LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-400310

Implementing Graph Pattern Queries on a Relational Database

Ian L. Kaplan, Ghaleb M. Abdulla, S Terry Brugger, Scott R. Kohn

January 8, 2008

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Graph Query Language : Implementing Graph Pattern Queries on a Relational Database

This page last changed on Jan 11, 2008 by kaplan4.

LLNL technical report LLNL-TR-400310. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Funding provided by LDRD 06-ERD-009.

Abstract

When a graph database is implemented on top of a relational database, queries in the graph query language are translated into relational SQL queries. Graph pattern queries are an important feature of a graph query language. Translating graph pattern queries into single SQL statements results in very poor query performance. By taking into account the pattern query structure and generating multiple SQL statements, pattern query performance can be dramatically improved. The performance problems encountered with the single SQL statements generated for pattern queries reflects a problem in the SQL query planner and optimizer. Addressing this problem would allow relational databases to better support semantic graph databases. Relational database systems that provide good support for graph databases may also be more flexible platforms for data warehouses.

Introduction

A semantic graph database is a database made up of typed graph vertices and typed edges that connected these vertices. For example, a graph database could include vertices with the type `person` or `organization`. An example of a person vertex might be {name=*Thomas J. Watson*, birthdate = 1914-01-14 }. This instance of a person vertex in the graph could be connected by an `EmployedBy` edge to the `organization` vertex *International Business Machines* and a `BornIn` edge to the city `Dayton, Ohio`. As this example demonstrates, a vertex in the graph database may have multiple attributes (e.g., name and birth date). Edges may, optionally, have attribute values as well.

A graph query language is a query language designed for a graph database. When a graph database is implemented on top of a relational database, queries in the graph query language are translated into relational SQL queries [1]. Some graph query operations can be efficiently implemented by translating the graph query into a single SQL statement. Examples include adjacency and graph filter queries. The adjacency function adds a set of selected adjacent links and vertices to a selected subgraph. The graph filter function filters a graph to return a subgraph consisting of only selected vertices or links. Graph pattern queries are an important feature of a graph query language. In contrast to the adjacency and graph filter functions, translating pattern queries into single SQL statements can result in very poor query performance.

In this report, the performance of single SQL statement pattern queries is contrasted with an approach that generates SQL for depth first paths through the pattern query. The performance of this latter approach can result in dramatically faster graph pattern query execution. Pattern queries that are translated into SQL for depth first paths will also perform better as the graph size increases.

Generating SQL for paths through the pattern query requires more complex software support in the graph

query engine. This extra complexity could be avoided if the relational SQL optimizer and query planner generated an efficient query plan for single SQL statements generated for pattern queries. The fact that this is not the case represents a failure of the SQL query optimizer, relative to graph pattern queries.

Graph Databases and Graph Queries

In some application areas, graph databases offer important advantages over relational databases:

1. Data from a wide variety of sources can be easily fused into a single semantic graph.
2. The structure of the graph is controlled by the semantic graph ontology. The graph structure can be changed by modifying the ontology. Within limits, software can migrate graph data from one graph ontology to another.
3. Graph queries provide a simple way to express graph database queries would be difficult to express in relational SQL.

Many graph pattern matching algorithms [6] are implemented as in memory algorithms. In most cases these algorithms can only be applied to data sets that will fit in computer memory. A semantic graph database can store data from myriad sources, resulting in very large graphs that cannot fit into computer memory. The pattern matching algorithms that are applied to out-of-memory graphs stored on persistent media like computer disk systems are data parallel algorithms. In this report these algorithms are implemented in relational SQL.

A graph database could be supported by custom database software. A graph query would be directly translated into a query plan for this custom database. Such a database would require significant development effort and might have to be modified for different hardware configurations. Translating graph queries into standard SQL for a relational database allows the graph database to be hosted on a variety of relational database systems, ranging from database servers to distributed relational databases.

The graph query language discussed here [5] supports the following graph functions:

- graph union, intersection and difference
- filter - filter a graph so that the query result contains only the vertices and edges defined in the query.
- adjacency - expand a defined subgraph by adding selected links and vertices.
- Path traversal between a set of source and destination points in the graph.
- Pattern queries.

The graph query language supports nested queries, where one query expression defines the input graph for another query expression.

The graph query language described here has been influenced by the work of David Jensen's group at the University of Massachusetts at Amherst on the QGraph graph query language [2, 3] and by the work that David Silberberg's group has done at the Johns Hopkins Advanced Physics Laboratory [4].

The binary graph operations (union, intersection and difference), filter and adjacency are implemented by translating the graph query into a single SQL statement. This approach is attractive because of its simplicity. A similar approach can be used for graph pattern queries. The SQL for a pattern query could also be constructed by traversing the data structure for the pattern query. As the results presented in this report show, when a single SQL statement is generated for a graph pattern query, the query performance

can be very poor.

Graph Database Ontologies

A graph database is created when a graph ontology is loaded into the semantic graph database system. The graph ontology defines the graph vertex types and the types of the edges that connect these vertices. The ontology also defines the attributes for each vertex (or edge). Multiple ontologies can be supported at the same time and there may be multiple graphs associated with each ontology.

The graph database schema that supports the graph ontologies is diagrammed in **Figure 1**, below. Each ontology has an associated set of tables that store the vertex attributes. In **Figure 1** the table es_42 stores attribute information for instances of *author* vertices and the table es_43 stores the attributes for instances of *article* vertices. The cloud represents the ontology information which includes mappings between the names in the query (author or article) and database tables (es_42 or es_43). Graph vertex attributes are shared by all graphs that are associated with a particular ontology. There is a unique edge table for each graph. The edge table columns are also diagrammed in **Figure 1**. Each graph also has an associated vertex table that contains all of the vertex identifiers and types for the graph. This table is not used by pattern query operations.

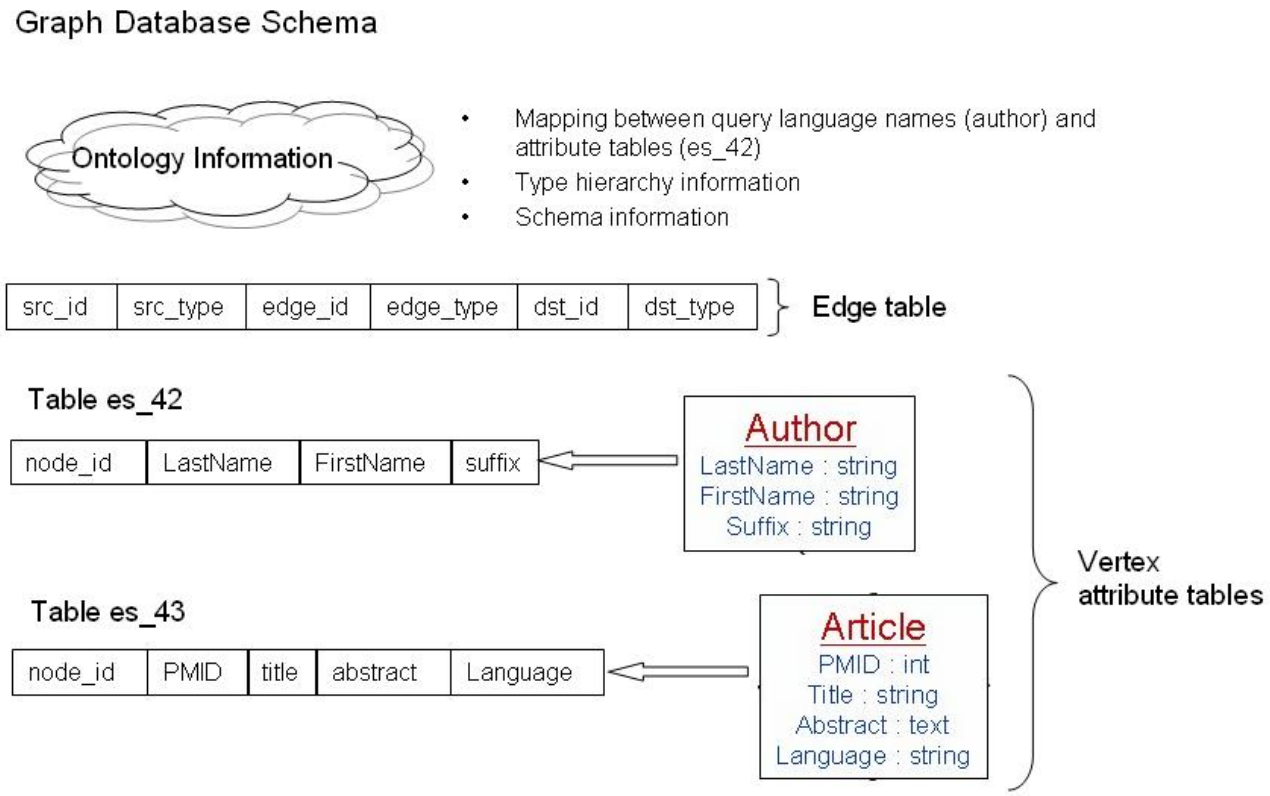


Figure 1

The SQL generated for the pattern queries discussed in this report largely consist of self-joins against the edge table. The edge table has three sets of indices, as shown in Figure 2, below. The primary key is on the {link_id, link_type} columns. There are also indices on the {src_node_id, src_node_type} and the {dst_node_id, dst_node_type} columns.

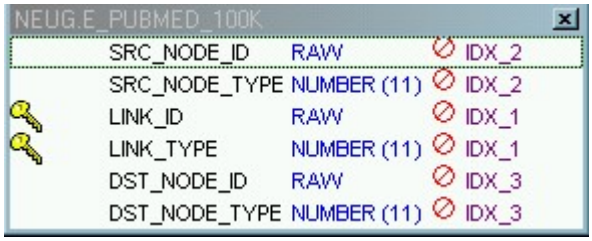


Figure 2

Pattern Queries

"I don't know what you mean by 'pattern query'," Alice said.

Humpty Dumpty smiled contemptuously. "Of course you don't – till I tell you. When I say *find all of the authors who have published two articles*, I mean *return a set of subgraphs that pair each author with combinations of two of the articles they have published*. It's a combinatoric choose(N,k) problem you see."

"But a pattern query doesn't mean 'choose(N, k)', it just means find that pattern in the graph" Alice objected.

"When I use a word," Humpty Dumpty said in a rather a scornful tone, "it means just what I choose it to mean – neither more nor less.

"The question is," said Alice, "whether you can make words mean different things."

With apologies to Lewis Carroll and his book *Through the Looking-Glass* (Macmillan, 1871)

Pattern queries seem intuitive. For example, the pattern query *find all of the authors who have published two articles* seems intuitively obvious. But what does this query really mean? Does it mean *find all of the authors who have published exactly two articles*? Or does it mean *find all of the authors who have published two or more articles*? If we use the latter meaning, what subgraphs should the query return?

The pattern query *find all of the authors who have published two articles* is constrained only by the types of the vertices (or the edges if we defined the edge relationship in the pattern query). As Humpty Dumpty states in the quote above, computational complexity of the pattern query *find all authors who have published two articles* is *choose(N, 2)*, where N is the number of articles published by a particular author. This is shown in **Figure 3**, below. The author Peter Watts has four publications: Starfish, Maelstrom, Behemoth and Blindsight.

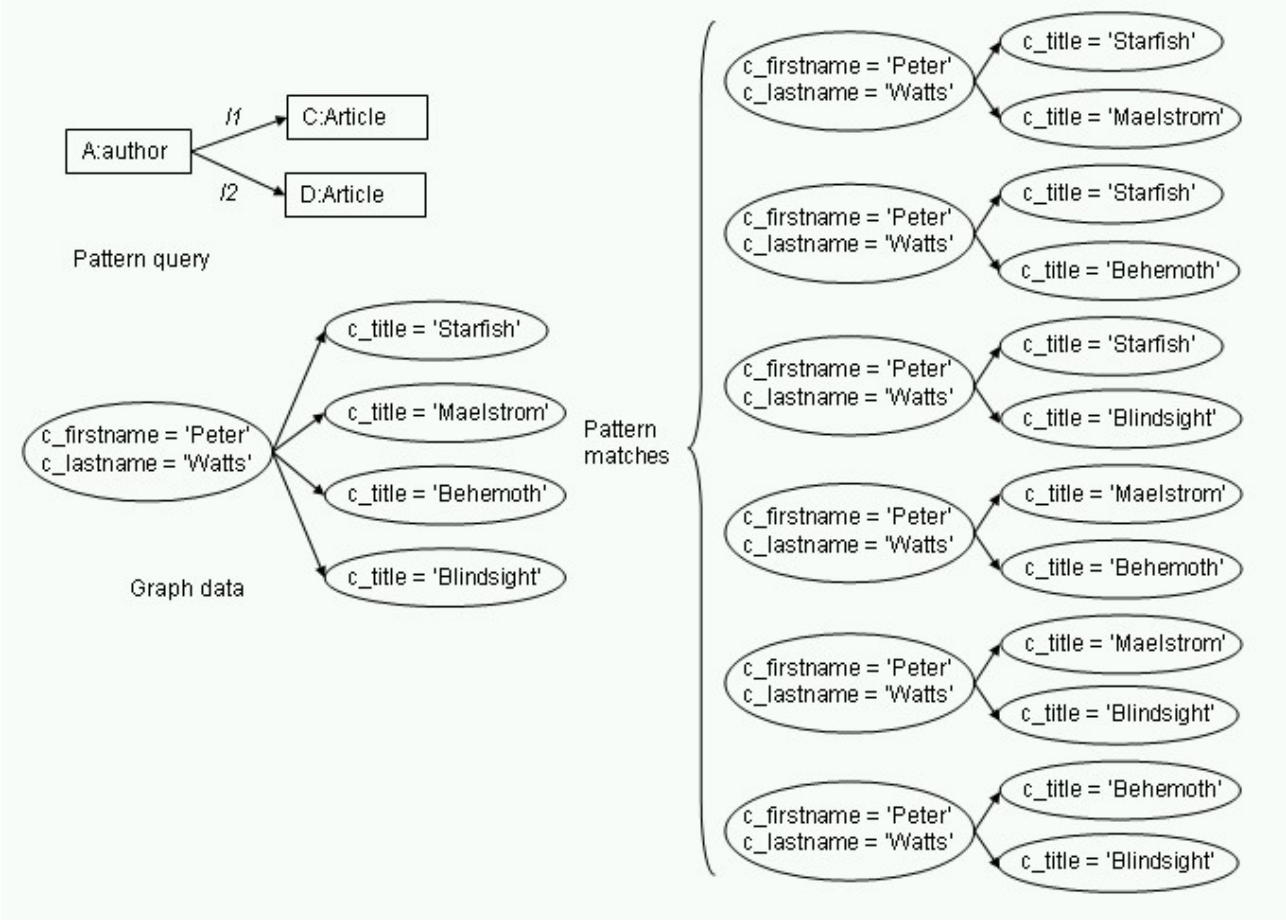


Figure 3

The number of unique patterns in this case is (3 * 4)/2 = 6. The number of subgraphs that are returned by choose(N, k) pattern queries can be calculated using the equation below:

choose(N, k) = $\frac{1}{k!}\prod_{i=(n-k)+1}^{n} i$

Why Allow Combinatoric Pattern Queries?

The behavior of pattern queries that are constrained only by the vertex type may be unexpected. The person posing the query may not expect to get back a set of subgraphs consisting of the combinatoric set of *choose(N, k)* matching subgraphs. Why allow queries that return a large unexpected result?

In some cases combinatoric pattern matching is the only technique that will find a matching pattern in the graph. In **Figure 4**, below, the pattern query (shown with rectangles) matches only one pattern in the data (shown with ovals). This pattern can only be discovered by trying the different combinations to fill in the matching subgraph.

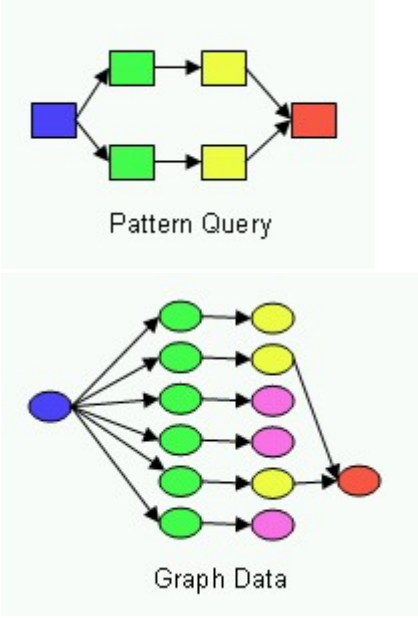


Figure 4

Combinatoric matches are only a problem when there are instances of high degree vertices for one of the types in the pattern. If the pattern consists of types with only a relatively small number of instances the combinatoric match may not be a problem.

Queries with Constraints

By adding constraints to a pattern query, the number of subgraphs returned can be reduced. There are two kinds of constraints:

1. a where clause that defines a constraint on the vertex attributes
2. a cardinality constraint that defines an exact number of edges.

These constraints can be mixed in a single pattern query.

An example of a query with where clause constraints is shown below in **Figure 5**.

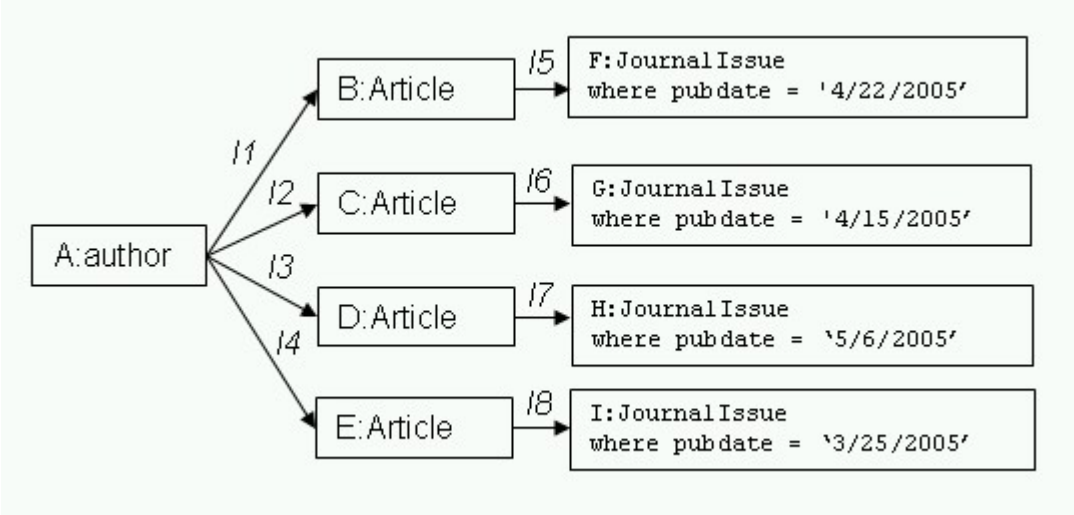


Figure 5

The pattern query in **Figure 6** includes a cardinality constraint. This query finds all of the authors who have published *exactly* two articles.

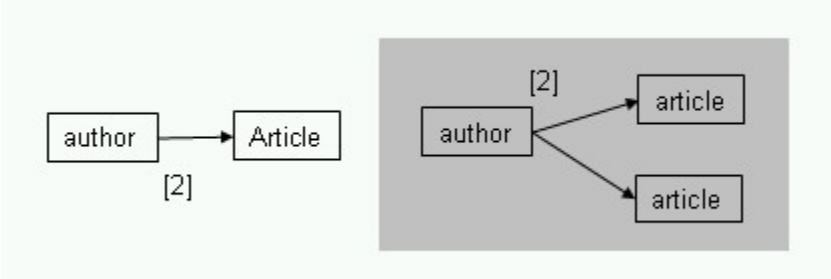


Figure 6

A cardinality constraint may also specify a range. **Figure 7** shows a query that will return subgraphs for authors who have published *exactly* two, three or four articles.

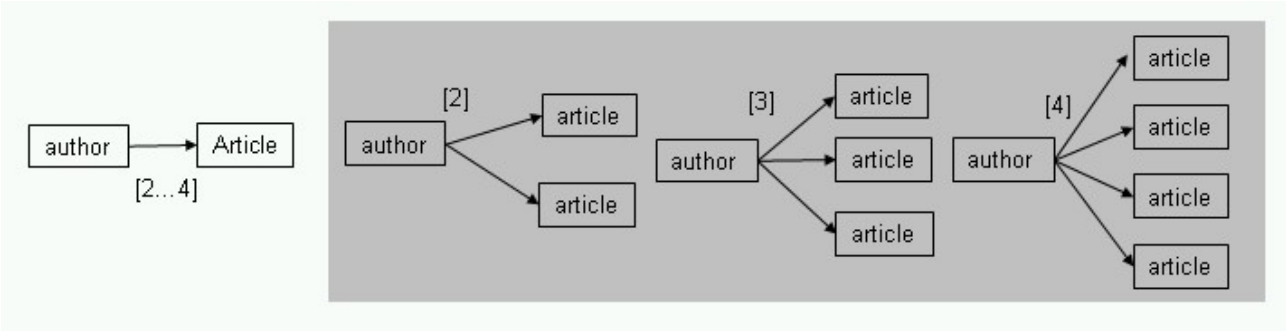


Figure 7

A cardinality constraint may have an open upper bound. The query in **Figure 8** will return subgraphs for authors have have published 2 to N articles, where N is the maximum number of articles published by any author.

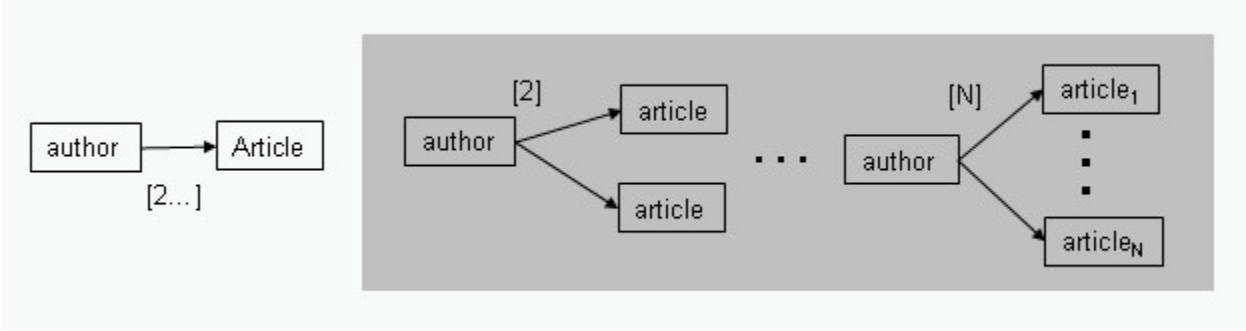


Figure 8

Pattern Query to SQL Translation

A graph pattern query is a connected graph. Execution of a pattern query is the process of finding subgraphs that exactly match the graph defined in the pattern query.

A graph pattern query is stored as a data structure in computer memory. The query is translated into SQL by traversing this data structure and building one or more SQL statements which are sent to the relational database to produce the pattern query result. The simplest way to translate pattern queries into SQL is to generate a single SQL statement, with an edge table join for each edge in the graph. Unfortunately on the database we used for the benchmarks in this report, the performance of these queries is poor. By traversing unique non-overlapping paths through the pattern query and generating SQL for each path, much faster pattern query execution can be achieved.

Avoiding Geometric Pattern Duplicates

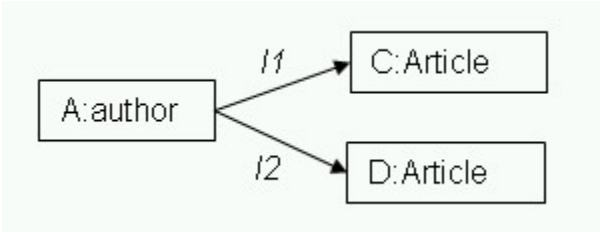


Figure 9

There is no inherent ordering of vertices in a graph. **Figure 10** shows two possible orderings for one subgraph returned by the pattern query in **Figure 9** (an author linked to two articles).

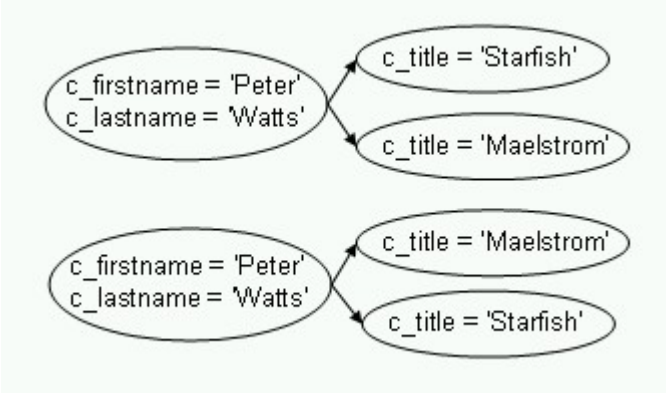


Figure 10

Patterns that differ only in their geometric arrangement are of no interest in graph pattern matching applications and greatly increase the size of the query result. To avoid returning these geometric duplicates the SQL that is generated for a pattern query forces an ordering to eliminate duplicate patterns. An example is shown below in the SQL that would be generated for the author to two articles query.

```
create table query_1000K as
SELECT l1.link_id link_1_id, l1.link_type link_1_type, l2.link_id link_2_id,
       l2.link_type link_2_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2
 WHERE l1.src_node_type = 190  /* author */
   AND l1.dst_node_type = 186  /* article */
   AND l2.dst_node_type = 186  /* article */
   AND l1.src_node_id = l2.src_node_id
   AND l1.dst_node_id > l2.dst_node_id;
```

The source and destination node (or vertex) ID is an integer or a cryptographic hash value that is globally unique for an ontology (e.g., no other node in a graph associated with that ontology has that ID value). The greater than constraint on the destination of the two author to article edges forces an ordering for the article vertices in the matching subgraphs. This eliminates geometric duplicates.

The Pattern Query Result Table

The result of a pattern query is the unique set of subgraphs that match the graph pattern. The SQL for pattern queries in this report return a result where each matching subgraph is a row in the result table. Each column of the row is a link in the matching subgraph. This presents a problem since pattern query result tables may have differing numbers of result table columns. The query engine will execute additional queries that write the query result into a query result table that has the same column structure for all pattern queries. These queries are outside of the scope of this report.

The Relational Database and Test Graph Size

All of the benchmarks discussed in this report have been run on a relational database that is hosted on a four processor database server. The graph used for benchmark testing consists of 1,002,970 vertices (i.e., about 1 million vertices) and 1,929,797 edges (i.e., about 2 million edges). The important metric here is the number of edges (e.g., the edge table size). The graph data used for these benchmarks is taken from the PubMed data set, which consists of journal article publication information. This graph is

considerably smaller than the graphs that would be seen in actual semantic graph applications.

Pattern Queries with Cardinality Constraints

The pattern query in **Figure 11** includes a cardinality constraint. This pattern query will find all of the authors who have published *exactly* two articles.

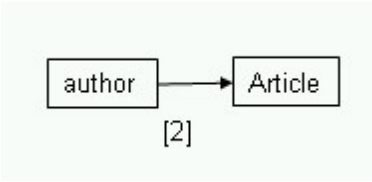


Figure 11

The SQL for a pattern query with a cardinality constraint requires nested SQL that finds the author vertices for the authors who have published *exactly* two articles. This nested SQL is shown below:

```
SELECT DEGREE.src_node_id
  FROM (SELECT DISTINCT links.src_node_id, COUNT (*) cnt
                  FROM e_pubmed_1000k links
                 WHERE links.src_node_type = 190  /* author */
                   AND links.dst_node_type = 186  /* article */
              GROUP BY links.src_node_id) DEGREE
 WHERE DEGREE.cnt = 2;
```

This SQL query returns only the author vertices, not the complete pattern. In order to return the complete pattern, this query is included as a subquery in a larger SQL query. This is shown below:

```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2,
(SELECT DEGREE.src_node_id
  FROM (SELECT DISTINCT links.src_node_id, COUNT (links.src_node_id) cnt
                  FROM e_pubmed_1000k links
                 WHERE links.src_node_type = 190                /* author */
                   AND links.dst_node_type = 186               /* article */
              GROUP BY links.src_node_id) DEGREE
 WHERE DEGREE.cnt = 2) src
 WHERE l1.src_node_id = src.src_node_id
   AND l1.dst_node_type = 186                                   /* article */
   AND l2.dst_node_type = 186                                   /* article */
   AND l1.src_node_id = l2.src_node_id
   AND l1.dst_node_id > l2.dst_node_id;
```

The above SQL statement, with the nested query to find author vertex cardinality caused problems for the query optimizer and query planner, resulting in poor query performance on the database we used for our tests. The performance of this query on the test graph is shown below:

time (seconds)	rows
8640 (2.4 hours)	66497

The fact that this query has such poor performance seems to be an aberration of the database we used for testing (perhaps a bug). Another relational database we used for testing executed this query in about

the same time as the query broken into two steps, discussed below.

To avoid this problem, this query was broken up into two steps:

1. Build a table of author vertices that are linked to two articles.
2. Return the author to article pattern that contains these articles.

```
CREATE TABLE degree AS
SELECT DEGREE.src_node_id
  FROM (SELECT DISTINCT links.src_node_id, COUNT (links.src_node_id) cnt
                  FROM e_pubmed_1000k links
                 WHERE links.src_node_type = 190                 /* author */
                   AND links.dst_node_type = 186                /* article */
              GROUP BY links.src_node_id) DEGREE
 WHERE DEGREE.cnt = 2;
```

```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2, degree
 WHERE l1.src_node_id = degree.src_node_id
   AND l1.dst_node_type = 186                                    /* article */
   AND l2.dst_node_type = 186                                    /* article */
   AND l1.src_node_id = l2.src_node_id
   AND l1.dst_node_id > l2.dst_node_id;
```

	time (seconds)	rows
Step 1	5.2	66497
Step 2	2.8	66497
Total time	8.0	

Optimizing Graph Pattern Query Execution

One of the simplest ways to implement graph pattern queries on top of a relational database is to translate a pattern query into a single SQL statement. The pattern query translation software traverses the pattern query data structure and builds an SQL statement for the pattern query. As the benchmarks below demonstrate, these SQL queries consist of multiple self-joins against the graph edge table.

Unfortunately the SQL optimizer and query planner does not execute these SQL statements efficiently. As the benchmarks below demonstrate, the poor execution performance for these SQL statements is not a result of the SQL statement complexity, as might be the case if there were multiple levels of query nesting. The problem seems to arise from the nature of graph pattern queries. Different edge table join orderings can produce intermediate query results that differ by orders of magnitude in size. **Figure 12**, below, shows a breath first and a depth first join ordering. Each color in the diagram is intended to represent a set of edge table joins. If the breath first ordering is used there would be a very large query intermediate. If a depth first join ordering is used, the query intermediate is orders of magnitude smaller resulting in dramatically faster query execution times.

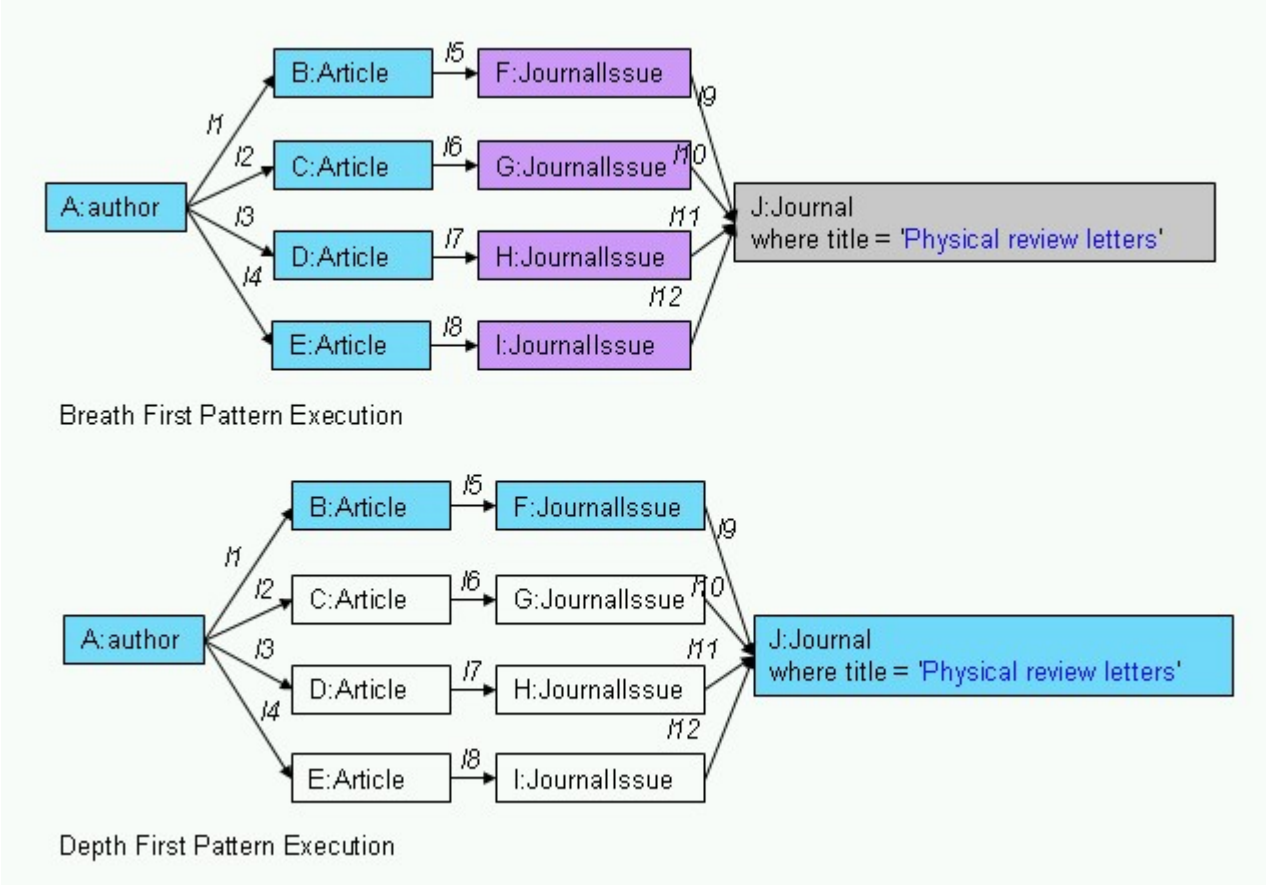


Figure 12

The benchmarks in this report compare pattern queries implemented by single SQL statements with patterns implemented by SQL generated for depth first paths through the pattern query. As these benchmarks demonstrate, multiple SQL statements that take the pattern structure into account yield much better pattern query performance than single SQL statements.

query	number of subgraphs (rows)	single SQL statement (seconds)	paths through the pattern (seconds)
Where constraint	1	8640 (2.4 hours)	~3.0
Query 1	39036649	54000 (15 hours)	1014 (~17 minutes)
Query 2	957272	16 hours (did not complete)	21.8
Query 3	19067116	3480.0 (58 minutes)	387 (6.45 minutes)
Query 4	7630	6840.0 (1.9 hours)	282.0 (4.7 minutes)

Discussion

Translating graph pattern queries into relational SQL that has acceptable performance can be a difficult problem. The benchmarks results described in this report were obtained for a small graph of only 1 million vertices and 2 million edges. The graphs used in real semantic graph applications would be much larger. These larger graphs may require more powerful databases systems than the database server used in these benchmarks. But hardware alone will not deliver pattern query performance. Pattern query

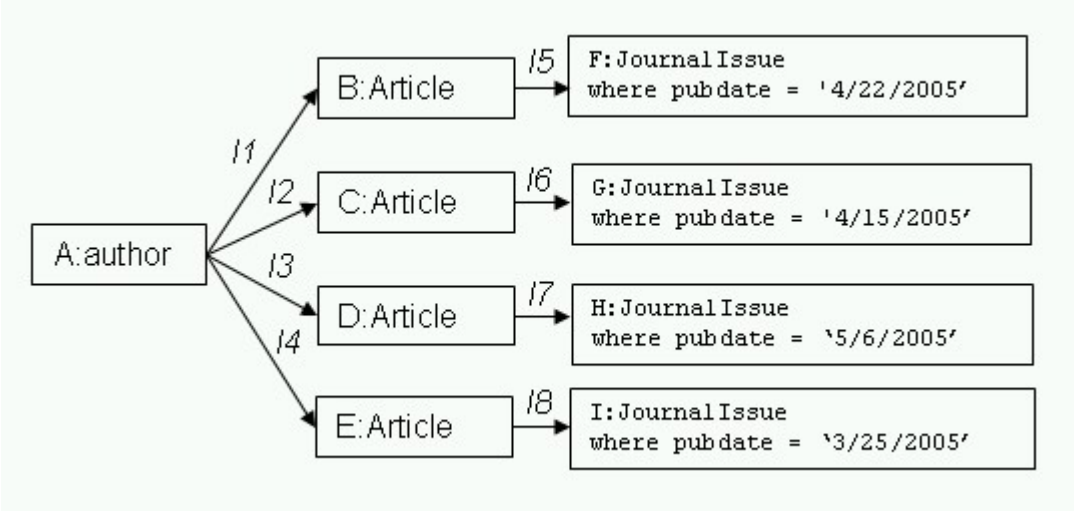
execution must minimize the size of query intermediate results. The SQL optimizer cannot be relied on to efficiently execute single SQL statements that implement pattern queries. The results in this report show that acceptable pattern query execution performance requires SQL generation that optimizes for the pattern query structure.

The extra complexity entailed in translating graph pattern queries into multiple SQL statements is required because the relational database query planner and optimizer does very poorly with the single SQL statements generated for pattern queries. Graph pattern query translation must take into account the structure of the pattern query and generate multiple SQL statements to efficiently implement the pattern query. This complexity could be avoided if the relational database query optimizer were extended to handle pattern queries efficiently. One approach that the SQL optimizer might take is to rebuild the pattern query from the set of joins defined in a single SQL statement for the pattern. Once the pattern is reconstructed, the query optimizer could use a query plan that ordered joins using depth first paths through the pattern, as described in this report.

Graph databases and graph query languages allow queries to be easily formulated that are difficult to express in relational SQL. These queries are commonly used in applications involving crime or intelligence investigation. These are specialized areas and it could be argued that the effort to extend the query optimizer for graph pattern queries would not benefit a majority of the database user community. However, the problems encountered for pattern queries may also exist for data warehouse applications. The database schema chosen for many data warehouses may be overly constrained to avoid the kinds of performance problems discussed in this report. The result is a data warehouse that is tuned for certain queries but performs very badly for others. An SQL optimizer that could deliver acceptable performance for pattern queries might allow more general data warehouse schemas. The result would be a more flexible data warehouse implementation.

Pattern Query Benchmark Results

A Pattern Query with a Where Clause Constraint



This is a highly constrained pattern query. There is only one instance of this sub-graph in the test graph. The simplest way to generate SQL for this query is to traverse the data structure for the pattern query

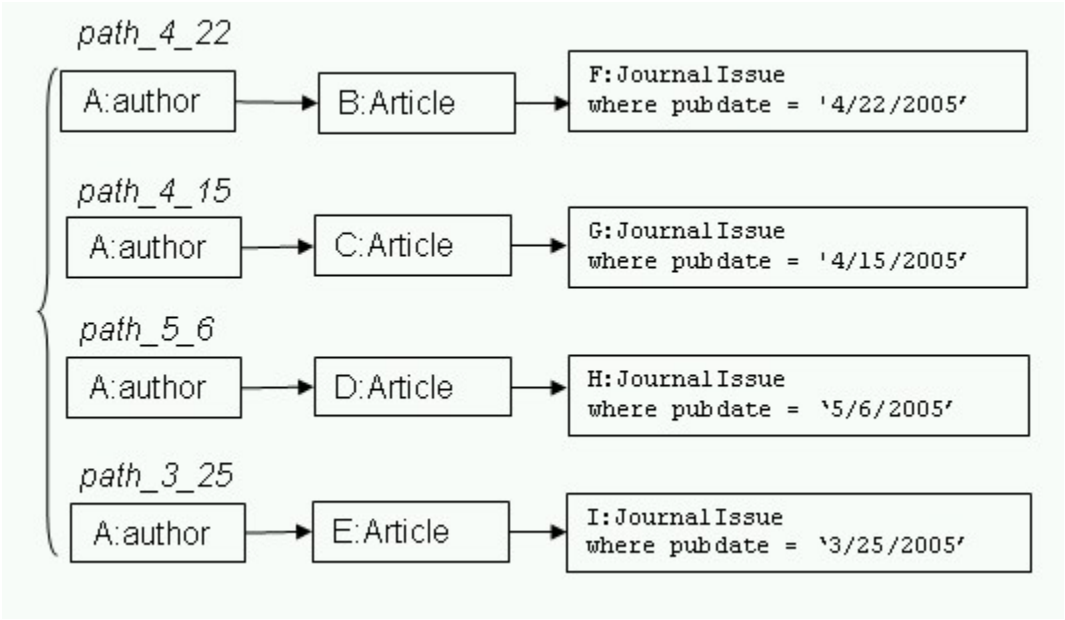
and generate a single SQL statement. The SQL statement that might be generated is shown below. The es_194 table is the vertex table that contains the attribute values for the JournalIssue vertices.

```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type,
       l3.link_id link_3_id, l3.link_type link_3_type,
       l4.link_id link_4_id, l4.link_type link_4_type,
       l5.link_id link_5_id, l5.link_type link_5_type,
       l6.link_id link_6_id, l6.link_type link_6_type,
       l7.link_id link_7_id, l7.link_type link_7_type,
       l8.link_id link_8_id, l8.link_type link_8_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3, e_pubmed_1000k l4,
       e_pubmed_1000k l5, e_pubmed_1000k l6, e_pubmed_1000k l7, e_pubmed_1000k l8,
           es_194 ji1, es_194 ji2, es_194 ji3, es_194 ji4
 WHERE l1.src_node_type = 190  /* author */
   AND l1.dst_node_type = 186  /* article */
   AND l2.dst_node_type = 186  /* article */
   AND l3.dst_node_type = 186  /* article */
   AND l4.dst_node_type = 186  /* article */
   AND l1.src_node_id = l2.src_node_id
   AND l1.src_node_id = l3.src_node_id
   AND l1.src_node_id = l4.src_node_id
   AND l1.dst_node_id > l2.dst_node_id
   AND l2.dst_node_id > l3.dst_node_id
   AND l3.dst_node_id > l4.dst_node_id
   AND l5.src_node_id = l1.dst_node_id
   AND l5.dst_node_type = 194  /* JournalIssue */
   AND l6.src_node_id = l2.dst_node_id
   AND l6.dst_node_type = 194  /* JournalIssue */
   AND l7.src_node_id = l3.dst_node_id
   AND l7.dst_node_type = 194  /* JournalIssue */
   AND l8.src_node_id = l4.dst_node_id
   AND l8.dst_node_type = 194  /* JournalIssue */
   AND ji1.c_pubdate = to_date('4/22/2005', 'MM/DD/YYYY')
   AND l5.dst_node_id = ji1.guid
   AND ji2.c_pubdate = to_date('4/15/2005', 'MM/DD/YYYY')
   AND l6.dst_node_id = ji2.guid
   AND ji3.c_pubdate = to_date('5/6/2005', 'MM/DD/YYYY')
   AND l7.dst_node_id = ji3.guid
   AND ji4.c_pubdate = to_date('3/25/2005', 'MM/DD/YYYY')
   AND l8.dst_node_id = ji4.guid;
```

time (seconds)	rows
8640 (2.4 hours)	1

This query takes 2.4 hours on a million vertex test graph. Graphs that would be used in real analytic problems would be orders of magnitude larger, resulting in even longer query times.

This query can also be efficiently implemented by dividing it into four paths through the pattern query, one path for each leaf condition in the pattern query tree. Each of the paths is then joined in a final step. This approach requires five SQL queries: one for each of the four paths, plus the final join.




```
CREATE TABLE path_4_22 AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type,
       l1.link_id e1_link_id, l1.link_type e1_link_type,
       l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type,
       l2.link_id e2_link_id, l2.link_type e2_link_type,
       l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2, es_194 ji
WHERE l1.src_node_type = 190  /* author */
  AND l1.dst_node_type = 186  /* article */
  AND l2.src_node_id = l1.dst_node_id
  AND l2.dst_node_type = 194  /* JournalIssue */
  AND ji.c_pubdate = to_date('4/22/2005', 'MM/DD/YYYY')
  AND l2.dst_node_id = ji.guid
  ;
```

```
CREATE TABLE path_4_15 AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type,
       l1.link_id e1_link_id, l1.link_type e1_link_type,
       l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type,
       l2.link_id e2_link_id, l2.link_type e2_link_type,
       l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2, es_194 ji
WHERE l1.src_node_type = 190  /* author */
  AND l1.dst_node_type = 186  /* article */
  AND l2.src_node_id = l1.dst_node_id
  AND l2.dst_node_type = 194  /* JournalIssue */
  AND ji.c_pubdate = to_date('4/15/2005', 'MM/DD/YYYY')
  AND l2.dst_node_id = ji.guid
  ;
```

```
CREATE TABLE path_5_6 AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type,
       l1.link_id e1_link_id, l1.link_type e1_link_type,
       l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type,
       l2.link_id e2_link_id, l2.link_type e2_link_type,
       l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2, es_194 ji
WHERE l1.src_node_type = 190  /* author */
  AND l1.dst_node_type = 186  /* article */
  AND l2.src_node_id = l1.dst_node_id
  AND l2.dst_node_type = 194  /* JournalIssue */
```

```
  AND ji.c_pubdate = to_date('5/6/2005', 'MM/DD/YYYY')
  AND l2.dst_node_id = ji.guid
  ;
```

```
CREATE TABLE path_3_25 AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type,
       l1.link_id e1_link_id, l1.link_type e1_link_type,
       l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type,
       l2.link_id e2_link_id, l2.link_type e2_link_type,
       l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2, es_194 ji
WHERE l1.src_node_type = 190  /* author */
  AND l1.dst_node_type = 186  /* article */
  AND l2.src_node_id = l1.dst_node_id
  AND l2.dst_node_type = 194  /* JournalIssue */
  AND ji.c_pubdate = to_date('3/25/2005', 'MM/DD/YYYY')
  AND l2.dst_node_id = ji.guid
  ;
```

```
CREATE TABLE query_result AS
SELECT p1.e1_link_id link_1_id, p1.e1_link_type link_1_type, p1.e2_link_id, link_5_id
p1.e2_link_type link_5_type,
       p2.e1_link_id link_2_id, p2.e1_link_type link_2_type, p2.e2_link_id, link_6_id
p2.e2_link_type link_6_type,
       p3.e1_link_id link_3_id, p3.e1_link_type link_3_type, p3.e2_link_id, link_7_id
p3.e2_link_type link_7_type,
       p4.e1_link_id link_4_id, p4.e1_link_type link_4_type, p4.e2_link_id, link_8_id
p4.e2_link_type link_8_type
FROM path_4_22 p1, path_4_15 p2, path_5_6 p3, path_3_25 p4
WHERE p1.e1_src_node_id = p2.e1_src_node_id /* make sure that the author vertices are the same
*/
  AND p2.e1_src_node_id = p3.e1_src_node_id
  AND p3.e1_src_node_id = p4.e1_src_node_id
  AND p1.e1_dst_node_id > p2.e1_dst_node_id /* order the article vertices */
  AND p2.e1_dst_node_id > p3.e1_dst_node_id
  AND p3.e1_dst_node_id > p4.e1_dst_node_id
  ;
```

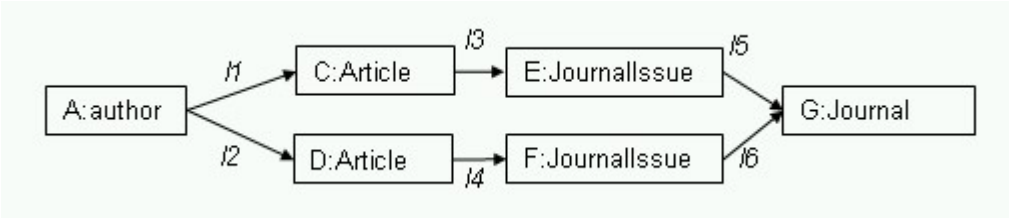
	time (seconds)	rows
path_4_22	0.344	1559
path_4_15	1.0	1388
path_5_6	0.828	1660
path_3_25	0.467	493
join the paths	0.280	1
total	~3.0	

Pattern Queries Constrained by Vertex Type

As discussed above, pattern queries that are constrained only by the types of the pattern vertices do combinatoric pattern matching. In this section we give a few examples of the how these queries can be translated into single SQL statements. The performance of these single SQL statements is very poor. By generating SQL for paths through the pattern and then joining these paths, query performance can be dramatically improved.

Query 1

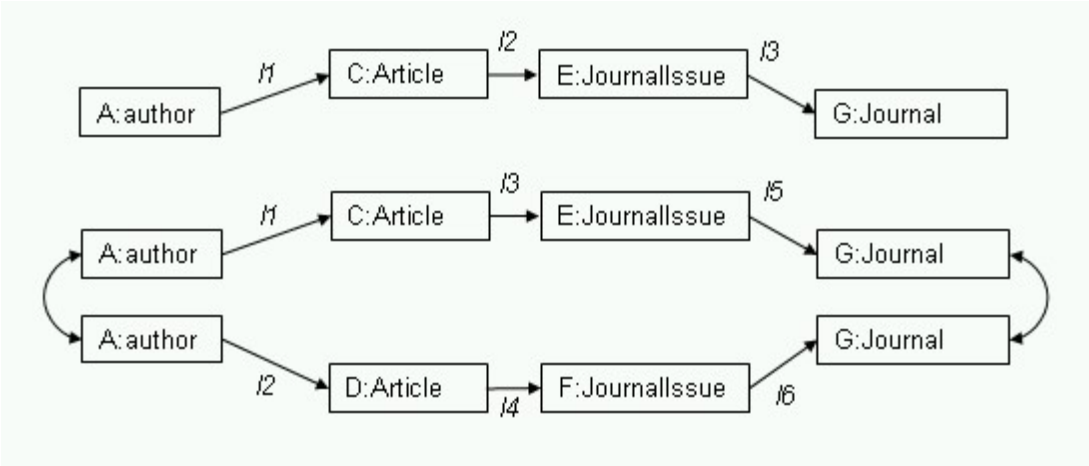
Find subgraphs that consist of an author that has published two articles in the same journal.




```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type,
       l3.link_id link_3_id, l3.link_type link_3_type,
       l4.link_id link_4_id, l4.link_type link_4_type,
       l5.link_id link_5_id, l5.link_type link_5_type,
       l6.link_id link_6_id, l6.link_type link_6_type
 FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3,
      e_pubmed_1000k l4, e_pubmed_1000k l5, e_pubmed_1000k l6
 WHERE l1.src_node_type = 190  /* author */
   AND l1.dst_node_type = 186  /* article */
   AND l2.dst_node_type = 186  /* article */
   AND l1.src_node_id = l2.src_node_id
   AND l1.dst_node_id > l2.dst_node_id
   AND l3.dst_node_type = 194  /* JournalIssue */
   AND l3.src_node_id = l1.dst_node_id
   AND l4.dst_node_type = 194  /* JournalIssue */
   AND l4.src_node_id = l2.dst_node_id
   AND l5.dst_node_type = 191  /* Journal */
   AND l5.src_node_id = l3.dst_node_id
   AND l6.src_node_id = l4.dst_node_id
   AND l5.dst_node_id = l6.dst_node_id;
```

time (seconds)	rows
54000 (15 hours)	39036649

The subgraphs that match Query 1 can be found much more rapidly using two sub-queries. The first query finds a linear path through the pattern (author --> article --> JournalIssue --> journal). The second query performs a self-join on the result of this query. This self-join finds the paths that share a common author and journal, but have different articles. The join also orders the article vertices so that the resulting subgraphs are unique.



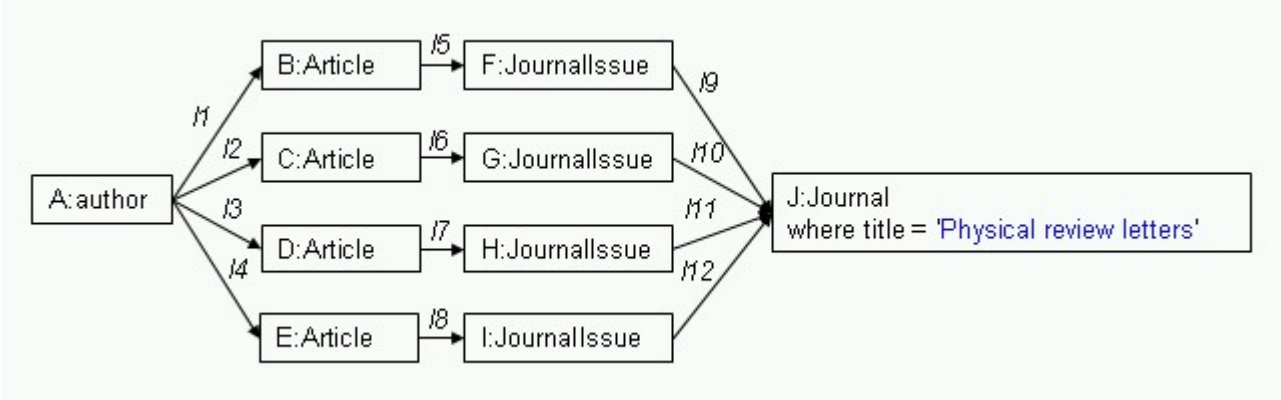
```
CREATE TABLE query_half AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type,
       l1.link_id e1_link_id, l1.link_type e1_link_type,
       l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type,
       l2.link_id e2_link_id, l2.link_type e2_link_type,
       l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type,
       l3.src_node_id e3_src_node_id, l3.src_node_type e3_src_node_type,
       l3.link_id e3_link_id, l3.link_type e3_link_type,
       l3.dst_node_id e3_dst_node_id, l3.dst_node_type e3_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3
WHERE l1.src_node_type = 190  /* author */
  AND l1.dst_node_type = 186  /* article */
  AND l2.src_node_id = l1.dst_node_id
  AND l2.dst_node_type = 194  /* JournalIssue */
  AND l3.src_node_id = l2.dst_node_id
  AND l3.dst_node_type = 191  /* Journal */
  ;
```

```
CREATE TABLE query_1000k AS
SELECT h1.e1_link_id, h1.e1_link_type,
       h1.e2_link_id e3_link_id, h1.e2_link_type e3_link_type,
       h1.e3_link_id e5_link_id, h1.e3_link_type e5_link_type,
       h2.e1_link_id e2_link_id, h2.e1_link_type e2_link_type,
       h2.e2_link_id e4_link_id, h2.e2_link_type e4_link_type,
       h2.e3_link_id e6_link_id, h2.e3_link_type e6_link_type
  FROM query_half h1, query_half h2
 WHERE h1.e1_src_node_id = h2.e1_src_node_id
   AND h1.e3_dst_node_id = h2.e3_dst_node_id
   AND h1.e1_dst_node_id > h2.e1_dst_node_id
   ;
```

	time (seconds)	rows
find author to journal path	114.0	13696874
join path result	900 (15 minutes)	39036649
total	1014 (~17 minutes)	

Query 2

Find subgraphs that consist of an author who has published four articles in the journal *Physical Review Letters*.

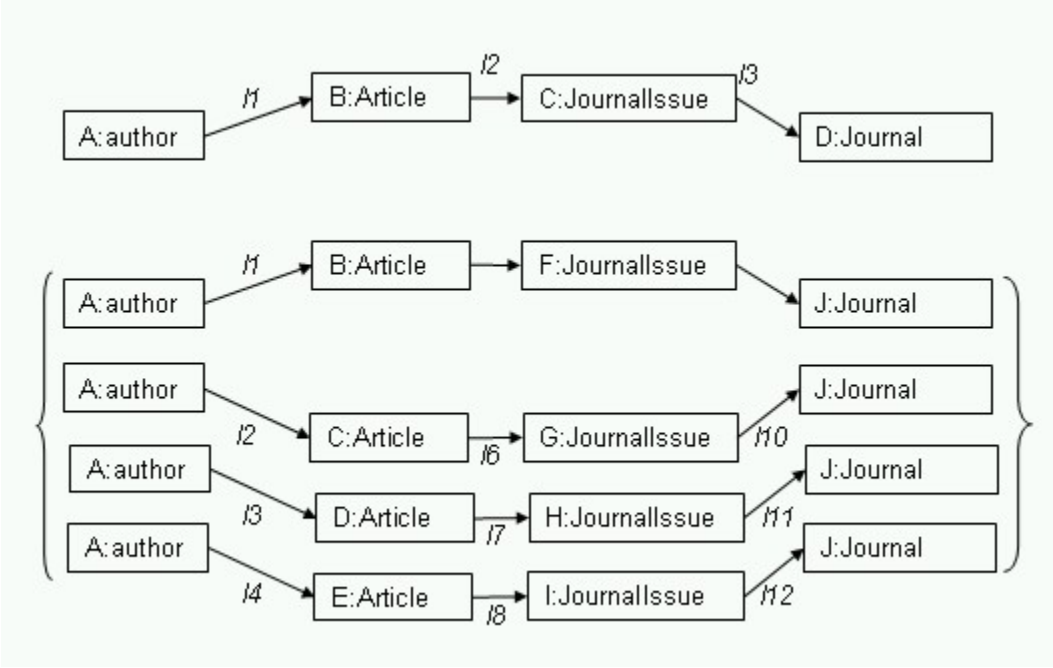


A single SQL query that implements this pattern query is shown below:

```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type,
       l3.link_id link_3_id, l3.link_type link_3_type,
       l4.link_id link_4_id, l4.link_type link_4_type,
       l5.link_id link_5_id, l5.link_type link_5_type,
       l6.link_id link_6_id, l6.link_type link_6_type,
       l7.link_id link_7_id, l7.link_type link_7_type,
       l8.link_id link_8_id, l8.link_type link_8_type,
       l9.link_id link_9_id, l9.link_type link_9_type,
       l10.link_id link_10_id, l10.link_type link_10_type,
       l11.link_id link_11_id, l11.link_type link_11_type,
       l12.link_id link_12_id, l12.link_type link_12_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3, e_pubmed_1000k l4,
       e_pubmed_1000k l5, e_pubmed_1000k l6, e_pubmed_1000k l7, e_pubmed_1000k l8,
       e_pubmed_1000k l9, e_pubmed_1000k l10, e_pubmed_1000k l11, e_pubmed_1000k l12,
       es_191 journal
 WHERE l1.src_node_type = 190  /* author */
   AND l1.dst_node_type = 186  /* article */
   AND l2.dst_node_type = 186  /* article */
   AND l3.dst_node_type = 186  /* article */
   AND l4.dst_node_type = 186  /* article */
   AND l1.src_node_id = l2.src_node_id
   AND l1.src_node_id = l3.src_node_id
   AND l1.src_node_id = l4.src_node_id
   AND l1.dst_node_id > l2.dst_node_id
   AND l2.dst_node_id > l3.dst_node_id
   AND l3.dst_node_id > l4.dst_node_id
   AND l5.src_node_id = l1.dst_node_id
   AND l5.dst_node_type = 194  /* JournalIssue */
   AND l6.src_node_id = l2.dst_node_id
   AND l6.dst_node_type = 194  /* JournalIssue */
   AND l7.src_node_id = l3.dst_node_id
   AND l7.dst_node_type = 194  /* JournalIssue */
   AND l8.src_node_id = l4.dst_node_id
   AND l8.dst_node_type = 194  /* JournalIssue */
   AND l9.src_node_id = l5.dst_node_id
   AND l9.dst_node_type = 191  /* journal */
   AND journal.c_title = 'Physical review letters'
   AND l9.dst_node_id = journal.guid
   AND l10.src_node_id = l6.dst_node_id
   AND l10.dst_node_id = l9.dst_node_id
   AND l11.src_node_id = l7.dst_node_id
   AND l11.dst_node_id = l9.dst_node_id
   AND l12.src_node_id = l8.dst_node_id
   AND l12.dst_node_id = l9.dst_node_id;
```

time (hours)	rows
16 hours	did not complete

If the intermediate joins used by the query planner execute the edge table self joins in a breath first order there can be large intermediate results. A depth first query strategy, where SQL is generated for paths through the pattern yields dramatically better query performance. This is shown below where the first query calculates a path through the pattern and the second query joins the paths.




```
CREATE TABLE query_half AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type,
       l1.link_id e1_link_id, l1.link_type e1_link_type,
       l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type,
       l2.link_id e2_link_id, l2.link_type e2_link_type,
       l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type,
       l3.src_node_id e3_src_node_id, l3.src_node_type e3_src_node_type,
       l3.link_id e3_link_id, l3.link_type e3_link_type,
       l3.dst_node_id e3_dst_node_id, l3.dst_node_type e3_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3,
    (select guid from es_191 where c_title = 'Physical review letters') journal
WHERE l1.src_node_type = 190  /* author */
  AND l1.dst_node_type = 186  /* article */
  AND l2.src_node_id = l1.dst_node_id
  AND l2.dst_node_type = 194  /* JournalIssue */
  AND l3.src_node_id = l2.dst_node_id
  AND l3.dst_node_type = 191  /* Journal */
  AND l3.dst_node_id = journal.guid
  ;
```

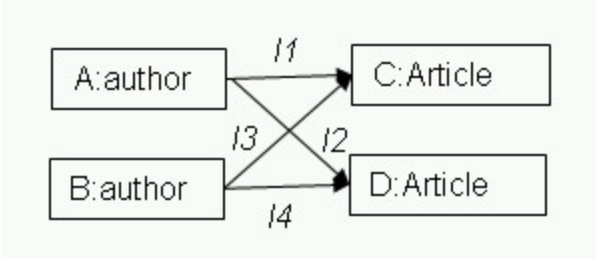
```
CREATE TABLE query_1000k AS
SELECT h1.e1_link_id,              h1.e1_link_type,
       h1.e2_link_id e5_link_id,  h1.e2_link_type e5_link_type,
       h1.e3_link_id e9_link_id,  h1.e3_link_type e9_link_type,
       h2.e1_link_id e2_link_id,  h2.e1_link_type e2_link_type,
       h2.e2_link_id e6_link_id,  h2.e2_link_type e6_link_type,
       h2.e3_link_id e10_link_id, h2.e3_link_type e10_link_type,
       h3.e1_link_id e3_link_id,  h3.e1_link_type e3_link_type,
       h3.e2_link_id e7_link_id,  h3.e2_link_type e7_link_type,
       h3.e3_link_id e11_link_id, h3.e3_link_type e11_link_type,
       h4.e1_link_id e4_link_id,  h4.e1_link_type e4_link_type,
       h4.e2_link_id e8_link_id,  h4.e2_link_type e8_link_type,
       h4.e3_link_id e12_link_id, h4.e3_link_type e12_link_type
  FROM query_half h1, query_half h2, query_half h3, query_half h4
 WHERE h1.e1_src_node_id = h2.e1_src_node_id  /* the source vertices at the start of the
pattern are the same */
   AND h2.e1_src_node_id = h3.e1_src_node_id
   AND h3.e1_src_node_id = h4.e1_src_node_id
   AND h1.e3_dst_node_id = h2.e3_dst_node_id  /* the dest. vertices at the end of the pattern
are the same */
   AND h2.e3_dst_node_id = h3.e3_dst_node_id
   AND h3.e3_dst_node_id = h4.e3_dst_node_id
   AND h1.e1_dst_node_id > h2.e1_dst_node_id  /* order the article vertices */
```

```
    AND h2.e1_dst_node_id > h3.e1_dst_node_id
    AND h3.e1_dst_node_id > h4.e1_dst_node_id
    ;
```

	time (seconds)	rows
path query	1.8	27418
part 2	20.0	957272
total	21.8	

Query 3

Find the subgraphs that consist of two authors who have co-authored two papers.

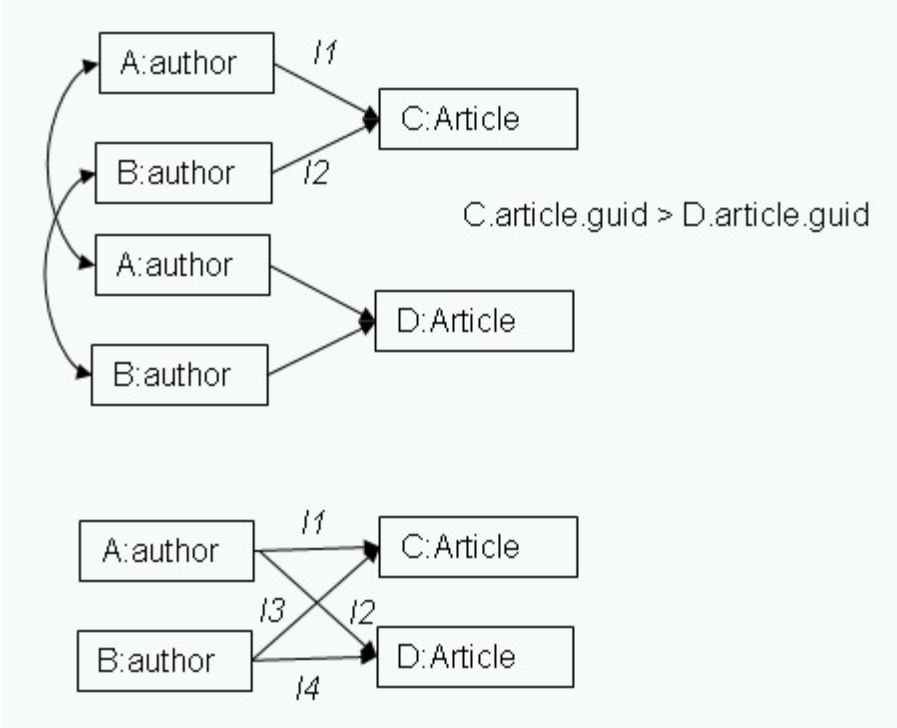


A single SQL statement that implements this query is shown below:

```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type,
       l3.link_id link_3_id, l3.link_type link_3_type,
       l4.link_id link_4_id, l4.link_type link_4_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3, e_pubmed_1000k l4
 WHERE l1.src_node_type = 190  /* author */
   AND l1.dst_node_type = 186  /* article */
   AND l2.dst_node_type = 186  /* article */
   AND l1.src_node_id = l2.src_node_id  /* l1.src and l2.src are the same vertex */
   AND l1.dst_node_id > l2.dst_node_id  /* l1.dst and l2.dst are different vertices */
   AND l3.src_node_type = 190  /* author */
   AND l3.dst_node_id = l1.dst_node_id  /* l3.dst is the same vertex as l1.dst */
   AND l4.dst_node_id = l2.dst_node_id  /* l4.dst is the same vertex as l2.dst */
   AND l3.src_node_id = l4.src_node_id  /* l3.src and l4.src are the same vertex */
   AND l1.src_node_id > l3.src_node_id;
```

time (seconds)	rows
3480.0 (58 minutes)	19067116

This query can be calculated more efficiently by breaking it into two sub-queries and then joining the result, as shown in the diagram below.




```
CREATE TABLE query_half AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type, l1.link_id e1_link_id,
l1.link_type e1_link_type, l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type, l2.link_id e2_link_id,
l2.link_type e2_link_type, l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type
FROM e_pubmed_1000k l1, e_pubmed_1000k l2
WHERE l1.src_node_type = 190 /* author */
  AND l1.dst_node_type = 186 /* article */
  AND l2.src_node_type = 190 /* author */
  AND l1.dst_node_id = l2.dst_node_id
  AND l1.src_node_id > l2.src_node_id;
```

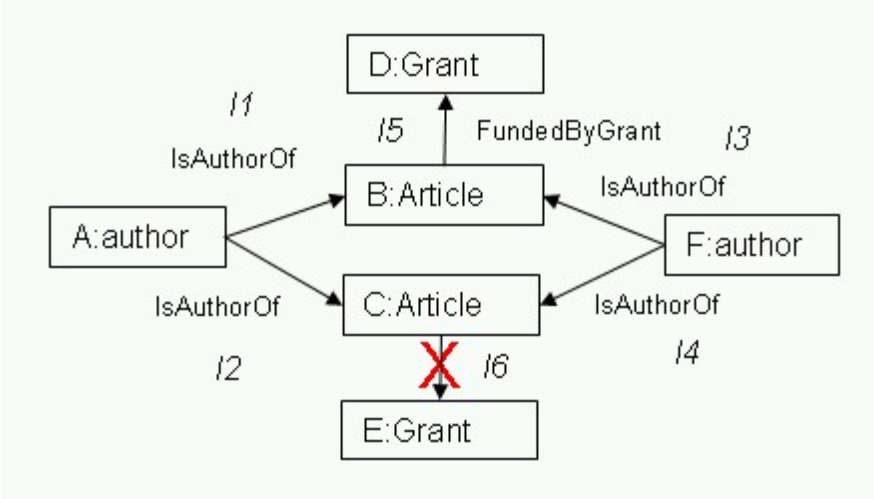
```
CREATE TABLE query_result AS
SELECT h1.e1_link_id e1_link_id, h1.e1_link_type e1_link_type, h1.e2_link_id e3_link_id,
h1.e2_link_type e3_link_type,
       h2.e1_link_id e2_link_id, h2.e1_link_type e2_link_type, h2.e2_link_id e4_link_id,
h2.e2_link_type e4_link_type
FROM query_half h1, query_half h2
WHERE h1.e1_src_node_id = h2.e1_src_node_id
  AND h1.e2_src_node_id = h2.e2_src_node_id
  AND h1.e1_src_node_id > h2.e2_src_node_id
  AND h1.e1_dst_node_id > h2.e1_dst_node_id
  ;
```

	time (seconds)	rows
part 1	51.0	7297174
part 2	336.0	19067116
total	387 (6.45 minutes)	

Query 4

In knowledge discovery applications it can be important to be able to specify a query where an edge is *excluded* from the pattern. The query below specifies a pattern that contains an article that has an

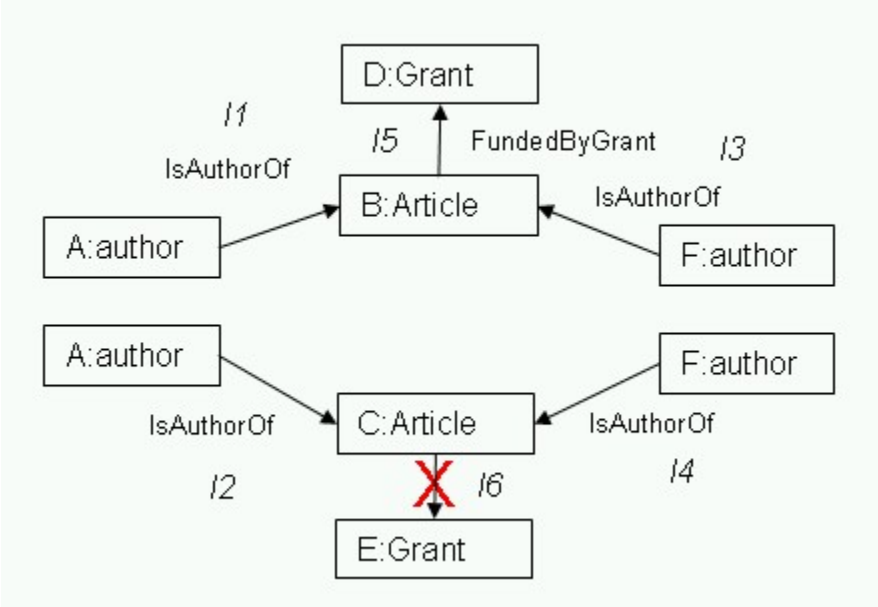
associated grant and an article that does not have an associated grant. A single SQL statement for this query is shown below.




```
CREATE TABLE query_1000k AS
SELECT l1.link_id link_1_id, l1.link_type link_1_type,
       l2.link_id link_2_id, l2.link_type link_2_type,
       l3.link_id link_3_id, l3.link_type link_3_type,
       l4.link_id link_4_id, l4.link_type link_4_type,
       l5.link_id link_5_id, l5.link_type link_5_type
  FROM e_pubmed_1000k l1, e_pubmed_1000k l2, e_pubmed_1000k l3,
       e_pubmed_1000k l4, e_pubmed_1000k l5
 WHERE l1.link_type = 313      /* IsAuthorOf */
   AND l1.src_node_type = 190  /* author */
   AND l1.dst_node_type = 186  /* article */
   AND l2.link_type = 313      /* IsAuthorOf */
   AND l2.dst_node_type = 186  /* article */
   AND l1.src_node_id = l2.src_node_id  /* l1.src and l2.src are the same vertex */
   AND l1.dst_node_id > l2.dst_node_id  /* l1.dst and l2.dst are different vertices */
   AND l5.link_type = 312      /* FundedByGrant */
   AND l5.src_node_id = l1.dst_node_id
   AND l5.dst_node_type = 185  /* Grant */
   AND l3.link_type = 313      /* IsAuthorOf */
   AND l3.src_node_type = 190  /* author */
   AND l3.dst_node_id = l1.dst_node_id  /* l3.dst is the same vertex as l1.dst */
   AND l4.link_type = 313      /* IsAuthorOf */
   AND l4.dst_node_id = l2.dst_node_id  /* l4.dst is the same vertex as l2.dst */
   AND l3.src_node_id = l4.src_node_id  /* l3.src and l4.src are the same vertex */
   AND l1.src_node_id > l3.src_node_id
   AND NOT EXISTS(
          SELECT 1
            FROM e_pubmed_1000k l6
           WHERE l6.link_type = 312                        /* FundedByGrant */
           AND l6.src_node_id = l4.dst_node_id
             AND l6.dst_node_type = 185  /* grant */
          );
```

time (seconds)	rows
6840.0 (1.9 hours)	7630

Calculating paths through the query yields considerably better performance:




```
CREATE TABLE query_half_grant AS
SELECT l1.src_node_id e1_src_node_id, l1.src_node_type e1_src_node_type, l1.link_id e1_link_id,
l1.link_type e1_link_type, l1.dst_node_id e1_dst_node_id, l1.dst_node_type e1_dst_node_type,
       l3.src_node_id e3_src_node_id, l3.src_node_type e3_src_node_type, l3.link_id e3_link_id,
l3.link_type e3_link_type, l3.dst_node_id e3_dst_node_id, l3.dst_node_type e3_dst_node_type,
       l5.src_node_id e5_src_node_id, l5.src_node_type e5_src_node_type, l5.link_id e5_link_id,
l5.link_type e5_link_type, l5.dst_node_id e5_dst_node_id, l5.dst_node_type e5_dst_node_type,
FROM e_pubmed_1000k l1, e_pubmed_1000k l3, e_pubmed_1000k l5
WHERE l1.src_node_type = 190 /* author */
  AND l1.link_type = 313      /* IsAuthorOf */
  AND l1.dst_node_type = 186 /* article */
  AND l3.src_node_type = 190 /* author */
  AND l3.link_type = 313      /* IsAuthorOf */
  AND l1.dst_node_id = l3.dst_node_id
  AND l1.src_node_id > l3.src_node_id
  AND l5.src_node_type = 186  /* article */
  AND l5.src_node_id = l3.dst_node_id
  AND l5.dst_node_type = 185  /* grant */
  ;
```

```
CREATE TABLE query_half_no_grant AS
SELECT l2.src_node_id e2_src_node_id, l2.src_node_type e2_src_node_type, l2.link_id e2_link_id,
l2.link_type e2_link_type, l2.dst_node_id e2_dst_node_id, l2.dst_node_type e2_dst_node_type,
       l4.src_node_id e4_src_node_id, l4.src_node_type e4_src_node_type, l4.link_id e4_link_id,
l4.link_type e4_link_type, l4.dst_node_id e4_dst_node_id, l4.dst_node_type e4_dst_node_type
FROM e_pubmed_1000k l2, e_pubmed_1000k l4
WHERE l2.src_node_type = 190 /* author */
  AND l2.link_type = 313    /* IsAuthorOf */
  AND l2.dst_node_type = 186 /* article */
  AND l4.src_node_type = 190 /* author */
  AND l4.link_type = 313    /* IsAuthorOf */
  AND l2.dst_node_id = l4.dst_node_id
  AND l2.src_node_id > l4.src_node_id
   AND NOT EXISTS(
          SELECT 1
            FROM e_pubmed_1000k l6
           WHERE l6.link_type = 312                          /* FundedByGrant */
           AND l6.src_node_id = l4.dst_node_id
           AND l6.dst_node_type = 185  /* grant */
       )
   ;
```

```
CREATE TABLE query_result AS
SELECT h1.e1_link_id, h1.e1_link_type, h1.e3_link_id, h1.e3_link_type, h1.e5_link_id,
```

```
  h1.e5_link_type,
        h2.e2_link_id, h2.e2_link_type, h2.e4_link_id, h2.e4_link_type
FROM query_half_grant h1, query_half_no_grant h2
WHERE h1.e1_src_node_id = h2.e2_src_node_id
  AND h1.e3_src_node_id = h2.e4_src_node_id
  AND h1.e1_src_node_id > h2.e4_src_node_id
  AND h1.e3_dst_node_id > h2.e4_dst_node_id
  ;
```

	time (seconds)	rows
part 1 (query_half_grant)	162.0	548855
part 2 (query_half_no_grant)	78.0	7085770
join	42	7630
total	282.0 (4.7 minutes)	

Acknowledgments

Justin Levandoski did some of the early foundation work on translating pattern queries into single SQL statements and benchmarking these queries on multiple databases.

References

1. *SQL 1999: Understanding Relational Language Components* by Jim Melton and Alan R. Simon, Morgan Kaufmann, 2002
2. *A visual language for querying and updating graphs* by H. Blau, N. Immerman and D. Jensen, 2002, University of Massachusetts Amherst Computer Science Technical Report 2002-037
3. *Proximity 4.2 QGraph Guide* by David Jensen, 2006, University of Massachusetts Amherst
4. *The Graph Query Language*, David Silberberg, July 18, 2006, from a presentation at the Lawrence Livermore National Laboratory
5. *A Semantic Graph Query Language* by Kaplan, I.L., October 16, 2006, Lawrence Livermore National Laboratory Technical Report UCRL-TR-225447
6. *An Algorithm for Subgraph Isomorphism* by Ullman, J, Journal of the ACM, January 1976 (Volume 23, Issue 1, Pgs 31 - 42)