Graph data management for molecular and cell biology

B. A. Eckman
P. G. Brown

As high-throughput biology begins to generate large volumes of systems biology data, the need grows for robust, efficient database systems to support investigations of metabolic and signaling pathways, chemical reaction networks, gene regulatory networks, and protein interaction networks. Network data is frequently represented as graphs, and researchers need to navigate, query and manipulate this data in ways that are not well supported by standard relational database management systems (RDBMSs). Current approaches to managing graphs in an RDBMS rely on either external procedural logic to execute the graph algorithms or clumsy and inefficient algorithms implemented in Structured Query Language (SQL). In this paper we describe the Systems Biology Graph Extender, a research prototype that extends the IBM RDBMS—DB2® Universal Database software—with graph objects and operations to support declarative SQL queries over biological networks and other graph structures. Supported operations include neighborhood queries, shortest path queries, spanning trees, graph transposition, and graph matching. In a federated database environment, graph operations may be applied to data stored in any format, whether remote or local, relational or nonrelational. A single federated query may include both graph-based predicates and predicates over related data sources, such as microarray expression levels, clinical prognosis and outcome, or the function of orthologous proteins (i.e., proteins that are evolutionarily related to those in another species) in mouse disease models.

Introduction

Graph structures in systems biology

Graph structures or networks are ubiquitous in post-genomic molecular and cell biology. The quest to identify gene function has increasingly focused on the ways in which genes and gene products interact with other genes, proteins, and small molecules in the cell. The term *systems biology* has many definitions; here is one offered by the Institute for Systems Biology: "Systems biology is the study of an organism, viewed as an *integrated* and *interacting network* of genes, proteins and biochemical reactions which give rise to life" [1]. While other definitions of systems biology might emphasize the interactions of cells within tissues and tissues within organs, and an ecologically minded researcher might focus on predator–prey relationships within an ecosystem, all of them share a common element: *they involve a complex set of relationships among interacting components.* These relationships are normally modeled using graph structures and operations.

Publicly available databases of biomolecular interactions and biological pathways are proliferating; they comprise an increasing volume of both computationally predicted and experimentally determined data. Examples include IntAct [2], DIP [3], BIND** [4], KEGG [5], and the Yeast, Fly, or Worm General Repositories for Interaction Datasets (GRIDs) [6]. Although they are not the focus of this paper, networks of related biological terms, such as the Gene

0018-8646/06/$5.00 © 2006 IBM

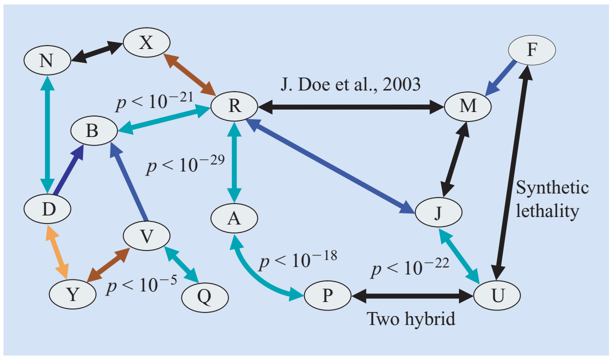


Figure 1

Biological connection graph. Nodes are genes or proteins. Edges represent a heterogeneous set of relationships, e.g., orthology or paralogy (turquoise, weighted with p values), protein–protein interactions (black, labeled with literature references or the experimental method used), domain fusion (orange), and chromosomal proximity (brown).

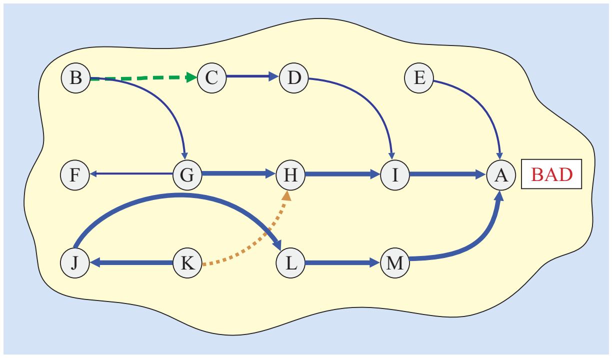


Figure 2

Biochemical pathway that includes a disease-related protein. Nodes represent proteins. Arc colors encode different biochemical interactions between proteins, e.g., *stimulates*, *activates*, *inhibits*. Edge weights, signifying confidence in the interactions, are represented visually by the thickness of the arcs.

Ontology [7], MeSH [8], and UMLS [9], are also readily modeled as graph structures.[1] The following are examples of relationships between biomolecular entities (A and B) that constitute networks:

- A is similar to B [e.g., relationships of orthology (evolutionary relationships across species) or paralogy (evolutionary relationships within species)].
- A interacts with B.
- A regulates the expression of B.
- A inhibits the activity of B.
- A stimulates the activity of B.
- A binds to B [e.g., protein–DNA binding, protein dimers, ligand binding, G-protein (guanine nucleotide-binding protein) coupling].

A fruitful method of representing the totality of information known about the function of a gene or a protein, i.e., its relationships with other biomolecular components, is to construct a biological connection graph (**Figure 1**). Like all graphs, a biological connection graph consists of a set of *nodes* or *vertices*, which represent the interacting components, and *edges*, which represent relationships between the nodes. Edges represented as arrows or arcs may be *colored* or labeled to signify the relationship type (binding, orthology, and so forth). Edges may be *weighted* to signify the likelihood of the relationship (using, for example, the p value, or the probability that a relationship could be due to chance alone), the confidence with which the relationship is posited, or the yield associated with an edge in a biochemical reaction graph (e.g., $A \rightarrow B$ with a 60% yield, $A \rightarrow C$ with a 40% yield). Edges can be *directed* (e.g., A regulates B) or *undirected* (e.g., A and B are orthologs). In this way, a heterogeneous collection of relationship types can be integrated into a single graph in which the large, complex network of connections can be queried, investigated, and reasoned about.

Motivating examples

The following are typical examples of real-world biological questions that may be represented as queries over graph structures:

1. *Finding potentially druggable targets* (i.e., proteins that can bind with high affinity and specificity to small, druglike compounds [10]). To disrupt the activity of protein A (e.g., because it is disease-related but not a member of a druggable protein family), find all proteins B within a path length k upstream of A in a biological pathway graph, considering only interactions of a certain type (color) whose confidence value (weight) exceeds a certain threshold (**Figure 2**).
2. *Graph topology reflecting function*. Interacting proteins tend to form highly connected clusters within interaction networks. Salwinski and Eisenberg [11], citing Ravasz et al. [12], suggest that we can assess the quality of a prospective interaction by examining the length of the shortest path between potential interactors. In signaling networks, *subgraph matching* may be used to find graph motifs that may

[1]Our IBM Research colleagues Peter Schwarz and Julia Rice have used the Systems Biology Graph Extender for just this purpose.

correspond to repeated functional modules, possibly conserved through evolution [13].

3. *Inferring functional relationships*. In predicting interlogs (i.e., evolutionarily conserved protein–protein interactions) [14], protein pairs are more likely to interact if their orthologs interact in one or more other species. More generally, following Marcotte et al. [15] and the Predictome [16] and STRING [17] projects, one may infer functional relationships among proteins (new edges in the connection graph) based on a variety of links between proteins within and across species, e.g., orthology relationships, chromosomal proximity, phylogenetic profiling, and domain fusion (**Figure 3**).
4. *Exploring neighborhoods in function space*. Construct a graph consisting of protein nodes from multiple species, connected by functionally relevant edges of various types (e.g., orthology, paralogy, interaction, domain fusion, and chromosomal proximity). Find all neighbors within k hops of a protein of interest following only interaction edges based on experimental evidence and orthology edges within a threshold evolutionary distance. See the section on simple graph queries below for an example Structured Query Language (SQL) query.

Further examples of biologically relevant graph queries can be found in [18, 19].

This paper describes the Systems Biology Graph Extender (SBGE), a research prototype that extends the IBM relational database management system (RDBMS)—IBM DB2* Universal Database—with graph objects and operations to support queries over biological networks, connection graphs, and other graph structures. The SBGE allows bioinformaticians (researchers who specialize in bioinformatics, the application of computer science and information technology to biological investigations) and developers to exploit the power of graph algorithms to answer important questions in biological research. The paper is organized as follows. To make our discussion accessible to both computer scientists and biologists, in the next section we give a brief overview of SQL, the relational model, and graph theory. We then describe the SBGE data model and query language extensions in detail, followed by examples of ways in which the SBGE can be used to answer real-world biological questions. In the last two sections, we briefly discuss related work and directions for further research and development and then highlight the key features of the SBGE and present our conclusions.

Background

In this section we provide a brief overview of the relational data model and the capabilities of SQL for those who may not be familiar with it. We present graph theory more formally and point out how it pertains to mainstream data processing.

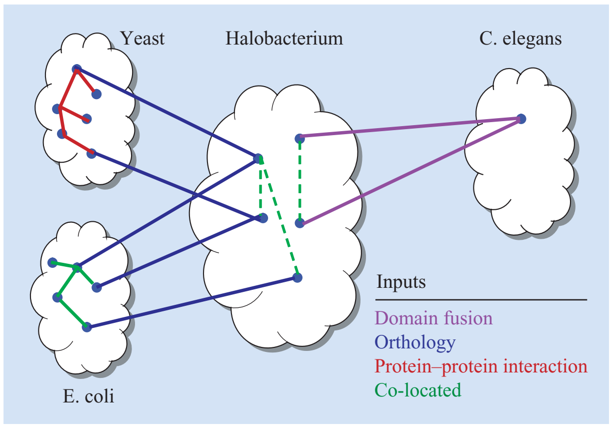


Figure 3

Schematic method of illustrating the inferred functional relationships in the Halobacterium proteome based on a variety of inter- and intra-proteome relationships. The output of graph analysis consists of the predicted green-dashed relationships.

SQL and the relational model

In an RDBMS, data is viewed as tables composed of rows and columns, as in a spreadsheet. The data is queried using SQL, a *declarative query language*. A declarative language allows users to describe the results they wish to retrieve rather than the steps used to retrieve those results; users specify what they want, not how they want to get it. SQL allows a variety of powerful set-based operations, including filtering out extraneous data from a dataset, performing `UNION`s and `INTERSECTION`s of sets of data, and `JOIN`ing sets of data on a common field. For more on RDBMSs and their advantages for managing biological data, see [20].

Although it has many strengths, SQL does not support such arbitrarily complex operations on data as regular expression pattern matching on strings or BLAST [21] alignments on protein sequences. To build in more complex operations, DB2 and most commercial RDBMSs expand SQL with special-purpose functions called *user-defined functions* (UDFs). UDFs can be used anywhere in an SQL query that an expression of their return type can be used. For example, a function returning a table can be used in the SQL `FROM` clause, and a function returning a scalar can be used in the `SELECT` and `WHERE` clauses.

The SBGE uses functions to extend DB2 with graph operations. We can easily represent graphs in relational tables as sets of nodes and sets of edges, but we cannot easily express all relevant operations—e.g., path of length *k*, neighborhood queries, and graph matching—on graphs in SQL. So why use SQL at all? Because the mature technology in a commercial RDBMS allows us to efficiently define and retrieve subgraphs based on node and edge annotations, labels, and weights. RDBMSs scale well for large graphs because they can perform searches without requiring the entire graph database to be loaded into main memory. In sum, our approach is to manage graph data using SQL and to manage graph operations using functional extensions to SQL.

From the computer science point of view, biological graph-structured data is interesting because it is the following:

- *Large and getting larger*. Leser [18] reports that the STRING database of predicted protein links has more than 200,000 associations. The Ingenuity** Pathways Knowledge Base [22] contains millions of pathway interactions extracted from the scientific literature. Any single user query will deal with a relatively small subset of the entire graph.
- *Shared* among a community of users, each of whom cares about potentially overlapping subsets of the large, integrated network of known and inferred connections among genes and proteins. For example, one group of scientists in a large organization might be interested in the human apoptosis pathway, and another group might be focused on the network of evolutionary relationships among apoptosis-related genes in a variety of eukaryotic species.
- *Subject to constant change* and revision. For example, functional links among human proteins inferred from yeast interactions must be updated as false positives, and negatives are eliminated from the yeast datasets. In many graph applications, edge properties (likelihoods and measures of confidence) are central to the problem definition; while the graph structure may be fixed, its edge properties, and therefore the optimal answers to graph questions, will vary.
- A *valuable asset* to the organization that paid for generating or discovering it. Since data is arguably the most important commodity in science, managing it is worth doing right. RDBMSs were developed to support business data management information systems and therefore place a premium on quality-of-service properties such as data integrity (consistency with certain rules), availability (efficient concurrent access for many users), recoverability (robustness to system and software failure), and security.

Requirements such as scalability, multiuser access control, and robustness make the management of graph data a classic database management system (DBMS) problem. From the biology point of view, extending an RDBMS with graph operations means that all of the industrial-strength features that make RDBMSs the technology of choice for data management are automatically available for graph data management. These features include query optimization, efficient management of a hierarchy of storage devices via caching and buffering to enhance performance, referential integrity, recoverability, scalability, transaction management to permit concurrent updates, and a declarative query language [23].

Overview of graph theory

In this section we briefly outline the graph data model employed by the SBGE. Graph theory is a very large, verdant field of study, and we have deliberately chosen to start with the basics. In formal terms, a Graph **G** is a triple $\langle \boldsymbol{N}, \boldsymbol{E}, \boldsymbol{W} \rangle$, where $\boldsymbol{N} = \{\boldsymbol{n}_i\}$ is a set of *nodes*, $\boldsymbol{E} \subseteq \boldsymbol{N} \bowtie \boldsymbol{N}$ is a set of *edges*, each edge consisting of a pair of nodes $\langle \boldsymbol{n}_i, \boldsymbol{n}_j \rangle$, and $\boldsymbol{W}$ is a mapping $\boldsymbol{E} \mapsto \boldsymbol{R}^n$, corresponding to the *payload* or set of properties of each edge. The payload is defined as a vector of real numbers because in practice, edge properties can be quite complex.

A graph whose edges have direction—i.e., are oriented from a source node to a destination node—is called a *directed graph*, or *digraph*. A graph or digraph that has multiple edges among nodes is a *multigraph* or *multidigraph*, respectively. An edge that connects a node to itself is called a *loop*. The SBGE currently supports graphs and digraphs with loops, though it could, in principle, support multigraphs as well. For simplicity, we use the word *graph* throughout this paper when referring to both graphs and digraphs.

Figure 4(a) shows a simple graph consisting of eight nodes and 11 edges. Each node and each edge is identified by a non-zero integer value. Each edge label is of the form *i:p*, where *i* is the edge identifier and *p* is its payload, in this example a single real number value. (In our implementation, SBGE edge payloads actually support a vector of seven 64-bit double-precision values for weights and seven 8-bit characters for labels or colors.)

Systems Biology Graph Extender (SBGE)

In this section we introduce the data model and the query language extensions that make up the SBGE.

Graph management in SQL

Representing graphs in an RDBMS is straightforward. A simple table holding one edge per row is all that is required. A more elaborate schema might normalize information relating to the nodes or the edge payloads into separate tables related by foreign keys to the edge table. **Figure 4(b)** presents the SQL Data Definition

Language (DDL) needed to define such a table, and **Figure 4(c)** shows how the table would look if it were populated with data from the graph in Figure 4(a). This kind of graph representation is usually referred to as a *node association list*. (For simplicity, the payload is represented here as a scalar, while in the general case it is a vector.)

Data model

The SBGE data model has three primary features. First, it defines a graph as a first-class SQL *data type* (i.e., a data type that can be manipulated in the same ways as other types in the SQL language). This permits developers to work with graph instances using SQL in the same way that they work with textual strings or numbers. For example, a graph type can be used in the definition of an SQL table, functions can accept or return instances of the graph type, and developers can manipulate instances of the graph type in SQL queries using methods over the type. In formal terms, we have introduced the notion of graphs (and digraphs) as a domain within the framework of the relational model. By encapsulating graphs and graph operations within a data type and a set of associated operators, we are able to employ algorithms to compute graph operations that are not available to a query executor limited to the set of relational operators.

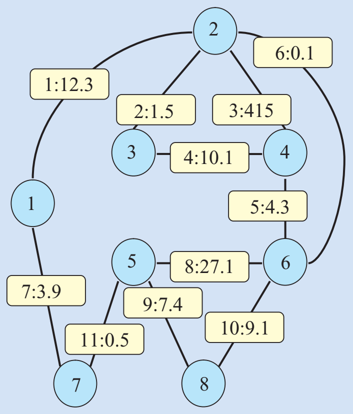
Second, the SBGE defines an aggregate operator that turns a list of rows (that is, the result of an SQL `SELECT` query) into graph instances. In formal terms, we have introduced a new relational operator, which takes as input a relation with a certain set of attributes (an edge list) and returns another relation consisting of a single attribute of the graph domain.

Third, our extensions define a new class of relational operators, which takes a single graph instance and decomposes it back into simpler data structures, including edge lists, as in Figure 4(c).

Thus we extend the relational algebra by implementing functions that are closed over relational tables and graphs encapsulated as user-defined types. Developers can start with graph data stored in an SQL table—or computed using an SQL `SELECT` query—convert it to an instance of a graph object, perform some operation on the graph object, and then store the result back in the database as either a set of graph instances or an edge list.

Implementation

In this section we illustrate, in a slightly simplified syntax, the SQL extensions making up the SBGE. For simplicity, we omit the DDL statements involved in adding these features to the RDBMS. Instead, we focus on how application developers would use the extensions and, in particular, how they might use the SBGE for reasoning about the Gene Ontology database [7]. More specifically,



(a)

```
CREATE TABLE Edge_List (
  ID       INTEGER  NOT NULL  PRIMARY KEY,
  Node_A   INTEGER  NOT NULL,
  Node_B   INTEGER  NOT NULL,
  Payload  FLOAT    NOT NULL
);
```

(b)

ID	*Node_A*	*Node_B*	*Payload*
1	1	2	12.3
2	2	3	1.5
3	2	4	415.7
4	3	4	10.1
5	4	6	4.3
6	2	6	0.1
7	1	7	3.9
8	5	6	27.1
9	5	8	7.4
10	6	8	9.1
11	5	7	0.5

(c)

Figure 4

(a) Simple graph; (b) DDL for simple edge table; (c) edge-list representation of simple graph (a).

we focus on two tables within that schema: `TERMS` and `TERM2TERM`. A subset of the Gene Ontology Schema is shown in **Figure 5**.

The purpose of Gene Ontology is to provide for consistent use of terminology in life sciences and systems biology research. These tables hold information about the terms in the ontology (words and phrases employed by scientists studying gene function) and relationships among these terms, for example, when a term such as *apoptosis* is a specialized form of another term, *cell death*.

```
CREATE TABLE TERMS (
   ID          INTEGER      NOT NULL PRIMARY KEY,
   NAME        VARCHAR(255) NOT NULL WITH DEFAULT '',
   TERM_TYPE   VARCHAR(55)  NOT NULL WITH DEFAULT '',
   ACC         VARCHAR(255) NOT NULL WITH DEFAULT '',
   IS_OBSOLETE INTEGER      NOT NULL WITH DEFAULT 0,
   IS_ROOT     INTEGER      NOT NULL WITH DEFAULT 0
);
CREATE TABLE TERM2TERM (
   ID          INTEGER     NOT NULL PRIMARY KEY,
   REL_TYPE    INTEGER     NOT NULL WITH DEFAULT 0,
   TERM1       INTEGER     NOT NULL
                    REFERENCES TERMS (ID),
   TERM2       INTEGER     NOT NULL
                    REFERENCES TERMS (ID)
);
```

Figure 5

Subset of Gene Ontology schema.

Graph data type
The SBGE adds graphs and digraphs as new domains or data types to SQL. The new types are primarily used in table definitions. They specify the kind of data a table column is to contain. The SQL statement

```
CREATE TABLE Graph_Data_Table (
     Id      INTEGER       PRIMARY KEY,
     Comment VARCHAR (255) NOT NULL,
     Data    Graph         NOT NULL
);
```

illustrates how a developer may create a table to store graph data objects alongside more conventional SQL data types.

Simple graph construction
The simplest operation in the SBGE is a function that takes as input the minimum amount of information needed to construct a graph that consists of a single edge. The following illustrates a query that constructs one simple graph for each row in the `TERM2TERM` table:

```
WITH Small_Graphs (Graph_Data) AS (
    SELECT Graph (   TT.Id,
                     TT.Term1,
                     TT.Term2)
       FROM Term2Term TT
     )
SELECT Status ( G. Graph_Data)
  FROM Small_Graphs G;
```

It is in the following form: For each row in `TT`, construct a simple one-edge graph and add it to the table variable `Small_Graphs`. Then return a status for each row (each graph) in the table variable. Note that although edges in Gene Ontology are directed—term A is a specialization of term B, but not vice versa—for simplicity, we continue focusing on the semantics of graphs, not digraphs.

The query above introduces a necessary complication. The SBGE manages graph data in a binary format, making it necessary to apply some operation over each graph data object to make it readable. In this case, the function `Status( )` reports some useful details about the graph to which it is applied: node count, edge count, and as many of the graph properties as can be efficiently computed, such as whether or not the graph is a tree.

Compiling graphs
A more complex extension involves combining multiple graphs into a single large graph. We chose the term *compilation* because the process calls for something more than just creating a union of all of the edges in the input graphs. As part of the compilation process, it is possible to infer automatically certain properties about the graph that will result. For example, a graph is a tree if a) the number of nodes exceeds the number of edges by 1, and b) the graph consists of a single component.

The compilation process requires the SBGE to deal gracefully with constraints such as the following:

- When two input graphs possess identical edges in terms of having the same source and destination nodes (with the same orientation, if relevant), what is to be done with the payloads? The graph compiler may choose a) the minimum or b) the maximum payload from among those on offer, or else c) combine the payloads.
- The graph must contain no ambiguity with respect to edge identities. If the same edge identification is used erroneously for two different source/destination combinations, the SBGE will produce an error.

The SBGE compiles graphs with a User-Defined Aggregate (UDA) operation [24, 25]—an extensibility mechanism provided by most modern RDBMS products. In SQL, aggregate functions take a list of data values and compute some result over the input list, typically a statistic such as a sum or an arithmetic mean. UDAs generalize this concept. A UDA takes as input a list of data values of some type (often user-defined) and produces as output a result reflecting some operation over the entire set.

The graph compilation query below is in the following form: For each row in `TT`, construct a simple one-edge graph and add it to the table variable `Small_Graphs`;

compile all of the rows (graphs) in the table variable into a single large graph and report its status:

```
WITH Small_Graphs (Graph_Data) AS (
  SELECT Graph (  TT.Id,
                  TT.Term1,
                  TT.TERM2)
  FROM Term2Term  TT
)
SELECT Status (
         GraphCompile (G. Graph_Data))
  FROM Small_Graphs G;
```

In contrast to the previous query, which produces as many small graphs as there are rows in the `TERM2TERM` table, this query produces exactly one graph. All of the logical edges in the `TERM2TERM` table are included in the query result graph. Duplicate edge rows in the `TERM2TERM` table are found in the resulting graph only once.

Having constructed a graph, we can save to a table. This is achieved by using the query below in the following form: Insert a row containing the single large graph, an `Id`, and a comment into a relational table; compile all of the simple graphs into a single large graph; for each row in `TT`, construct a simple one-edge graph:

```
INSERT INTO Graph_Data_Table
( Id, Comment, Data)
SELECT 101,
       'Simple Example Graph',
       Graph_Compile (
                 Graph ( TT.Id,
                         TT.Term1,
                         TT.TERM2))
  FROM Term2Term TT;
```

Implementing graph compilation as an aggregate provides a number of advantages over implementing the same operation as an SQL stored procedure. The most important of these is flexibility. SBGE UDFs can be combined in the style of a functional programming language with the results of each function being passed as an argument to another. A single query can compile a number of graphs, compare them or filter them according to some criteria, and then store the results in a table. Another advantage of implementing graph compilation as a UDA lies in the way that the SBGE can exploit the RDBMS intraquery parallelism: Stored procedures and external code cannot be parallelized so easily. One of our plans for future work is to exploit this innate RDBMS parallelization (see the section on further research and development below).

One way to characterize our work is to point out that our emphasis is on supporting operations that involve a very large number of small to medium-sized graphs rather than complex operations on a single very large graph. Of course, by embedding our SBGE within the RDBMS, we can store and reason about graphs that have as many edges as one may have rows in an SQL table. In practice, any one operation over an extremely large graph tends to filter out uninteresting edges and nodes, a function that can be efficiently performed by the RDBMS engine.

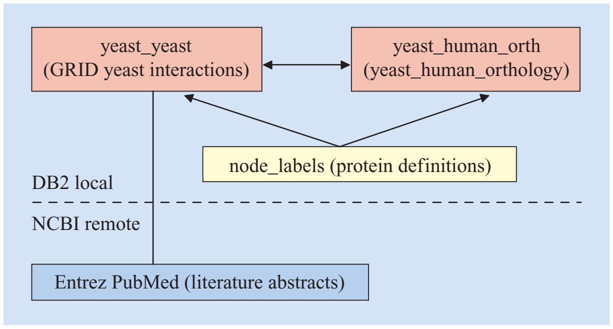

yeast_yeast
(GRID yeast interactions)
yeast_human_orth
(yeast_human_orthology)
node_labels (protein definitions)
DB2 local
NCBI remote
Entrez PubMed (literature abstracts)

Figure 6

Database schema for example queries.

Graph operations

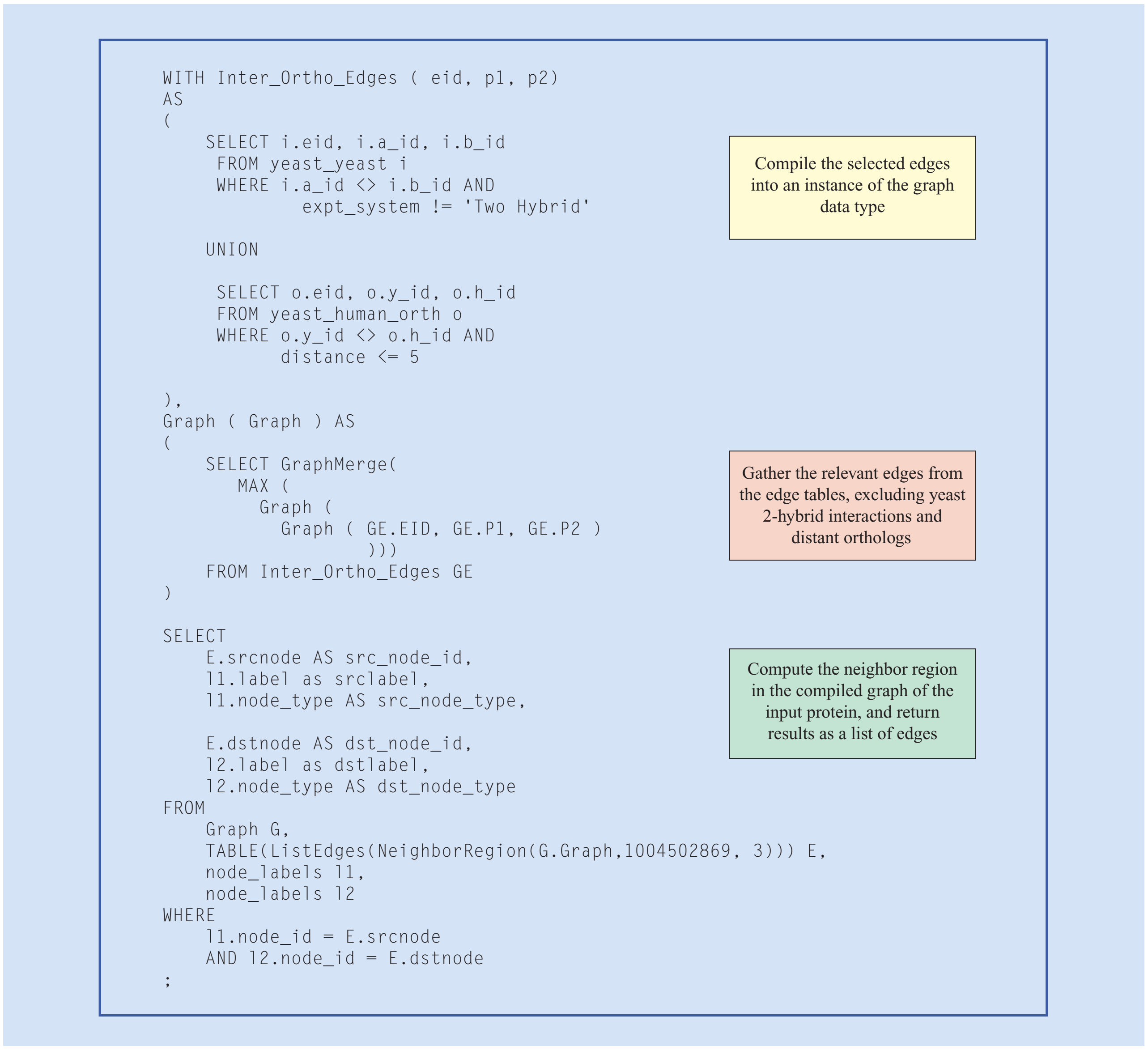
The functions that comprise the SBGE are summarized in **Table 1**, along with descriptions of what they do. These functions represent four major categories: extracting graph properties, comparing graphs, computing graphs from graphs, and decomposing a graph into its component parts.

Examples of biologically relevant graph queries

To illustrate the ease and elegance of performing graph operations on biological data using the SBGE and to demonstrate its effectiveness in answering real-world biology questions, we have assembled a simple biological connection graph consisting of three local database tables (shown in the DB2 section of **Figure 6**) and one remote annotation data source at the National Center for Biotechnology Information (NCBI). The `yeast_yeast` table, downloaded from the Yeast General Repository for Interaction Datasets (GRID) [6], contains 19,789 interactions (edges) among 4,917 yeast proteins (nodes). The interactions are labeled with the experimental technique used to detect the interaction and a PubMed identifier linking the interaction with the relevant reference in the PubMed [8] database of annotated abstracts from the biomedical literature. The `yeast_human_orth` table contains orthology relationships among 1,781 human proteins and 1,683 yeast proteins (1,781 edges and 3,464 nodes) contributed by our collaborator, Dr. Dennis Wall of the Harvard Medical School. Orthology edges are given weights

Table 1 Graph operations supported by the SBGE.

Operation	*Description*
	Extract graph properties
Extract graph properties	Given a graph, return such properties as number of nodes, number of edges, whether or not a directed graph has cycles, number of components contained in the graph, sum of graph edge payloads, etc.
	Compare graphs
Test subgraph inclusion	Compare two graphs A and B to see whether graph A is a subgraph of graph B. The function returns `TRUE` `iff` (if and only if) for every edge in A there exists an equivalent edge in B (i.e., an edge with the same source and destination node IDs). Note that subgraph and equality comparisons implemented in the SBGE are computationally simpler than general isomorphism, and they can be accomplished in time $O(\|E\|)$.
Test graph equality	Compare two graphs A and B for equality. The function returns `TRUE` `iff` for every edge in A there exists an equivalent edge in B (that is, an edge with the same source and destination node IDs), and for every edge in B, there exists an equivalent edge in A. Graph A equals B `iff` graph A is a subgraph of B and B is a subgraph of A.
Compute shared nodes and edges	Compare two graphs to see whether they intersect. There are two types of intersection: *node intersection* (where graphs A and B share a node with the same ID) and *edge intersection* (where graphs A and B share an edge with the same source and destination ID and, optionally, payload). These functions are computed in $O(\|N\|)$ and $O(\|E\|)$ respectively.
	Compute graphs from graphs
Perform elementary transformations	Transpose a directed input graph (create a new graph with all of the edge orientations reversed), or compute the transitive closure of an input graph (a new graph with an edge between every pair of nodes that are connected by some path in the input graph).
Compute intersection, union, or disjunction	Compute the intersection, union, or disjunction of two graphs.
Compute connected components	A single graph G may contain several *connected component* subgraphs; i.e., each of the nodes in subgraph A is connected via some path to all of the other nodes in A, and the same for subgraph B, but the nodes in A are not connected to the nodes in B, and vice versa. In the case of digraphs, the components in which each of the nodes can be reached from all of the other nodes are called *strongly connected components*.
Compute shortest path and shortest path tree	Given a graph G, starting at some node n, compute a graph (path) that is the sequence of edges from n to a destination node s for which the sum of the edge weights is minimal over the set of all such paths. Similarly, compute the shortest path tree, i.e., the tree rooted at n that consists of the set of shortest paths from n to each s in G.
Compute distance between nodes	Compute the distance between two nodes as a sum of the weights along the shortest path edges.
Compute minimum spanning tree	Given a graph G, compute a new graph consisting of the set of edges that constitute the minimum spanning tree for G, i.e., an acyclic subgraph of G that connects all of the nodes and whose total weight is minimized.
Compute neighbor regions	Given a graph G and a start node n, return the region of G that can be reached from n by a path of some maximum number k of edges. We speak of the k-in or k-out neighbor region of a graph (digraph).
	Decompose a graph into its component parts
Shred graph into list of its edges	Given an input graph, output a list of its edges. This list constitutes the same type of relational table from which G was originally constructed. Intuitively, this is the inverse of graph compilation.
Shred graph into list of nodes	Given an input graph, return a list of all of its nodes together with details about the number of incident edges each node has. One might use this function to find periphery nodes in a graph, i.e., nodes with exactly one incident edge.
Perform topological sort	Output the nodes in a sorted order based on their dependencies such that if the graph contains an edge (u, v), then u appears before v in the ordering.



```
WITH Inter_Ortho_Edges ( eid, p1, p2)
AS
(
    SELECT i.eid, i.a_id, i.b_id
     FROM yeast_yeast i
     WHERE i.a_id <> i.b_id AND
             expt_system != 'Two Hybrid'

    UNION

     SELECT o.eid, o.y_id, o.h_id
     FROM yeast_human_orth o
     WHERE o.y_id <> o.h_id AND
           distance <= 5

),
Graph ( Graph ) AS
(
    SELECT GraphMerge(
       MAX (
         Graph (
           Graph ( GE.EID, GE.P1, GE.P2 )
                   )))
    FROM Inter_Ortho_Edges GE
)

SELECT
    E.srcnode AS src_node_id,
    l1.label as srclabel,
    l1.node_type AS src_node_type,

    E.dstnode AS dst_node_id,
    l2.label as dstlabel,
    l2.node_type AS dst_node_type
FROM
    Graph G,
    TABLE(ListEdges(NeighborRegion(G.Graph,1004502869, 3))) E,
    node_labels l1,
    node_labels l2
WHERE
    l1.node_id = E.srcnode
    AND l2.node_id = E.dstnode
;
```

Compile the selected edges into an instance of the graph data type

Gather the relevant edges from the edge tables, excluding yeast 2-hybrid interactions and distant orthologs

Compute the neighbor region in the compiled graph of the input protein, and return results as a list of edges

Figure 7

Exploring neighborhoods in function space.

between 0 and 10 that represent an estimate of their evolutionary distance. The `node_labels` table contains human-readable protein definitions downloaded from GenBank [26] and the Saccharomyces Genome Database (SGD) [27].

Simple graph queries

We now give a simple graph query of the type described in motivating Example 4 in the Introduction. Using the yeast–yeast interaction data and the yeast–human orthology data noted in Figure 6 and described above, **Figure 7** shows the SQL query to find all neighbors within three hops of the human chloride channel 3 protein (ID #1004502869) following all interaction edges except those found by the yeast two hybrid method and following only fairly close orthology edges (distance ≤ 5).

The query is composed of three sections. In the first section (in yellow), the relevant edges are gathered from the yeast interaction graph and the yeast–human orthology graph into a table variable called `Inter_Ortho_Edges`, enforcing the constraints on edge type and weight. In the second section (peach), these

Table 2 Partial results of Figure 7 query.

src_label	src_node_type	dst_label	dst_node_type
YPT6	yeast	RAB6B	human
GEF1	yeast	chloride channel 3	human
GEF1	yeast	RIC1	yeast
YPT6	yeast	YPR197C	yeast
RIC1	yeast	YPR197C	yeast
YPT6	yeast	YPR084W	yeast
RIC1	yeast	YPR084W	yeast
YPT6	yeast	MRL1	yeast
RIC1	yeast	MRL1	yeast
YPT6	yeast	MAK3	yeast
RIC1	yeast	MAK3	yeast
YPT6	yeast	YPR050C	yeast
RIC1	yeast	YPR050C	yeast
YPT6	yeast	SRO7	yeast
RIC1	yeast	SRO7	yeast
YPT6	yeast	DSS4	yeast
RIC1	yeast	DSS4	yeast
YPT6	yeast	TFP3	yeast
RIC1	yeast	TFP3	yeast
YPT6	yeast	OXR1	yeast
RIC1	yeast	OXR1	yeast
YPT6	yeast	APL5	yeast
RIC1	yeast	APL5	yeast
YPT6	yeast	BEM4	yeast
RIC1	yeast	BEM4	yeast
YPT6	yeast	VPS30	yeast
RIC1	yeast	VPS30	yeast
RIC1	yeast	YPL105C	yeast
YPT6	yeast	LGE1	yeast
RIC1	yeast	LGE1	yeast

edges are compiled into an instance called `Graph` of the graph data type. In the third section (green), the `NeighborRegion( )` function returns a graph consisting of everything within three hops of the input protein, and the `ListEdges( )` function decomposes this temporary graph into a relational table of edges. Finally, this temporary edge table is joined with the human-readable definitions in the `node_labels` table to produce the tabular output of the SQL query (**Table 2**), which can then be simply listed for the user or formatted for graph display applications.

Most queries involving graph operations follow the same pattern as our example, but simply substitute different edge tables (yellow section) and different graph functions (green section). For example, the following functions would be used to implement the following motivating examples:

1. Finding potentially druggable targets: `Upstream_Neighbors( )`.
2. Graph topology reflecting function: `Shortest_Path( )` and `Subgraph( )`.
3. Inferring relationships: `Shortest_Path( )`.

Concerning Example 2, some graph-matching problems in systems biology require full subgraph isomorphism, the exact solution of which is known to be NP-complete (i.e., its algorithmic complexity precludes exhaustive computation). Computational biologists have developed a variety of heuristic methods to approach this problem. They are typically implemented in the Java** programming language. Rather than choose to support one of these methods over the others, we provide Java code that converts from the C internal graph representation used in the SBGE to a Java graph class representation. This allows SBGE functions to be seamlessly composed in a single SQL query with UDFs written in Java. We have tested this approach with colleagues from the Computational Biology Center at the IBM Thomas J. Watson Research Center and found that calling their Java functions within an SQL query and composing them with SBGE functions greatly enhanced the system maintainability and ease of use without significantly affecting performance.

Federated queries
As high-throughput biology begins to generate large volumes of systems biology data, there is a growing need for robust, efficient systems to support investigations of metabolic and signaling pathways, chemical reaction networks, gene regulatory networks, and protein interaction networks. In systems biology research in the post-genomic era, the variety of data sources and number of techniques available to discover, represent, and predict functional relationships among biomolecular entities is large and increasing. Investigators must deal not only with a variety of network databases (e.g., KEGG [5], EcoCyc [28], and the Molecule Pages of the Alliance for Cellular Signaling [29]) and protein interaction databases (e.g., GRID, DIP, BIND, and IntAct), but also with related data such as nucleotide and protein sequences and annotations (GenBank [30], UniProt [31], Entrez Gene [8]), orthologous clusters (COGs [8]), protein structure (PDB [32]), chemical compound structures and properties (e.g., Daylight** Toolkit [33] and MDL** ISIS [34]),

public and private repositories of microarray expression data (GEO [8]), and special-purpose databases (e.g., GPCRDB [35], ENZYME [36], and TRANSFAC** [37]). Biological connection graphs comprising multiple organisms require integration with a variety of model organism databases (e.g., MGD [38], Flybase [39], and SGD** [40]). The results of analytic applications such as BLAST, Daylight, and MDL, and a variety of network modeling and simulation tools must also form part of an integrated system that supports systems biology research.

Scientists need to be able to answer questions that integrate all relevant data, whether it comes from an RDBMS, flat files, Extensible Markup Language (XML), Web sites, document management systems, applications, or special-purpose systems. They need to search through large volumes of data and correlate information in complex ways. To derive the greatest advantage from this data requires full query-based access to all of the most up-to-date information available, irrespective of where it is stored or its format, with the flexibility to customize queries easily to meet the needs of a variety of individual investigators. A key element in the IBM response to the challenge of heterogeneous database integration is federated database technology [41].

IBM federated technology

IBM WebSphere* Information Integrator software builds on an earlier system, IBM DiscoveryLink* software [42, 43], by using federated database technology to provide integrated access to life sciences data sources. Rather than simply duplicating and loading all data sources of interest into a common local RDBMS, the federated middleware "wraps" the actual data sources in place, providing an extensible framework and encapsulating the details of the sources and how they are accessed. In this way, the WebSphere Information Integrator middleware provides users with a virtual database to which they can pose arbitrarily complex queries in the high-level, non-procedural query language SQL. The WebSphere Information Integrator middleware efficiently answers these queries, even though the necessary data may be scattered across several different sources and those sources may not themselves possess all of the functionality needed to answer such a query. In other words, its query engine can optimize queries and compensate for SQL function that may be lacking in a data source. Additionally, queries can exploit the specialized functions of a data source so that no functionality is lost in accessing the source through the federated middleware.

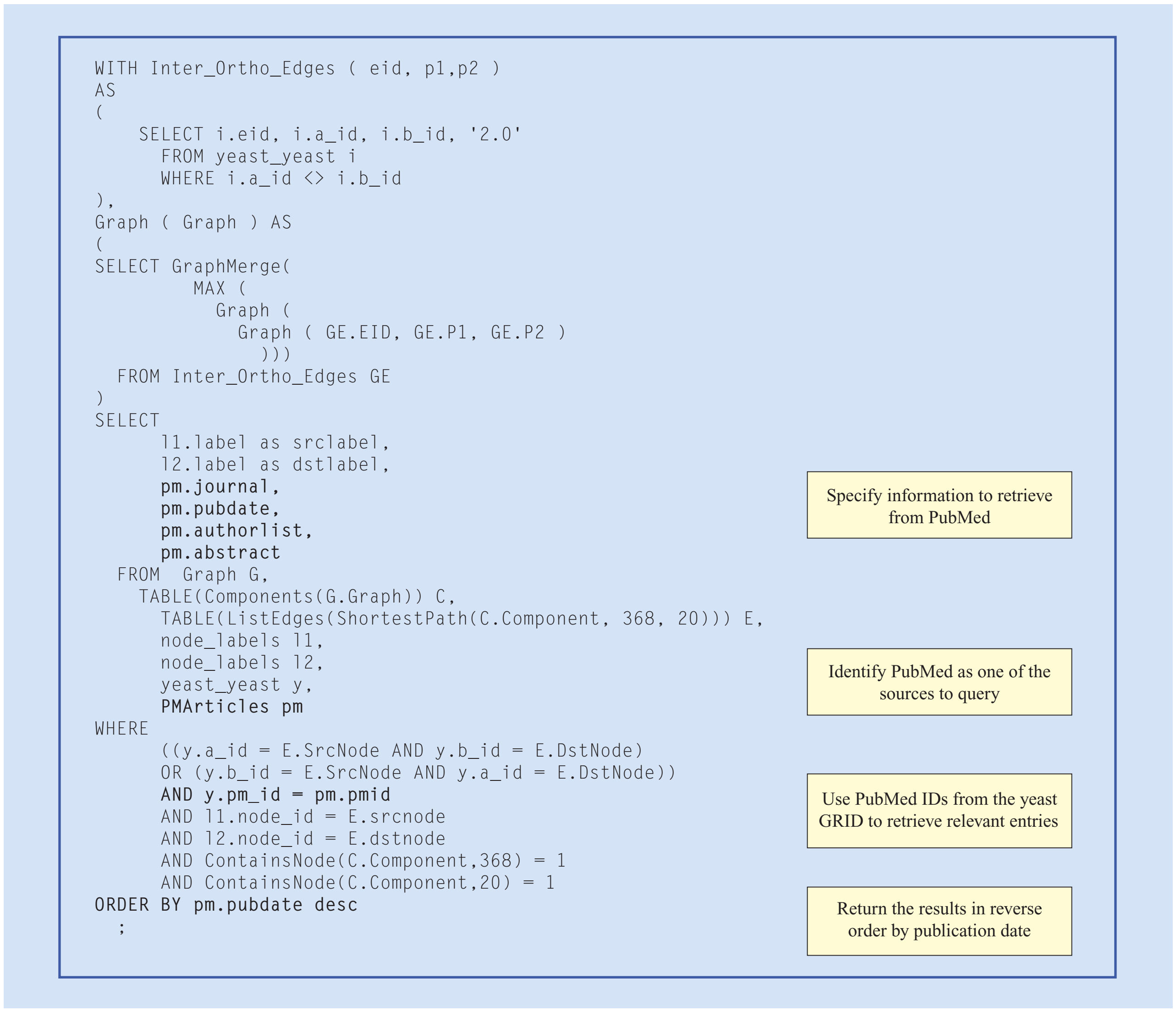
Examples of federated queries

The real power of SBGE is revealed in the context of a data federation. With the WebSphere Information Integrator middleware, users can write single declarative SQL queries that span multiple data sources. They can be written in a variety of formats and distributed over the Internet and within an organization as if they were components of a single relational database. The SBGE graph functions can operate on edge data taken from a DB2 or Oracle** database, XML files, Web sites, or a variety of other sources. Annotations on nodes (e.g., proteins and genes) can be drawn from a wide variety of sources, both publicly available and local to an organization. These annotations can be retrieved simply as part of the query result, or they can be used in specifying constraints to limit the nodes and edges that are compiled into an instance of the graph data type.

The following example uses the database schema given in Figure 6. We can find the shortest path in the yeast interaction graph between yeast proteins ARL1, a soluble GTPase (ID #368), and FUN26, a nucleoside transporter (ID #20). In addition to the interaction edges that lie on the shortest path and the names of the interacting proteins, we can return the PubMed authors, journal, title, and abstract where the interaction is reported. The single SQL query to perform this work is shown in **Figure 8**.

Additional examples of biological questions spanning multiple heterogeneous distributed data sources that may be answered using single SQL queries that include SBGE graph functions are as follows:

- To predict candidate genes in juvenile diabetes: Find all genes that 1) are located in chromosomal regions identified through association studies (input); 2) are expressed specifically in the pancreas; 3) are known to contain single nucleotide polymorphisms (SNPs); and 4) lie within *k* hops in a biological interaction graph from 5) a gene known to be involved in metabolism or immune function. This query integrates the expression data in UniGene [44] or a local enterprise microarray database (part 2) with the polymorphism data in dbSNP [45] (part 3), the SBGE `Neighbor_Region( )` function (part 4), and the protein function classifications in the Gene Ontology (part 5).
- Find a subgraph of a large reaction pathway graph that 1) has the same structure and 2) involves an enzyme from the same family as the enzyme in the input subgraph, and retrieve annotations on the 3) proteins and 4) compounds in the subgraph. This query might integrate the SBGE `subgraph( )` function with ENZYME, UniProt, and a database of compounds in an Oracle database.
- Return genes and their neighbors in known pathways that are at least twofold up- or down-regulated after lung transplant rejection compared to the immediate



```
WITH Inter_Ortho_Edges ( eid, p1,p2 )
AS
(
    SELECT i.eid, i.a_id, i.b_id, '2.0'
      FROM yeast_yeast i
      WHERE i.a_id <> i.b_id
),
Graph ( Graph ) AS
(
SELECT GraphMerge(
         MAX (
           Graph (
             Graph ( GE.EID, GE.P1, GE.P2 )
               )))
  FROM Inter_Ortho_Edges GE
)
SELECT
      l1.label as srclabel,
      l2.label as dstlabel,
      pm.journal,
      pm.pubdate,
      pm.authorlist,
      pm.abstract
  FROM  Graph G,
    TABLE(Components(G.Graph)) C,
      TABLE(ListEdges(ShortestPath(C.Component, 368, 20))) E,
      node_labels l1,
      node_labels l2,
      yeast_yeast y,
      PMArticles pm
WHERE
      ((y.a_id = E.SrcNode AND y.b_id = E.DstNode)
      OR (y.b_id = E.SrcNode AND y.a_id = E.DstNode))
      AND y.pm_id = pm.pmid
      AND l1.node_id = E.srcnode
      AND l2.node_id = E.dstnode
      AND ContainsNode(C.Component,368) = 1
      AND ContainsNode(C.Component,20) = 1
ORDER BY pm.pubdate desc
  ;
```

Specify information to retrieve from PubMed
Identify PubMed as one of the sources to query
Use PubMed IDs from the yeast GRID to retrieve relevant entries
Return the results in reverse order by publication date

Figure 8

Federated query with graph operations.

post-transplant state in the same patient. This query uses the SBGE `Neighbor_Region( )` function.

- To investigate disease processes, find all pathways where a compound of known efficacy inhibits or slows a reaction and retrieve Gene Ontology classifications and UniProt functions for all genes and proteins in the pathway. This query involves filtering on edge labels ("inhibits" or "slows").

Discussion

Directions for further research and development

While the SBGE has been shown to be useful for systems biology research, there are many areas in which further research is needed. To aid in compiling very large graphs, we are working on parallelizing graph compilation by taking advantage of innate RDBMS parallelized query processing for aggregate functions. Our initial implementation of the SBGE requires that each graph fit into main memory. To our knowledge, most public domain graph libraries (e.g., the Boost C++ library and the JDigraph Java library) make this same assumption. For many applications, this is not a problem: An SBGE graph of one million nodes and five million edges occupies a little less than one gigabyte of memory.[2]

[2]Each graph instance can contain up to two gigabytes of data. The relationship between memory size and the number of nodes and edges is complex: $mem = graph\ header$ (~1 K) + 48 bytes · $|N|$ · 1.2 + 96 bytes · $|E|$.

An RDBMS-based approach, however, should scale gracefully and cope with increasing data volumes. To handle graphs larger than main memory, we plan to take advantage of RDBMS features and tricks, such as temporary tables, to hold intermediate results.

The current implementation of the SBGE is written in C/C++, a language that gives us optimal performance but one in which code quality requires a degree of software engineering experience and time investment not available to many systems biologists. Other languages (notably Java and C#) represent more productive development environments, though at some cost in performance. When developing speculative algorithms, particularly algorithms to approximate results whose complexity precludes exhaustive computation, it makes sense to write UDFs in these languages. In the simple graph queries section above, we described such a scenario and our C-to-Java format converter for serialized graphs. To further support Java developers, we anticipate releasing a set of Java classes that directly access the graph structure.

Enhancements are also needed from the point of view of representing biology. Currently the SBGE does not provide any special help with data provenance issues [23], but we realize that it is important to be able to identify and excise pathway edge predictions based on data that has since been superseded by more advanced experimental techniques. Further, the temporal element is often important in pathway analysis. We can currently accommodate pathways that differ by, say, developmental stage, just as we do pathways that differ by tissue or cell type, cell line, or species, simply by representing the stage of each pathway as a separate graph. However, the SBGE cannot currently support more complex temporal queries, such as those based on J. F. Allen's interval-based temporal logic [46]: e.g., find subgraphs in a large pathway graph in which the reaction involving reactants A and B temporally overlaps the reaction involving reactants C and D; or find subgraphs in which the reaction involving A and B ends just as the reaction involving C and D emerges. In a more practical vein, we plan to build loaders to enable users to import graph data represented in the emerging XML standard formats, e.g., Biological Pathways Exchange (BioPAX) [47] and PSI-MI [48].

Related work

For some classes of DBMSs, graphs can be represented directly within the data model. Network DBMS products were built around the idea of storing data objects and the associations among them and then using procedural code to navigate the resulting graph. In more recent times, one of the reasons for developing object-oriented DBMS technology was that SQL DBMS engines were perceived to be very inefficient in the way they managed graph data models.

Advocates for these systems underestimated the utility of declarative queries in information management and the value of integrating graph queries with more traditional relational queries. Database developers have proven unwilling to give up the productivity of SQL in exchange for improved performance in a relatively minor function. Specialized graph DBMSs, which usually employ a declarative query language specific to graphs, suffer from the same deficiency.

Because of the difficulties of supporting graphs in SQL, some commercial RDBMSs adopt a hybrid approach, relying on the RDBMS for storage but layering logic outside to perform the graph algorithms, e.g., in Java classes. From the point of view of the RDBMS, this approach makes graph operations a dead end; i.e., the results of a graph operation cannot be joined with other data sources, counted, stored to disk with guaranteed transaction atomicity, or manipulated in any other way via the declarative SQL query language.

In terms of related work specific to biological networks, KEGG [49] is a preeminent database of biomolecular pathways. It enables the user to retrieve pathways by various methods, such as by protein or compound, but it does not support searches on the topology of the networks or enable the retrieval of subgraphs of the networks.

BioCyc [50] is a venerable project, one of the first to represent and reason about pathways. It uses a frames-based representation implemented in Lisp software. The entire graph database is imported into main memory on initialization rather than enabling the user to define subgraphs of interest; thus, it is not readily scalable to very large graphs.

The BioPathways Graph Data Manager Project [19] aims to construct a general-purpose graph data management system that will be adapted to support biopathways and protein interaction network databases for microbial organisms. Based on a functional programming language model (leaning toward an implementation in Standard ML), this project has gathered and analyzed an impressively wide range of graph-based scenarios in biology but has not yet been fully implemented.

The Pathway Query Language (PQL) [18] is similar to the SBGE in that it is built on top of an RDBMS and addresses similar use cases. However, because it is based on stored procedures rather than UDFs, its ability to compose graph functions in a single query is limited. Because it assumes that all paths in the graph have been precomputed before querying begins, it may be able to handle complex path constraints more easily than the SBGE, but at a significantly higher maintenance cost.

Conclusion

In this paper we have introduced the Systems Biology Graph Extender (SBGE) and described its approach and implementation. Through real-world examples of exploring functional relationships among biomolecular entities via graph queries, we hope to have demonstrated that integrating graph operations into an RDBMS can provide systems biologists with unparalleled opportunities for exploring and predicting functional relationships among biomolecular entities. The SBGE allows graph operations such as neighborhood queries and shortest paths to be seamlessly integrated with the retrieval, sorting, grouping, and filtering of related data—all within a single declarative SQL statement. Extending an RDBMS with graph operations means that all of the robust features that make RDBMSs the data management technology of choice are automatically available for graph data management: for example, query optimization, efficient storage management, integrity checks, recoverability, and scalability.

From the computer science point of view, the heart of the SBGE is a large group of graph-theoretic operations that include compare, combine, find components, compute shortest paths, and flows. These operations are implemented as UDFs that operate on the graph data structures used to hold graph data in memory and to store it on disk. We demonstrated how these graph operations could be combined with one another and with other SQL query fragments to support a number of higher-level operations.

The highlights of the SBGE are the following:

- The SBGE allows graph data to be organized either in SQL tables or as encapsulated data objects. The SBGE allows for a smooth transition between representations.
- A wide variety of graph operations are supported, with the extension being responsible for selecting the optimal algorithms for an operation given the properties of the input graphs.
- Simple graph operations can be combined to compose more complex ones in a single declarative SQL query.
- The extensions can be combined with most other SQL query language features. Because they are well encapsulated within SQL, the user can treat them like any other SQL functions without needing to know the graph algorithms used.

The real power of the SBGE is revealed when it is used in tandem with federated database technology. In this context, the SBGE graph functions can operate on virtually any network data, regardless of its format or location. Annotations on nodes (such as proteins or genes) can be drawn from a wide variety of sources, both public and proprietary. Systems biologists are provided with the richest possible set of data and functions to support their quest to discover, represent, and predict functional relationships, and thus to push the boundaries of scientific understanding.

Acknowledgments

We thank the reviewers for their helpful comments and express our warm appreciation to the following: David Searls, Pankaj Agarwal, and Bill Reisdorf of GlaxoSmithKline Pharmaceuticals for early ideas on biological connection graphs; Paul Shannon and Marta Janer of the Institute for Systems Biology for examples of biologically important graph queries; Dennis Wall of the Harvard Medical School for generously contributing his human–yeast orthology distance data; colleagues from IBM Research and IBM Software Group who helped us organize and participated in the IBM Academy of Technology Study on Database Directions in Systems Biology in March 2003, especially Steve Burbeck, Don Chamberlin, Julia Rice, and June Andersen; Stella Mitchell of the IBM Thomas J. Watson Research Center for implementing the C-to-Java graph converter; and Peter Schwarz, Yu Deng, and Julia Rice of the IBM Almaden Research Center for road-testing an early version of the SBGE in their investigations of similarity metrics over ontologies.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Unleashed Informatics Limited, Ingenuity Systems, Inc., Sun Microsystems, Inc., Daylight Chemical Information Systems, Inc., MDL Information Systems, Inc., BIOBASE Biological Databases, The Board of Trustees, Leland Stanford Junior University, or Oracle Corporation in the United States, other countries, or both.

References

1. The Institute for Systems Biology; see *http://www.systemsbiology.org*.
2. H. Hermjakob, L. Montecchi-Palazzi, C. Lewington, S. Mudali, S. Kerrien, S. Orchard, M. Vingron, B. Roechert, P. Roepstorff, A. Valencia, H. Margalit, J. Armstrong, A. Bairoch, G. Cesareni, D. Sherman, and R. Apweiler, "IntAct: An Open Source Molecular Interaction Database," *Nucl. Acids Res.* **32**, Database issue, D452–D455 (2004).
3. L. Salwinski, C. S. Miller, A. J. Smith, F. K. Pettit, J. U. Bowie, and D. Eisenberg, "The Database of Interacting Proteins: 2004 Update," *Nucl. Acids Res.* **32**, Database issue, D449–D451 (2004).
4. G. D. Bader, D. Betel, and C. W. V. Hogue, "BIND: The Biomolecular Interaction Network Database," *Nucl. Acids Res.* **31**, No. 1, 248–250 (2003).

5. M. Kanehisa, S. Goto, S. Kawashima, Y. Okuno, and M. Hattori, "The KEGG Resource for Deciphering the Genome," *Nucl. Acids Res.* **32**, Database issue, D277–D280 (2004).
6. B.-J. Breitkreutz, C. Stark, and M. Tyers, "The GRID: The General Repository for Interaction Datasets," *Genome Biol.* **4**, No. 3, R23 (2003).
7. M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock, "Gene Ontology: Tool for the Unification of Biology. The Gene Ontology Consortium," *Nature Genet.* **25**, No. 1, 25–29 (2000).
8. D. L. Wheeler, T. Barrett, D. A. Benson, S. H. Bryant, K. Canese, D. M. Church, M. DiCuccio, R. Edgar, S. Federhen, W. Helmberg, D. L. Kenton, O. Khovayko, D. J. Lipman, T. L. Madden, D. R. Maglott, J. Ostell, J. U. Pontius, K. D. Pruitt, G. D. Schuler, L. M. Schriml, E. Sequeira, S. T. Sherry, K. Sirotkin, G. Starchenko, T. O. Suzek, R. Tatusov, T. A. Tatusova, L. Wagner, and E. Yaschenko, "Database Resources of the National Center for Biotechnology Information," *Nucl. Acids Res.* **33**, Database issue, D39–D45 (2005).
9. S. J. Nelson, T. Powell, and B. L. Humphreys, "The Unified Medical Language System (UMLS) Project," *Encyclopedia of Library and Information Science*, M. J. Bates, M. N. Maack, and M. Drake, Eds. Marcel Dekker, Inc., New York, 2002, pp. 369–378.
10. A. L. Hopkins and C. R. Groom, "The Druggable Genome," *Nature Rev. Drug Discovery* **1**, No. 9, 727–730 (2002).
11. L. Salwinski and D. Eisenberg, "Computational Methods of Analysis of Protein–Protein Interactions," *Curr. Opin. Struct. Biol.* **13**, No. 3, 377–382 (2003).
12. E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabasi, "Hierarchical Organization of Modularity in Metabolic Networks," *Science* **297**, No. 5586, 1551–1555 (2002).
13. A. Ma'ayan, S. L. Jenkins, S. Neves, A. Hasseldine, E. Grace, B. Dubin-Thaler, N. J. Eungdamrong, G. Weng, P. T. Ram, J. J. Rice, A. Kershenbaum, G. A. Stolovitzky, R. D. Blitzer, and R. Iyengar, "Formation of Regulatory Patterns During Signal Propagation in a Mammalian Cellular Network," *Science* **309**, No. 5737, 1078–1083 (2005).
14. A. J. Walhout, R. Sordella, X. Lu, J. L. Hartley, G. F. Temple, M. A. Brasch, N. Thierry-Mieg, and M. Vidal, "Protein Interaction Mapping in *C. elegans* Using Proteins Involved in Vulval Development," *Science* **287**, No. 5450, 116–122 (2000).
15. E. M. Marcotte, M. Pellegrini, M. J. Thompson, T. O. Yeates, and D. Eisenberg, "A Combined Algorithm for Genome-Wide Prediction of Protein Function," *Nature* **402**, No. 6757, 83–86 (1999).
16. J. C. Mellor, I. Yanai, K. H. Clodfelter, J. Mintseris, and C. DeLisi, "Predictome: A Database of Putative Functional Links Between Proteins," *Nucl. Acids Res.* **30**, No. 1, 306–309 (2002).
17. C. von Mering, L. J. Jensen, B. Snel, S. D. Hooper, M. Krupp, M. Foglierini, N. Jouffre, M. A. Huynen, and P. Bork, "STRING: Known and Predicted Protein–Protein Associations, Integrated and Transferred Across Organisms," *Nucl. Acids Res.* **33**, Database issue, D433–D437 (2005).
18. U. Leser, "A Query Language for Biological Networks," *Proceedings of the 3rd European Conference on Computational Biology*, Madrid, Spain, 2005, pp. ii33–ii39.
19. F. Olken and K. D. Keck, Biopathways Graph Data Manager (BGDM) Project; see *http://hpcrd.lbl.gov/staff/olken/graphdm/graphdm.htm.*
20. B. A. Eckman, "A Practitioner's Guide to Data Management and Data Integration in Bioinformatics," *Bioinformatics: Managing Scientific Data*, Z. Lacroix and T. Critchlow, Eds., Morgan Kaufmann Publishers, San Francisco, 2003, pp. 35–73.
21. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *J. Molec. Biol.* **215**, No. 3, 403–410 (1990).
22. Ingenuity Systems, *Ingenuity Pathways Knowledge Base*; see *http://www.ingenuity.com/products/pathways_knowledge.html.*
23. H. V. Jagadish and F. Olken, "Database Management for Life Science Research: Summary Report of the Workshop on Data Management for Molecular and Cell Biology at the National Library of Medicine, Bethesda, MD, February 2–3, 2003," *OMICS: J. Integrat. Biol.* **7**, No. 1, 131–137 (2003).
24. M. Stonebraker and P. G. Brown, with D. Moore, *Object-Relational DBMSs: Tracking the Next Great Wave*, Morgan Kaufmann Publishers, New York, 1999.
25. H. Wang and C. Zaniolo, "User-Defined Aggregates in Database Languages," *Proceedings of the 7th International Workshop on Database Programming Languages: Research Issues in Structured and Semistructured Database Programming*, Kinloch Rannoch, Scotland, 1999, pp. 43–60.
26. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler, "GenBank," *Nucl. Acids Res.* **34**, Database issue, D16–D20 (2006).
27. K. R. Christie, S. Weng, R. Balakrishnan, M. C. Costanzo, K. Dolinski, S. S. Dwight, S. R. Engel, B. Feierbach, D. G. Fisk, J. E. Hirschman, E. L. Hong, L. Issel-Tarver, R. Nash, A. Sethuraman, B. Starr, C. L. Theesfeld, R. Andrada, G. Binkley, Q. Dong, C. Lane, M. Schroeder, D. Botstein, and J. M. Cherry, "*Saccharomyces* Genome Database (SGD) Provides Tools to Identify and Analyze Sequences from *Saccharomyces cerevisiae* and Related Sequences from Other Organisms," *Nucl. Acids Res.* **32**, Database issue, D311–D314 (2004).
28. P. D. Karp, M. Riley, M. Saier, I. T. Paulsen, J. Collado-Vides, S. M. Paley, A. Pellegrini-Toole, C. Bonavides, and S. Gama-Castro, "The EcoCyc Database," *Nucl. Acids Res.* **30**, No. 1, 56–58 (2002).
29. Molecule Pages: A Comprehensive Signaling Database; see *http://www.signaling-gateway.org/molecule/.*
30. D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler, "GenBank," *Nucl. Acids Res.* **33**, Database issue, D34–D38 (2005).
31. A. Bairoch, R. Apweiler, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L.-S. L. Yeh, "The Universal Protein Resource (UniProt)," *Nucl. Acids Res.* **33**, Database issue, D154–D159 (2005).
32. N. Deshpande, K. J. Addess, W. F. Bluhm, J. C. Merino-Ott, W. Townsend-Merino, Q. Zhang, C. Knezevich, L. Xie, L. Chen, Z. Feng, R. K. Green, J. L. Flippen-Anderson, J. Westbrook, H. M. Berman, and P. E. Bourne, "The RCSB Protein Data Bank: A Redesigned Query System and Relational Database Based on the mmCIF Schema," *Nucl. Acids Res.* **33**, Database issue, D233–D237 (2005).
33. Daylight Chemical Information Systems, Inc., Daylight Toolkit; see *http://www.daylight.com/products/toolkit.html.*
34. MDL Information Systems, Inc., ISIS; see *http://www.mdli.com/products/framework/isis/.*
35. F. Horn, J. Weare, M. W. Beukers, S. Horsch, A. Bairoch, W. Chen, O. Edvardsen, F. Campagne, and G. Vriend, "GPCRDB: An Information System for G Protein-Coupled Receptors," *Nucl. Acids Res.* **26**, No. 1, 275–279 (1998).
36. A. Bairoch, "The ENZYME Database in 2000," *Nucl. Acids Res.* **28**, No. 1, 304–305 (2000).
37. V. Matys, E. Fricke, R. Geffers, E. Gossling, M. Haubrock, R. Hehl, K. Hornischer, D. Karas, A. E. Kel, O. V. Kel-Margoulis, D. U. Kloos, S. Land, B. Lewicki-Potapov, H. Michael, R. Munch, I. Reuter, S. Rotert, H. Saxel, M. Scheer, S. Thiele, and E. Wingender, "TRANSFAC: Transcriptional Regulation, from Patterns to Profiles," *Nucl. Acids Res.* **31**, No. 1, 374–378 (2003).
38. J. T. Eppig, C. J. Bult, J. A. Kadin, J. E. Richardson, J. A. Blake, A. Anagnostopoulos, R. M. Baldarelli, M. Baya, J. S.

Beal, S. M. Bello, W. J. Boddy, D. W. Bradt, D. L. Burkart, N. E. Butler, J. Campbell, M. A. Cassell, L. E. Corbani, S. L. Cousins, D. J. Dahmen, H. Dene, A. D. Diehl, H. J. Drabkin, K. S. Frazer, P. Frost, L. H. Glass, C. W. Goldsmith, P. L. Grant, M. Lennon-Pierce, J. Lewis, I. Lu, L. J. Maltais, M. McAndrews-Hill, L. McClellan, D. B. Miers, L. A. Miller, L. Ni, J. E. Ormsby, D. Qi, T. B. Reddy, D. J. Reed, B. Richards-Smith, D. R. Shaw, R. Sinclair, C. L. Smith, P. Szauter, M. B. Walker, D. O. Walton, L. L. Washburn, I. T. Witham, and Y. Zhu, "The Mouse Genome Database (MGD): From Genes to Mice—A Community Resource for Mouse Biology," *Nucl. Acids Res.* **33**, Database issue, D471–D475 (2005).

39. R. A. Drysdale and M. A. Crosby, "FlyBase: Genes and Gene Models," *Nucl. Acids Res.* **33**, Database issue, D390–D395 (2005).
40. *Saccharomyces* Genome Database; see *http://www.yeastgenome.org/*.
41. D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM Trans. Info. Syst.* **3**, No. 3, 253–278 (1985).
42. L. M. Haas, P. M. Schwarz, P. Kodali, E. Kotlar, J. E. Rice, and W. C. Swope, "DiscoveryLink: A System for Integrated Access to Life Sciences Data Sources," *IBM Syst. J.* **40**, No. 2, 489–511 (2001).
43. L. Haas, B. A. Eckman, P. Kodali, E. Lin, J. E. Rice, and P. M. Schwarz, "DiscoveryLink," *Bioinformatics: Managing Scientific Data*, Z. Lacroix and T. Critchlow, Eds., Morgan Kaufmann Publishers, San Francisco, 2003, p. 428.
44. M. S. Boguski and G. D. Schuler, "ESTablishing a Human Transcript Map," *Nature Genet.* **10**, No. 4, 369–371 (1995).
45. S. T. Sherry, M. H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin, "dbSNP: The NCBI Database of Genetic Variation," *Nucl. Acids Res.* **29**, No. 1, 308–311 (2001).
46. J. F. Allen and P. J. Hayes, "Moments and Points in an Interval-Based Temporal Logic," *Comput. Intell.* **5**, No. 4, 225–238 (1990).
47. BioPAX: Biological Pathways Exchange; see *http://www.biopax.org/*.
48. H. Hermjakob, L. Montecchi-Palazzi, G. Bader, J. Wojcik, L. Salwinski, A. Ceol, S. Moore, S. Orchard, U. Sarkans, C. von Mering, B. Roechert, S. Poux, E. Jung, H. Mersch, P. Kersey, M. Lappe, Y. Li, R. Zeng, D. Rana, M. Nikolski, H. Husi, C. Brun, K. Shanker, S. G. N. Grant, C. Sander, P. Bork, W. Zhu, A. Pandey, A. Brazma, B. Jacq, M. Vidal, D. Sherman, P. Legrain, G. Cesareni, I. Xenarios, D. Eisenberg, B. Steipe, C. Hogue, and R. Apweiler, "The HUPO PSI's Molecular Interaction Format—A Community Standard for the Representation of Protein Interaction Data," *Nature Biotechnol.* **22**, No. 2, 177–183 (2004).
49. Kyoto Encyclopedia of Genes and Genomes (KEGG); see *http://www.genome.jp/kegg/*.
50. BioCyc Database Collection; see *http://www.biocyc.org/*.

Received October 3, 2005; accepted for publication March 6, 2006; Internet publication October 17, 2006

Barbara A. Eckman *IBM Software Group, 1475 Phoenixville Pike, West Chester, Pennsylvania 19380 (baeckman@us.ibm.com).* Dr. Eckman, a Senior Technical Staff Member in Healthcare and Life Sciences Solutions, joined IBM in 2001 with ten years' experience in bioinformatics research at a Human Genome Project Center and two major pharmaceutical companies. Her current research interests include optimizing scientific workflows using mediator technology, advanced technologies for healthcare interoperability, and extending DBMS technology to address questions in life sciences and clinical research. Dr. Eckman holds a Ph.D. degree from the University of Pennsylvania.

Paul G. Brown *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (pbrown1@us.ibm.com).* Mr. Brown is a Senior Technical Staff Member in the Advanced Databases Research Group. His research interests include scalable systems for data analysis, improving DBMS engine performance, and developing advanced RDBMS extensions. Mr. Brown previously worked at the University of California at Berkeley, Illustra, and INFORMIX.