines represent tree edges, dashed lines are non-tree edges.

example generating the GRIPP index structure for graphs with 50,000 nodes and 100,000 edges takes less than 130 seconds, which is faster than all competing methods.

3.2 Querying GRIPP – Single Node Pairs

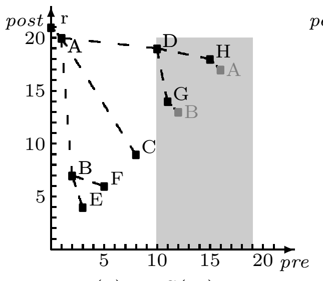
We will now depict how to efficiently answer reachability queries for a fixed pair of nodes bound to node variables in the query using GRIPP. Recall that reachability queries in trees can be answered with a single lookup because all reachable nodes of a node v have a preorder value that is contained within the borders given by v_{pre} and v_{post}. When querying the GRIPP index structure this way we face two problems. First, a node v may have many instances v' in $IND(G)$. But every non-tree instance of v in $IND(G)$ is a leaf node in $O(G)$ and has no successors in $O(G)$. We therefore always use the tree instance of a node for querying. The second problem is that in the preorder range (also called *reachable instance set* of v, $RIS(v)$) of a tree instance v' we will only find instances of nodes that are reachable from v' in $O(G)$. Nodes reachable from v in G but not from v' in $O(G)$ will be missed, as during index creation we do not traverse child nodes when we reach a node over a non-tree edge. We only insert a non-tree instance in $IND(G)$ and therefore successor of non-tree instances might not have an instance in $RIS(v)$. To account for that we have to extend the search using the *hop technique*. To find all reachable nodes of v in G we basically perform a depth-first search over the index structure using non-tree instances in reachable instance sets. In Figure 5(a) $RIS(D)$ contains non-tree instances of nodes B and A, i.e., both are hop nodes for D.

To make the search more efficient we developed three pruning strategies, namely the simple, the skip, and the stop strategy. Using the simple and skip pruning strategy we avoid posing queries for preorder ranges which we have already checked. During the search we keep a list U of all nodes that have been used to retrieve a reachable instance set. Now assume we have found a new hop node h. If that new hop node is equal to or successor of a previously used hop node we apply the simple pruning strategy and prune that node from being used as new hop node ($RIS(h)$ has previously been retrieved). Otherwise, if hop node h is ancestor to used hop nodes, we do not want to consider the range of the used hop nodes again, we therefore skip those ranges using the skip strategy. In the case where h is sibling to all nodes in U we have to retrieve the entire reachable instance of h.

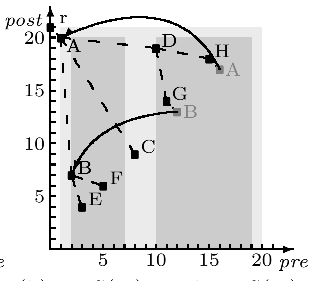
The stop strategy requires some additionally precomputed values, the stop nodes. A stop node s is a node in G for which for every non-tree instance in $RIS(s)$ there exists a corresponding tree instance in the set. This means, that all nodes reachable from s in G are reachable from s' in $O(G)$. As soon as we reach a stop node during the search we can immediately return without checking every hop node in $RIS(s)$. In Figure 5(a) nodes r, A, B, E, F, and C are stop nodes.

To answer $reach(v, w)$ we proceed as follows. We first find the tree instance v' of v and retrieve $RIS(v)$. If $w \in RIS(v)$ we finish and return TRUE, otherwise we have to extend the search using non-tree instances in $RIS(v)$ (preferably a non-tree instance of a stop node). We extend the search until we find an instance of w or no further usable hop nodes are available.

Consider Figure 5(b) and $reach(D, r)$. We find non-tree instances of nodes B and A in $RIS(D)$ and first use node B as hop node. As $RIS(B)$ does not contain the an instance of r or further hop nodes we proceed with node A as hop node. As A is a stop node we do not have to consider the non-tree instances in $RIS(A)$, we only have to check if $r \in RIS(A)$. As this is not the case $reach(D, r) =$ FALSE.



(a) $RIS(D)$



(b) $RIS(D)$ and $RIS(B)$ in dark gray; $RIS(A)$ in light gray.

Figure 5: The example shows $reach(D, r)$ evaluated on the GRIPP index structure from Figure 4(b). Nodes A and B are hop nodes for D.

3.3 Querying GRIPP – Set-Oriented

To apply the GRIPP query strategies for PQL we also have to consider set-oriented operations. So far, we only answered $reach(v, w)$ for single bindings of nodes v and w to node variables in the query. But in a typical PQL query several nodes from G may be bound to the node variables. Therefore, we also require an implementation of the path existence operator for set-oriented operations. We start with the following, given the binding of node v to one node variable and a set of bindings W to the other node variable, find all nodes in W that are reachable from v.

An obvious implementation would be to query the GRIPP index structure $|W|$ times for $reach(v, w)$ with $w \in W$. This strategy requires time linear in the number of nodes in W.

A different approach is to implement an additional search strategy for sets of nodes. Recall, in the search strategy for single node pairs, i.e., $reach(v, w)$, we stop extending the search if we find w in a reachable instance set or if there are no further usable hop nodes. For set-oriented operations we would have to check which nodes $w \in W$ are contained in $RIS(h)$. We could only stop if all nodes in W are reached or no further usable hop nodes exist. As some nodes in W may not be reached from v we have to explore the entire search space. As this is time consuming we use a different strategy.

For the implementation of set-oriented querying we proceed as follows. We first select the tree instance of v, retrieve $RIS(v)$ and proceed the search using hop nodes. Here as well we apply the pruning strategies described in the last section. We stop the search when no further usable hop nodes are available. During the search we store the pre- and postorder ranges of all nodes that have been used to retrieve a reachable instance set in a list U. When we have finished the search we use this list U to determine which nodes $w \in W$ have an instance with a preorder value that is contained in the range of at least one node $u \in U$. Using an RDBMS we can implement this as join operation between the GRIPP index table (for the instances of nodes in W) and instances in U. We experimentally evaluate both query strategies to answer $reach(v, W)$, which we present in the following.

3.3.1 Experimental Evaluation

To experimentally evaluate both approaches we use a generated random graph with 10,000 nodes and 20,000 edges and evaluated $reach(v, W)$ for different sizes of W. We implemented both query strategies for GRIPP as stored procedures in a commercial object-relational database system. Tests were performed on a DELL dual Xeon machine with 4 GB RAM. Queries were run without rebooting the database. The query times for $reach(v, W)$ are averaged over 1,000 queries, with a randomly selected node v and n randomly selected nodes w. The number of nodes in set W is given as parameter n.

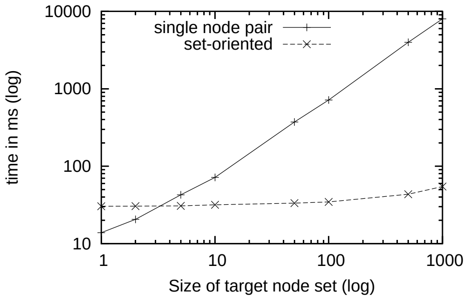


Figure 6: Average query time for both strategies and increasing size of set W.

Figure 6 shows the average query times for the single node pair strategy and the set-oriented strategy to evaluate $reach(v, W)$. The size of the set of nodes W ranges from 1 to 1,000 nodes. The figure shows that the average query times for the set-oriented approach is almost constant over different sizes of the graph. In contrast, the query time for the single node pair approach grows linear with the number of nodes in set W. The figure also shows that for very small sizes of W, i.e., up to three nodes, querying GRIPP using the single node pair query strategy is advantageous. The reason is that for those searches we can immediately stop if we find node w in a reachable instance set. In contrast, for the set-based search we have to explore the entire space and can only stop when no further usable hop nodes are available. For few nodes in set W the set-oriented approach therefore might explore a larger search space than necessary.

4. FUTURE WORK

So far, we only considered the logical plan space and some implementations of operators in the physical plan space. We still need to address two major issues, which are evaluating path and path length queries and assigning cost functions to implementations of operators.

4.1 Path and Path Length Queries

Using GRIPP we can evaluate $reach(v, W)$ very efficiently. But as soon as we pose conditions on the length of the path or even on path properties, e.g., the containment of a node in the path, evaluating that condition using GRIPP is not very efficient as we evaluated in [23]. Other indexing strategies such as computing the transitive closure [1] or 2-Hop-Cover [19] with distance information only works for small graphs. As the networks under consideration are usually large these index structures are not generally applicable. We therefore have to find new indexing and query strategies to evaluate path and path length queries.

4.2 Cost Functions

Given the implementation of operators we have to assign a cost function to every implementation. A cost function usually takes as input parameters the size of the input and sizes of intermediate results. So far, we have not assigned a cost function to the search strategies for GRIPP, but this will be the next step.

In addition to assigning cost functions to operators we also must estimate the result size of the newly defined operators. For the hierarchy operator we can use techniques described in [14] for data warehousing, as in that area hierarchy queries are also required. To estimate the result size of the path operator we can apply sampling techniques as described in [16].

5. CONCLUSION

Our aim is to implement a system that uses a PQL query as input, parses the query, applies cost-based query optimization to execute the query efficiently, and then displays the result graphically. We believe that optimizing graph queries is not just needed for biological networks, but also in other areas. As several other graphs, such as social networks or web graphs exhibit the same structure as biological networks [2] this work can easily be adapted. In this paper we proposed a framework for cost-based optimization of graph queries expressed in PQL. We gave details on implementations of some operators for graph queries. Some issues still remain open such as assigning cost functions or finding more efficient strategies to answer path length and path queries, which we will address in future work.

6. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proc. of the ACM SIGMOD Conference*, pages 253–262, 1989. ACM Press.

[2] A.-L. Barabąsi and Z. N. Oltvai. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics*, 5(2):101–113, Feb 2004.

[3] I. Borodina and J. Nielsen. From genomes to in silico cells via metabolic networks. *Current Opinion in Biotechnology*, 16(3):350–355, Jun 2005.

[4] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proc. of the PODS Conference*, pages 34–43, 1998. ACM Press.

[5] M.P. Consens and A.O. Mendelzon. GraphLog: a Visual Formalism for Real Life Recursion. In *Proc. of the PODS Conference*, pages 404–416, 1990. ACM Press.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 2001.

[7] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the STOC Conference*, pages 365–372, 1987. ACM Press.

[8] B. Eckman and P. G. Brown Graph data management for molecular and cell biology. *IBM J. Res & Dev.*, 50(6):545 – 560, Nov 2006.

[9] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[10] J. van Helden, A. Naim, R. Mancuso, M. Eldridge, et al. Representing and analysing molecular and cellular function using the computer. *Journal of Biological Chemistry*, 381(9-10):921–935, 2000.

[11] Y. E. Ioannidis. Query Optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[12] M. Kanehisa, S. Goto, S. Kavashima, Y. Okuno, and M. Hattori. The KEGG resource for deciphering the genome. *Nucleic Acids Research*, 32:D277–D280, 2004.

[13] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF, 2002. In *Proc. of the WWW Conference*, 2002.

[14] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal Histograms for Hierarchical Range Queries. In *Proc. of the PODS Conference*, pages 196–204, 2000. ACM Press.

[15] U. Leser. A query language for biological networks. *Bioinformatics*, 21 Suppl 2:ii33–ii39, Sep 2005.

[16] R. J. Lipton and J. F. Naughton. Query Size Estimation by Adaptive Sampling. In *Proc. of the PODS Conference* pages 40–46, 1990. ACM Press.

[17] M.V. Mannino and L.D. Shapiro. Extensions to Query Languages for Graph Traversal Problems. *IEEE Trans. Knowl. Data Eng.*, 2:353–363, 1990.

[18] J. McHugh and J. Widom. Query Optimization for XML. In *Proc. of the VLDB Conference*, pages 315–326, 1999. Morgan Kaufmann.

[19] R. Schenkel, A. Theobald, and G. Weikum. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proc. of the ICDE Conference*, pages 360–371, 2005. IEEE Computer Society.

[20] F. Sohler and R. Zimmer. Identifying active transcription factors and kinases from expression data using pathway queries. *Bioinformatics*, 21 Suppl 2:ii115-ii122, Sep 2005.

[21] U. Stelzl, U. Worm, M. Lalowski, Ch. Haenig, F. H. Brembeck, et al. A human protein-protein interaction network: a resource for annotating the proteome. *Cell*, 122(6):957–968, Sep 2005.

[22] S. Trißl and U. Leser. Querying Ontologies in Relational Database Systems. In *Proc. of the DILS*, volume 3615 of *Lecture Notes in Computer Science*, pages 63–79, 2005. Springer.

[23] S. Trißl and U. Leser. GRIPP - Indexing and Querying Graphs based on Pre- and Postorder Numbering. Technical Report No. 207, Humboldt-Universität zu Berlin, 2006.

[24] S. Trißl and U. Leser. Fast and Practical Indexing and Querying of Very Large Graphs. In *Proc. of the ACM SIGMOD Conference*, *to appear*, 2007. ACM Press.

[25] V. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proc. of the ICDE Conference*, pages 443–454, 2003. IEEE Computer Society.