Query Optimization for a Graph Database with Visual Queries

Greg Butler, Guang Wang, Yue Wang, and Liqian Zou

Department of Computer Science and Software Engineering,
Concordia University,
1455 de Maisonneuve Blvd. West,
Montréal, Québec, H3G 1M8, Canada
{gregb, gwang_1, wang_yue, l_zou}@cse.concordia.ca

Abstract. We have constructed a graph database system where a query can be expressed intuitively as a diagram. The query result is also visualized as a diagram based on the intrinsic relationship among the returned data. In this database system, CORAL plays the role of a query execution engine to evaluate queries and deduce results. In order to understand the effectiveness of CORAL optimization techniques on visual query processing.We present and analyze the performance and scalability of CORAL's query rewriting strategies, which include *Supplementary Magic Templates*, *Magic Templates*, *Context Factoring*, *Naïve Backtracking*, and *Without Rewriting* method. Our research surprisingly shows that the *Without Rewriting* method takes the minimum total time to process the benchmark queries. Furthermore, CORAL's default optimization method *Supplementary Magic Templates* is not uniformly the best choice for every query. The "optimization" of visual queries is beneficial if one could select the right optimization approach for each query.

1 Introduction

Scientific and industrial projects have been generating large volumes of data. This tremendous amount of data need storage and analysis. A key issue is that the data management software needs to be easy–to–use, yet provides fast response time. It is not trivial to make a database system simple and intuitive enough for the end-users to query in a sophisticated way. Our graph database system [1] is toward solving this important problem. We believe that the diagrammatic query and visual result display will ease the task of data management and data analysis. We have applied the benefits of deductive query language, diagrammatic queries, and data visualization so as to provide the end-users, who are not familiar with or do not want to bother with writing SQL queries, a helpful system to pose queries and represent query results in a diagram.

In brief, our graphical user interface allows the end–users to construct queries by drawing diagrams. It is implemented in JAVA. The supported graphical query language we have implemented is GraphLog [2]. The query result set is visualized as diagrams with the same icon and style as in the query. CORAL [3] is the system's database engine. The raw data are stored in a MySQL [4] relational

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 602–616, 2006.

database. The detailed description about our system's query formulation and result visualization mechanism can be found at [1].

In order to handle the huge size data in a real–world application, we need to study the possible optimization techniques that are effective for diagrammatical queries so as to speed up the query processing. There have been quite a number of graph database systems presented in the literature. In general, they have been lack of full studies of the query language expressiveness, semantics and optimization. To the best of our knowledge, no performance studies of optimization strategies for graph databases have been done prior to our work.

Our first step towards an optimization solution is to understand the effect of CORAL's optimization techniques on the diagrammatical queries in our context. In this paper, we will present the study of CORAL's query optimization techniques in our context. We have used a benchmark of 24 queries across a range of different data sizes. This query set on the University model are carefully chosen and typical enough to evaluate our system's ability to express queries at different complex levels. The data sets used for our experiment range from 640,000 pieces of ground facts to 5,100,000. The wide range of data sets are comprehensive enough to examine our system's capability to handle large data sets.

The capabilities of the system has been expanded to include all features of GraphLog. Our tests and demonstration were performed with a system capable of handling selection, projection, queries with negation, and transitive queries. It also supports blobs, which help to modularize queries with hierarchical relationships and layout the results in the orthogonal shape.

The rest of this paper is organized as follows: Section 1.1 introduces the evaluation framework. Section 2 illustrates the structure of the Graph Database System. Section 2.1 gives an example of query and its translations across the system. Section 3 presents and analyzes the performance experiment results for various CORAL query optimization strategies. Section 4 discusses the related work and Section 5 concludes the paper.

1.1 Test Benchmark

The benchmark used in the test was a framework for object-oriented query language evaluation [5]. It was built on a University Model. Originally, it was a guideline for designing a new query language and improving the performance of existing languages.

The University Model is a simplified version of a university administration system that manages the personal and academic information of students and staff members. The structural relationships among the classes defined in the schema are given in Figure 1.

The class `Person` has two subclasses `Staff` and `Student`. `Visiting_staff` is the subclass of the `Staff`. Both an object of class `Staff` and an object of class `Student` can be an object of class `Tutor`. A student may be supervised by one or more staff members. A staff member may be a supervisor for one or more students. However, some students do not have a supervisor and some staff members do not supervise a student. Every staff member works in a `Department`

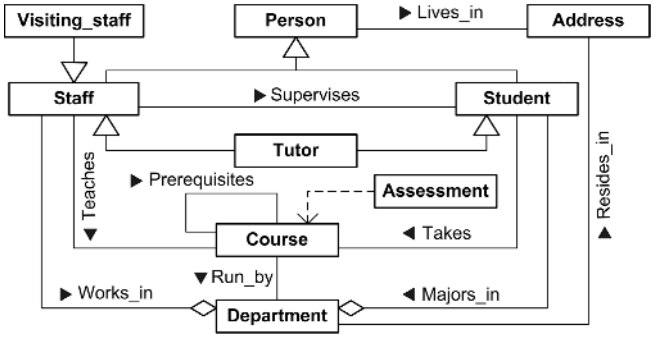


Fig. 1. The university model structure

and every student studies a specific major in a department. Each department offers `Course` for students. The staff members are lecturers for these courses. In some cases, the students need to take and pass the prerequisite courses before taking a course. Each course has an object of `Assessment` that specifies its credits and schools terms that offer it. Every person lives in a place that is defined by `Address`. Each department has a location, which is an object of class `Address`.

The benchmark contains four evaluation dimensions: *expressive power*, *support of object-orientation*, *support of collections* and *usability*. Each evaluation dimension is composed of a set of criteria and each criteria is assessed by a set of proposed queries on the University Model (Appendix A). The support of *object-orientation* mainly concerns the object identity, method calling, complex objects, class hierarchy and dynamic binding of the system. The *expressive power* approach is to test the object manipulation features of the system, such as nested queries and relational completeness. The support of *collections* tries to find a set of operations on the system that can obtain consistent performance on different collection classes. It is also used to test on the mixing of and conversion between different collection classes. The *usability* aims at examining the ease of using query notations.

2 System Architecture

Our Graph Database System takes up four subsystems. They are **Graphical User Interface**, **TGL Translator**, **Query Processing Engine** and **MySQL Data Storage**. The system architecture is shown in Figure 2.

The **Graphical User Interface (GUI)** is the system's interface to end–users. Users can draw a query diagram in the query editor. The GUI translates the user's query that is defined as a diagram into XML format and sends it to the next layer of the system: TGL Translator. The GUI is also responsible to visualize the query result set into a graph.

The **TGL Translator** consists of the *query translation component* and the *result translation component*. The query translation component accepts the query

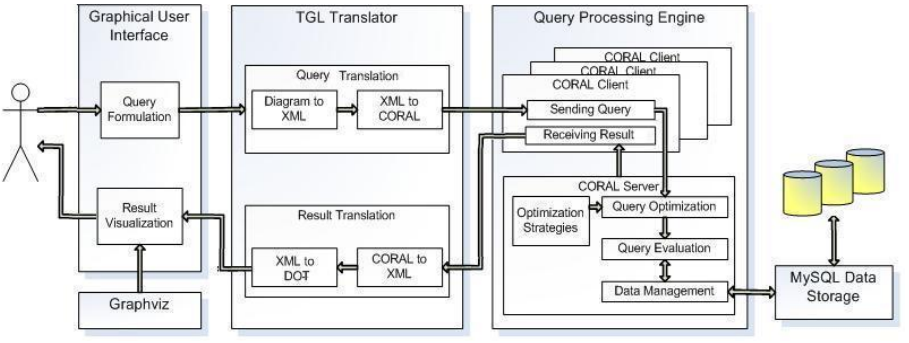


Fig. 2. Graph database system architecture

diagram from the GUI and first translates it into the XML format with pre–defined tags. Then it translates the XML–formatted query into a CORAL query program. The result translation component translates the query result returned by CORAL into XML and passes the XML–formatted query result to the upper GUI layer.

The **Query Processing Engine** is responsible for evaluating the query and deducing the result. It consists of two components: the CORAL client and the CORAL server. The CORAL client interacts with the TGL Translator and the MySQL Data Storage. It is responsible to receive the query plan from the TGL Translator and the query result from the CORAL server. The CORAL client terminates when the query finishes, whereas the CORAL server will live until the user requires to shut it down.

The CORAL Server is the deductive engine to optimize the query and execute the query. The *query optimization* part transforms the incoming queries to an internal representation based on the optimization(rewriting) methods used in the query. In the *optimization strategies*, several control annotations are added to the original query program. This optimized program is transferred to the *query evaluation* part. The *query evaluation* part takes the annotated program and in-memory facts as input and executes the program under the direction of annotations. The *data management* part is in charge of maintaining and manipulating the facts for each query. It loads data from the client interface of MySQL Data Storage and converts the data into CORAL facts on demand [11].

The **MySQL Data Storage** stores the data source physically in MySQL database. The conventional data manipulations can be performed on data in MySQL. The CORAL server initiates a connection with MySQL. All records in the target database are loaded into the CORAL server's computer main memory as a runtime database for CORAL.

2.1 A Query Example

The translation of queries adds flexibility to the system. The transformation of a query from a diagram to XML representation is a process of depicting

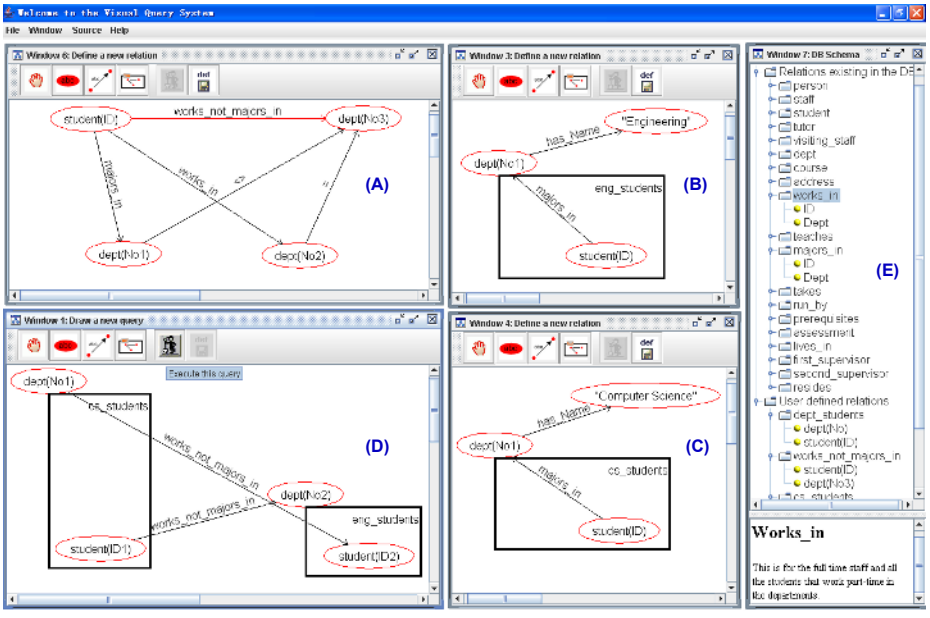


Fig. 3. The query interface. The user posed a query which returns all of the Computer Science and Engineering students who do part-time jobs across these two departments. **(A)** Defining a new relation called `works_not_majors_in`. All students who work part–time in a department that is different from their major department satisfy this concept. **(B)** Defining another new relation `eng_students`. A **blob** is used, which is a container containing all students in the Engineering department. **(C)** Similarly, defining a relation called `eng_students` for all the Computer Science students. **(D)** This is the query diagram. It makes use of the relations defined in the first three windows. **(E)** An overview of all the relations present in the database and the user–defined relations.

the query diagram in format of XML with pre–defined tags. The structure of an XML representation for a query diagram follows the Transferable Graphic Language (TGL) schema. The TGL translator builds up a mapping between an XML document that conforms to the TGL schema and a CORAL program. The detailed description about how this mapping is done can be found in [6].

In order to illustrate this procedure, we provide an example query, which returns all of the Computer Science and Engineering students who are doing part–time jobs across these two departments. Its query diagram in our graph database system is shown in Figure 3. The detailed description about our system's query formulation and result visualization mechanism can be found at [1].

The TGL translator translates both the relation definition diagrams and the query diagram into the XML–formatted documents. The following XML document is translated from the query diagram in Figure 3. There are four elements under the `<distinguished-show>` element, meaning that these four elements should be returned as the query result. It consists of two `<edge>` elements for `works_not_majors_in` as well as two `<blob>` elements named `cs_students` and

eng_students. In contrast, the nodes student(ID1), student(ID2), dept(ID1) and dept(ID2) are under the <content> element. They make up the context of the query diagram.

```
<graphlog>
<showGraphlog>
 <id>tempQueryResult</id>
 <distinguished-show>
   <edge>
     <id>EID3_0</id>
     <predicate>works_not_majors_in</predicate>
     <FromNodeID>NID0003</FromNodeID>
     <ToNodeID>NID0000</ToNodeID>
   </edge>
   <edge>
     <id>EID1_2</id>
     <predicate>works_not_majors_in</predicate>
     <FromNodeID>NID0001</FromNodeID>
     <ToNodeID>NID0002</ToNodeID>
    </edge>
   <blob>
     <id>BID0006</id>
     <predicate>cs_students</predicate>
     <outerNodeID>NID0000</outerNodeID>
     <innerNodeID>NID0001</innerNodeID>
   </blob>
   <blob>
     <id>BID0007</id>
     <predicate>eng_students</predicate>
     <outerNodeID>NID0002</outerNodeID>
     <innerNodeID>NID0003</innerNodeID>
   </blob>
 </distinguished-show>
 <content>
   <node>
        <id>NID0000</id>
        <entity>
          <name>dept</name>
          <field>No1</field>
        </entity>
   </node>
   <node>
        <id>NID0001</id>
        <entity>
          <name>student</name>
          <field>ID1</field>
        </entity>
   </node>
   <node>
        <id>NID0002</id>
        <entity>
          <name>dept</name>
          <field>No2</field>
        </entity>
   </node>
   <node>
        <id>NID0003</id>
        <entity>
          <name>student</name>
          <field>ID2</field>
        </entity>
   </node>
 </content>
</showGraphlog>
</graphlog>
```

The following two CORAL program modules are generated by the TGL Translator. They are translated as the definition of relation works_not_majors_in and relation cs_students. The TGL translator generates a similar CORAL query program for eng_students. The CORAL query program translated directly from the XML query in the previous page makes use of these three CORAL relation definitions in order to generate the final answer.

```
module cs_students.
export cs_students(ff).
eid1_0(ID,No1) :- majors_in(ID,No1).
cs_students (No1,ID) :-
   dept(No1,"Computing Science"),
   eid1_0(ID,No1).
end_module.
```

```
module works_not_majors_in.
export works_not_majors_in(ff).
  eid0_3(ID,No2) :- works_in(ID,No2).
  eid0_2(ID,No1) :- majors_in(ID,No1).
  works_not_majors_in(ID , No3):-
        eid0_3(ID,No2),
        eid0_2(ID,No1),
        No2 = No3, No1 <> No3.
end_module.
```

The CORAL query program translated directly from the XML query in the pervious page makes use of these three CORAL relation definitions in order to generate the final answer.

Figure 4 shows the query result for the example query. The diagram clearly shows two clusters, one for CS department students and the other for Engineering department students. In addition, all of these students are working part-time

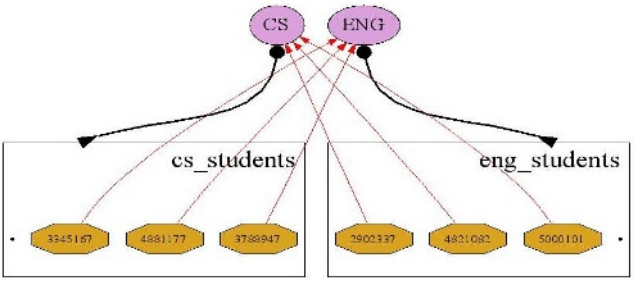


Fig. 4. Query result for the example query

across these two departments: three Engineering students identified by their student IDs are working in the CS department, and three CS students are working in the Engineering department.

3 Optimization Experiment

Deductive databases allow a view to be defined using logical rules, and allow logical queries against the view. Since the rules allow recursive definitions, the resulting expressive power of the query language is greater than the relational query languages. Graph query languages are even more expressive, and provide a visual representation.

In our graph database system, CORAL works as the deductive engine to evaluate queries and deduce results. We chose CORAL for its flexibility to connect to relational database, and its declarative nature for ease of translation from/to a GraphLog query diagram. In order to find the possible optimization solutions for our visual queries, we first test the effectiveness of CORAL's optimization strategies in our context.

3.1 Preliminaries

CORAL is a deductive database system that supports a declarative language. Every CORAL program is a collection of modules, each of which can be separately compiled into CORAL internal data structures. The modules may include facts and rules. In a declarative environment, a fact is the same thing as a tuple in a relation or a row in an SQL table. A rule is a way to derive new facts. We can say the facts are the unconditional rules. The collection of all facts are stored physically in the database, called the *existential database*. The set of all facts that we can derive from the base set of facts are not stored physically in the database, called the *intensional database*.

Modules are the units of optimization and also the units of evaluation. Evaluation techniques can be chosen on a per-module basis, and different modules with different evaluation techniques can interact in a transparent fashion. Although

CORAL developed a number of query evaluation strategies, it still uses heuristic programming rather than a cost estimation package to choose evaluation methods.

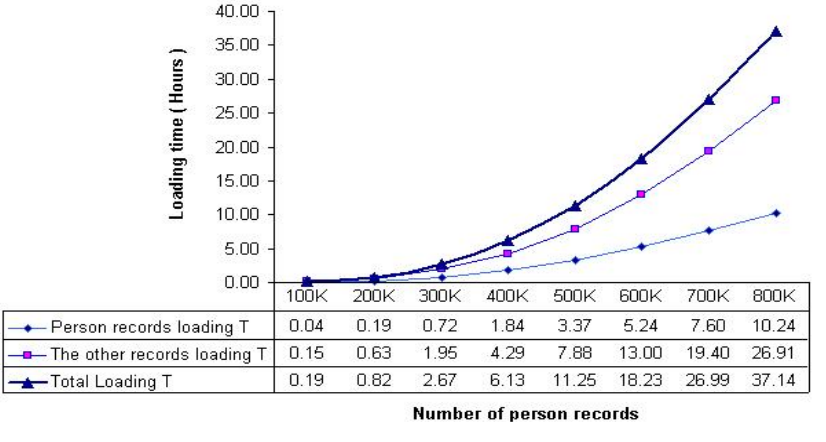
The user can specify high-level annotations at the level of each module, to guide the query optimization. User-level annotations can be added directly to the source code and they give the programmer freedom to control query's optimization as well as evaluation. CORAL's user-level annotations are divided into Rewriting Annotations, Execution Annotations, and Per-Predicate Annotations. Presently, CORAL's Rewriting Annotations, include *Supplementary Magic Templates* [7], *Magic Templates* [8], *Context Factoring* [9], *Naïve backtracking* [10], and *Without Rewriting* method. CORAL's default rewriting (optimization) method is *Supplementary Magic Templates*.

3.2 Data Loading

The test platform was a SunFire 280R with two 900MHz UltraSparc-III+ CPUs and 4GB physical memory. The operating system is Solaris 9. We tried the five annotations mentioned above individually on each query and recorded the query execution time on the CORAL server.

We have used eight data sets for our experiment to subjectively evaluate CORAL's optimization performance. Our script generates a MySQL program which inserts the data records into MySQL database. The total number of records in a data set is a function of the number of person records. The number of courses and addresses are constants, which is 500 and 2000 respectively. The size of the other tables depends on the number of person records, and(or) the number of course records, and(or) the number of address records.

Figure 5 plots the data loading time from MySQL data storage to CORAL run-time workspace for each data set. The x–axis shows the variable of number of person records, and y–axis, shows the range of the variable of data loading time in units of hours. Thus, the graph is showing us the change in loading



	100K	200K	300K	400K	500K	600K	700K	800K
Person records loading T	0.04	0.19	0.72	1.84	3.37	5.24	7.60	10.24
The other records loading T	0.15	0.63	1.95	4.29	7.88	13.00	19.40	26.91
Total Loading T	0.19	0.82	2.67	6.13	11.25	18.23	26.99	37.14

Fig. 5. MySQL to CORAL workspace data loading time

time from MySQL to CORAL over the increasing data set size. The curve at the bottom shows the time CORAL takes to load the person records. The curve in the middle shows represents the time CORAL takes to load all of the rest recodes. The curve at the top most of the graph demonstrates the total time CORAL takes to load all of the records in the database.

On the graph, it is easy to see that the total time CORAL takes to load MySQL data steadily rise over the data set size, from a low of about 0.2 hour in the data set based on 100,000 person records to a level of about 37 hours in the largest data set based on 800, 000 person records. It is observed that when the data set size is doubled, the loading time is about six times longer. It is tolerable for CORAL to spend hours loading in the data, as it is only the one- time cost for loading all the data to the CORAL workspace.

3.3 Query Processing

In each test run, all five optimization techniques mentioned above were tried out on each query. Figure 6 is showing CORAL total execution time for 24 queries across five optimization methods at each test run by using the cluster columns.

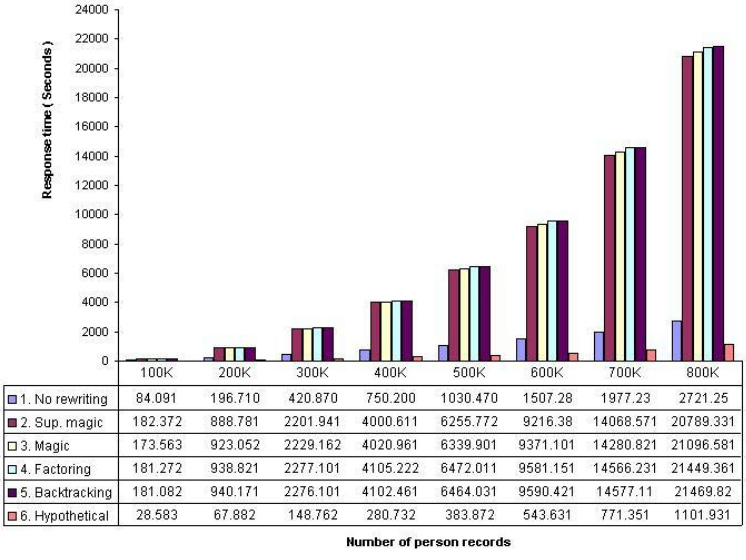
In detail, the x–axis shows the variable of number of person records, and the y–axis shows the range of the variable of CORAL execution time in units of seconds. On the graph, there are altogether eight cluster columns representing eight test runs. One cluster column is for one test run on a particular data set. Each cluster is made up of six vertical bars, for six kinds of CORAL optimization methods, in sequence of *No rewriting*, *Supplementary magic*, *Magic*, *Factoring*, *Naïve backtracking*, and *Hypothetical method*. The higher a vertical bar, the longer time that CORAL takes to execute all of the 24 queries using the optimization method the vertical bar represents.

CORAL has implemented the first five optimization methods. The sixth optimization method, as its name tells, is our assumption of such a method's presence. This "hypothetical method" can intelligently choose a fastest optimization method among CORAL's five optimization methods. It is not implemented as an optimization method in CORAL.

We have used 30 queries in the test benchmark for our experiment. However, in Figure 6, we omitted some queries (Q12, Q15, Q16, Q28, Q29, Q30). For the case of Q12, Q15 and Q16, CORAL does not return the query result after 15–20 minutes even for the smallest data set. We found out that the problem of Q12 is that its translated CORAL program involves cycles on negation of a sub–query, which is not supported by CORAL. Q15's problem is that, it has to include the same relation twice in the query program body. Q16 introduces a free variable which makes the CORAL program unsafe. For the case of Q28, Q29, and Q30, none of CORAL's rewriting methods has a response after 15–20 minutes except for the smallest data set.

3.4 Analysis

With the experiment results summarized in Figure 6, we have the following three main findings:



	100K	200K	300K	400K	500K	600K	700K	800K
1. No rewriting	84.091	196.710	420.870	750.200	1030.470	1507.28	1977.23	2721.25
2. Sup. magic	182.372	888.781	2201.941	4000.611	6255.772	9216.38	14068.571	20789.331
3. Magic	173.563	923.052	2229.162	4020.961	6339.901	9371.101	14280.821	21096.581
4. Factoring	181.272	938.821	2277.101	4105.222	6472.011	9581.151	14566.231	21449.361
5. Backtracking	181.082	940.171	2276.101	4102.461	6464.031	9590.421	14577.11	21469.82
6. Hypothetical	28.583	67.882	148.762	280.732	383.872	543.631	771.351	1101.931

Fig. 6. CORAL response time for 8 test runs (24 queries in each run)

(A) There is neither sudden drop nor rise in terms of each vertical bar (representing a rewriting method). Regardless of the rewriting strategies, the CORAL's response time for all of 24 queries continually rise over the data set sizes, from a low of about 29 seconds in the smallest data set to a level of 24,000 seconds ($\approx$ 6 hours). It is a reasonable growth as the larger data set consumes longer time for CORAL to execute the queries.

(B) It is common for every test run that CORAL's four rewriting methods take much longer than its no–rewriting method. This implies that CORAL's four rewriting methods slow down the query processing in certain queries: namely Q11 and Q13, and Q19 (From Table 1). For these three queries, CORAL's rewriting methods should not be used. Q11 asks for students taking a course given by "Steve Johnson". Q13 asks for all the names of the students. Q19 asks for students living in the area of Hillhead, Kelvinside and Dowanhill.

(C) The"hypothetical method" significantly outperforms the other five optimization techniques. Recall that this "hypothetical method" is one of our assumptions about CORAL's optimization strategies. It can intelligently pick the fastest optimization method among CORAL's five optimization methods.

A common behavior among CORAL's optimization methods is that the CORAL response time for each query of a certain optimization method continually rises up over the data set. Under an optimization method, the larger the data set, the longer time that CORAL takes for a query. Table 1 shows the CORAL response time particularly for the largest data set based on 800,000 person records. The numeric values in the table is in unit of seconds. Combining it with the other seven experiment results, we have discovered the following interesting facts:

Table 1. CORAL response time in unit of seconds with 5 rewriting strategies for the data set of 800,000 persons, totally 5,000,000 records

	No rewriting	Sup. magic	Magic	Factoring	Back tracking
Q1	3.000	1.570	1.600	1.570	1.560
Q2	13.550	13.850	14.700	14.180	14.480
Q3	105.550	0.040	0.030	0.030	0.040
Q4	492.950	1.270	0.810	0.830	0.830
Q5	0.010	0.020	0.010	0.020	0.030
Q6	1.320	1.320	1.340	1.320	1.320
Q7	527.690	105.980	113.650	114.740	114.880
Q8	6.160	6.380	6.410	6.430	6.450
Q9	12.330	12.290	12.720	12.620	12.620
Q10	16.480	18.760	20.060	20.180	20.240
Q11	11.540	18561.200	19013.000	19000.800	19022.400
Q13	791.740	1032.840	1216.960	1226.420	1229.840
Q14	0.150	0.190	0.180	0.200	0.190
Q17	12.490	13.090	13.610	13.050	13.380
Q18	9.650	0.001	0.001	0.001	0.010
Q19	109.110	994.400	662.670	1012.790	1007.290
Q20	2.370	2.440	2.440	2.460	2.450
Q21	0.110	0.110	0.110	0.100	0.100
Q22	233.650	1.110	0.110	0.130	0.130
Q23	134.740	17.500	11.690	17.040	17.070
Q24	0.150	0.160	0.160	0.150	0.160
Q25	0.100	0.110	0.120	0.110	0.110
Q26	0.810	0.800	0.810	0.800	0.800
Q27	235.600	3.900	3.390	3.390	3.440

(1) CORAL's rewriting techniques have pronounced effect on 8 queries (Q1, Q3, Q4, Q7, Q18, Q22, Q23, Q27), with the improvement from the low of 2 times faster in Q1 with Supplementary Magic method for the data set based on 800,000 person records, to the level of 2500 times faster in Q18 for the same data set. In contrast, the remaining 13 queries do not benefit or suffer when using CORAL's rewriting techniques in every experiment, since the improvement with an optimization technique is marginal ($\leq 20\%$).

(2) CORAL's default optimization method *Supplementary Magic* is not guaranteed to defeat CORAL's the other three optimization methods under all circumstances. The evidence is from Q3, Q4, Q22, Q23, and Q27 across all the test runs. Due to the space limitation, we only discuss the case of 800,000 person records data set as an evidence, as it is shown in Table 1. In Q3, both *Magic* and *Factoring* takes only 75% of the time consumed by applying *Supplementary Magic* method. In Q4 and Q22, *Magic* takes as low as 63% and 66% respectively of the time consumed by using *Supplementary Magic* method. For Q22's case, the *Supplementary Magic* method is even more worse: the other three methods takes only around 10% of the time consumed by *Supplementary Magic*.

(3) There is no universal best nor worst optimization technique. If one optimization technique fails, so do the other three techniques.

We categorized the benchmark queries into three classes: queries that benefit from optimization, queries that suffer from optimization, and queries that keep neutral about optimization. We have summarized their CORAL query program(s) characteristics in the Table 2.

Table 2. Characterizing queries based upon their optimization effect

	Optimization is beneficial								Optimization is harmful					
	Q1	Q3	Q4	Q7	Q18	Q22	Q23	Q27	Q11	Q13	Q19	Q28	Q29	Q30
atomic values	+			+		+	+	+	o		o	o	o	
don't care symbols	+	+									o		o	
≥ 3 joins		+	+	+			+		o			o	o	o
recursion					+									
relations union						+		+			o	o		
naïve										o				

4 Related Work

There have been numerous graph database systems presented in the literature [15, 16, 17, 18, 19, 20, 21, 22]. In general, they have been research prototypes that lack full studies of the query language expressiveness, semantics and optimization. The implementations have been limited. To the best of our knowledge no performance studies of optimization strategies for graph databases have been done prior to our work.

Our work builds upon the Hy+ system [2] developed by Alberto Mendelzon in Toronto. They designed the GraphLog language, studied its expressive power, and implemented a prototype system in Smalltalk. Their implementation translated GraphLog into several logic-based systems including CORAL. However, the only performance study [23] that they did was a comparison of the naïve translation to Datalog/CORAL with a translation to factored Datalog using automata.

In the field of deductive databases there has been extensive research on the optimization of queries for Datalog (and its variants). The major interest has been the optimization of recursive queries. Ceri et al. [14] provide an excellent summary of the field. The evaluation or comparison of optimization strategies is typified by Bancilhon and Ramakrishnan [12, 13] who develop analytical cost models for the optimization strategies when applied to four queries (related to the parent and ancestor relations) and then generate numerical data from the analytical models using synthetic data driven by three shapes — tree, inverted tree, and cylinder — for the "family tree". The state-of-the-art is perhaps best summarized in a quote [24]: *"Related work on the performance of recursive queries and their evaluation algorithms has considered either worst case performance, or performance over structured synthetic databases, or empirically measured performance over randomly generated relations."* The community has not developed extensive benchmarks nor carried out extensive performance comparisons.

5 Conclusion

In this paper, we have studied the optimization of visual queries for our graph database using a benchmark of 24 queries across a range of different data sizes. Our aim is to understand the effectiveness of CORAL's optimization techniques

on the diagrammatic queries. Recall that our database supports defining queries in a diagrammatic form and visualization of the query results. With the extensive experiment results, we are able to conclude that it is beneficial to optimize a visual query. Nevertheless, within the scope of the 24 benchmark queries, applying one optimization technique uniformly in general was worse than applying no optimization to the queries. It is important to utilize the optimization strategies in CORAL when appropriate as there is very slow execution for some queries if no optimization is used. Our research indicates that, a "smart" selection that is able to determine which kind of rewriting method to apply on a given query may profoundly improve the performance of the query execution engine.

References

1. Butler, G., Wang, G., Wang, Y., Zou, L.: A Graph Database with Visual Queries for Genomics. Procs. of the 3rd Asia-Pacific Bioinformatics Conf. (2005) 31–40
2. Consens, M.P.,Eigler, F.Ch., Hasan, M.Z., Mendelzon, A.O., Noik, E.G., Ryman, A.G., Vista, D.: Architecture & Applications of the Hy+ Visualization System. IBM Systems Journal, Vol. 33, No. 3 (1994) 458–476
3. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The CORAL Deductive System. VLDB Journal, Vol. 3, No. 2 (1994) 161–210
4. Widenjus, M., Axmark, D.: MySQL Reference Manual. O'Reilly (2002)
5. Chan, K.C., Trinder, P.W., Welland, R.: Evaluating Object-Oriented Query Languages. The Computer Journal, Vol 37, No. 10 (1994) 858–872
6. Zou, L.: GraphLog: Its Representation in XML & Translation to CORAL. Masters Thesis. Dept. of Computer Science, Concordia University (2003)
7. Beeri, C., Ramakrishnan, R.: On the Power of Magic. Procs. of the ACM Symp. on Principles of Database Systems (1987) 269–283
8. Ramakrishnam, R.: Magic Templates: A Spellbinding Approach to Logic Programs. Procs. of the Intl. Conf. on Logic Programming (1988) 140–159
9. Naughton, J.F., Seshadri, S.: Argument Reduction Through Factoring. Procs. of the 15th Intl. Conf. on Very Large Databases (1989) 173–182
10. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: Rule ordering in bottom-up fixpoint evaluation of logic programs. Procs of the 16th Intl. Conf. on Very Large Databases (1990) 359–371
11. Wang, G.: Linking CORAL to MySQL & PostgreSQL. Master Thesis. Dept. of Computer Science, Concordia University (2004)
12. Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies. Procs. of ACM SIGMOD (1986) 16–52
13. Bancilhon, F., Ramakrishnan, R.: Performance evaluation of data intensive logic programs. Foundations of Deductive Databases & Logic Programming. J. Minker ed., Morgan Kaufmann (1988) 439–517
14. Ceri, S., Gottlob, G., Tanca, L.: What You Always Wanted to Know About Datalog. IEEE Trans. on Knowledge & Data Eng., Vol. 1, No. 1. (1989) 146–166
15. Giugno, R., Shasha, D.: A Fast & Universal Method for Querying Graphs. Proc. of the Intl. Conf. in Pattern Recognition. (2002) 112–115
16. Cruz, I.F., Leveille, P.S.: Implementation of a Constraint-Based Visualization System. Procs. of IEEE Intl. Symp. on Visual Languages (2000) 13–21
17. Gyssens, M., Paredaens, J., Gutch, D.V.: A graph-oriented object model for database end-user interfaces. Procs. of ACM SIGMOD (1990) 24–33

18. Paredaens, J., Peelman, P., Tanca, L.: G-Log: A Declarative Graphical Query Language. Procs. of 2nd Intl. Conf. on Deductive & Object–oriented Databases (1991) 108–128
19. Poulovassilis, A., Hild, S.G.: Hyperlog: a graph-based system for database browsing, querying & update. Trans. on Knowledge & Data Eng., Vol. 13, No. 2 (2001)
20. Olston, C.: VIQING: Visual Interactive QueryING. Procs. of 4th IEEE Symp. on Visual Languages. (1998) 162–169
21. Erwig, M.: XING: a visual XML query language. Journal of Visual Languages & Computing, Vol 14 (2003) 5–45
22. Ni, W., Ling, T.W.: GLASS: A Graphical Query Language for Semi-Structured Data. Procs. of 8th Intl. Conf. on Database Systems for Advanced Applications (2003) 362–369
23. Vista, D., Wood, P.T.: Efficient Evaluation of Visual Queries Using Deductive Databases. Workshop on Programming with Logic Databases. (1993) 143–161
24. Seshadri S., Naughton, J.F.: On the expected size of recursive Datalog queries. Procs. of ACM Symp. on Principles of Database Systems. (1991) 268–279

Appendix A: Benchmark Evaluation Dimensions and Queries

1. Support of object-orientation
   (a) Method calling
   Q1. Return staff members named Steve Johnson.
   (b) Dynamic binding
   Q2. Return staff members earning more than 2000 per month.
   (c) Complex objects
   Q3. Return tutors living in Glasgow.
   (d) Object identity
   Q4 Return tutors working and studying in the same department.
   (e) Class hierarchy
   Q5 Return all visiting staff in the university.
   Q6 Return all visiting staff members in the university who earn more than 2000 per month.
2. Expressive power
   (a) Multiple generators
   Q7 Return students studying in the same department as Steve Johnson.
   (b) Dependent generators
   Q8 Return courses taken by the students.
   (c) Returning new objects
   Q9 Return students and the courses taken by them.
   (d) Nested queries
   Q10 Return students and the courses taken by them that have more than one credit.
   (e) Quantifiers
   Q11 Return students taking a course given by Steve Johnson.
   Q12 Return students taking only courses given by Steve Johnson.
   (f) Relational completeness
   Q13 Return the names of students.
   Q14 Return all the possible combinations between departments and courses.
   Q15 Return staff members and students in the Computing Science Department.
   Q16 Return areas where students, but no staff live.

(g) Nested relational extension
Q17 Return income tax of staff as 40

(h) Recursion
Q18 Return all direct and indirect prerequisite courses of the "DB4" course.

3. Support of collections
   (a) Collection literals
   Q19 Return students living in the following areas: Hillhead, Kelvinside and Dowanhill.
   (b) Collection equality
   Q20 Return courses with no prerequisite courses.
   (c) Aggregate functions
   Q21 Return courses with less than two assessments.
   (d) Positioning and ordering
   Q22 Return the first and second supervisors of Steve Johnson.
   Q23 Return students having Steve Johnson before Bob Campbell in their supervisor lists.
   (e) Occurrences and counting
   Q24 Return courses with 4 assessments of the same percentage weight.
   Q25 Return the number of assessments worth 25
   (f) Converting collections
   Q26 Return the salary of tutors and keep the possible duplicate values.
   (g) Combining collections
   Q27 Return the students supervised by Steve Johnson.
   (h) Mixing collections
   Q28 Return courses taught by the supervisors of Steve Johnson
4. Usability
   (a) Local definitions
   Q29 Return students whose major departments are in either Hillhead Street or University Avenue.
   (b) Query functions
   Q30 Return students taking some course run by their departments.